PageRank implementation with OpenCL

EECS587 Final Project Report

Yixing Jiang (ethanjyx)

December 18th, 2014

1 Introduction

Pagerank algorithm [9] is a classic algorithm in evaluating the quality of a web link. The project idea is to implement PageRank algorithm with OpenCL. The PageRank algorithm involves calculating pagerank values based on link graph. In the calculation, the new pagerank value of a node depends on the pagerank values of other nodes that point to this node from previous iteration. Thus there will be a lot of communications involved in this process. Also, the new pagerank value of one node does not interfere with that of another node, which allows parallelization. All in all, to calculate pagerank values with OpenCL is a difficult and interesting idea to explore.

In this report, I will demonstrate the ideas and optimizations I developed in this project - including a non-promising idea with map, a naive implementation with an atomic function in kernel, optimization for copying data and a final implementation of compressed sparse row. Besides, I have tested my implementations and optimizations with a small dataset containing 6012 nodes and another dataset containing more than 5 million nodes. In the end, I achieved great speedups. I have also included my worth-noting findings about OpenCL throughout this report.

2 Initial Idea with Mapping

I started this project with an implementation idea with mapping from every node to all its outgoing links. This idea came up because the initial dataset I acquired did not contain continuous node ids. After researching for container support in OpenCL but not finding anything very helpful, I gave this idea up.

2.1 Algorithm

The algorithm will be as follows in every iteration.

```
for every node i:
    newPr(i) = (1 − d) / N; // require mapping for newPr(i)
    for all every node j there is a link from j to i:
        newPr(i) += d ∗ oldPr(j) / numOutlinks(j);
        // require mapping for oldPr and numOutlinks
    end−for
end−for
```

With incontinuous node ids, data structures such as newPr (which means the new pagerank value for the current round), oldPr, numOutlinks will need associate containers such as map.

2.2 OpenCL Container Support

OpenCL has limited support with containers. Only raw arrays with primitive data types are allowed, while none of C++ STL containers are supported. Boost-compute [2] is the best third party library I have found that partially supports map feature, but it is still premature and not well documented.

In this Boost-compute library, a container called flat_map was implemented. Friendly API was provided for host code, but to perform a lookup in kernel code you will have to literally do a binary search with the buffer transferred from the map, which is still very inconvenient. Also, to insert into this map, time complexity will be O(N).

I installed the library on my Macbook Pro and it worked as illustrated in the example. However, I did not have much confidence in the correctness and support I can get from it. Also, it is a a bad idea to access the whole chunck of buffer in kernel, which should be memory intensive and slow. Therefore, I did not try out implementing this and started thinking about my next steps. Anyways, Boost-compute library is definitely something worth noticing and contributing to.

3 Implementation with Raw Arrays

OpenCL has barely any support for standard library containers. The best option is using arrays. I was able to find a dataset with continuous node ids. In this implementation edges will be stored into arrays and get parallelized to process.

3.1 Dataset

After some searching online, a dataset is found [8] containing continuous node ids, which makes it convenient to use arrays to construct data structures. This dataset contains 6012 nodes and 23875 edges. This dataset will be used for all the following evaluations.

3.2 Algorithm

Here is the algorithm for the kernel function.

```
kernel void pagerank(global int* inlinks,
                     global int* outlinks,
                     global int* numOutlinks,
                     global float* oldpr,
                     global float* newpr,
                     global float* d)
{
    size_t i = get_global_id(0);
    int in = inlinks[i];
    int out = outlinks[i];
    float contribution = d * oldpr[in] / numOutlinks[in];
    AtomicAdd(&newpr[out], contribution);
}
```

Every edge has an inlink and an outlink. In this naive implementation, inlinks and outlinks are stored as separate arrays. This algorithm will parallelize the work by every thread looks at only one edge at a time. By indexing the same position at inlinks and outlinks array, we get the node id pair for an edge. Then the contribution for the incoming link is computed and added with an atomic function AtomicAdd() (this function will be discussed further in Section 4.3).

3.3 Evaluation

As a benchmarking standard, another pure C version (Non-OpenCL in the figure) with the same algorithm was written. In Figure 1, three timings were listed - OpenCL version using GPU device, OpenCL version using CPU device and Non-OpenCL device.

