Fast Diameter Computation of Large Sparse Graphs using GPUs

Giso H. Dal
Institute for Computing and Information Sciences,
Radboud University Nijmegen, The Netherlands
gdal@cs.ru.nl

Walter A. Kosters and Frank W. Takes
Leiden Institute of Advanced Computer Science,
Leiden University, The Netherlands
{kosters,ftakes}@liacs.nl

***Abstract*—In this paper we propose a highly parallel GPU-based bounding algorithm for computing the exact diameter of large real-world sparse graphs. The diameter is defined as the length of the longest shortest path between vertices in the graph, and serves as a relevant property of all types of graphs that are nowadays frequently studied. Examples include social networks, webgraphs and routing networks. We verify the performance of our parallel approach on a set of large graphs comprised of millions of vertices, and using a CUDA GPU observe an increase in performance of up to $21.1\times$ compared to a CPU algorithm using the same strategy. Based on these results, we provide a characterization of the types of graphs that are well-suited for traversal by means of our parallel diameter algorithm. We furthermore include a comparison of different GPU algorithms for single-source shortest path computations, which is not only a crucial step in computing the diameter, but also relevant in many other distance and neighborhood-based algorithms.**

***Keywords*-diameter; graph traversal; eccentricity; CUDA; sparse graphs**

I. Introduction

The field of graph traversal algorithms has been stimulated by the arrival of Graphics Processing Units (GPUs), as they deliver high performance at a relatively low cost [1]. The trend of ever increasing memory has made it possible to run algorithms on very large real-world graphs consisting of millions of vertices and edges. Although today's frameworks make the GPU easily accessible to programmers, developing high performance algorithms for traversing graphs is a challenging task due to strict guidelines of the parallel architecture.

The *diameter* of a graph is defined as the longest shortest path length, or alternatively as the highest eccentricity value over all vertices. In turn, the *eccentricity* of a vertex is the length of a longest shortest path connecting that vertex to any other vertex. The diameter is traditionally computed using the *All-Pairs Shortest Path* (APSP) algorithm, running in $\mathcal{O}(mn)$ time for graphs with n vertices and m edges. To reduce computation time at the cost of exactness, estimation methods that base the diameter on eccentricity values determined from a set of randomly selected vertices can be used [2]. We are specifically interested in exact diameter computation, allowing us to observe the actual paths that realize the diameter. Our parallel implementation is based on the observations and bounding strategies proposed in [3], [4], which drastically reduces the total number of eccentricity computations which have to be performed to find the exact value of the diameter.

Graph representation is important when optimizing algorithms on the GPU, and the main challenge lies in fully utilizing the parallel architecture [5] of the GPU. In previous work, the APSP problem for graphs (and thus the diameter problem) was solved in parallel by means of the Floyd-Warshall algorithm, based on graph implementations using adjacency matrices [6], [7], [8], [9]. Although it is common to represent graphs as an adjacency matrix, when dealing with sparse graphs with a large number of nodes and edges, the use of adjacency lists is necessary to reduce the usage of unnecessary space [9], [10]. Most methods break up the APSP into multiple *Single-Source Shortest Path* (SSSP) problems and then compute the length of these paths in parallel.

In case of the diameter, we are interested in the longest shortest path length, and a parallel version of Dijkstra's algorithm could be used [11], [12]. However, when dealing with unweighted graphs, Dijkstra's algorithm is essentially the same as *Breadth First Search* (BFS), and the SSSP algorithm for traversing the graph can be simplified and optimized further, which we will discuss in this paper. We will attempt to characterize structural properties of the graph that influence the extent to which a GPU algorithm for traversing the graph can efficiently be applied. In our experiments, we will apply our GPU algorithm to a variety of large real world graphs of different types, including social networks, citation networks, communication networks and web graphs, using the sequential CPU algorithm as a performance baseline.

The main contribution of this paper is exposing the parallel nature of our bounding strategy for finding the diameter, and its well-suitedness for GPU implementation. Instead of focusing on solving multiple SSSP problems in parallel like [7], [8], [9], [10], [13], we focus on reducing the number of SSSP problems to solve. An adjacency list is used to represent sparse graphs efficiently as previously done by [9], [10], with the addition of having padded each individual list to more easily comply with GPU memory access optimizations. We propose multiple algorithm variants that improve upon [1], [14] in providing best-of-both-worlds variants that are reproducible through provided pseudo code, and additionally we give a metric to choose a specific variant at different stages of the algorithm. Furthermore, the proposed algorithm follows the most strict definition of access optimization restrictions in terms of coalescing, meaning it will perform well on CUDA GPUs of any given compute capability.

The rest of this paper is organized as follows. We cover background knowledge on the subjects of graph diameter and the GPU programming model in Section II. Related work is presented in Section III. Section IV discusses the algorithm for computing the diameter on the GPU, as well as different single-source shortest path algorithms. Results of applying these algorithms as well as the diameter algorithm are discussed in Section V. We conclude with a summary and suggestions for future work in Section VI.

II. Background

This section offers some insight into the knowledge used in this paper. Section II-A discusses the graph diameter and Section II-B covers the important parts of the CUDA programming model.

