Sparse Matrix Vector Multiplication on GPU

Dhruv Dogra, Rutgers University

Abstract

In Sparse Matrix Vector Multiplication, we take a sparse matrix (matrix in which most elements are zero) and calculate its product with a vector. Graphics Processing Units (GPUs) have massively parallel architecture consisting of thousands of core which can be utilized to solve multiple tasks in parallel. In this project we look at Segment Scan implementation of SPMV and its performance on different matrices.

1. Segment Scan Implementation

Segment scan implementation is based on parallel prefix sum within a thread warp. Prefix sum scans through a set of numbers and compute sum based on some "predicates". To understand this suppose we are given a sequence of numbers of X with key K associated with it:

$$K = \{ 0 , 0 , 0, 1, 1, 2, 2\}$$

$$X = \{x0, x1, x2, x3, x4, x5, x6\}$$

Then the prefix sum for this input sequence will be as follows:

$$Y = \{x0, x0+x1, x0+x1+x2, x3, x3+x4, x5, x5+x6\}$$

In our implementation row indexes act as a predicate guard which means that only the multiplication value which corresponds to a particular row will be added together. Before implementing the kernel function we had to preprocess the input matrix. We placed all the non-zero elements consecutively using mergesort algorithm. Mergesort has the best possible time complexity for a comparison based sort.

We also used shared memory in order to store intermediate values. This helped in increasing our speed of task as shared memory access are quicker compared to reading from RAM. Once our product is computed we write only once from shared memory to the main memory.

There are few things we need to keep in mind while implementing segment scan. It could be that one row can span multiple warps and it's not necessary that total threads are a multiple of matrix elements. To handle these cases we used atomicAdd operation and used an if check to ensure we don't work with uninitialized elements.

Issues faced in our implementation is of two types, namely thread divergence and memory coalescing. Let's look at them in the next two sections.

1.1. Thread Divergence

This is one of the major problem present in segment scan. All the threads inside a warp execute the same instruction but it may happen that only few of threads are actually doing the work. The rest of threads are idle thereby wasting computational resources. This problem occurs in segment scan due to the control statement present in the kernel code.

Following is the code snippet for segment scan kernel:

```
__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];
}
```

The 'if' block causes thread divergence and depending on the number of non-zeroes the workload varies for each thread inside same warp.

1.2 Memory Coalescing

Data is retrieved from RAM in chunks and these memory accesses are costly. In an ideal scenario we would like to minimize the number of memory accesses. This is possible if the data we need is present next to each other in RAM. As we have reordered the matrix, as per row, we face memory coalescing problem when we try to access vector data. The non-zeroes positions vary greatly and depending on these positions we get vector data.

Table 1: Performance on 384 CUDA cores						
Matrix	10*192	8*256	5*384	4*512	2*768	2*1024
cant	472	461	462	461	461	464
	5094	2669	2858	2696	3502	2776
circuit 5M	5057	5069	5180	5057	4933	5064
	26810	23960	25966	24188	31570	24787
consph	749	749	748	745	748	760
	4181	3976	4266	4012	5243	4141
pdb1HYS	534	534	543	534	544	537
	2951	2834	2977	2781	3720	2850
pwtk	1446	1423	1404	1445	1435	1498
	7944	7454	7975	7530	9880	10492
Rail4284	3271	3281	3218	3198	3335	3349
	15735	14724	15990	14770	19117	15138
Rma10	557	575	573	573	576	582
	3339	3055	3249	3042	3961	3220
Watson_2	479	519	485	479	480	479
	3061	2815	2847	2787	3633	2832
webbase	766	765	767	757	754	764
	4672	4394	4722	4442	5564	4518
Fullchip	7333	7434	7186	7363	7409	7299
	35676	33027	37273	33460	43578	34417
mc2depi	456	450	459	455	459	458
	3024	2864	3041	2986	3766	2952
Shipesec1	947	952	941	980	939	952
	5417	5092	5500	5384	6748	5298
Turon_m	255	226	226	228	230	228
	1561	1830	1543	1455	1809	1559
Mac_5000	315	316	318	315	318	316
	2115	2051	2163	2047	2541	2115