From Figure 1 we can find that Non-OpenCL code even runs faster than OpenCL version. Ideally OpenCL code is parallelized and should run faster than Non-OpenCL code. This is definitely not desirable.

3.4 Analysis

In this section we will discuss what might cause this slow speed, and potential optimization plans.

Basically we can consider one iteration of Pagerank calculation as two steps:

1. Calc: does the actual computation and generates a new round of pagerank values.
2. Copy: copy array of new pagerank values into old array of pagerank values.

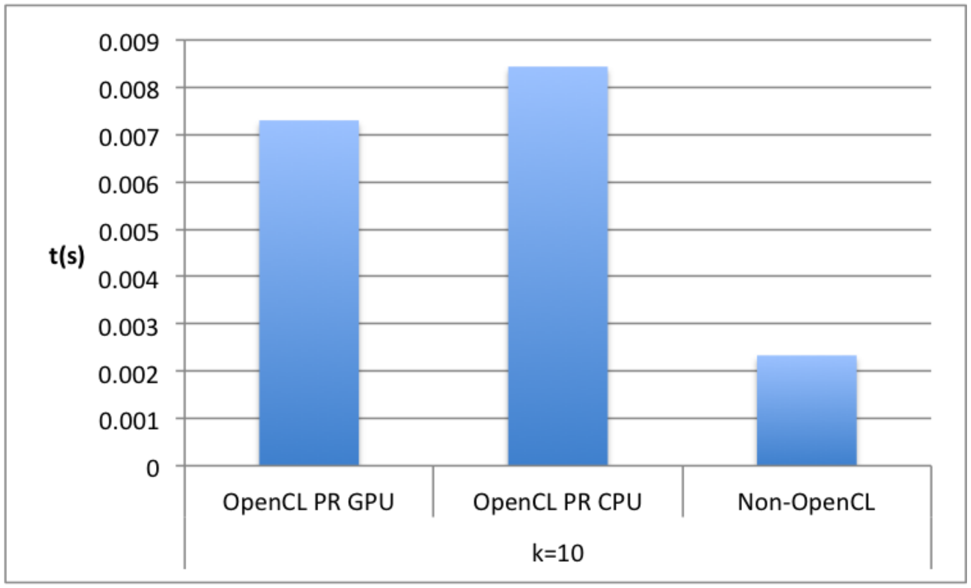

Figure 1: 10 iterations of PageRank. Overall runtime comparisons.

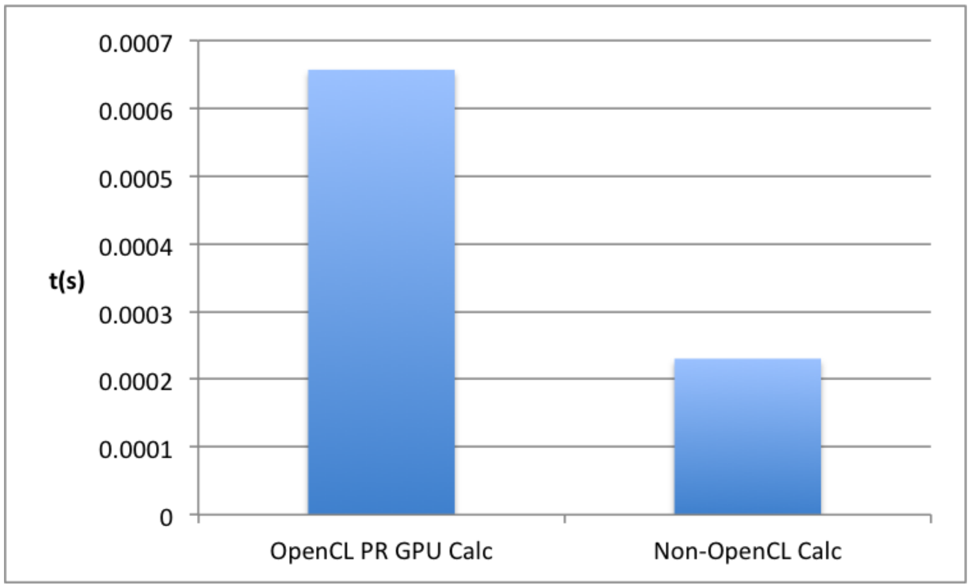

Figure 2: Timing comparison for Calc

To analyze these two parts, timings were done for these two parts separately as shown in Figure 2 and Figure 3. First we analyze the runtime for the Calc step only. We can still find this part is running slower than non-OpenCL code. To solve this, I analyzed the potential problem in Section 4.3 and updated my algorithm for Calc in Section 5.