A. Graph Diameter

Let $G(\mathcal{V}, \mathcal{E})$ be an undirected graph with vertices $\mathcal{V} = \{0, \ldots, n-1\}$ and edges $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$. We use n for the number of vertices and m to denote the number of edges. The *distance* between vertices $u, v \in \mathcal{V}$ is the length of the (or a) shortest path that connects them. The *eccentricity* of vertex $v \in \mathcal{V}$ is the greatest distance between v and any other vertex $w \in \mathcal{V} - \{v\}$. The *diameter* $D(G)$ of graph G equals the maximum eccentricity of any vertex $v \in \mathcal{V}$. In other words, the diameter is the length of a longest shortest path in the graph. We focus on the largest *weakly connected component* (WCC) of a graph, which is the largest subgraph of G that is connected when all edges are considered to be undirected.

In order to find the diameter of a graph, traditionally the shortest path must be calculated between all pairs of vertices. The *Single Source Shortest Path* (SSSP) problem finds such a path from source vertex $v \in \mathcal{V}$ to every vertex $w \in \mathcal{V} - \{v\}$. The *All Pairs Shortest Path* (APSP) problem finds a shortest path from any vertex $v \in \mathcal{V}$ to any vertex $w \in \mathcal{V} - \{v\}$.

B. Compute Unified Device Architecture

The *Compute Unified Device Architecture* (CUDA) is a parallel programming framework developed for graphics cards made by NVIDIA. The architecture we focus on is called *Fermi* [15], [5], [16]. Distinction is made between the different Fermi architectures by their *compute capability*. We consider compute capability 2.0 from now on, which is determined by the graphics card used during the experiments.

In order to write efficient programs with the CUDA programming model one must take notice of its key components. Section II-B1 will discuss the programming model and Section II-B2 will discuss the memory model.

1) Programming Model: CUDA functions or *kernels* are executed by threads organized in a hierarchical structure as illustrated by Figure 1. A *warp* consists of 32 threads that are executed in parallel. When branching the threads belonging to the same warp, they synchronize implicitly at the next instruction that must be executed by all threads of that warp. Execution is sequential until that time. A *block* consists of warps and is executed on a single *streaming multiprocessor* (SM). Multiple blocks may be assigned to the same SM. A *grid* consist of the blocks that execute the kernel.

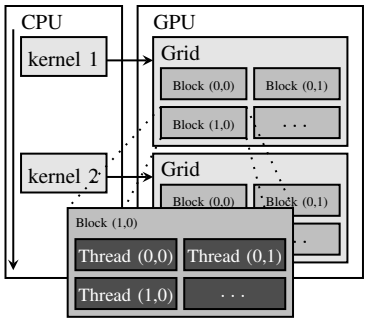

Figure 1 CUDA Programming Model

There is an indexing scheme in place in order to identify threads and blocks. Using this indexing scheme, branching is supported at the thread and block level.

2) Memory Model: The hierarchy of the programming model is also of influence in the memory model as illustrated in Figure 2. Threads within a block can communicate or share data through *shared memory*. Each block is allowed to access *global memory*, through which blocks can communicate and share data. The host can transfer data to and from global memory and *constant memory*.

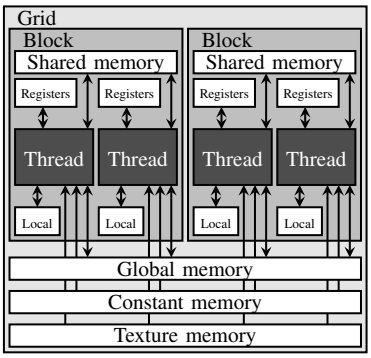

Figure 2 CUDA Memory Model

Table I shows the scope and lifetime of data located on different memories. A parallel mechanism requires memory accesses to be optimized by the programmer. This regards global and shared memory. Assume that global memory is divided into consecutive 128-bit segments. We say that global memory accesses are optimized or *coalesced*, when the following rules are obeyed by each thread:

1) Access 4-byte values,
2) Access consecutive addresses (4-byte spacing),
3) Threads in a warp must access the same segment.

TABLE I Scope and lifetime of different types of memory

Memory	Scope	Lifetime
Register	Thread	Kernel
Local	Thread	Kernel
Shared	Block	Kernel
Global	Grid	Program
Constant	Grid	Program

This is the most strict definition of coalescing, which applies to compute capabilities < 1.2. Later compute capability coalescing requirements are relaxed and therefore subsume the rules given above. This also holds for the even further relaxed access requirements to shared memory. This means that regardless of memory type and compute capability, these rules lead to optimal bandwidth. This was important to mention because our algorithm is complacent toward the stricter phrasing of coalescing with regard to the bounding strategy, as will become clear later on, and is therefore suitable to run on GPUs with different compute capabilities. When access patterns make it impossible to obey the aforementioned rules, shared memory can be used as an intermediate step in instances, as its rules are less constricting. We speak of a *bank conflict* in the context of shared memory when one or both of the following rules are broken assuming the same segmentation as for global memory:

1) Access 4-byte values,
2) Threads in a warp must access different addresses in the same segment, *or*
3) Access the same offset within different segments.

Algorithm optimization is greatly determined by how well the previously discussed guidelines and restrictions are followed.

