Mateusz Machalica April 24, 2014 r.

GPU accelerated implementation of Brandes algorithm

Baseline algorithm

Denote the input graph by $G = \langle V, E \rangle$. There are two natural ways of parallelizing Brandes algorithm:

- a source-based parallelisation, where each thread computes betweenness centrality score for all paths with a given source
- a loop-based parallelisation, where all threads cooperate in execution of loops withing single source iteration

The source-based algorithm is expected to scale linearly in number of processors, since computations for different sources are fully independent and final reduction of intermediate scores can be done in parallel. The problem with this approach is the need of temporary data of $O(|V|)$ size per each thread, therefore for massively parallel processing units, where number of threads p is of the order of 1000, we would need $O(p|V|)$ memory. Even if such memory requirements can be met by the main memory, there is no cache of such size and most of memory accesses will be serialized. There are other limitations of GPU processors that drive our attention to the second approach. Source-based parallelism can be efficiently implemented on CPU, where for $p \leq 8$ we can easily fit G and all intermediate data in the cache.

We implement loop-based parallelism (also known as vertex-based) by assigning a thread to each vertex and processing all vertices with a given distance from the source in a superstep. This way a single source computation requires a number of supersteps no greater than d – the diameter of G. Details of this approach are described in [1]. Underlying graph representation is closely related to CSR format as described in [1], we concatenate adjacency lists of a graph in the order given by vertex IDs and store them in an array `adj`, an additional array `ptr` stores for each vertex `v` an index in `adj` of the first element of its adjacency list. Since lists are concatenated in the proper order, we can access adjacency list of `v` by reading consecutive elements of `adj` between index `ptr[v]` and `ptr[v + 1]`.

Performance measurements methodology

Performance measurements were conducted with the following (mostly social graphs) instances from [5]:

- `ego-Facebook`
- `soc-Epinions1`
- `soc-Slashdot0811`
- `soc-Slashdot0922`
- `wiki-Vote`
- `p2p-Gnutella31`
- `soc-sign-epinions`
- `loc-Brightkite`

For each instance we measured total execution time, including preprocessing and OpenCL device setup. Directed graphs were made undirected by symmetrisation of corresponding adjacency matrices. We measured relative total time changes for each of above graphs, this way we assigned equal weights to the graphs of different sizes. Measurements were averaged over two runs per each machine (`nvidia1` and `nvidia2`).

General optimisation techniques

Removal of vertices of degree less than two

It is easy to see that vertices of degree no greater than one have betweenness centrality score equal to zero. A less obvious observation is that betweenness centrality score for a tree can be computed in linear time by recursively removing vertices of degree less than two and appropriately updating scores. These observations lead to an algorithm which reduces the size of input graph and ensures that all remaining vertices have degree at least two. All necessary proofs can be found in [2]. There is a significant difference is the reduction algorithm proposed

in [2] and the one used by our implementation. The one described in the paper contains other reductions and is far from linear, our implementation contains cache-efficient linear algorithm. We have abandoned the idea of implementing other reductions described in [2] as they do not improve single threaded running time significantly and are quite complicated.

The reduction has an obvious benefit of decreasing instance size. For social graphs from [5], we have obtained an average reduction of number of vertices by $(50 \pm 13)\%$. The less obvious benefit is *equalisation* of threads within a warp, we no longer can have a situation that some threads in a warp have nothing to do, while the others must process plenty of adjacent edges.

Ordering of the graph

It is beneficial to ensure proper ordering of `adj` (and consequently `ptr`). Since in a single superstep we process all vertices within a given distance from the source, we would like all threads in the same warp to be assigned to the vertices within this distance. It intuitively means that we would like vertices which are close in G to have close IDs. One of possible ways to improve vertex to vertex ID assignment in the graph is to sort it according to BFS visit times from an arbitrary source. Unfortunately this optimisation brings decrease of running time of $(3 \pm 2)\%$ only.

GPU-specific optimisations

SoA for intermediate data

Brandes algorithm uses intermediate data in fairly regular way which might suggest that one can benefit from storing it in the array of structures. Our initial intuition (following [3]) was that storing all intermediate data for a single vertex in a structure and fetching entire structure at the beginning of each kernel would bring significant performance boost. Our measurements showed that this is absolutely not the case and that appropriate approach is to lay out data in structure of arrays, which decreases running time by $(12 \pm 6)\%$ when compared with AoS layout. The other benefit of SoA layout is the possibility of efficient splitting kernels into smaller units which turned out to be a crucial for one of the optimisations covered later in this document.

Vertex virtualization

Technical report [1] describes optimisation which involves assigning multiple threads to process a single vertex in such a way that no thread processes more than D adjacent edges. Optimal value of D was found experimentally. Note that this optimisation involves changing two addition operations to atomic versions, it turns out that it still brings decrease of running time by $(17 \pm 11)\%$. Since there is no atomic addition operation for floats in OpenCL 1.1, we had to emulate it as described in [4] – we later show how to get rid of this inefficiency.

