CUDA Force-directed Graph Drawing

Qidu He (qiduh), Di Jin (djin2)

December 2018

Abstract

We parallelize a forced-directed algorithm that considers attractive/repulsive force between any two nodes to draw aesthetically-pleasing graph on both CPU and NVIDIA GPU and compare their performance. We also accelerate the algorithm by applying parallel Barnes-Hut Simulation and achieve more than 10000x speedup on large dataset.

1 Background

1.1 Force-Directed Algorithm

Graph drawing shows a graph based on the topological relationship between vertices and edges. One category of typical algorithms to draw graphs in an aesthetically-pleasing way is forced-directed method. The idea of a force-directed layout algorithm is to consider a force between any two nodes. In this project, we want to implement and optimize a specific version called Fruchterman-Reingold. The nodes are represented by steel rings and the edges are springs between them. The attractive force is analogous to the spring force and the repulsive force is analogous to the electrical force. The basic goal is to minimize the energy of the system by moving the nodes and changing the forces between them. The following image is an example - Social network visualization. Suppose k is the constant describing the optimal length of edge, and d is the distance between two nodes.

Attractive force is $fa(x) = x^2/k$

Repulsive force is $fr(x) = k^2/x$

The algorithm initially assigns each node a random position. It iteratively calculates the repulsive force between all pairs of vertices and the attractive force for each edge. Position of node is changed based on the resultant force vector it receives. The algorithm ends when all node positions do not change or the number of iterations exceeds a certain threshold.

The key data structures are the node and edges. For each nodes, we need to calculate repulsive force from all other nodes and attractive force for nodes directly connected with it. The input of this algorithm is a file contains the

number of nodes and all edges represented by start point index and end point index. The output of the algorithm is a list of coordinates of all nodes.

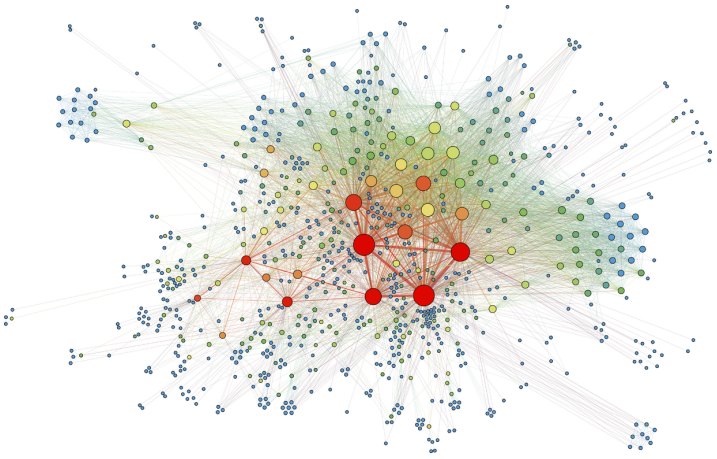

Figure 1: An example of Force-Directed Layout

The algorithm is time consuming. The time complexity of one iteration is $O(n^2)$ where n is the number of nodes. The part of computing repulsive force is most time consuming because all pairs of nodes in the graph needs to be considered. This part can also benefit from parallelism most as the computation has no dependency on others in a single iteration.

```
for i := 1 -> iterations
    for v in V
        v.disp = 0;
        for u in V //repulsive force
            d := v.pos - u.pos
            v.disp += (d/|d|) * fr(|d|)
        for e in E //attractive force
            d := e.v.pos - e.u.pos
            e.v.disp -= (d/|d|) * fa(|d|)

    for v in V //update
        v.pos += (v.disp/|v.disp|) * min(v.disp, t)

    t = cool(t)
```

Figure 2: Pseudo Code of Fruchterman-Reingol

As is shown in the pseudo code, the workload in one iteration contains two part, computing attractive forces, computing repulsive forces and updating positions. The next position of one node depends on every other node, as the accumulated force calculation depends on distances. The position of nodes can be computed in parallel by assigning one node to one thread at a time. The part of computing repulsive force has good space locality and is amenable to SIMD execution, as each node needs to interact with all other nodes, which can be done in the same order of all threads.