III. Related Work

Shortest path problems are fundamental in the field of combinatorial optimization and have many applications [17]. There are many sequential algorithms solving the SSSP problem. Dijkstra's algorithm solves this for positive weighted graphs in time complexity $\mathcal{O}(n \log n + m)$ [18], and for positive weighted planar graphs in $\mathcal{O}(n)$ time [19], which in both cases is optimal [20]. Parallel implementations of Dijkstra's algorithm exist but are oriented towards CPUs and work-efficient parallel processes [11], [12]. A different approach is required for the GPU architecture as it uses a much greater number of parallel processes or threads, as demonstrated in [10]. As the diameter of a graph can easily be derived from the result of an APSP algorithm, parallel algorithms based on the Floyd-Warshall algorithm could be used to determine the diameter [6], [7], [8], [9]. Most of these algorithms use adjacency matrices, which prohibit the study of large graphs with millions of nodes.

Parallel BFS implementations have been proposed by [1], [10], [21], [22], [23], and the hierarchy of the programming model has been further exploited by [23], and [14]. There are two reoccurring components they all have in common. Each BFS implementation uses level synchronization and the typical FIFO approach is replaced by a so-called frontier keeping track of vertices already visited. Each vertex is assigned a thread that uses the frontier to store the distance to the source vertex at each level. Both optimizations regard hardware correspondence.

IV. Methodology

Using CUDA, we developed an algorithm for the GPU for calculating the graph diameter. Graph representation is discussed in Section IV-B and is key in high performance GPU computing. In Section IV-C we discuss a parallel eccentricity algorithm separately, as it is the dominating component of the diameter computation, runtime wise. To reduce the number of eccentricity computations, we start with the proposal of a highly parallel bounding strategy in Section IV-A and elaborate on the components it consists of. We finish by discussing algorithm variants in Section IV-D.

A. Diameter Bounding Strategy

The graph diameter problem can be broken down into many SSSP problems, pruning and bounding strategies help to reduce the search space of the problem, which becomes very important when working with millions of vertices and edges. In spite of the sequential nature of the diameter procedure proposed in Algorithm 1, the individual functions that it consists of are highly parallel.

Although the diameter procedure runs on the CPU, all computations are done on the GPU. The CPU is only used to launch kernels and to monitor the stopping condition. Each function is contained in its own kernel, as crowding kernels with multiple functions causes individual threads to use more shared resources than necessary. In turn, this reduces the total number of threads, referred to as *occupancy*, that can execute the kernel and thus impacts performance negatively.

Algorithm 1 Diameter computation using CUDA

```
procedure DIAMETER()
    B_l[0,...,n-1] ← 0
    B_u[0,...,n-1] ← n
    F[0,...,n-1] ← 0
    while max(B_l[0,...,n-1]) ≠ max(B_u[0,...,n-1]) do
        vertex ← SELECT_CANDIDATE(C, B_l, B_u)
        eccentricity ← ECCENTRICITY(vertex, F)
        UPDATE_BOUNDS_KERNEL(eccentricity, B_l, B_u, F)
        UPDATE_CANDIDATES_KERNEL(C, B_l, B_u)
    return max(B_l[0,...,n-1])
```

Frontier array F stores shortest distances to the source vertex but we first select a vertex from array C holding *candidates*. We compute the eccentricity for that candidate (see Section IV-C), and with the aid of arrays B_u and B_l we do the bookkeeping of the respective eccentricity upper and lower bound for each remaining candidate. These bounds are determined using observations made by [3], stating that every vertex is reachable from vertex v in less steps than its eccentricity value $\varepsilon(v)$, i.e., the length of its longest shortest path. Let $d(v, u)$ be the distance between vertex v and u, then it takes exactly $d(v, u)$ steps to reach u from v and at most an additional $\varepsilon(v)$ steps to reach every other vertex from u. Furthermore, it is impossible for an upper bound to be lower than the global lower bound, and vice versa in undirected graphs. The number of candidates can be reduced using these upper and lower bounds, and we come back to this later.

Candidate selection is of great influence on our bounding strategy. The eccentricity of a selected vertex determines the extent to which the bounds can reduce the number of computations that come afterwords. Different strategies have been investigated in [3]. We choose a candidate by alternately

selecting a vertex with the global highest bound and selecting a vertex with the global smallest bound. This seems to contribute the most with regards to pruning candidates.

Algorithm 2 shows the functions for the bookkeeping of candidate vertices and bounds. We update the upper and lower bounds with the eccentricity value determined from the previous selected vertex at each iteration of the diameter procedure. As bounds grow toward each other, the eccentricity of a vertex is determined when the lower bound is equal to the upper bound, without having to explicitly compute the eccentricity for that vertex. Candidates are pruned when they can no longer contribute to finding the diameter, cf. the bounds discussed in [3].