Table 2: Performance on 192 CUDA cores						
Matrix	10*192	8*256	5*384	4*512	2*768	2*1024
cant	532	524	518	530	518	514
	6760	6464	4042	3962	4656	4093
circuit 5M	5829	5810	5810	5821	5985	5919
	60304	57442	35959	35662	41763	36258
consph	855	845	848	853	857	861
	10037	9616	6019	5961	6966	6129
pdb1HYS	628	663	613	630	669	631
	7100	6776	4154	4083	4854	4179
pwtk	1710	1646	1659	1640	1713	1817
	19111	18192	10890	10617	12772	10835
Rail4284	3932	4081	3945	3927	3955	3906
	37172	35646	22197	21668	25445	21944
Rma10	649	651	657	661	650	662
	7720	7360	4548	4452	5288	4569
watson_2	558	549	541	550	551	532
	6134	5850	3741	3713	4181	3738
webbase	863	881	857	851	854	855
	9753	9330	5863	5745	6701	5835
FullChip	8840	8987	8683	8532	8541	8551
	84597	80984	50840	50067	58257	51179
mc2depi	520	534	516	534	535	508
	6530	6199	3875	3897	4523	4013
Shipesec1	1097	1080	1053	1089	1079	1059
	13027	12429	7682	7514	8862	7728
Turon_m	260	261	262	257	256	250
	3097	2938	1888	1886	2164	1919
Mac_5000	354	346	347	367	363	365
	4404	4290	2882	2809	3149	2799

We did experiments on two machines with different computational powers and created one table for each. Table 1 results were derived from machine containing GPU model GT 730, while Table 2 were from GPU GT 630. Each matrix has 2 rows indicating timing. The first row has unit in milliseconds and it represents the time it took to preprocess data. The second unit is in microseconds and represents kernel time.

We can see from the results mentioned in the tables that Machine 1 performed better than Machine 2. There is marginal improvement in preprocessing time but GPU performance shows massive improvement. The kernel time is almost halved for the configuration 10*192 and 8*256. As the block size increases this difference reduces but still it remains quite high.

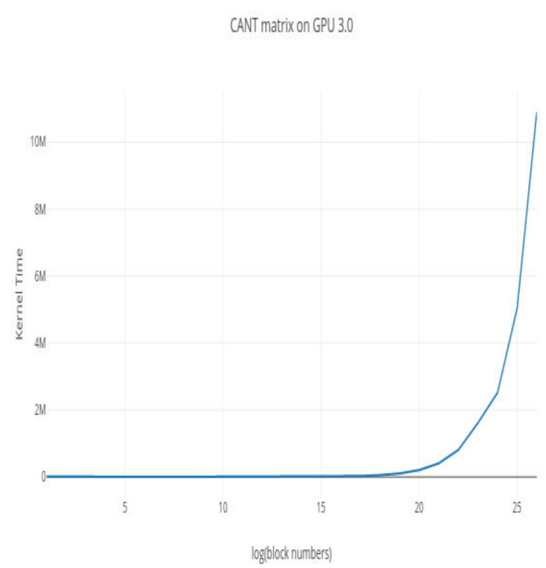
1.3 Performance with constant block size

In this section we'll see what happens when keep the number of threads per block to a constant (256) and try different values for block numbers. We intend to do this for the given sparse matrices on GPUs with distinct computing capabilities.

So CUDA capability with 3.0 and above the maximum number of blocks in x-dimension can be 2^31-1. We shall create graphs to see how the kernel time changes as we increase block number from 2 to highest feasible number.

Let's take **cant** matrix to do an in-depth analysis which we can then extend to other matrices. Also please note the units of graphs-
X-axis = Logarithmic base of 2 of block numbers
Y-axis = Time in microseconds

We used block numbers in the power of 2. So 1 on x-axis means that block number was set to 2, 10 implies block number is 2^10 or 1024.

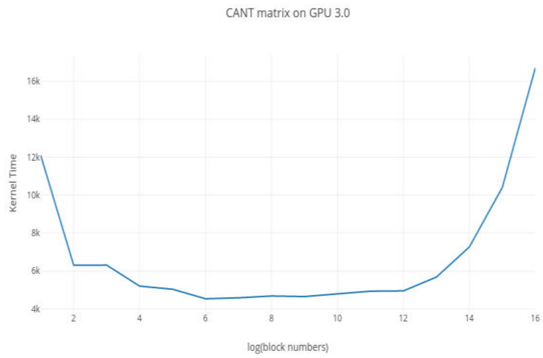


In this graph we can see that as the block numbers increases the time increases exponentially as well. Let's look at the table using which this graph was created. As we can see the Kernel time decreases at first as the block number increases. But as the block number value goes on increasing exponentially the kernel time increases in the same way. Values marked in red indicate when *Errors* were getting reported. When block number was set 2^27, the kernel launch was timed out and no result was given back.

