Sparse Matrix-Vector Multiplication on GPU

Ashish Jindal, Rutgers University

Abstract

The Sparse Matrix-Vector Multiplication (SpMV) kernel computes a vector y as the product of a n by m sparse matrix A and a dense vector x. The massive parallelism of graphics processing units (GPUs) offers tremendous performance which can be exploited for problems like sparse vector multiplication. In this report we discuss 3 implementations of SpMV and show how reordering of data can affect performance of the kernel and also demonstrate methods to exploit memory coalescing for some performance boost.

1. Implementation using atomic operations

An atomic function performs a read-modify-write atomic operation on one 32-bit or 64-bit word residing in global or shared memory. For example, atomicAdd() reads a word at some address in global or shared memory, adds a number to it, and writes the result back to the same address. The operation is atomic in the sense that it is guaranteed to be performed without interference from other threads. In other words, no other thread can access this address until the operation is complete.

Atomic operations in a parallel environment present a real challenge because they serialize execution. Instead of seeing an n Processor parallel speedup, applications that perform an atomic operation on a single counter will only exhibit a sequential runtime. In other words, incrementing a single counter with atomicAdd() means that the counter has to be locked, thus forcing all the parallel threads to stop and wait so they can individually perform the increment operation — one after the other.

The implementation is fairly simple and straightforward and gives very good performance despite the use of atomic operations, but it can also be tuned further based on ordering of data in row major form or column major form as discussed below.

1.1. Data in Column major order

In this data ordering we sort the input list of non-zero elements in column major order. This way all the column 1 values are ordered before column 2 and so on. This means that data reads on the vector 'x' will be clustered in column wise manner and hence would give better cache reads. When we are processing all column 1-2-3 values the processor, which are meant to be multiplied with corresponding 1st 2nd and 3rd value in the vector 'x' again and again, processor will automatically optimize the performance of reads by bringing the corresponding values of vector 'x' into texture cache, thus facilitating very good read performance on vector 'x'. This works well only for matrices with uniform distribution of data among all columns, as only then it can take full advantage of memory coalescing.

1.2. Data in Row major order

In this data ordering we sort the input list of non-zero elements in row major order. This way all the row 1 values are ordered before row 2 and so on. This means that data reads on the vector 'x' will be quite random ias different row items can be accessing various column values in any order. This implementation is not able to fully utilize cache for reads on vector 'x'. and hence has bad performance compared to its column major counterpart.

Another problem with implementation can be that the writes on output vector will not happen in parallel, as multiple threads will be processing data items for the same row which will end up simultaneously executing atomicAdd on same memory locations multiple times in the output vector.

Following figure illustrates the access pattern, in a system with 4 threads and 9 non-zero elements in COO format –

row	[0	0	1	1	2	2	2	3	3]
col	[0	1	1	2	0	2	3	1	3]
val	[1	7	2	8	5	3	9	6	4]

Iteration 0	[0	1	2	3					]
Iteration 1	[				0	1	2	3	]
Iteration 2	[								0]

Following are the time of execution of kernels for various input matrices in column major order -

Matrix	Time (us) 512 x 32	Time (us) 512 x 64	Time (us) 512 x 128	Time (us) 256 x 256
cant	5671	3157	1943	1947
circuit5M_dc	57576	31013	19828	19972
consph	8429	4658	2788	2792
FullChip	84050	45038	30778	30223
mac_econ_fwd500	4359	2460	1626	1628
mc2depi	5950	3403	2211	2186
pdb1HYS	5827	3208	1979	1977
pwtk	15454	8372	4989	4633
rail4284	41953	21988	13969	13977
rma10	6358	3570	2187	2149
scircuit	3342	1925	1268	1304
shipsec1	10865	5976	3553	3544
turon_m	3068	1758	1188	1173
watson_2	6229	3460	2201	2201
webbase-1M	9059	5034	3436	3337

We can observe in the above stats that, as the number of the threads in the system increase the performance of this implementation also gets better. Also it seems that the performance is not affected by the number of blocks and block size in the kernel till the total number of threads remain constant.

