Summer Project Summary: GPU Graph Analytics

Runzhou Han
rzhan@bu.edu
adviser: Howie Huang, Yuede Ji

Abstract

Connected component (CC) is a fundamental concept in graph theory. In undirected maps, CC means a subgraph where two vertices are connected to each other. There are many algorithms based on CPU to detect all the CCs in a graph. Frequently used CPU algorithms *Depth First Search* (DFS). During these years, as graphic processing unit(GPU) has been introduced to a larger range of use, people successfully implemented GPU in graph analysis algorithms. In the past decade, GPU algorithms to detect CC in a graph has been a hot spot and a lot of algorithms was introduced to people. Different from CPU CC detection, as limited by some of GPUs properties in design, DFS has been proved to be difficult to achieve on GPUs. Instead, *Breadth First Search* (BFS) is a common way to detect CCs on GPUs, as GPU is designed to have large amount of computing threads which make it possible to analyze huge graph in parallel. GPU BFS is discussed by many people and has been proved to have ideal performance on GPU. However, there are still space for GPU BFS to improve its performance. Our project discussed some possible ways to improve performance of GPU BFS.

I. GPU Architecture

Before we program on GPU, the step 0 is to learn GPU architecture. To make it easier, it's better for us to begin with a comparison between CPU and GPU. CPU basically contains its own logic (or Control), ALUs, memory (Cache and DRAM). CPU has very mature and independent logic and adequate number of ALUs, which promise multiple functions can run on CPU. Comparing with CPU architecture, GPU is designed to focus more on computing ability. GPU has less logic units and memory but much more ALUs than CPU. These ALUs are divided into grids, blocks and threads to make it convenient to achieve parallel computing. In most GPU algorithms, CPU is used as a logic provider and it assign and balance computing work load to GPUs. Figure 1 can clearly demonstrate the differences between CPU architecture and GPU architecture [1].

Figure 1. CPU architecture and GPU architecture.

GPU memory can be divided into global memory, shared memory, texture memory and constant memory. Figure 2 by Yuede Ji gives an intuitive understanding of GPU architecture [2].

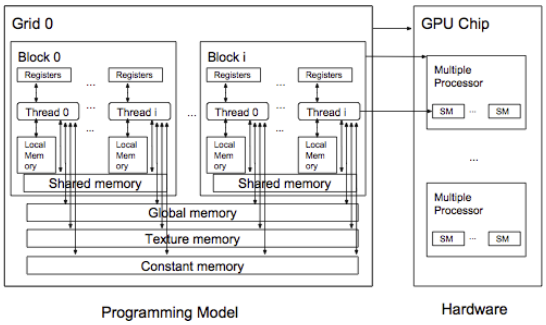

Figure 2. GPU architecture.

II. CUDA programing

After we understand GPU architecture, the next step is to learn GPU coding standard, CUDA. CUDA is an extension of C/C++, which means the basic syntax and writing style is very similar to C/C++. Basic CUDA operations, includes applying for memory, starting a kernel on a device (GPU) with proper number of blocks/threads, and delivering data between device (GPU) and host(CPU). Parallel programming is the most important part of CUDA programing. As every computing unit (thread) has a unique index and computation in threads can be parallel, in a sense, to achieve parallel computing is to tune the thread index and the index of an array in order to keep their consistence. CUDA programming can be basically summary into following steps:

1) Apply for GPU memory.
2) Copy data from CPU memory to GPU memory.
3) Choose the number of blocks/threads to start the kernel function (GPU function) and let GPUs do computations.
4) After the kernel finish the computation, copy the result from GPU memory back to CPU memory.
5) Free device memory.

Pseudo-code is given as follow:

```
const int length = 10;
const int blocks = 2;
Const int threads = 2;

__global__ void kernel(int * array_gpu)
        {       …       }

int main()
{
        cudaSetDevice(0);
        // Set device as the current GPU for the calling host thread
        int * array = (int *)malloc(sizeof(int) * length);
        int *array_gpu;
        cudaMalloc((void**) &array_gpu, sizeof(int) * length);
        //Apply for memory on device
        cudaMemcpy(array_gpu, array, sizeof(int) * length, cudaMemcpyHostToDevice);
        //Copy array fron CPU to GPU
        kernel<<<blocks, threads>>>(array_gpu);
        //Choose block number and thread number and start kernel function
        cudaMemcpy(array, array_gpu, sizeof(int) * length, cudaMemcpyDeviceToHost);
        //Copy the result from GPU back to CPU
        free(array);
        cudaFree(array_gpu);
        //Free memory
```

```
        return 0;
}
```

III. Graph Representation

A graph is composed of vertices and edges. We usually use numbers to identify vertices and use lines that matches them to represent edges, like figure 3.