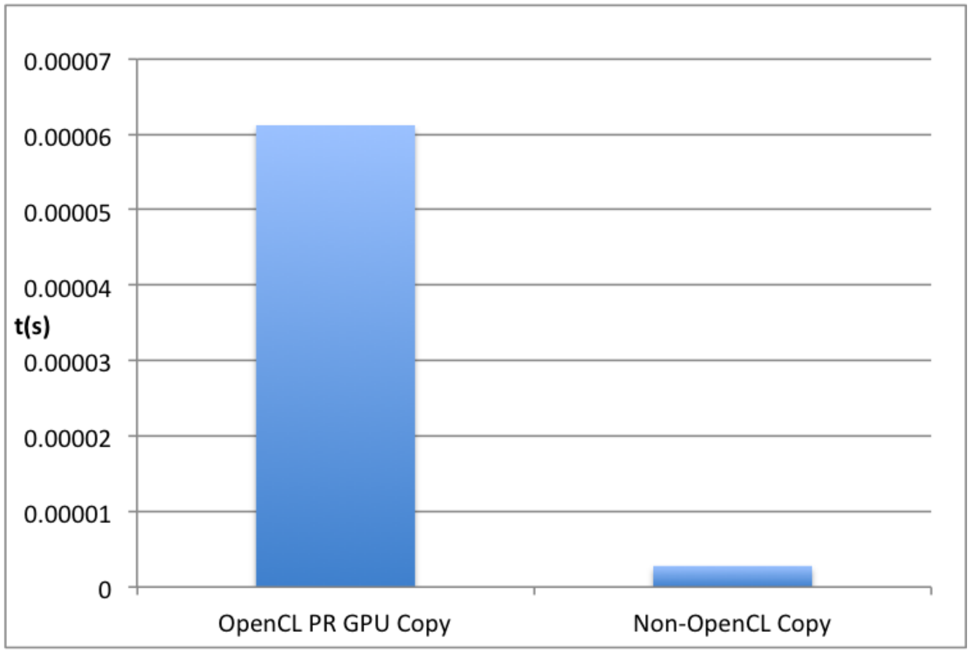

Figure 3: Timing comparison for Copy

Then let us look at the runtime comparison for the Copy step. From Figure 3 we can tell that Copy takes far more time for OpenCL than Non-OpenCL code. The only difference in implementation is that OpenCL uses gcl_memcpy while Non-OpenCL code uses memcpy. gcl_memcpy is much slower because it involves cross device replication of data. In Section 4 I will discuss my optimization for the slow Copy step.

4 Optimization: Improving Copying

In this section I will discuss my optimization in copying new pagerank values into old pagerank values.

4.1 Algorithm

```
kernel void pagerank(
                    global int* inlinks,
                    global int* outlinks,
                    global int* numOutlinks,
                    global float* oldpr,
                    global float* newpr)
{
    size_t i = get_global_id(0);
    int in = inlinks[i];
    int out = outlinks[i];
    AtomicAdd(&newpr[out], oldpr[in]);
}

kernel void exchange(global float* oldpr,
                        global float* newpr,
                        global int* numOutlinks)
{
    size_t i = get_global_id(0);
    if (numOutlinks[i])
        oldpr[i] = d * newpr[i] / numOutlinks[i];
    else
        oldpr[i] = 0;
    newpr[i] = (1-d) / N;
}
```

In this optimization, exchange() function is added to parallelize the linear copying work. Moreover, we can actually precompute the contribution of every node when copying. Since this contribution value will be reused for every other node that one node points to, it will improve the efficiency.