However, when computing attractive force, the space locality is not that good. The nodes each nodes connected are usually very different from each other. And the access pattern are random. It's not so amenable to SIMD execution as the connectivity of nodes varies from each other. Fortunately, this part is not so time consuming as repulsive force computation, since large graphs are usually sparse and have much less edges than n^2.

1.2 Barnes-Hut Trees

Barnes Hut simulation is an algorithm to accelarate N-body simulation. It recursively divides n bodies into groups by storing them in a quadtree (octree in 3D simulation). Each node of the tree represents a region in the space. The root node represent the whole space. Each node of the tree can have up to 4 children, each children represents one quarter of the region. The space is recursively subdivided into quarters until each subdivision contains 0 or 1 bodies. The leaf nodes of the tree are bodies. In this project, we use Barnes-Hut trees to solve the problem of high time complexity in repulsive forces computing. It can reduce the time complexity of one iteration of the whole algorithm from $O(n^2)$ to $O(nlogn+m)$. To achieve best performance, we also parallelize the Barnes-Hut simulation.

The key data structure of Barnes-Hut simulation are the quad trees. The two key operations are building the tree and computing forces with the help of the tree. The input of the algorithm is a list of node coordinates and the output is the updated coordinates of the node.

The most time consuming part of the algorithm is building the tree and computing forces with the tree. Both two parts can benefit from parallelism. However, locks are needed to maintain the consistency of the tree when inserting nodes in parallel. We solve the problem with fine-grained locks, which will be described in detail in next section. The locality is not so good as the access pattern of a tree is usually random. Both parts are not amenable to SIMD execution as the number of steps varies for each node. However, we take advantage of SIMD execution for extra precision in computing forces.

2 Approach

We first implement the sequential version of force-directed algorithm. Afterwards, we parallelize it on both CPU and GPU. To further optimize it, we apply Barnes-Hut Tree.

2.1 Sequential Version

In sequential version, we first sort edges according to the start point of it and compute the range of edges for each node. After that, in each iteration, we traverse through all nodes. For each node, repulsive forces from all other

vertices and attractive forces for each connected edge are computed and accumulated. After computation of all nodes are done, the new positions will be updated.

2.2 Parallel Version on CPU (OpenMP)

To better analyze the perfomance with machine and parallelism of program, we take extra time to implement parallel version on CPU through OpenMP other than sequential version on CPU with parallel version on GPU. This version is tested with Xeon Phi.

Each iteration is divided into two parts - force/displacement calculation and update. Add OpenMP annotation for vertexes loop in both parts.

2.3 Parallel Version on GPU (Naïve)

Our naïve GPU version is written in CUDA, tested on latedays with Tesla K40m. In this implementation, nodes are assigned to threads in a Round-Robin way. In each iteration, two kernel functions are launched, the force-computing kernel function and the position updating one.

In force-computing kernel, a thread takes charge of the whole force computation of nodes assigned to it. It will first iterate through all the nodes and compute the repulsive forces from these nodes based on distances. The forces are accumulated in registers and stored as displacements in nodes after computation finishes.

In the position updating kernel, nodes are also assigned to threads in Round-Robin fashion. The position are updated according to the displacement stored in each node.

2.4 Parallel Version on GPU (Barnes-Hut Tree)

To further optimize the performance on GPU, we implement another parallel version with Barnes-Hut Tree. We mainly focus on repulsive force calculation optimization as the graph is sparse for normal dataset. This version perform best among all methods for large dataset.

The implementation is referenced on paper [3]. The work in each iteration is divided into 6 parts, which are realized by 6 CUDA kernels (The work of each kernel is described in detail blow). To achieve best performance, all kernels are parallelized. There are several optimization compared to traditional Barnes-Hut algorithm:

- Minimize Thread Divergence: Group similar work together in the a warp. Combine operations (e.g., fuse similar traversals).
- Minimize Global Memory Accesses: Read common data (shared by threads in a block) once and cache in shared memory. Select a good allocation order for data. Only poll the missing items while waiting.

- Minimize Lock Overhead: Lock as little as possible (e.g., only a child pointer instead of entire node, only last node instead of entire path to node). Use Lightweight Locks such as atomic operation.