Algorithm 2 Updating of bounds and candidate vertices

```
▷ T is the total number of threads
▷ TID is a unique global thread id
function UPDATE_BOUNDS_KERNEL(eccentricity, B_l, B_u, F)
    for i ← TID to n−1 : i ← i + T do
        B_l[i] ← max(B_l[i], max(F[i], eccentricity − F[i]))
        B_u[i] ← min(B_u[i], eccentricity + F[i])

function UPDATE_CANDIDATES_KERNEL(C, B_l, B_u)
    upperbound ← max(B_u[0, ..., n−1])
    lowerbound ← max(B_l[0, ..., n−1])
    for i ← TID to n−1 : i ← i + T do
        if C[i] = 1 and (B_u[i] = B_l[i]    or
          (B_u[i] ≤ lowerbound and B_l[i] · 2 ≥ upperbound) ) then
            C[i] ← 0
```

A precondition to the functions shown in Algorithm 2 is that frontier F must contain the distance of the path from the source vertex to every other vertex. This influences the eccentricity function in that it cannot use a queue structure. A queue is organized in such a way that the vertices with the longest shortest paths are located at the beginning. Reorganizing this queue in a coalesced fashion is not possible. Both functions in Algorithm 2 have fully coalesced access patterns maximizing bandwidth. This reveals their well-suitedness for GPU optimization. Section V will show that in practice these functions only account for a small portion of the total runtime, yet contribute a lot. We leave the eccentricity function to Section IV-C as this function is most dominantly present in the total runtime, and it is more difficult to optimize on the GPU.

B. Graph Representation

Although it is easy to develop an implementation with optimized access patterns as discussed in Section II-B with an adjacency matrix representation, it takes $\mathcal{O}(n^2)$ space. We focus on sparse graphs, thus much space would be wasted. An *adjacency list* does not waste space, yet poses many problems when trying to optimize memory access patterns. Figure 3 shows how the index of each vertex is used to point to the starting position of its own adjacency list. The representation consists of three arrays. Array V_b contains the base addresses or starting positions of the adjacency lists, V_s contains the size of each adjacency list and array E consists of the concatenated adjacency lists.

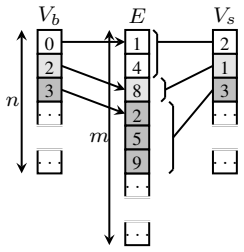

Figure 3 Concatenated adjacency list example

Depending on the sparseness of the graph, an intermediate representation can be considered where E is a matrix holding a list of adjacency lists depicted by Figure 4. In this case V_b would become obsolete and each adjacency list can be accessed using the vertex identifier as index. Matrix E would be of size $n \times \ell$, where ℓ is the size of the longest adjacency list. For memory access optimization reasons discussed in Section II-B we must increase ℓ to the next multiple of 32. As the compactness depends on the length of the longest list, we further reduced it by using the representation where each *individual* adjacency list is padded to a multiple of 32. This can be achieved with a simple $\mathcal{O}(n)$ time algorithm. We have identified this to be the most compact representation that can still comply with access optimization rules.

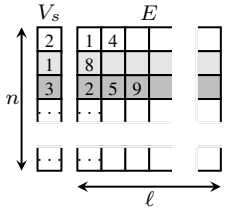

Figure 4 List of adjacency lists example

C. Eccentricity

The eccentricity of vertex $v \in \mathcal{V}$ can be computed by the use of one *Breadth First Search* (BFS), which takes $\mathcal{O}(m)$ time. When talking about the BFS approach we have the notion of *levels*. A level contains every vertex of which the distance to the source vertex is the same. We refer to vertices as being *active* when the edges of these vertices need to be traversed at the current level in order to create the next. Distances are stored in a *frontier* as the algorithm progresses.

We have based the eccentricity function shown by Algorithm 3 on the BFS proposed by [10]. It should be used with the graph representation described by Figure 3. We use the frontier array F of size n to keep track of active vertices at each level, and thus need level synchronization.

The **for** loop in Algorithm 3 guarantees that each block is responsible for a different section of frontier F. This is shown by Figure 5 for the instance that the kernel is launched with

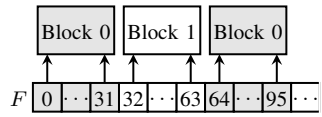

Figure 5 Frontier access pattern

Algorithm 3 Eccentricity function

```
▷ T is the total number of threads
▷ TID is a unique global thread id
function ECCENTRICITY_KERNEL(F, &level)
    for i ← TID to n−1 : i ← i + T do
        if F[i] = level then
            for j ← 0 to V_s[i] do
                if F[E[V_b[i] + j]] = −1 then
                    F[E[V_b[i] + j]] ← level + 1
    if F changed then
        level ← level + 1

function ECCENTRICITY(v)
    level ← 0
    llevel ← 0
    F[0, ..., n−1] ← −1
    F[v] ← level
    while llevel = level do
        ECCENTRICITY_KERNEL(F, level)
        llevel ← llevel + 1
    return level
```

two blocks consisting of 32 threads. This access pattern is also used in the other kernels and ensures coalesced access if the number of threads that make up a block at kernel launch is a multiple of 32.

D. Variants

We have focused on the parallelization of the eccentricity function. Below, we discuss six different variations of Algorithm 3 computing the eccentricity value. In the next section, we compare the different algorithm variants.

Standard: The *Standard* eccentricity algorithm is shown in Algorithm 3. A thread is assigned to each vertex. When a thread makes a change in the frontier at the current level, we know that there is a next level and therefore need to continue another iteration of the **while** loop.