Coalesced accesses to adjacency lists

Note that in vertex virtualization approach each thread assigned to the same vertex of G acceses a continuous region of `adj` of size D. We can coalesce accesses to `adj` within a single vertex by assigning adjacent edges to virtual vertices (threads) in a round-robin fashion as described in [1]. Measured decrease in running time due to this optimisation is negligible $(5 \pm 4)\%$, but it greatly simplifies graph preprocessing phase.

Replacing atomics with parallel reduction

Probably the most important GPU-specific optimisation involved replacing atomic additions with parallel reduction. Each virtual vertex, instead of adding its value to vertex-wide aggregate, put it in a temporary array. A separate kernel was launched with one thread per vertex in order to aggregate partial results from all virtual vertices into a single number. This optimisation is responsible for $(38 \pm 8)\%$ decrease of running time.

Fine grained data dependencies a.k.a. keeping GPU busy

The overhead of launching kernel or scheduling data transfers can be conceptually split into a host-side and device-side[1]. We cannot do much about device-side one but we can try to hide host-side overhead by proper design and implementation of CPU driver (the code responsible for gluing kernels together).

Ideally, we would like to design our driver (and kernels) in such a way that all it does is to schedule execution of certain kernels and wait for final results during entire GPU-based computation. This is unfortunately impossible in most cases, but we can do pretty much as well by making sure that the command queue is not empty until the

[1] By device-side overhead we mean the overhead associated with copying kernel arguments, starting it and enforcing global memory barriers (implied by OpenCL specification) after the execution.

end of the computation, effectively overlapping kernel or memory transfer setup with other kernels execution. Details of this optimisation are documented in the driver code, we will however briefly cover the main idea here.

OpenCL command queue by defaults ensures sequential consistency between kernels and memory transfers and guarantees nothing when it comes to host-device interaction unless one invokes `clFinish()` call, which returns after command queue becomes empty and the last command completes. Our driver implementation uses OpenCL `cl_event` in order to enforce slightly relaxed consistency between the host and the device, we still make use of in-order command queue though. We have managed to replace all but the last one[2] invocations of `clFinish()` with calls to `clWaitForEvents()` in such a way that we never wait for the last scheduled command (we might however wait for the second-to-last one). Obviously all memory transfers are asynchronous, just like kernel invocations.

Tricky parts and possible mistakes

Kernels profiling

The tricky part lies in the implementation of kernel and memory transfer profiling. It is easy to get a 30% run time increase if one tries to wait for every kernel invocation – the reason for this was discussed in previous section. The trick here is to aggregate events in an array and process them after we are sure that most or all of them have already finished. This way we can use OpenCL profiling and accurately measure kernels run time with only 5% – 8% performance hit.

Elaborated explanation from previous section justifies a better in terms of performance, but less accurate, method of measuring total execution time of all kernels. Since proper design of the driver ensures non-emptiness of commands queue until the very end of the algorithm, we can approximate total execution time using CPU wall clock. Our implementation provides both methods so the user can see the difference[3] between both approaches and decide whether it is necessary to loose on performance.

Miscellaneous optimisations and cheats

Asynchronous device initialisation

According to our tests entire process of device initialisation and kernels preparation takes from 300 ms to 500 ms. Since reading the graph and CPU preprocessing takes less than 200 ms for test instances, we can hide this overhead by initialising device in parallel.

CPU and GPU cooperation

Additionally we have implemented CPU source-parallel version of Brandes algorithm. In order to ensure low-overhead allocation of sources for CPU and GPU processing, we use atomic integer and distribute sources to processing units (CPU workers or GPU driver) in nearly round-robin fashion. Optimal number of worker CPU threads was determined experimentally[4]. We have observed linear scaling of CPU implementation performance in terms of number of worker threads (as long as number of threads does not exceed number of CPU cores).

References

[1] A. E. Sariyuce, K. Kaya, E. Saule, U. V. Catalyurek
„Betweenness Centrality on GPUs and Heterogeneous Architectures”

[2] A. E. Sariyuce, K. Kaya, E. Saule, U. V. Catalyurek
„Shattering and Compressing Networks for Betweenness Centrality”

[3] J. Luitjens
„Global Memory Usage and Strategy”

[4] I. Suhorukov
„OpenCL 1.1: Atomic operations on floating point values”
`http://suhorukov.blogspot.com/2011/12/opencl-11-atomic-operations-on-floating.html`

[2] The one after the last command of entire algorithm. After careful reading of OpenCL 1.1 specification one can conclude that we can get rid of this call too. We have decided to leave it as it is, since it is encountered only once and makes no measurable difference in performance.

[3] Our measurements show that the difference highly depends on the algorithm setup and graph size.

[4] We can actually guess it, since we have p cores it is intuitively the most beneficial to use one thread for GPU driver and $p-1$ threads for CPU workers. Our tests confirmed this intuition

[5] J. Leskovec
„Stanford Large Network Dataset Collection" http://snap.stanford.edu/data/