2.4.1 Compute Bounding Box

The first step is to determine the smallest region that contains all nodes as the root region. In other words, the upper bound and lower bound of coordinates should be computed. For convenience, this step is done by calling the min_element and max_element from Thrust library directly. The coordinate of the root node and the initial radius are computed as follows.

$x_{root} = (x_{min} + x_{max})/2$

$y_{root} = (y_{min} + y_{max})/2$

$radius = max(x_{max} - x_{min}, y_{max} - y_{min})/2$

2.4.2 Build Tree

To explore space locality while avoid pointer chasing situation, the quad tree is represented with three arrays. A child array stores the children of tree nodes and two array that stores the coordinates of tree nodes. For each nodes, the array stores the indices of its four children. Since the number of leaves equals to the number of vertices in the original graph, the number of nodes in the tree is bounded. In the very beginning, the root node is placed at the end of the array.

Vertices are assigned to threads in an interleaved way. When a thread tries to insert a vertex to the quad tree. It will first go down from the root of the tree, find the path by compare its indices with the indices of the cell nodes, until it hit a NULL pointer (which is represented by a -1 in the child array) or another vertex. There are three possible situations. First, the cell it wants to visit is locked. In this case, it will wait until the cell is unlocked. Second, it hits a NULL pointer, which means the last node it visited is a cell node. In this situation, the thread will first try to lock the position of that NULL pointer with a special value (-2 in our implementation). If it locked the position successfully, the thread will assign the index of the vertex to it. Otherwise the thread will continue going down until hit a NULL pointer. Another situation is that the node hits a vertex. In this situation, the position points to the vertex is also locked first. Then the current region is divided into four smaller regions and a new cell containing both vertices will be allocated. This step will be done recursively until the two vertices fall in two different regions. After that, the position will be unlocked by assigning it with the index of the first allocated cell.

For both the coordinates array and child array, new cells are allocated from the end of the array to the front. The structure of the arrays are shown in the figure below. The first N elements in the arrays are the vertices. In the middle there is some empty space. The cell nodes are placed at the end.

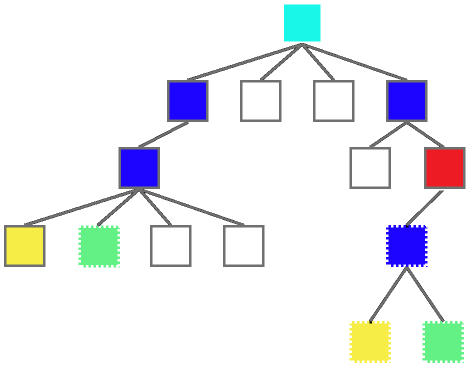

Figure 3: Process of building a tree, blue nodes are cell nodes, yellow nodes are old leaf nodes, green nodes are newly inserted leaf nodes, red nodes are locked nodes

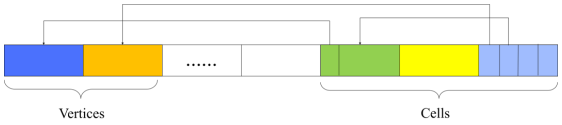

Figure 4: Tree Representation with The Child Array, vertex nodes are in the front of the array, cell nodes are allocated from the end

2.4.3 Summarize Internal Node Information

In this step, the center and the mass(the number of vertices) of each cell nodes are computed. At first, the mass of cell nodes are initialized to -1, indicating that it is not yet computed. Cell nodes are summerized in bottom up order. The computation of one cell node depends on the computation result of its four children. Since cell nodes are allocated from the end of the array. Nodes with small indices are likely to be near the leaves. Cell nodes are assigned to threads in an interleaved fashion. If the children of the node is not ready, it will keep waiting and add the indices of missing children to a stack. After the computation is done. The mass of that node will be updated. Thread fences are used to maintain the order.

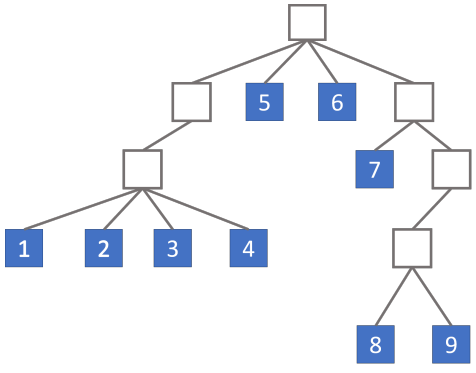

Figure 5: Sample Result of Sorting

2.4.4 Sort the Vertices

Place vertexes into an array such that they appear in the same order as in-order traversal of the quadtree. In this way, group spatially close (in tree) vertexes together. Since the number of vertexes in every subtree is known, the sorting could be parallelized.