Pointers: The *Pointers* algorithm is an optimization of Standard, realized by forcing some variables residing in caches into registers. Also, array V_b holding the starting positions of the adjacency lists now holds actual global memory addresses pointing to these starting positions instead of integer index values. This reduces overall memory address translation cost.

Busy-Wait: The *Busy-Wait* algorithm attempts to reduce memory accesses by idling blocks at certain levels where they have no work to perform. The frontier is read entirely at each level, even though a small portion of the vertices need to be traversed. A queue structure would not have this problem, yet GPUs need this ambiguity in order to fully utilize the parallel architecture. Each block is responsible for a disjoint section of the frontier. Equation 1 is used to determine to whom the section belongs that is written to:

$$block \quad = \quad \lfloor v/TB \rfloor \bmod B. \tag{1}$$

Here, v is the vertex that is activated, TB is the number of threads per block and B is the number of blocks that execute the kernel. Figure 6 shows how the result of the equation is stored in array A which is used to determine at the start (once, so no polling) of each level if a block should traverse the frontier or not.

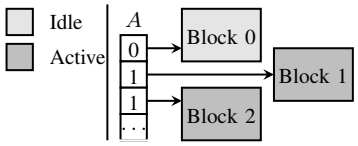

Figure 6 Busy-Wait strategy

Textures: The *Textures* algorithm is a simple variation with a fundamental difference. Array E holding the edges no longer resides in global memory but in texture memory. Texture memory is optimized for spacial locality, meaning that the assumption is made that neighboring addresses are likely to be accessed. This is the case when using adjacency lists.

Warp: The *Warp* algorithm increases parallelism when dealing with long edge lists by processing them with a *warp* instead of by just one thread. A warp is a collection of 32 threads which are executed at the same time [5]. Previously discussed variations of the algorithm are thread-centric, whereas this algorithm is warp-centric. Algorithm 4 shows a code snippet that replaces the outermost **for** loop of the eccentricity function in Algorithm 3. We first copy the appropriate section from the frontier to an array in shared memory. This way the threads within a warp can process the frontier in an optimized way at the same time. Once an active vertex is found, the warp traverses the edge list (the innermost **for** loop) with an optimized access pattern when using the algorithm in combination with the data representation presented by Figure 4.

Algorithm 4 Warp-Centric eccentricity function

```
▷ WID is the warp id within a block
▷ TID is the thread id within a block
▷ BID is the block id
▷ BS is the block size (#threads)
▷ GS is the grid size (#blocks)
...
for i ← BID · BS to n−1 : i ← i + BS · GS do
    cache[TID] ← F[i + TID]
    for w ← WID · 32 to (WID + 1) · 32 do
        if cache[w] = level then
            for t ← TID mod 32 to V_s[i + w] do
                if F[E[Vb[i + w] + t]] = −1 then
                    F[E[Vb[i + w] + t]] ← level + 1
...
```

Hybrid: The *Hybrid* algorithm attempts to combine the best attributes of a thread-centric algorithm (Standard) and a warp-centric (Warp) algorithm. At the beginning of each level during the eccentricity computation a choice is made to use either the thread or the warp-centric function. The result of this choice depends on the number of active vertices and the length of the adjacency lists.

TABLE II Real world datasets and their properties

Dataset	N	n_{wcc}	m_{wcc}	$\triangle$	$\overline{C}$	D	D_{90}	I	T	S
Amazon0601	5	403364	3387224	3986507	1.1769	21	7.6	28	143.68	13.34
as-skitter	7	1694616	11094209	28769868	2.5932	25	5.9	6	138.85	8.75
cit-Patents	3	3764117	16511741	7515023	0.4551	22	9.4	111	4515.64	21.11
com-amazon	5	334863	925872	667129	0.7205	44	15	7	24.71	11.95
com-dblp	8	317080	1049866	2224385	2.1187	22	8.1	8	21.78	13.56
com-LiveJournal	1	3997962	34681189	177820130	5.1273	18	6.4	9	519.04	16.12
com-youtube	1	1134890	2987624	3056386	1.0230	21	6.5	2	17.26	13.66
Email-EuAll	2	224832	395270	267313	0.6763	13	4.5	3	8.95	5.85
roadNet-CA	6	1957027	5520776	120676	0.0219	850	500	181	12381.40	2.45
roadNet-PA	6	1087562	3083028	67150	0.0218	782	539	61	2508.87	2.22
roadNet-TX	6	1351137	3758402	82869	0.0220	1049	670	83	5194.15	1.84
soc-LiveJournal1	1	4843953	68983820	285730264	4.1420	18	6.5	6	437.18	16.01
soc-Pokec	1	1632803	30622564	32557458	1.0632	11	5.3	3	110.88	14.01
web-BerkStan	4	654782	7499425	64690980	8.6261	669	10	5	153.72	2.06
web-Google	4	855802	5066842	13391903	2.6430	22	8.1	5	47.38	12.25
web-NotreDame	4	325729	1497134	8910005	5.9514	46	9.3	3	16.35	4.05
web-Stanford	4	255265	2234572	11329473	5.0701	740	9.8	7	111.23	1.88
WikiTalk	2	2388953	5018445	9203519	1.8339	9	4	7	97.02	11.07

V. Experimental Results

In this section we report on two types of experiments. Using real world graphs, we make a comparison between multiple GPU algorithms and we assess their performance compared to a typical sequential CPU implementation. We preferred a GPU version over a parallel CPU version, because the access patterns of the bounding strategy are optimal for a GPU and thus seemed a better fit. We used The Little Green Machine (LGM) supercomputer [24] for experimentation, which uses *Intel Xeon E5620* CPUs and *NVIDIA GTX 480* GPUs. Each GPU implementation runs at full occupancy (requiring 21 registers to prevent memory spilling). With regard to the GTX 480 this means that each of the available 15 streaming multiprocessors has 1536 threads executing a kernel in parallel. In [25] a performance improvement was reported under lower occupancy, yet this proved not to be true in our case.