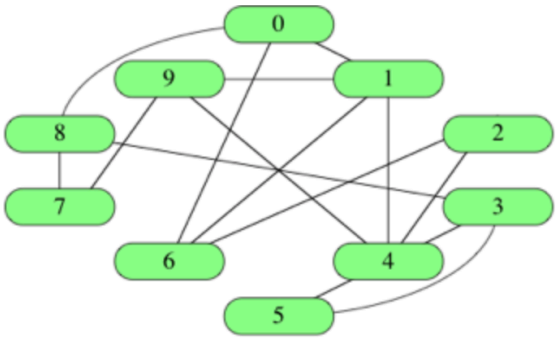

Figure 3. Example of a graph

To analyze a graph, firstly we should choose a proper abstraction to describe it. The frequently used graph representation includes edge lists, adjacency matrices, adjacency lists. Here we choose *Compressed Sparse Row* (CSR) to represent a graph because memory is efficiently used when representing large graphs and it's easy to iterate. A CSR is created from an edge list. It consists of a begin position array and an edge array. The edge array lists all the adjacent vertices of a vertex from vertex 0 to vertex n (here wo suppose vertex number starts from 0). The begin position list tells us the begin position and the end position of adjacent vertices of a specific vertex in the edge array. Figure 4 gives an intuitive understanding of CSR.

Compressed Row Storage / Compressed Sparse Row

	i	1			2		3			4		5	6	7	O
Row Start Index	k	1			5		7			11		13	14	15	Ni+1
	k	1	2	3	4	5	6	7	8	9	10	11	12	13	14		
Column Index	j	1	2	3	4	2	3	3	4	5	6	4	5	6	6		NNZ
Data	a(i,j)	11	12	13	14	22	23	33	34	35	36	44	45	56	66		NNZ

Row Start Index : **1 ≤ i ≤ Ni** : if Data(k) = a(i,j) then RowStartIndex(i) ≤ k < RowStartIndex(i+1)
i = Ni+1 : k = NNZ+1
Order is 2*NNZ + Ni + 1 (=35)

Figure 4. How to build a CSR

Therefore, if we have the begin position array and the edge array, we can iterate all the adjacent vertices of a given vertex using the following pseudo-code:

```
int * begin_position;
int * edges;
CSR(begin_position, edges);
// Build CSR
for(int i = begin_position[vertex]; i < begin_position[vertex+1]; i++ )
{
        int adjacent_vertex = edges[i];
        // iterate all the adjacent vertices in the edge array of the given vertex
}
```

IV. Single thread BFS on CPU

Before we do BFS on GPU, let's first have a look at BFS on CPU. The primary idea of conventional CPU BFS is based on a queue. With a given start vertex, we load all of its adjacent vertices into a frontier, and mark them as read. Then, for each vertex in the queue, we read its adjacent vertices into the queue. We clear the former frontier and the queue is regarded as the new frontier. Then we clear the queue. The signal of the end of BFS is no vertex can be load into the queue, and by the end of the BFS, all the vertices are marked as read. The pseudo-code of the process is given below:

```
int BFS(begin_position_array, edge_array, start_point, vertex_status)
{
        vertex_statu = {unread, … , unread};
        frontier = {start_point};
        while frontier != {}
        {
                for vertex in frontier
                {
                        vertex_status[vertex] = read;
                        int u = begin_position_array[vertex];
                        int v = begin_position_array[vertex+1];
                        for child_vertex in edge_array[u,v]
                        {
                                queue ←child_vertex;
                        }
                }
                frontier clear;
                frontier ← queue;
                queue clear;
        }
}
```

i. Top-Down-BFS[3]

In previous section, we choose a vertex to be a 'parent' and we iterate the CSR array to find its adjacent vertices, which is regard as its 'child'. This parent to child mode is called top-down-BFS.

ii. Bottom-Up-BFS[3]

Top-down-BFS is wildly used and easy to understand. However, it also has some disadvantages. For example, if the frontier is very large, we have to examine a lot of vertices in the CSR array to check if they are marked as read. With the BFS goes on, more and more vertices are marked as read so in the advanced searching steps most of the vertices are read. We consider these read vertices are failure because they don't generate next generation. The failures are meaningless so this operation may waste a lot of unnecessary time.