4.2 Evaluation

In Figure 4 we show the timing comparison for between and after this optimization. We can see that the runtime has improved a bit. But still, it is not even faster than Non-OpenCL code. In the next subsection I will analyze why.

4.3 Problem with atomic function

By looking at the kernel code, it can be easily found that AtomicAdd() may take too much time to do, and most importantly, it may be blocking. AtomicAdd() here is different from the OpenCL standard atomic_add() [1]. atomic_add() of OpenCL, weirdly, only supports atomic addition of integers, while the AtomicAdd() here is a function built on top of other atomic functions in OpenCL library [4].

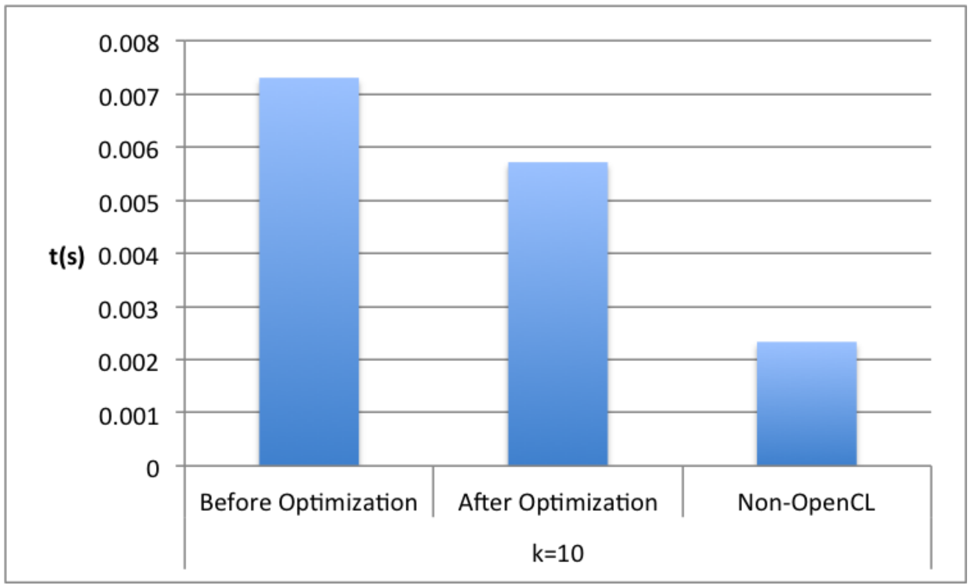

Figure 4: Timing comparison for between and after this optimization

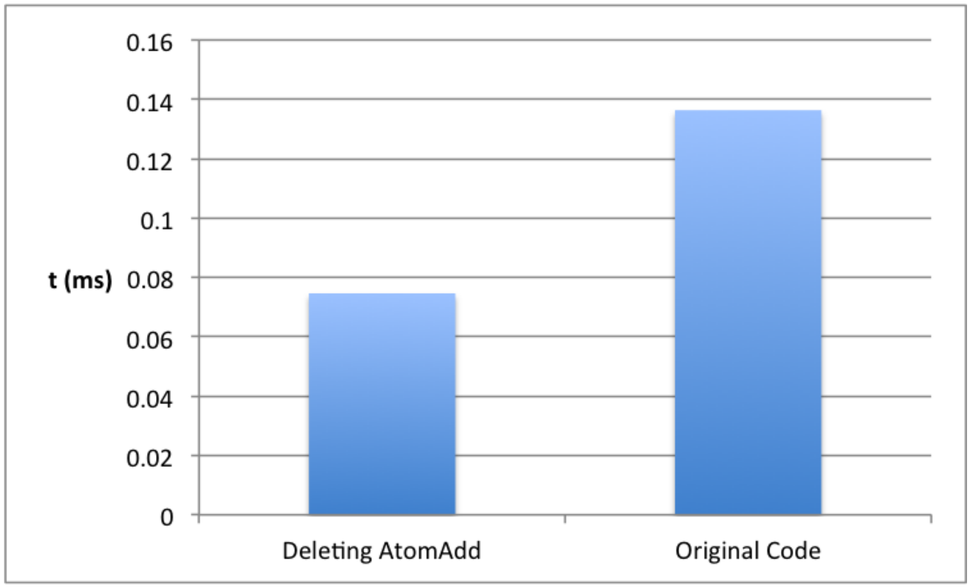

Figure 5: Timing comparison between with and without the atomic function call (Deleting AtomAdd means commenting that line out)

From Figure 5 we can see that this atomic function will take almost half of the overall running time of calculation. To avoid using atomic function, we should arrange that the pagerank value for a specific node will be calculated in only one thread. This encourages us to think about parallelizing in terms of nodes instead of edges.