The sequential CPU implementation obtained from [4] uses a typical FIFO queue when computing eccentricity values with the BFS graph traversal method. It utilizes the same bounding strategy as proposed in Section IV. We elaborate on the datasets used during experimentation in Section V-A. Section V-B focuses on the diameter computation and its components, and Section V-C further investigates the two algorithm types that the Hybrid algorithm consists of.

A. Datasets

We have tested with eight different graph or network types, namely social networks (1), communication networks (2), citation networks (3), web graphs (4), product co-purchasing networks (5), road networks (6), autonomous system graphs (7) and collaboration networks (8). The datasets are obtained from [26] and are described in Table II. Large real world sparse graphs have been selected with 400,000 to 70,000,000 edges and are either undirected or converted to undirected graphs. Larger graphs would not fit into our 1.5GB of memory which translates to roughly 400 million edges, without accounting for space required for the algorithm. Table II shows the network type N as enumerated above, the number of vertices n_{wcc} and the number of edges m_{wcc} in the WCC, the number of triangles $\triangle$, the average clustering coefficient $\overline{C}$, the diameter D and the 90-percentile effective diameter D_{90}. It gives the total runtime T of the diameter computation in milliseconds, the speedup factor S in comparison to the CPU implementation, and the number of iterations I of Algorithm 1 it takes to compute the diameter, i.e., the number of eccentricity computations that are required to determine the exact diameter.

TABLE III Runtime and speedup of diameter components

Component	Avg runtime %	Avg speedup
Initialization	0.09	39.96
Candidate selection	1.17	11.70
Eccentricity	97.13	5.55
Updating bounds	1.29	46.59
Updating candidates	0.31	46.38

B. Diameter Computation Results

Each component of the diameter computation has been tested separately of which the results can be seen in Table III. The table holds averages over all tested datasets of the Pointers algorithm. It shows that the Eccentricity component dominates the total runtime and it also shows that the four components contributing to the bounding strategy have very high speedups and claim less than 3% of total runtime.

One relation between the graph properties that appeared to influence the performance is shown in Figure 7. The relation between D, D_{90} and the speedup factor can be explained