2.4.5 Compute Forces

We assign vertexes to threads in a round-robin way. For each vertex, first calculate repulsive force between the vertex with any other vertexes approximately through the quadtree obtained in previous steps. Afterwards, calculate attractive force on all edges connected to the vertex. At the same time, accumulate corresponding displacement.

During repulsive force calculation, one thread will traverse some "prefix" of the tree instead of the whole tree, which is the key idea of Barnes-Hut algorithm. However, since all threads in a warp execute same commands, every thread has to traverse the union of the tree prefixes of all threads in the warp. Sorting could reduce thread divergence to some degree since vertexes close in the quadtree are also close in the sorted array and the prefix are similar for close vertexes. To further eliminate the divergence, every thread expands tree prefix to encompass the entire union. The accuracy is improved without incurring additional runtime.

In this way, threads in a warp always access same global data. To avoid repeated memory access, only one thread per warp will read the pertinent data and cache it in shared memory.

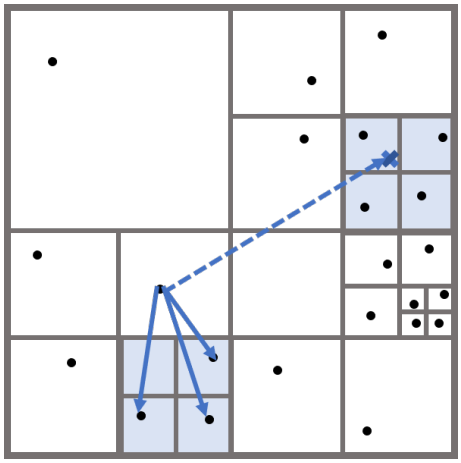

Figure 6: Repulsive force calculation diagram

```
depth := 0;
while (depth >= 0){
    while (/*more nodes to visit*/){
        if (/*first thread in the warp*/){
            // read next node and put in shared memory
        }
    }
    if (/*node is not null*/){
        // get node from shared memory and compute distance
        if ((/*node is a vertex*/)|| all(/*distance >= cutoff*/)){
            // compute repulsive force
        }
        else {
            depth++; // descend to next tree level
            if (/*first thread in the warp*/){
                // push node's children onto stack
            }
        }
    }
    depth--;
}
```

Figure 7: Pseudo code of key part

2.4.6 Update Positions

We assign vertexes to threads in a round-robin way. For each vertex, update the position according to calculated displacement in the previous step.

3 Results

The goal of this project is to parallelize force-directed algorithm on GPU, so we implement naive parallel version and Barnes-Hut tree parallel version with CUDA. To better analyze and understand the performance, we also implement sequential version and OpenMP parallel version on CPU.

During the exploration and after completing the programming, we conduct a number of experiments to analyze our performance as well as seek for better optimization. In general, there are two kinds of experiments. One is to evaluate the performance of program with different elements, such as different number of threads, different size of dataset, different implementation methods and so on. Another is to extract some statistical data of the program like average accessed node and broken execution time in Barnes-Hut algorithm. Results of experiment and corresponding analysis will be demonstrated below.

3.1 Experiments

3.1.1 Introduction of datasets and machines

To better explain our experiments, we first briefly introduce all datasets (real world data chosen from KONECT datasets except for 'Test') and machines that have been used in the project.