Table containing block numbers and kernel time on GPU with CUDA capability 3.0.

Block Number = Power of 2	Kernel Time
1	12108
2	6309
3	6312
4	5204
5	5040
6	4537
7	4588
8	4684
9	4654
10	4785
11	4937
12	4952
13	5679
14	7270
15	10406
16	16708
17	29283
18	54428
19	104722
20	205341
21	406577
22	808924
23	1614034
24	2520478
25	5039325
26	10900512

Let's take a look at a subpart of this graph where we can see the "bathtub" curve.



If we zoom in on the first graph and take the range of x-axis from 1 to 16 we get the above graph. This graph shows that we attain the best possible kernel time and then time increases exponentially.

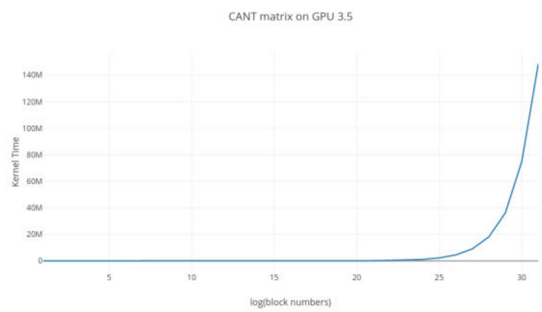
Let's see the performance of the code on GPU with CUDA capability 3.5. The table shows the data we collected as the block numbers were changed.

Table containing block numbers and kernel time on GPU with CUDA capability 3.5.

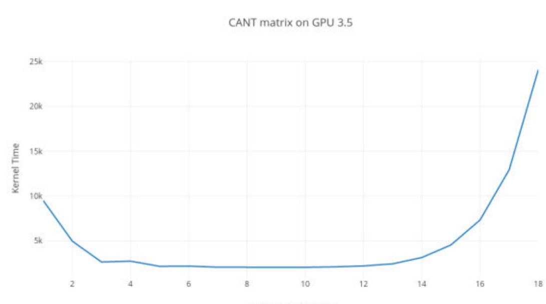
Block Number = Power of 2	Kernel Time
1	9510
2	4983
3	2668
4	2756
5	2181
6	2205
7	2084
8	2076
9	2071
10	2071
11	2122
12	2225
13	2461
14	3159
15	4561
16	7346
17	12939
18	24089
19	46451
20	91143
21	180496
22	359280
23	716775
24	1130790
25	2262150
26	4519284
27	9036377
28	18079876
29	36180225
30	75092652
31	148882043

NOTE: In the last row, block number was set to 2^31 – 1 and not 2^31.

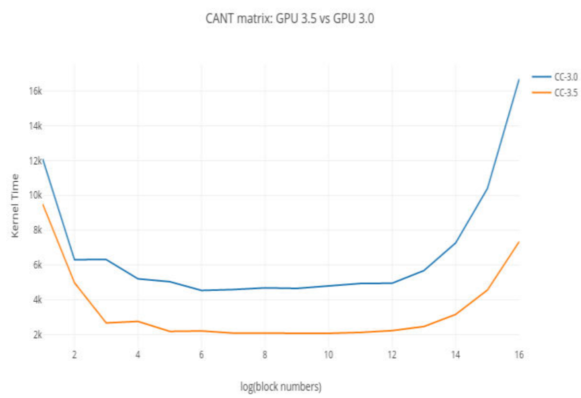
For this GPU we managed to launch the kernel with maximum allowable block number (2^31 - 1). The overall behavior on this machine is same as the previous machine. Initially there is a drop in kernel time and then it increases exponentially. In fact the rate at which we achieve the optimum block number value is same but the magnitude varies. In other words, on GPU 3 and 3.5 we get optimum time when block number is 2^6 but the kernel time is different for both the GPUs.



Above is the general graph cant matrix on GPU with CUDA capability 3.5. Below graph is the bathtub version which is obtained when we set x-axis range from 1 to 18.

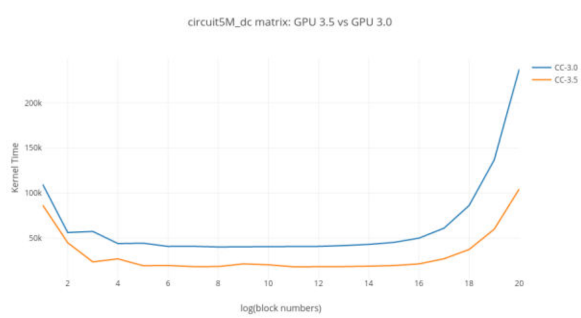


Let's look at the side by side comparison of cant matrix on two machines.