A better way is to do BFS in a reverse searching direction. Here we introduce bottom-up-BFS.

Different from top-down-BFS where the parent is given and we look for the child, in bottom-up-BFS, the child is given and we look for its parent. For a given vertex, we iterate its adjacent vertices, and once we find a vertex read, we regard this vertex as the child's parent. As in the last few steps of a BFS most of the vertices are read, it's very efficient to find a parent for a child. The pseudo-code of bottom-up-BFS is given below:

```
int BFS(begin_position_array, edge_array, start_point, vertex_status, all_vertices)
{
        vertex_statu = {unread, … , unread}
        vertex_status[start_point] = read
        stop = false
        while stop == false
        {
        stop = true
        for vertex in all_vertices
```

```
        {
                if vertex_status[vertex] = unread

                int u = begin_position_array[vertex]
                int v = begin_position_array[vertex+1]
                for adjacent_vertex in edge_array[u,v]
                {
                        if vertex_status[adjacent_vertex] = read
                        {
                                vertex_status[vertex] = read
                                stop = false
                                break
                        }
                }
        }
    }
}
```

iii. Hybrid BFS[3]

When comparing the two BFSs, we may find that top-down-BFS has better performance in the first half of the searching process, while bottom-up-BFS has better performance in the second half of the searching process. Actually, according to Scott Beamer's study [3], the time comparison is shown in figure 5.

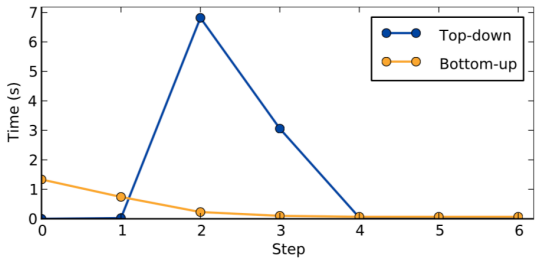

Figure 5. Time comparison between top-down-BFS and bottom-up-BFS.

We can see that in the first step of the searching, top-down-BFS almost cost no time because all the vertices in the frontier are unread. However, between the first step and the forth step, bottom-up-BFS has much better performance than top-down-BFS. Base on this fact, it's easy for us to think about mixing them up into a hybrid algorithm. The new BFS can switch between top-down and bottom-up depending on a threshold. The threshold can be a constant (static switch) or a variant depending on some factors (dynamic switch). In Beamer's study, the threshold is designed to be a comparison between number of vertices in the current frontier and a quotient between number of unread vertices and a custom factor.

V. Multi-thread BFS on GPU

Based on the understanding of CPU BFS, we are able to transplant BFS to GPU. In CPU BFS, we need to rather iterate the frontier or the entire vertex set to generate new frontiers in different sizes, which means we need to apply for memory for new frontier in every loop. However, in CUDA programming we can't apply for memory in kernel function but can only do it outside the kernel function. Therefore, we need to choose an array that doesn't change in size as our frontier. In our GPU BFS, the vertex status array is an ideal choice as its size is a known constant and we can directly check whether a vertex has been read. The basic idea of top-down-BFS on GPU is：

1) Choose a start point and mark its level as 1, and define a Boolean variable as stop sign of BFS.
2) Iterate the vertex status array, find out the vertices with the target level (for first loop is 1).

3) Iterate CSR array to find out these qualified vertices' (parent) adjacent vertices(children).
4) If the parent has child/children, mark child's level as its parents' level plus 1 and set the stop sign as false. If the parent doesn't have any children, set the stop sign as true and break the iteration.

VI. Connected component

In graph theory, a connected component of an undirected graph is a subgraph in which any two vertices are connected to each other by path, and which is connected to no additional vertices in the supergraph [4].

i. Weakly connected component (WCC)

In directed graphs, a WCC is a maximal subgraph of a directed graph such that for every pair of vertices u, v in the subgraph, there is an undirected path from u to v and a directed path from v to u.

ii. Strongly connected component (SCC)

In directed graphs, if every vertex of a subgraph is reachable from every other vertex, we call it a strongly connected component.