Table 1: Dataset

Name	#Vertexes	#Edges	Description
Test	200	199	Man-made data with two centers for testing
Physicians	241	1098	Vertexes represent physicians and edges represent friendship
Wiki-Vote	8340	103722	Wikipedia who-votes-on-whom network
Gowalla	196591	950327	Location-based social network
Texas	1379917	1921660	Road network of Texas
Hyves	1402673	2777419	Social network of Hyves, a Dutch social website

Table 2: Machine

Machine/Method	Information
CPU1/Sequential	Model name: Intel(R) Xeon(R) CPU E5-1660 v4 @ 3.20GHz CPU max MHz:3800.0000 CPU min MHz:1200.0000 BogoMIPS:6385.71 L1d cache:32K L1i cache:32K
CPU2/OpenMP	Model name: Intel(R) Xeon Phi(TM) Coprocessor 5110P Number of cores: 60 (4-threads per core, AVX512 ("16-wide") instruction support) Processor Base Frequency: 1.05 GHz Cache: 30 MB L2 Memory: 8 GB RAM (320GB/sec memory bandwidth)
GPU/Naive + Barnes-Hut	Model name: GeForce GTX 1080 CUDA Cores: 2560 Processor Clock: 1733MHz Standard Memory Config:8 GB GDDR5X Memory Interface Width 256-bit Shared memory: 96 KB Cache: 48KB L1

3.1.2 Comparison among Methods

We obtain execution time of different methods on different datasets. Results are displayed in Table 3. The sequential version runs on CPU1, OpenMP version runs on CPU2(Xeon Phi) with 236 threads, and the remaining two versions run on GPU with 32(blocks per grid)*512(threads per block) threads. Notice that there is no data for Texas/Hyves dataset with sequential version since the runtime is too long. Figure 8 demonstrates the speedup of parallelized program compared to the original sequential version.

Table 3: Runtime(seconds) per iteration

Dataset	Sequential	OpenMP	Naïve Parallel	Barnes-Hut
Test	0.0032	0.0024	0.00019	0.00083
Physicians	0.0048	0.0047	0.00024	0.00084
WikiVote	1.97	0.012	0.0061	0.0026
Gowalla	1096.71	3.54	1.73	0.078
Texas	/	170.3	87.87	2.72
Hyves	/	176.14	91.91	2.8

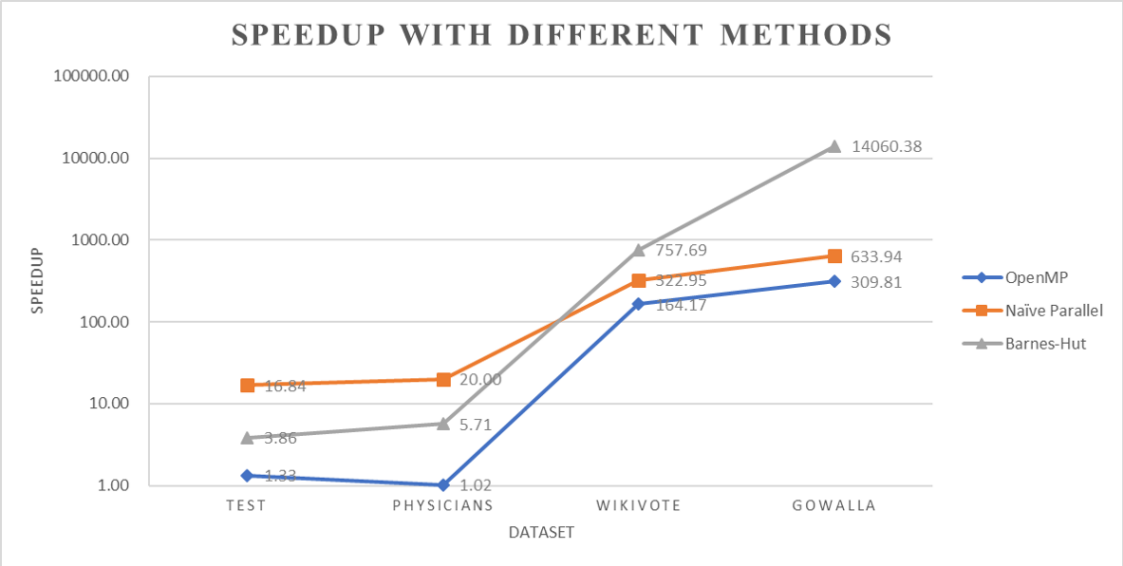

Figure 8: Speedup compared to sequential version

Naive v.s. Barnes-Hut: As Figure 8 shows, there is a intersection between gray line and orange line. It means that naive parallel version performs better with small dataset while Barnes-Hut algorithm performs better with large dataset. Moreover, when the number of nodes increases, the gap becomes larger. It is reasonable because overhead in Barnes-Hut algorithm like building tree becomes smaller as data scales, which is explained in details in section 3.1.4. If overhead is no object, Barnes-Hut will definitely beat naive parallel version since it reduces lots of calculation by approximate repulsive force.