Following are the time of execution of kernels for various input matrices in row major order -

Matrix	Time (us) 512 x 32	Time (us) 512 x 64	Time (us) 512 x 128	Time (us) 256 x 256
cant	8374	5261	3460	3489
circuit5M_dc	58296	31415	22002	21673
consph	12487	7463	5519	5568
FullChip	92975	55211	39613	39985
mac_econ_fwd500	4635	2641	2110	2150
mc2depi	6100	3449	2250	2276
pdb1HYS	9200	5434	3815	3849
pwtk	22847	13217	8609	8546
rail4284	52955	29222	23260	23199
rma10	9926	6131	4181	4256
scircuit	3371	1942	1489	1491
shipsec1	15512	8853	6414	6384
turon_m	2872	1653	1128	1142
watson_2	6779	3724	2790	2834
webbase-1M	10218	5657	4016	4095

Performance in row-major data is mostly worse than the column-major counterpart except in some cases, where the distribution of data favours row-major format, thus facilitating more bandwidth and cache utilization.

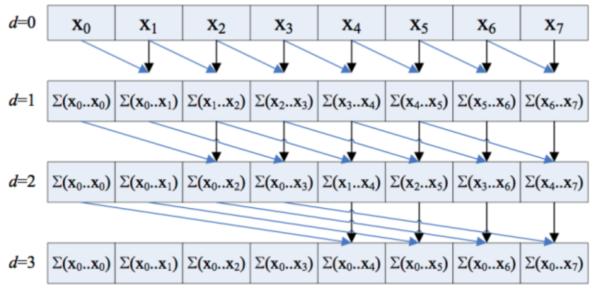
2. Implementation using segmented scan

This approach uses a parallel warp-wide reduction to sum per-thread results together. It implicitly relies on the fact that each warp processes only one row of the matrix and that the input is in row major order.

Following figure illustrates the memory access pattern, in a system with 4 threads and 9 non-zero elements in COO format for this approach -

row	[0	0	1	1	2	2	2	3	3]
col	[0	1	1	2	0	2	3	1	3]
val	[1	7	2	8	5	3	9	6	4]
Iteration 0	[0	1	2	3					]
Iteration 1	[				0	1	2	3	]
Iteration 2	[								0]

The segmented scan happen as illustrated in the following figure -


d=0
x0 x1 x2 x3 x4 x5 x6 x7
d=1
Σ(x0..x0) Σ(x0..x1) Σ(x1..x2) Σ(x2..x3) Σ(x3..x4) Σ(x4..x5) Σ(x5..x6) Σ(x6..x7)
d=2
Σ(x0..x0) Σ(x0..x1) Σ(x0..x2) Σ(x0..x3) Σ(x1..x4) Σ(x2..x5) Σ(x3..x6) Σ(x4..x7)
d=3
Σ(x0..x0) Σ(x0..x1) Σ(x0..x2) Σ(x0..x3) Σ(x0..x4) Σ(x0..x5) Σ(x0..x6) Σ(x0..x7)

But we need to take care of two corner cases -

1. Multiple rows can be processed in same warp, so segment scan should only add up data for same rows.
2. One row can span multiple warps, so each warp should be able to pass on its calculated value to the resulting product array for that corresponding row.

In our implementation we use shared memory to store results of segment scan, which is faster than global memory and when the segmented reduction is complete, we write the result back to the output array in the global memory. To take care of the case where same row may span multiple warps, we use atomicAdd instruction to write the result atomically to the corresponding position in the global array.

