Louvain algorithm full GPU implementation

Piotr Ambroszczyk

May 11, 2020

The implementation of Louvain algorithm from Community Detection on the GPU paper [1] with some modifications. In particular there are no calculations performed on the host side – the whole algorithm is parallelised to GPU. The implementation [1] uses `thrust` library.

1 Modifications and Optimisations

The biggest difference in my implementation and the original algorithm is single kernel for different degree buckets. Each vertex is assigned to some block, and threads in that block, in parallel, scan edges connected to the node. That way if a node has a small degree then the block assigned to it do the job fast, and switches to the next node. On the other hand, if the neighbourhood of the node is big, then 256 threads, reading from consecutive memory location, perform the job fast.

The final results are aggregated using simple reduction within the blocks, and `thrust::reduce` between them.

In the paper authors used double hashing where size of each hashmap was a prime number. I implemented double hashing as well, though the size of each hashmap was a power of 2. The reason for that was that computation of the next power of 2 can be easily parallelised insted of using binsearch in the precomputed array of prime numbers.

2 Quality tests

I have implemented small tests, the tests and outputs can be find in the `./test` folder. The modularity for them was calculated from the modularity formula by hand.

The program can be run with `-d` flag which activate debug info and, what is even more important, activates additional asserts checking whether the algotithm constrains are preserved, e.g. checks if the sum of k array is the same as the sum of all the weights up to possible float point error[2].

[1] As the paper authors recommended.

[2] The parameter needs to be chosen wisely, because for the big graphs relative error can be even greater then 0.001.

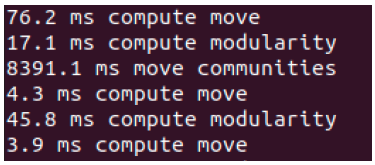

Figure 1: Parallel edges.

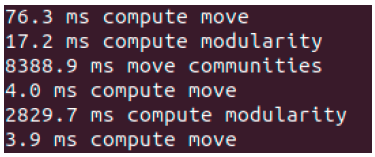

Figure 2: Only parallel nodes.

Last but not least, I have downloaded the hollywood-2009, audi_kw1, F1 and F2 graphs from SuiteSparse Matrix Collection, and checked if the calculated modularity is in the $[-1, 1]$ interval and increases over time.

Graph	Vertices	Edges	Host-Device memory transfer
holywood	1139905	57515616	204.9ms
audi_kw1	943695	39297771	142.8ms
F1	343791	13590452	59.0ms
F2	71505	2682895	14.9ms

Table 1: Sizes of graphs from SuiteSparse Matrix Collection.

3 Experiments

I have tested my implementation on the bruce machine with TITAN V graphic card. I have compiled my program with nvcc and flags
`-Xptxas -O3 -gencode=arch=compute_70,code=compute_70`.

At first, some parts of the implementation did not parallelised an access to the edges. In that solution each thread was responsible for separate node, and this thread performed calculations in a simple for loop over the node's edges. This approach appeared to be very slow, the Figure 1 and Figure 2 present time needed to modules of the program, when `calculate_modularity` was implemented with and without edge parallelism. This situation appeared after the first iteration of the outermost loop on the hollywool-2009 graph. Experiments showed that solution with a edge parallelism performs much better, especially when graph has irregular vertex degree or many accesses to the hashmap/weight memory are needed.

It is hard to compare results from [1] with presented implementation, because the authors used two different thresholds, one for the first modularity optimisation phase, and the second one for the further iterations. The paper authors also pointed that the performance was around 10 times worse when single threshold was used. When I run my CPU implementation it appeared to be much worse comparing with GPU version though one has to have in mind, that the CPU version was not optimised for this architecture. On the F2 graph,

GPU implementation took 0.24s to execute, CPU implementation took around 46s on Intel(R) Core(TM) i7-8650U CPU @ 1.90GHz so 192 speedup.

I have also run experiment with different numbers of blocks, and calculated speedup on aforementioned graphs – Table 2, Figure: 3.

Blocks number	**hollywood**	**audi_kw1**	**F1**	**F2**
1	203.9	232.2	96.9	18.0
2	109	118.3	49.6	9.2
4	60.5	60.5	25.6	4.8
8	35	31.5	13.3	2.5
16	22.3	16.2	6.9	1.4
32	15.7	8.7	3.8	0.7
64	12.4	4.9	2.1	0.45
128	11	2.9	1.3	0.31
256	10.1	1.9	1	0.24

Table 2: Time in seconds needed to execute the algorithm excluding data loading.

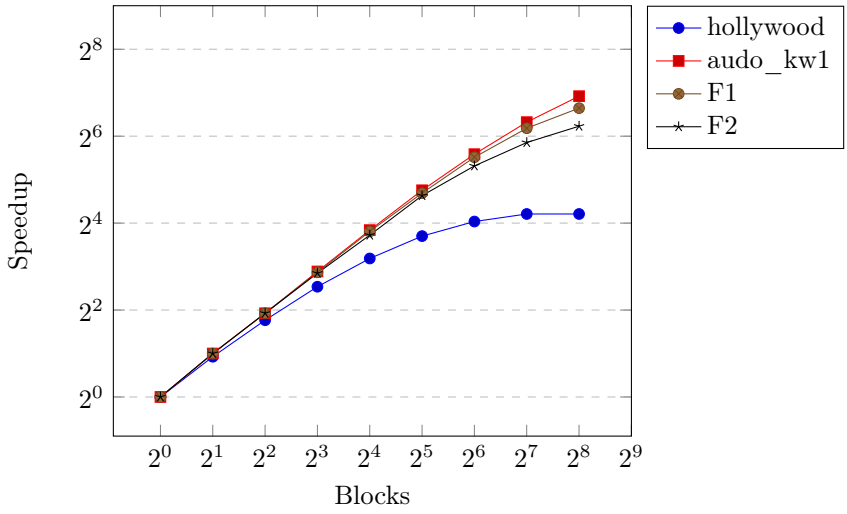

Figure 3: Speedup with respect to the number of blocks, 256 threads.

4 Miscellaneous

At first I was transforming the input data to the memory access efficient format using `std::sort` on tuples, though it appeared to be very slow on the graph

with around 100 million of edges; it took 2-3 minutes. I have moved it to GPU using `thrust::sort` and `thrust::reduce_with_key`, and the time reduced to less then a second.

Thrust library does not support debugging, so it was hard to profile the program. I have tried to separate thrust functions to the separate module which was compiled without debugging information though it did not work for all the graphs. I have managed to run profiler successfully on the audikw_1 graph and results are following.

Time(%)	Time	Calls	Avg	Min	Max	Name
61.28%	1.03998s	59	17.627ms	8.7360us	51.704ms	compute_move(int, int*, int*,
13.24%	224.61ms	67	3.3524ms	3.6800us	6.6392ms	calculate_modularity(int, int
7.30%	123.83ms	78	1.5875ms	992ns	41.422ms	[CUDA memcpy HtoD]
4.68%	79.371ms	67	1.1846ms	3.1360us	2.5759ms	initialize_degree(int, int*,
2.84%	48.144ms	7	6.8777ms	5.4400us	47.960ms	merge_community_fill_hashmap(

References

[1] Md. Naim and others. Community Detection on the GPU, URL: `http://dspace.uib.no/bitstream/handle/1956/16753/PaperIII.pdf`