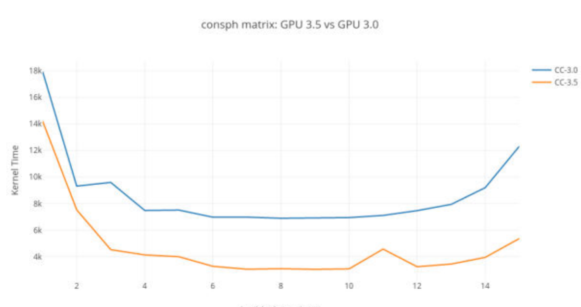
Following are the graphs obtained for the rest of the sparse matrices. We shall only be displaying the comparison graphs for them.

Circuit5M_dc matrix:



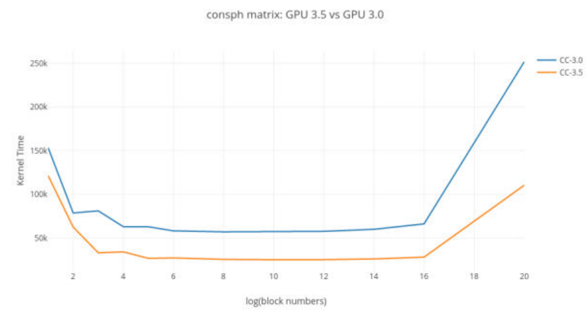
.

Consph matrix:

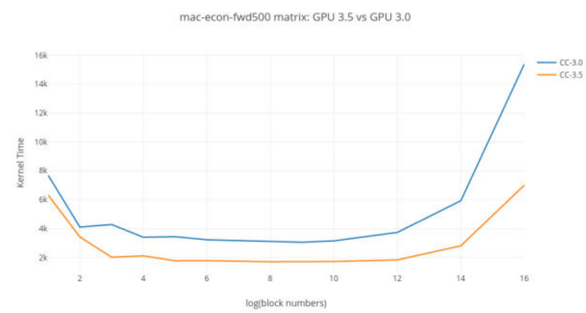


.

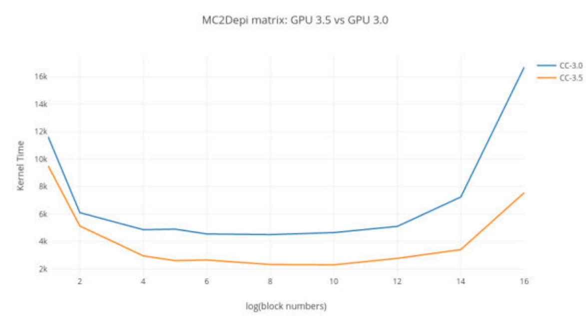
FullChip Matrix:



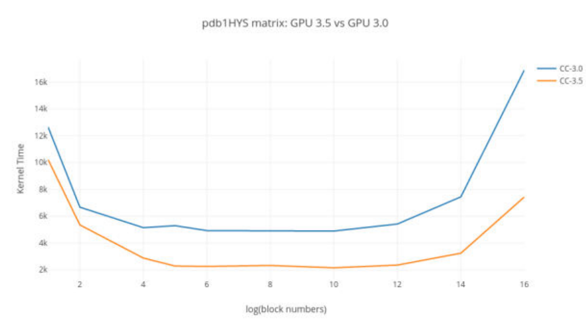
Mac-econ-fwd500 Matrix:



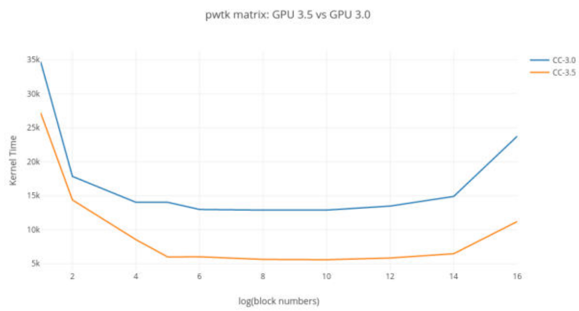
MC2Depi Matrix:



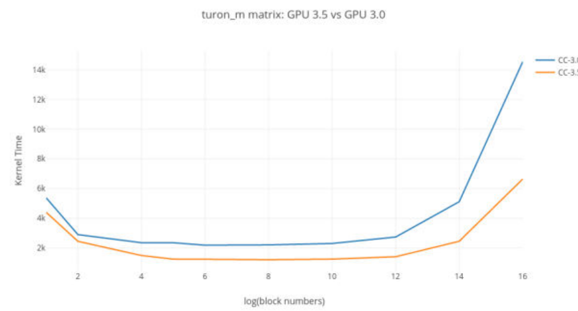
Pdb1HYS Matrix:



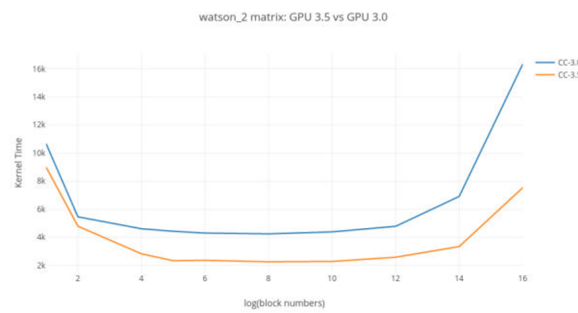
Pwtk Matrix:



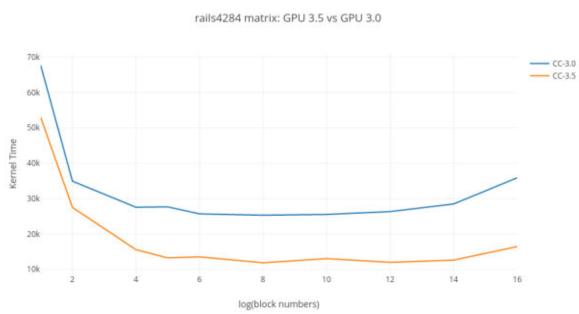
Turon_m Matrix:



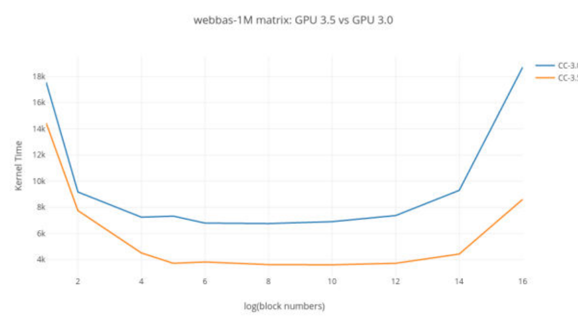
Watson_2 Matrix:



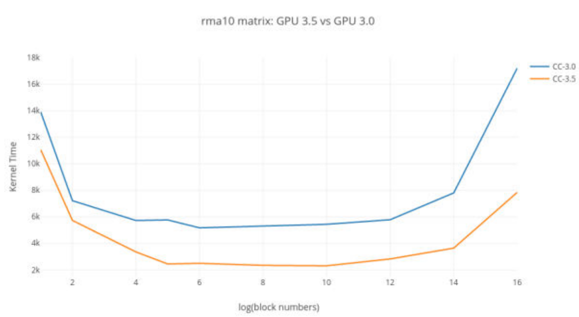
Rails4284 Matrix:



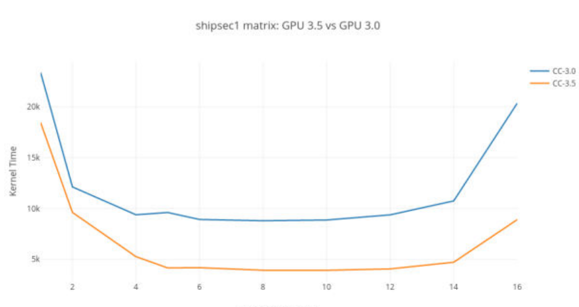
Webbase-1M Matrix:



RMA10 Matrix:



Shipsec1 Matrix:



For every matrix max block number GPU 3.0 fails to launch kernel whereas GPU 3.5 reports a back result after significant time but the output received is 100% incorrect.

2. References

- http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

- "Efficient Sparse Matrix-Vector Multiplication on CUDA" Nathan Bell and Michael Garland, in, "NVIDIA Technical Report NVR-2008-004",December 2008

- https://www.udacity.com/course/intro-to-parallel-programming--cs344

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