2.1. Problems with this approach

This approach has some disadvantages and major one of them is thread divergence. Following is a code snippet for segment scan code -

```
__device__ void
segmented_scan(const int lane, const int * rows, float * vals)
{
  // segmented scan in shared memory, assuming corresponding A values
  // are loaded into the shared memory array vals, the row indices loaded
  // into rows[] array in shared memory
  // lane is the thread offset in the thread warp

        if ( lane >= 1 && rows[threadIdx.x] == rows[threadIdx.x - 1] )
              vals[threadIdx.x] += vals[threadIdx.x - 1];
        if ( lane >= 2 && rows[threadIdx.x] == rows[threadIdx.x - 2] )
              vals[threadIdx.x] += vals[threadIdx.x - 2];
        if ( lane >= 4 && rows[threadIdx.x] == rows[threadIdx.x - 4] )
              vals[threadIdx.x] += vals[threadIdx.x - 4];
        if ( lane >= 8 && rows[threadIdx.x] == rows[threadIdx.x - 8] )
              vals[threadIdx.x] += vals[threadIdx.x - 8];
        if ( lane >= 16 && rows[threadIdx.x] == rows[threadIdx.x - 16] )
              vals[threadIdx.x] += vals[threadIdx.x - 16];
}
```

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.

In the segment scan approach, the divergence mainly comes from the number of logarithmic steps every thread has to execute. Depending on the number of non-zeros of each row in the 32 values being processed, a thread may execute 1 to 4 steps.

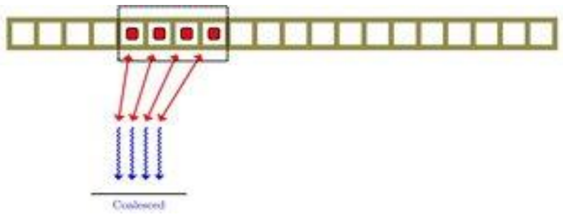
Also since we are statically partitioning the non-zero elements between available warps, that can also have performance issues if distribution of non-zero elements varies drastically among different rows. For e.g. if we have 1000 elements in just the first row and rest of the rows have 10 elements each, then while writing the result back to global memory the access will again be serialized as all threads will be trying to write to same block of memory simultaneously while processing the first row.

As the data in sorted in row-major format, we will also have memory coalescing problems while reading corresponding values from the vector, as they will be accessed in more or less random order.

2.2. Memory coalescing explained

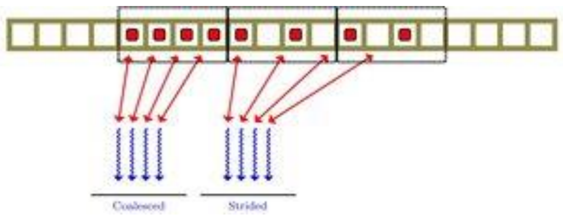
Memory is usually retrieved in large blocks from RAM.Some processing units will try to predict future memory accesses and cache ahead, while yet processing older parts of memory.Memory is cached in a hierarchy of successively larger-but-slower caches.

Therefore, making programs that can use predictable memory patterns is important. It is even more important with a threaded program, so that the memory requests do not jump all over; otherwise the processing unit will be waiting for memory requests to be fulfilled.



The memory accesses are close, and can be retrieved in one go/block (or the least number of requests).

However, if we increase the "*stride*" of the access between the threads, it will require many more memory accesses. Below: four more threads, with a stride of two.



Here you can see that these 4 threads require 2 memory block requests. The smaller the stride the better. The wider the stride, the more requests are potentially required.

Of course, worse than a large memory stride is a random memory access pattern. These will be nearly impossible to pipeline, cache or predict.

Following are the time of execution of kernels for various input matrices -

Matrix	Time (us) 512 x 32	Time (us) 512 x 64	Time (us) 512 x 128	Time (us) 256 x 256
cant	25610	13303	7434	6552
circuit5 M_dc	235873	121671	67033	57354
consph	38248	19870	10984	9646
FullChi p	331033	170569	94222	81451
mac_ec on_fwd 500	16052	8470	4892	4433
mc2depi	24918	13062	7104	6148
pdb1HY S	27515	14327	7912	6906
pwtk	74083	38308	20890	17946
rail4284	145550	75699	42468	37225
rma10	29814	15449	8598	7507
scircuit	12107	6443	3759	3270
shipsec1	49795	25807	14174	12281
turon_m	11246	6024	3413	2938
watson_ 2	22667	12041	6847	6146
webbase -1M	37539	19749	11097	9630

It can be observed in the above implementation that the performance is still worse than its atomic counterpart. Probably the thread divergence makes the performance of this implementation worse.