Naive v.s. OpenMP: The key ideas of these two methods are actually similar. The differences are machine and number of threads. Though OpenMP version runs on CPU, it does not perform much worse than GPU. It is not surperising since Xeon Phi is not normal CPU. In a sense, Xeon Phi, similar to GPU, is a co-processor good at parallel task. We also have tested the OpenMP program on CPU1, the speedup is much smaller. An interesting discovery is that though GPU's thread number is more than 30 times more than CPU2(Xeon Phi), the speedup is not the same thing. We think the reason is that the concepts of thread in CPU and GPU are not comparable. A single thread in Xeon Phi is much powerful than a single thread in GPU. Moreover, the thread in GPU will limited by warp. When dataset is small, the gap is large. A possible reason is that the relative overhead of thread creating is larger in Xeon Phi.

3.1.3 Scale with Number of Threads/Problem Size

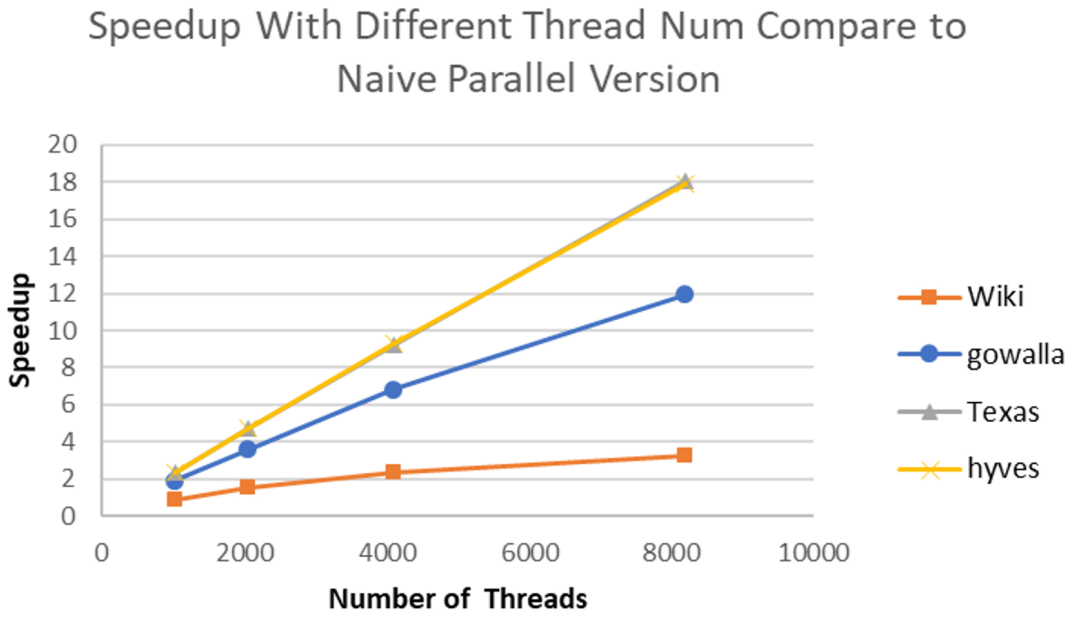

Figure 9: Speedup compared to the naive parallel version with 8192 threads

Figure 9 demonstrates speedup with different number of threads on different datasets. The code we chose as our baseline is our naive GPU version because it takes too long to compute one iteration or large dataset with a single thread. In our baseline, the configuration is 32 blocks and 256 threads in each block. We choose speedup rather than wall-clock time because it is easier to see whether the algorithm scales with speedups. In this experiment, we fix the grid size as 32 and change the number of thread in one block from 32 to 256. The machine we use is ghc with Nvidia GTX1080. The datasets we used are the same as last experiment.

Our algorithm scales well when datasets are large. For large datasets with more than one million points, like Texas and Hyves, the speedup is proportional to the number of threads.

