Assignment 3

Kshitij Shah - ks1223

Michael Sherman

Subha Srikanth

Dhruv Dogra

Objective

Implement Sparse Matrix-Vector Multiplication (SpMV) in CUDA using three different techniques, namely atomic, segment and our custom algorithms.

Algorithm Design

- **Atomic Algorithm**

  In this approach, different threads can process the same matrix row and communicate by atomic read-modify-write instructions. To ensure exclusive access to memory location atomic instructions are used. This may lead to serialization of execution. Though the kernel implementation is fairly simple, this algorithm gives good results.

  Following table contains the kernel execution time for different matrices:

	Time Taken (in microseconds)		
Input Matrix	**256 * 256**	**512 * 64**	**512 * 128**
cant	1924	3222	2051
circuit5M dc	20115	31178	19429
consph	2750	4690	2861
FullChip	30357	45337	31377
mac econ fwd500	1609	2473	1615
mc2depi	2188	3379	2180
pdb1HYS	1964	3220	1991
pwtk	4831	8309	4940
rail4284	14075	22011	14032
rma10	2130	3535	2280

scircuit	1280	1900	1365
shipsec1	3524	5985	3555
turon_m	1186	1747	1180
watson_2	2219	3474	2195
webbase-1M	3293	5032	3404

- **Segmented Scan**

A segmented scan is modification of a prefix sum with an equal sized array of flag bits to denote segment boundaries on which the scan should be performed. We sort the matrix in this algorithm before executing SpMV. This is done as input matrix is ordered by column. We convert it to a row ordered matrix. This realignment helps in improving memory access thereby increasing performance.

We use shared memory to store the results which improves the performance further. The main issue in this approach occurs due to thread divergence caused within the warp.

Following table contains the kernel execution time for different matrices:

	Time Taken (in microseconds)		
Input Matrix	**256 * 256**	**512 * 64**	**512 * 128**
cant	4643	16667	8576
circuit5M dc	40469	149043	75777
consph	6859	24932	12835
FullChip	57139	209793	106769
mac econ fwd500	3138	10614	5625
mc2depi	4484	15833	8132
pdb1HYS	4862	17480	8959
pwtk	12794	47332	24102
rail4284	25449	91348	46575
rma10	5364	19601	9586
scircuit	2449	7853	4127

shipsec1	8848	32167	16443
turon_m	2165	7173	3826
watson_2	4520	14736	7633
webbase-1M	6736	23787	12248

- **Design Algorithm**

  For our custom algorithm we decided to create a hybrid of previous two techniques. We reorder the input matrix, similar to segmented scan, but the kernel code is same as Atomic algorithm. This reordering improves locality of data, thereby increasing efficiency in memory access.

  Following table contains the kernel execution time for different matrices:

	Time Taken (in microseconds)		
Input Matrix	**256 * 256**	**512 * 64**	**512 * 128**
cant	3478	4861	3459
circuit5M dc	21712	31482	22029
consph	5546	7778	5503
FullChip	40864	54748	40193
mac econ fwd500	2132	2652	2133
mc2depi	2235	3417	2291
pdb1HYS	3832	5458	3824
pwtk	8602	12467	8562
rail4284	23301	29509	23304
rma10	4180	6039	4165
scircuit	1505	1929	1465
shipsec1	6357	9192	6395
turon_m	1144	1641	1128
watson_2	2822	3744	2913
webbase-1M	4076	5692	4045