VII. Finding WCC using multi-thread BFS on GPU

We have already discussed how to achieve multi-thread BFS on GPU, and the concept of WCC. As WCC is based on directed graph and the CSR in BFS is built on undirected graph, we need to think about a way to converted a directed graph into an undirected graph. There are some methods to do this. For example, in the edge list of a directed graph, if there exists an edge u→v, we can consider it as u→v and u←v, and insert the new edge u←v into the edge list. For our project, we used another way. We build a CSR for vertices on the left side of the edge list (forward CSR), and another CSR for vertices on the right side of the edge list (backward CSR). In BFS, given a vertex, we first check its adjacent vertices through the forward CSR and update their levels, and then do the same thing with the backward CSR. The idea of the code to detect WCC is：

1) Build forward and backward CSR.
2) Give a pivot.
3) Do forward/backward BFS.
4) Label the vertices found in BFS as WCC under the given pivot.
5) Label the levels in a WCC

We give the pseudo code of GPU top-down-BFS to detect WCC as following：

```
define array wcc = {-1,-1, … , -1}
build forward CSR
build backward CSR

for v in wcc
        if v = -1
                start_point = index of v
                v = index of v
                BFS(forward CSR)
                wcc[…] = index of v
                BFS(backward CSR)
                wcc[…] = index of v
        end if
end for
```

Figure 6 is the current performance of our BFS on some popular graphs. The GPU we used is P100. Both forward and backward CSR were used and we start with top-down BFS, switch to bottom-up BFS at layer 3.

graph	time/ms at layer 3
BD	54.2581
DB	86.9332
FL	21.528
LJ	30.8232
PK	11.1449
WE	59.6299
WL	1452.28
HD	35.4259

Figure 6. Time of searching the largest WCC in different graphs. Start with top-down

VIII. Summary

i. What we have done

1) CPU single thread top-down-BFS
2) CPU single thread bottom-up-BFS
3) CPU single thread hybrid BFS
4) GPU multi thread top-down-BFS
5) GPU multi thread top-down-BFS for WCC detection
6) GPU multi thread hybrid BFS (switch at layer 3) for largest WCC detection

ii. Challenge

1) Our work so far can only have good performance in detecting the largest WCC in the graph. I tried a recursive structure to detect all the WCCs but it's extremely slow because it caused multiple kernel launch times. How to avoid multiple kernel launch time can be a challenge.
2) Mapping the BFS to multiple GPUs.

IX. Related works

i. Gunrock: GPU graph analytics[5]

Gunrock is a multi-algorithm project for undirected graph analytics based on GPU. The algorithms included in the project are: BFS, Single-source shortest path(SSSP), Betweenness Certrality(BC), Connected components labeling(CCL), PageRank and Other Node Ranking Algorithms, Triangle Counting(TC). As our project is mainly on the improvement of GPU BFS, here we focus on the properties of Gunrock's BFS and we found that:

1) The format of graph representation is CSR
2) BFS gives the vertex level and predecessor's ID
3) Use different strategies on vertices with different degree(dynamic workload mapping)
4) Atomic operations to avoid concurrent vertex discovery
5) Use heuristics to reduce redundant entries in the output frontier (hash tables)
6) Reduce synchronization overhead by reduce atomic operations
7) Hybird BFS converting with hard threshold

What Gunrock has not done:
1) Asynchronous BFS
2) SCC analytics

ii. iSpan: Parallel identification of strongly connected components with spanning trees[6]

Reference

[1] http://www.cs.wm.edu/~kemper/cs654/slides/nvidia.pdf
[2] Yuede Ji's slide
[3] S. Beamer, K. Asanovic. Direction-Optimizing Breadth-First Search. *SC 2012.*
[4] Wikipedia: https://en.wikipedia.org/wiki/Connected_component_(graph_theory)
[5] Y. Wang, A. Davidson, Y. Pan. Gunrock: A High-Performance Graph Processing Library on the GPU. *PPoPP 2016.*
[6] Y. Ji, H. Huang. iSpan: Parallel identification of strongly connected components with spanning trees. *SC 2018.*