5 Final version with CSR

Get inspired by the idea of compressed sparse row (CSR) [7], I made another optimization and implemented the idea of CSR.

5.1 Algorithm

The algorithm is as follows:

```
kernel void pagerank(global int* pointers,
                     global int* inlinks,
                     global float* oldpr,
                     global float* newpr)
{
    size_t i = get_global_id(0);
    int index = 2 * i;
    int start = pointers[index];
    int end = pointers[index + 1];
    for(int j = start; j < end; ++j) {
        newpr[i] += oldpr[inlinks[j]];
    }
}
```

inlinks will contain a continuous subarray that have all incoming links of a node. pointers will contain the start and end point of the subarray for every node.

5.2 Evaluation

By implementing this, the speed of the program gets dramatically accelerated. From Figure 6 we can see that now with OpenCL GPU devices it is running faster than other two cases. However, the speedup is not expected and the 4 core CPU version is still slower than the non-OpenCL version. I believe this is caused by relatively large overhead in invoking the devices to work due to the small data(6012 nodes, 23875 edges). In the next section evaluation will be done based on a large dataset.

6 Speedup Analysis

Considering the inneglectable overhead for small dataset, here we perform a deeper analysis based on a larger dataset.

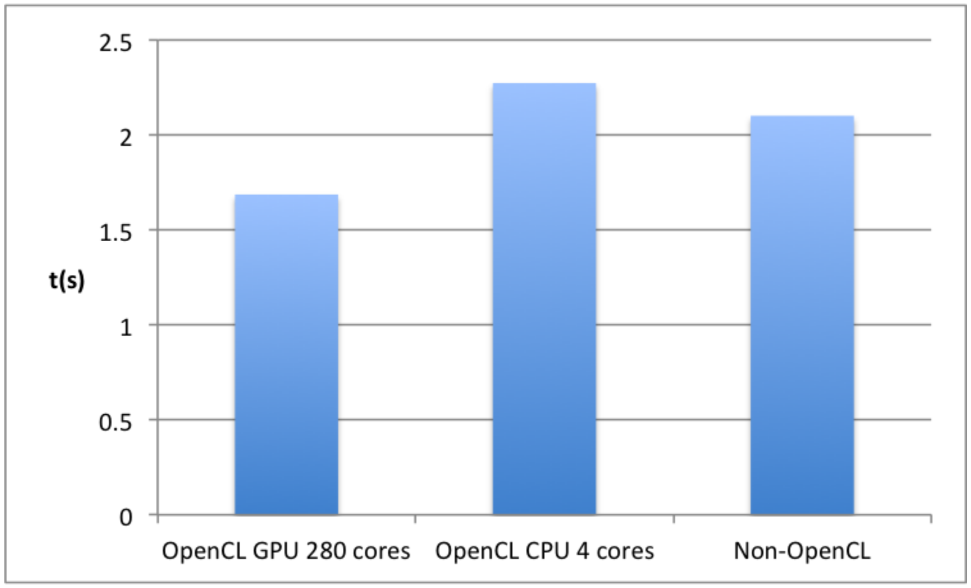

Figure 6: Timing comparison with the CSR algorithm, 10000 iterations

6.1 Dataset

The large dataset to be used here is another wikipedia link corpus. [6] It contains 5,716,808 nodes originally. However, the only device available to me is a 1GB Intel Iris. In order to prevent memory explosion, I will extract subgraph from the whole graph to do the experiments.

