Louvain community detection - report

Mateusz Biegański

February 21, 2021

Louvain community detection - report

Abstract

I present my implementation of highly concurrent version of Louvain community detection alghoritm, according to Md. Naim, Fredrik Manne *et al* paper **Community Detection on the GPU**. It is superior to another, earlier implementations, because of highly intensive usage of fast shared memory. My implementation is based on CUDA 10.2, and experiments were performed on TITAN V GPU.

Implementation

My implementation is mostly consistent with one presented in title paper. Both modularity optimization and graph merging are highly parallel. In modularity optimization I have extended version from paper, by implementing additional kernel, for handling vertices with degree at most 32, which is equal to size of single warp. I have used much warp-level primitives to make it efficient, they are described broader in Optimizations section. The most crucial thing is, that vertex's neighborhood is computed using block-common, shared memory. I take advantage of it during modularity optimization (computing total weights to neighboring communities) and during graph merging, computing same values, but for whole communities (sets of vertices).

Quality tests

Before I have written parallel solution in CUDA, I have written simple, sequential Python version of Louvain alghoritm. I also agreed proper results on minimal graphs with my colleagues. I append it to my solution as *seq.py*. It served me as a ground truth for agreeing values of Merging Phase on minimal examples with my CUDA version. Because parallel version is not deterministic, values may differ in bigger graphs.

Optimizations

Hashtables data locality

As a method of implementing my hasharrays I have chosen Linear probing, which takes advantage of memory locality, both during inserting and lookup, thus results in few uncached memory accesses.

Lack of modulo operator

Providing size of hash tables is power of 2, I can replace usages of modulo operator (%) during inserting or lookup, for the sake of logical AND operator, which is significantly faster.

Both lack of modulo and data locality can be seen in following code fragment (*hasharray.cu*):

```
    while (true) {
        if (hashtable[slot].key == key) {
            return hashtable[slot].value;
        }
        if (hashtable[slot].key == hashArrayNull) {
            return hashArrayNull;
        }
        slot = (slot + 1) & (table_size - 1);
    }

```

Warp-level primitives

Both in modularity optimization and in graph merging phases I use CUDA warp-level primitives. They are extremely fast, because they operate on common on-chip memory (L1) and are a great way to implement warp reductions or variable broadcasting. Following code performs warp reduction, computing maximum value of variable *var*:

```
    float var = tid;
    for (int offset = 16; offset > 0; offset /= 2) {
        var = fmaxf(var, __shfl_down_sync(mask, var, offset))
    // here thread with tid = 0 keeps value 31

```

I also used another primitives, i.a:

```
    uint32_t mask = __ballot_sync(0xFFFFFFFF, edgeNum < maxDegree / 2);
    // each thread in warp get bit mask with 1 on position k if and only
  if thread k holds edgeNum < maxDegree / 2 predicate

```

Bijective float-int representations

During modularity optimization, each thread computes it's modularity gain for it's associated community. These values are then reduced, and there is one community chosen, with best modularity increase, and, which is important, with lowest index, if multiple with same gain. Avoiding memory races leads to performance bottleneck, thus I take advantage of 64 bits atomicMax, encoding pair (gain, -community), both values in 32 bits, into single 64 bit integer and simply find it's maximum and then decoding. It's not obvious, because I had to find two-sided, order-preserving bijection, mapping float and int. Technically it was bit harder, because I needed unsigned int (U2 representation and it's leading minus bit is not conductive to computing maximum), but draft can be described by following functions:

```
__device__
__forceinline__
int32_t float_to_int(float f32_val) {
    int32_t tmp = __float_as_int(f32_val);
    return tmp ^ ((tmp >> 31) & 0x7fffffff);
}

__device__
__forceinline__
float int_to_float(int32_t i32_val) {
    int32_t tmp = i32_val ^ ((i32_val >> 31) & 0x7fffffff);
    return __int_as_float(tmp);
}

```

Usage of page-locked host memory

I used not only device-allocated memory, but also allocated in host RAM, which page-locked (for not to be swapped to disk).

NVIDIA Thrust routines

In some part of my implementations (especially graph merging) I have taken advantage of thrust::device_vector and associated routines, like reductions or transformations. It is convenient, because it is in accordance with modern C++ features like lambdas, being highly optimized at the same time.

Speedups measuring

In this section I present times that my solution obtained. Speedups are computed mostly against my or my colleague's python sequential solution. I have used matrices from Suite Sparse collection (https://sparse.tamu.edu)

NAME	VERTICES	EDGES	SEQ. TIME	PAR. TIME	SPEEDUP
bcsstk16	4,884	290,378	10,6	1.1	9.6
com-Amazon	334,863	1,851,744	96.18	31.0	3.1
com-dblp	317,080	2,099,732	106.7	22.2	4.8
offshore	259,789	4,242,673	99.3	5.8	17.1

Speedups I obtained are decent; it varies depending on edges/vertices ratio. You can see that the more edges per one vertex, the better performance of algorithm.

* All times measured are full times of program working