3.1.4 Break Execution Time

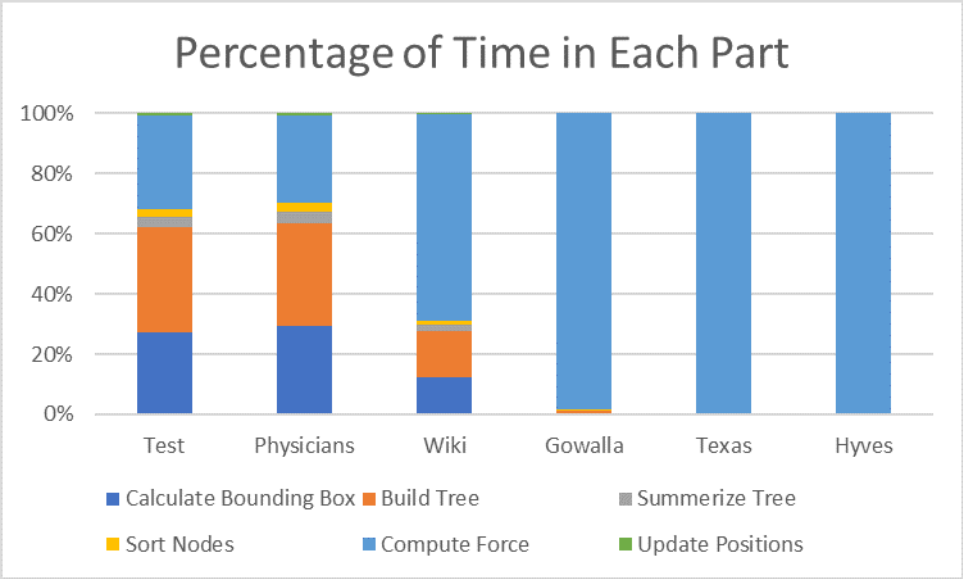

Figure 10: Percentage of Time of Different Kernels

Our focus is on the parallel Barnes-Hut implementation. So the following results are generated by running the parallel Barnes-Hut code. As is shown in 10, when the dataset is small, much time will be consumed in calculating the bounding box of all nodes and building the tree. However, once the size of dataset is larger than one hundred thousand, time of kernels other than computing forces are negligible. Theoretically, the time complexity of building a tree is the same as computing forces. But in our implementation, one warp of 32 vertices acting together. So the average nodes visited might be much larger than $log(n)$. In addition, attractive forces are computed in this step, which is also time consuming.

3.1.5 GPU Profile

The profile is generate from nvprof in CUDA toolbox, the dataset we used here is WikiVote.

- Multiprocessor activity percentage is high in Build Tree, Sort Nodes and Calculate Force is high. In these kernel functions, most of SMs are running all the time. However, the sm efficiency rate is low in Summerize Nodes. We guess the reason might be that in this kernel function, threads need to wait for its child to finish their calculation. The efficiency rate for Update Position is also low. The reason might be that the calculation is quite simple, so

Table 4: Dataset

	Build Tree	Summarize Nodes	Sort Nodes	Calculate Force	Update Position
Multiprocessor Activity	78.04%	42.71%	78.05%	74.49%	43.43%
Warp Execution Efficiency	29.98%	39.39%	54.16%	77.99%	99.87%
L2 read throughput	68.594GB/s	70.112GB/s	72.074GB/s	25.417GB/s	35.388GB/s
L2 write throughput	23.383GB/s	28.165GB/s	18.829GB/s	340.43MB/s	16.655GB/s
Stall(Instructions Fetch)	3.37%	6.59%	7.45%	13.31%	14.92%
Stall(Execution Dependency)	13.89%	16.99%	14.39%	40.34%	33.77%
Stall(Data Request)	75.69%	55.23%	53.23%	39.27%	20.96%
Global Memory Load Efficiency	19.39%	15.30%	47.54%	12.61%	99.95%
Global Memory Store Efficiency	16.90%	29.20%	12.63%	12.54%	99.95%

some threads might finished calculating before others being launched.

- The warp efficiency for Build Tree, Summarize Nodes and Sort Nodes are low. In these kernel functions, the number of instructions each threads execute might be very different from each other, since the path that nearby nodes goes through the tree are different. On the other size, in Calculate Force and Update Position, one warp of threads execute almost same number of instructions, which helps to achieve a high warp utilization.