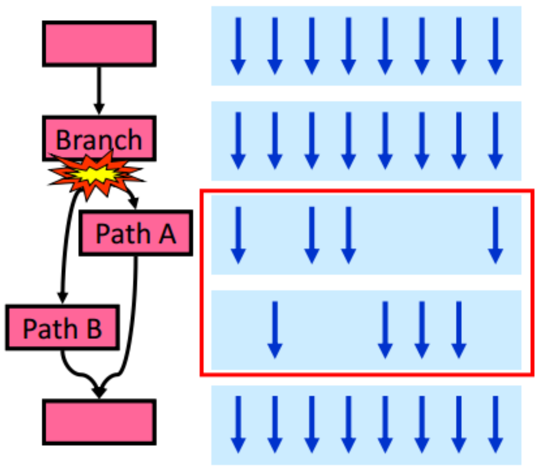
3. Improved performance by reordering

Threads are executed in warps of 32, with all threads in the warp executing the same instruction at the same time.

But, what happens if different threads in a warp need to do different things?

```
if (x < 0.0)
        z = x-2.0;
else
        z = sqrt(x);
```

This is called thread divergence.Thread divergence occurs when threads inside warps branches to different execution paths.



In the segment scan approach, the divergence mainly comes from the number of logarithmic steps every thread has to execute. Depending on the number of non-zeros of each row in the 32 values being processed, a thread may execute 1 to 4 steps.

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, rIndex[ ], cIndex[ ] in COO input 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 coordinate – rindex[0] (row id), cIndexl[0] (column id) is assigned to thread 0. If we change rIndex[ ] and cIndex[ ] to rIndexPrime[ ] and coord cIndexPrime[ ] 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 val[ ] to valPrime[ ] according to the permuted new non-zero elements order so that the segment scan code remains the same.

Following are the steps used to reorder elements in the input -

1. For every row's non-zero elements, divide them into no more than two components, 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. In our case we stop the decomposition when the size of remainder becomes less than 8.

To optimize this implementation further, we also played with number of non-zero elements processed by each warp and varied it from 32 to 4096 and decided to stick with a value of 64, which gives best performance.

The performance results are better than from the previous implementation of the segment scan and show that just by reordering data we can get a better performance. But if we compare its performance with the implementation using atomicAdd, it is still significantly slower. Probably larger and complex kernel means more instructions to be executed by each thread and then there is also the problem of memory coalescing as in this implementation we are accessing data in row-major order but in atomicAdd implementation we use col-major order of data

Following are the time of execution of kernels for various input matrices -

Matrix	Time (us) 512 x 32	Time (us) 512 x 64	Time (us) 512 x 128	Time (us) 256 x 256
cant	25574	13076	6875	6013
circuit5M_dc	232508	117829	63219	53002
consph	38353	19576	10531	8968
FullChip	332302	168586	90453	76998
mac_econ_fwd500	15993	8278	4619	4084
mc2depi	25513	13050	7078	5895
pdb1HYS	27849	14218	7675	6511
pwtk	72812	36983	19703	16303
rail4284	152646	77544	42066	36128
rma10	30136	15390	8291	7009
scircuit	12087	6265	3512	3064
shipsec1	49378	25178	13467	11311
turon_m	11132	5774	3240	2774
watson_2	22859	11771	6515	5638
webbase-1M	38332	19634	10633	9061

.

3. 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] Nathan Bell and Michael Garland. Efficient Sparse Matrix-Vector Multiplication on CUDA. NVIDIA Technical Report NVR-2008-004. December 2008.

[3] 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.

[4] 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.

[5] 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. Onthe-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.

[6] http://www.drdobbs.com/parallel/atomic-operations-and-low-wait-algorithm/240160177.

[7] https://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[8] http://www.nehalemlabs.net/prototype/blog/2014/06/23/parallel-programming-with-opencl -and-python-parallel-scan/.

[9] https://www.udacity.com/course/viewer#!/c-cs344/l-77202674/m-79993366.