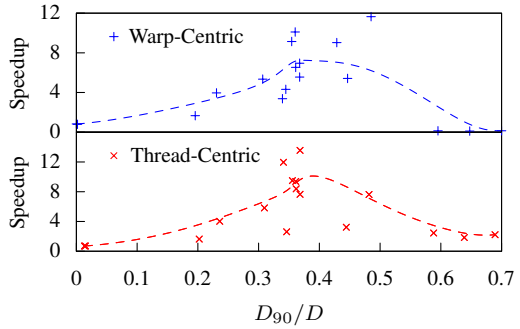

Figure 7 The relation between D and D_{90}

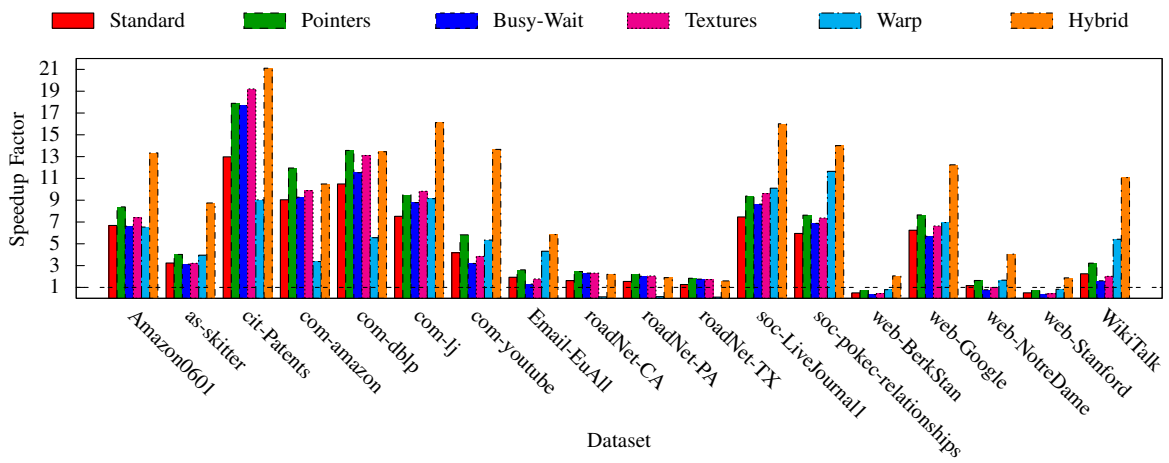

Figure 8 Real world graph results

by the imbalance in the division of labor among threads and levels. Large difference in list lengths, a large diameter and level synchronization result in loss of parallelism. Noticeable of the two worst performing datasets is that they have an extremely low D_{90}/D value in comparison to the other datasets (see Table II). It also seems that $D_{90}/D \approx 0.4$ performs best with both thread- and warp-centric algorithm type. The downwards slope in Figure 7 is due to the final three data points, which coincidentally belong to the same network type: Road networks. These networks appear to be characterizable by this relation. A final conclusion can be taken from Figure 8 on a higher level. Social networks, collaboration networks and citation networks generate datasets that all perform well. These network types appear to be well-suited for graph traversal on the GPU.

Figure 8 shows the speedup factors of our algorithms for every tested dataset. From an algorithmic perspective, the Pointers algorithm utilizes a crude optimization technique by forcing faster memory instructions. In comparison to the Standard algorithm this always seems to work. Although the Busy-Wait algorithm has less memory transactions with regard to the frontier, it is often slower than the standard version. This can be explained by considering the time it takes to run the additional control code. The performance of the Textures algorithm is the least predictable. Texture memory has a different cache than global memory and is optimized for spacial locality. This type of caching strategy can apparently be utilized in some instances. The Hybrid algorithm combines the best attributes of two algorithm types, and was inspired by the Warp algorithm performing very well during a small portion of the algorithm. This is why the combination nearly always comes out on top.

C. Adaptive Eccentricity Computation

Choosing which algorithm type to use at each level of the hybrid algorithm depends on the length of adjacency lists and the number of active vertices at that level. In an effort to find out when to use which type we created an adjacency list representation where each list is of equal length, and traversed it. Figure 9 shows the speedup factor of the warp-centric type

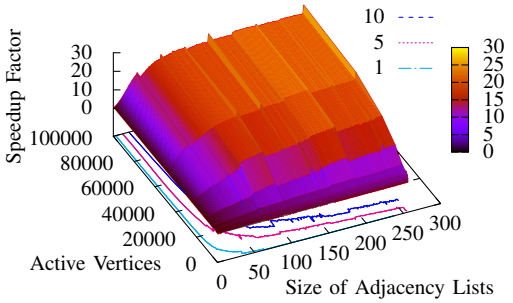

Figure 9 Relation between list size and the number of active vertices

compared to thread-centric. We have drawn the border of the $1\times$, $5\times$ and $10\times$ speedup on the lower two axes, which clearly reveals two relations: (1) long adjacency lists favor the warp-centric type (the reason is trivial as a warp should be much faster than a thread in traversing longer lists), and (2) fewer active vertices favor the thread-centric type, which does not have the overhead of first having to transfer the frontier array into shared memory and reading it multiple times from there. The speedup of the warp-centric type solely comes from long list traversal, thus its performance increases when there are more active vertices, i.e., more lists to traverse, which reduces relative overhead.

The result of Figure 9 was created by an unrealistic situation where adjacency lists are of equal length. We have used the Amazon0601 dataset to show multiple ways of indicating

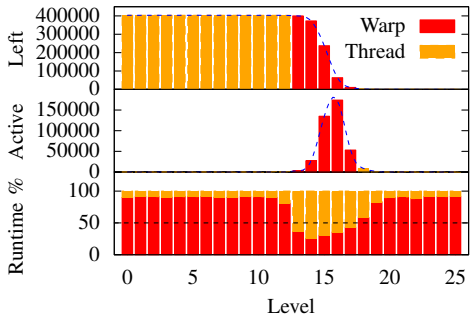

Figure 10 Measures to determine best algorithm type

when to use which type of algorithm in Figure 10 on a per level basis, by how many untraversed vertices are *left* and the number of *active* vertices. The *runtime percentage* is also compared of both algorithm types. An equal or smaller percentage means an equal or shorter runtime, respectively. We base our choice on the downward angle of how many vertices are left, which can be understood by comparing a similar visualization of the roadNet-PA dataset (where the warp-centric algorithm never performs better), we also never encounter a steep enough downward angle comparable to the area where the warp-centric algorithm performs better in Figure 10.

VI. Conclusion and Future Work

We have introduced and analyzed a highly parallel GPU-based algorithm to compute the exact diameter of large graphs with millions of nodes and edges efficiently. The bounding strategy has reduced the number of required BFS computations to compute the diameter from n to a constant number, usually less than 10. We have seen that the bounding strategy component of the algorithm is extremely efficient and well-suited for GPU implementation, and in combination with BFS reaches overall speedups of up to $21.1\times$ compared to the CPU implementation. The large graphs that we investigated have varying characteristics, and we have attempted to characterize which graphs are well-suited for traversal on GPUs, showing the relevance of the graph's D_{90}/D ratio for the speedup.

We have furthermore investigated different approaches for single BFS searches, in terms of data representation and computation, to compute the eccentricity of a vertex, which is the dominant component of the diameter algorithm and many other distance-based graph properties. It turns out that determining the best performing algorithm to compute the eccentricity highly depends on the stage of the BFS computation. We therefore proposed a hybrid technique comprised of the thread-centric and the warp-centric algorithm.