6.2 Timings and Analysis

Here we measure the runtime of using 280-core GPU and 4-core CPU (which are default for using GPU or CPU device on OpenCL), then compute and analyze the speedup.

size of subgraph	OpenCL CPU 4 Cores	OpenCL GPU 280 Cores	Speedup
100,000 nodes	0.2903s	0.0139s	20.9564
500,000 nodes	0.7644s	0.0282s	27.0909
1,000,000 nodes	2.1048s	0.0559s	36.0129
2,000,000 nodes	6.4298s	0.1259s	51.0700

Expected speedup is 280 / 4 = 70. Although the expected speedup is not reached, we are close and as we can see from Figure 7, as the graph gets larger, the speedup will get better. The reason is probably as the workload in calculating pagerank values gets heavier, the percentage of overhead is lower.

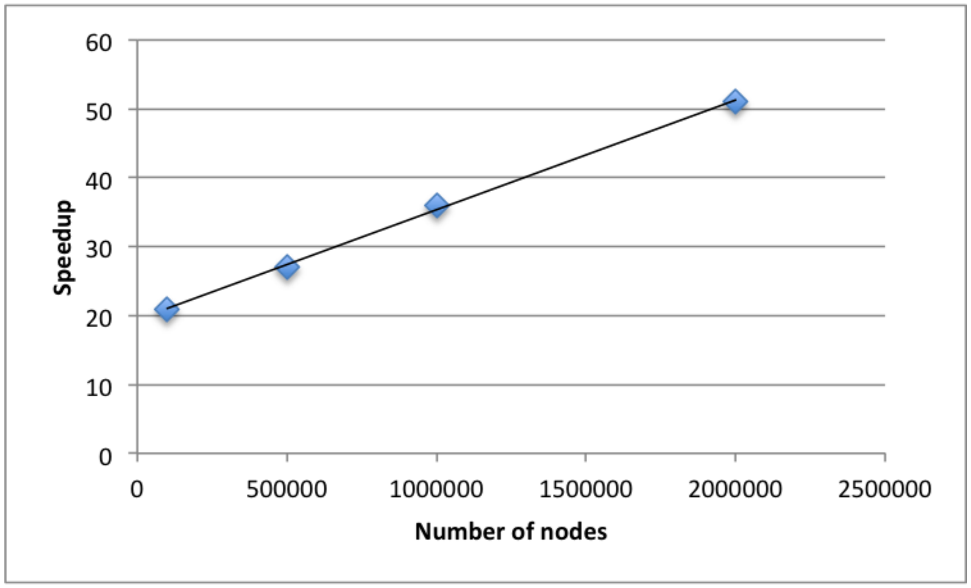

Figure 7: Relationship between number of nodes and speedup

6.3 Controlling number of cores with OpenCL

In order to check the speedup of the program, it is desired that we can control the number of cores. In the latest OpenCL, a new API called clCreateSubDevices [3] is released. However, as I was experimenting, device partition error was always returned. It turned out that device fission feature of OpenCL only supports CPU only [5].

7 Future Work

There are two points I would recommend for future work. First, in OpenCL users can change the workgroup size to tune the performance. I have observed some interesting behavior - the program runs much faster when using workgroup size only 1 than setting workgroup size to NULL which lets the program to select the best workgroup size. Second, I have no access to a graphics card with memory greater than 1GB. If possible, I would like to check the speedup for even greater number of nodes and see whether it will converge to the expected speedup.

8 Conclusion

Throughout the process, compressed sparse row was proved to be an efficient algorithm for Pagerank with OpenCL programming and excellent speedups have been achieved. OpenCL is still in dire needs for support and development.

References

[1] atomic_add - khronos group. `https://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/atomic_add.html`.

[2] Boost compute github repo. `https://github.com/kylelutz/compute`.

[3] clcreatesubdevices - khronos group. `https://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/clCreateSubDevices.html`.

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

[5] Opencl$^{\text{TM}}$ device fission for cpu performance. `https://software.intel.com/en-us/articles/opencl-device-fission-for-cpu-performance`.

[6] Using the wikipedia page-to-page link database. `http://haselgrove.id.au/wikipedia.htm`.

[7] Thilina Gunarathne, Bimalee Salpitikorala, Geoffrey Fox, and Arun Chauhan. Optimizing opencl kernels for iterative statistical applications on gpus.

[8] Amy Langville. Pagerank and web-based information retrieval. `http://www.limfinity.com/ir/`.

[9] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120.