- The L2 throughput for most kernel functions are the same. However, it is surprisingly slow in Calculate Force. We guess the reason is that only the first thread in a warp takes charge of reading the node from global memory and writing it to shared memory, which potentially slow down memory transmission.

- The reason of stall varies for different kernel functions. For Calculate Force, execution dependency and data request are the main reasons as many steps in this functions depends on previous results. In addition a thread needs to wait for the first thread to read node data from global memory and write into shard memory. In Build Tree, data request is responsible for almost half of stall. The reason is that threads need to compete for the lock with atomic instructions before modifying. This might force the processor to stall and wait for that lock.

- The global memory load and store efficiency is low for kernel functions except Update Position. The reason is that only in Update Position loads and stores can be done in batches and fully utilize the bus bandwidth, while in other kernel functions, loads and stores happen at different time for different threads. As a result, most of the bus bandwidth is wasted.

3.2 Analysis

3.2.1 Speedup Limitation

- **Synchronization Overhead**: Next position of one vertex depends on every other vertex, thus synchronization needs to be done after each iteration. Moreover, there are strict global fences between kernels in one iteration,

which negatively influence the performance.

- **Random Memory Access**:Time locality and space locality of memory access is poor, especially when dataset is large. For example, information of (approximately) all vertexes need to be accessed while calculation displacement of one vertex. It is similar to streaming data access.

- **Attractive Force Calculation**: Since normal datasets are usually sparse, i.e. the number of edges are small, the execution time of attractive force calculation has little impact. Therefore, we choose to optimize repulsive force calculation. The time complexity of attractive force calculation is proportional to the number of edges. Although not significant, the computing will limit the performance.

- **Thread Divergence**: Thread divergence will harm warp execution efficiency and further influence performance. In Barnes-Hut algorithm, thread divergence is pretty severe for kernels other than force calculation kernel and update position kernel. Moreover, though optimized for this problem, the efficiency of force calculation kernel is still less than 80%.

3.2.2 Components Analysis

The percentage of time spent in each region is show in 10. When the dataset is small, time spent in calculating bounding box, building tree and computing force. Percentage spent in computing force keeps growing when the size of dataset getting larger. There is room to improve. When we compute attractive forces, we still use the same method as naive version without optimization. In the future, we can make further optimization by dividing the graph into shreds and improve space locality of current algorithm.

3.2.3 Machine Choice

Our project targets at GPU. However, we actually have used three machines and made some comparison. We believe our original choice (GPU) is wise. The task, force-directed algorithm could make good use of the powerful parallel computing ability of GPU. Though the difference of speedup between OpenMP version and naive parallel version is not large, it is still worth using GPU. The mechanism of parallel is different between Xeon Phi and GPU, and the latter could implement more complicated algorithm such as Barnes-Hut, having higher potential.

References

[1] Fruchterman, T. M., & Reingold, E. M. (1991). Graph drawing by force-directed placement. Software: Practice and experience, 21(11), 1129-1164.

[2] Barnes, J., & Hut, P. (1986). A hierarchical O (N log N) force-calculation algorithm. nature, 324(6096), 446.

[3] Burtscher, M., & Pingali, K. (2011). An efficient CUDA implementation of the tree-based barnes hut n-body algorithm. In GPU computing Gems Emerald edition (pp. 75-92).

[4] konect network dataset – KONECT, April 2017.

[5] Jérôme Kunegis. KONECT – The Koblenz Network Collection. In Proc. Int. Conf. on World Wide Web Companion, pages 1343–1350, 2013.

4 Work Distribution

We closely worked together. The total credit for the project should be distributed 50%-50%.

Qidu He

- Write sequential version.
- Write naive parallel version.
- Write the last three kernels in parallel Barnes_Hut implementation (Sort Nodes, Compute Force, Update Position)
- Write and test OpenMP version.
- Write relevant part in the poster and the report

Di Jin

- Write sequential version.
- Write naive parallel version.
- Write the first three kernels in parallel Barnes_Hut implementation (Calculate Bounding Box, Build Tree, Summarize Nodes)
- Collect datasets and run experiments for sequential and GPU implementations.
- Write relevant part in the poster and the report

5 URL

https://qdhe.github.io/15618-final-project/