In future research we want to investigate a Floyd-Warshall hybrid implementation where multiple eccentricity computations are performed at the same time, while still preserving the bounding strategy. We have seen that the dominating component of the diameter computation is the eccentricity function, and dividing the work over multiple GPUs will definitely increase performance. We furthermore want to extend our GPU implementation such that it includes the computation of the radius, center and periphery of weighted and unweighted graphs.

Acknowledgement

The first author is partially supported by the ITEA2 MoSHCA project (ITEA 2 ip11027). The third author is supported by the NWO COMPASS project (grant #612.065.92).

References

[1] D. Merrill, M. Garland, and A. Grimshaw, "Scalable GPU graph traversal," in *Proceedings of the 17th ACM Symposium on Principles and Practice of Parallel Programming*, 2012, pp. 117–128.

[2] A. Sala, L. Cao, C. Wilson, R. Zablit, H. Zheng, and B. Y. Zhao, "Measurement-calibrated graph models for social network experiments," in *Proceedings of the 19th International Conference on World Wide Web*. ACM, 2010, pp. 861–870.

[3] F. W. Takes and W. A. Kosters, "Determining the diameter of small world networks," in *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*, 2011, pp. 1191–1196.

[4] F. W. Takes and W. A. Kosters, "Computing the eccentricity distribution of large graphs," *Algorithms*, vol. 6, no. 1, pp. 100–118, 2013.

[5] "CUDA C Programming Guide," accessed October 2013, *http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html*.

[6] A. Bleiweiss, "GPU accelerated pathfinding," in *Proceedings of the 23rd ACM Symposium on Graphics Hardware*, 2008, pp. 65–74.

[7] A. Buluç, J. R. Gilbert, and C. Budak, "Solving path problems on the GPU," *Parallel Computing*, vol. 36, no. 5, pp. 241–253, 2010.

[8] G. J. Katz and J. T. Kider Jr, "All-pairs shortest-paths for large graphs on the GPU," in *Proceedings of the 23rd ACM Symposium on Graphics Hardware*, 2008, pp. 47–55.

[9] T. Okuyama, F. Ino, and K. Hagihara, "A task parallel algorithm for computing the costs of all-pairs shortest paths on the CUDA-compatible GPU," in *Proceeding of the International Symposium on Parallel and Distributed Processing with Applications*, 2008, pp. 284–291.

[10] P. Harish and P. Narayanan, "Accelerating large graph algorithms on the GPU using CUDA," in *Proceedings of the 14th International Conference on High Performance Computing*. LNCS 4873, 2007, pp. 197–208.

[11] M. Erwig and F. Hagen, "The graph Voronoi diagram with applications," *Networks*, vol. 36, no. 3, pp. 156–163, 2000.

[12] J. L. Träff and C. D. Zaroliagis, "A simple parallel algorithm for the single-source shortest path problem on planar digraphs," in *Proceedings of the Parallel Algorithms for Irregularly Structured Problems*. LNCS 1117, 1996, pp. 183–194.

[13] P. Micikevicius, "General parallel computation on commodity graphics hardware: Case study with the all-pairs shortest paths problem," in *Parallel and Distributed Processing Techniques and Applications*, vol. 4, 2004, pp. 1359–1365.

[14] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun, "Accelerating CUDA graph algorithms at maximum warp," in *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, 2011, pp. 267–276.

[15] "CUDA C Best Practices Guide," accessed October 2013, *http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html*.

[16] "NVIDIA's Fermi: The First Complete GPU Computing Architecture," accessed October 2013, *http://www.nvidia.com/object/fermi-architecture.html*.

[17] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, *Network flows: Theory, algorithms, and applications*. Prentice Hall, 1993.

[18] J. R. Driscoll, H. N. Gabow, R. Shrairman, and R. E. Tarjan, "Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computation," *Communications of the ACM*, vol. 31, no. 11, pp. 1343–1354, 1988.

[19] M. L. Fredman and R. E. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms," *Journal of the ACM*, vol. 34, no. 3, pp. 596–615, 1987.

[20] M. Barbehenn, "A note on the complexity of Dijkstra's algorithm for graphs with weighted vertices," *IEEE Transactions on Computers*, vol. 47, no. 2, p. 263, 1998.

[21] S. Hong, T. Oguntebi, and K. Olukotun, "Efficient parallel graph exploration on multi-core CPU and GPU," in *Proceeding of the International Conference on Parallel Architectures and Compilation Techniques*, 2011, pp. 78–88.

[22] C. E. Leiserson and T. B. Schardl, "A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers)," in *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*, 2010, pp. 303–314.

[23] L. Luo, M. Wong, and W. Hwu, "An effective GPU implementation of breadth-first search," in *Proceedings of the 47th Design Automation Conference*, 2010, pp. 52–55.

[24] "The Little Green Machine (LGM) Supercomputer," accessed October 2013, *http://www.littlegreenmachine.org/*.

[25] V. Volkov, "Better performance at lower occupancy," in *Proceedings of the GPU Technology Conference*, 2010.

[26] "Stanford Large Network Dataset Collection," accessed October 2013, *http://snap.stanford.edu/data*.