POLITECNICO DI TORINO

Master of Science in Computer Engineering

Master Degree Thesis

Data-parallel implementation of the GRAIL index on many-core architectures

Supervisors
Prof. Stefano Quer

Candidates
Antonio Caiazza

Academic Year 2018 – 2019

Summary

Reachability is among the most common problems to address when working on graphs since it is the base for many other algorithms, and scalable solutions have become of paramount importance with the advent of distributed systems that process large amounts of data. Specifically many applications explore graphs with millions of nodes and vertices, which explains why the development of fast and scalable algorithms entails complex challenges.

Modern GPUs are highly parallel systems based on many-core architectures and have gained popularity in parallelizing algorithms that run on large data sets. In this context the NVIDIA CUDA platform has been used to provide a concrete implementation of the developed algorithms.

The main focus of this work is to analyze the GRAIL algorithm, which creates a scalable index for answering reachability queries on large graphs, in order to design an implementation that exhibits a greater amount of data parallelism. The GRAIL index relies heavily on depth-first search, which is among the hardest algorithms to develop on parallel systems, so an alternative approach based on breadth-first search will be explored and significant effort will be devoted towards analyzing the difficulties encountered and the solutions chosen to overcome them.

Finally this work will provide a concrete implementation on CUDA of the proposed algorithms, a comparison between the indexing and search times for the CPU and the GPU based versions and insight on how to conduct further research in future.

Acknowledgements

Il primo ringraziamento va ai miei coinquilini che, tra una caffettiera fusa, una partita a Risiko ed una chiacchierata improvvisata al momento giusto, hanno reso casa la mia prima esperienza fuori casa.

Approfitto per ringraziare Gabriele, senza il quale avrei corso il rischio di seguire qualche lezione a Torino, mentre per amici come Enrico e Lucilla, Federico, Marika, Anna, Matteo, Miguel Angel e Domenico le parole non bastano: sappiate che se siete in questa lista apprezzo la vostra amicizia oltre ogni dire.

Un particolare ringraziamento va ad Ale, con cui ho condiviso un lungo percorso fatto di discussioni sulla consistenza dei biscotti e, nel tempo libero, di progetti ed esami.

A Mariana che, dopo infiniti calendari di esami fatti e mai rispettati, non vede l'ora di tornare a svegliarsi dopo di me.

Ai miei zii, Salo ed Ivette, che mi hanno supportato a prescindere dalla distanza e che in meno di 14 giorni all'anno sono sempre riusciti a trasmettermi tutto il loro affetto.

Infine, il ringraziamento più grande va a mia madre, a cui in questi anni è bastata la voce per riportarmi a casa. A mio padre, che sin da piccolo mi ha insegnato a lanciare il cuore oltre l'ostacolo. A mio fratello, che non dovrà più sentire *"Samy, callate!"* mentre studio e che mi riporta costantemente a ciò che conta. Grazie per la costante fiducia in me, non avrei neanche intrapreso questo percorso senza di voi.

E a te, cane.

Contents

List of Tables

List of Figures

Part I

GRAIL and CUDA environment

Chapter 1

Introduction

1.1 General context

Graphs have always had significant relevance in computer science research given the versatility with which they represent data across distinct application domains, which explains their value to many scientific areas. It is not uncommon to see that, in several fields, graphs are used to model entities and relationships that would not be usually thought of as a set of vertices connected by edges. For instance, biological networks rely on graphs to represent genes as vertices and their interactions as edges, whereas most enterprises use software that represent domain specific products as nodes and commercial transactions as edges. Furthermore knowledge representation systems, such as ontologies, are able to derive new information by using vertices to symbolize concepts and edges to describe relationships among these concepts.

However unrelated these fields may be, it is evident that graphs are of little use if it is not possible to answer reachability queries on the data that they represent: this problem is called reachability, which is defined in graph theory as the ability of a vertex to reach its peers within the same graph. There exist a variety of methods for addressing this problem, but most reachability algorithms include a pre-processing phase that creates data structures, usually referred to as indices, that speed up the query resolution time at the expense of requiring more space to store these structures. Existing algorithms differentiate themselves according to several factors, but in recent times their biggest limitation lies in their capacity to process large graphs efficiently, and so scalability has become one of the most significant metrics for analyzing reachability methods.

Interestingly, recent times have seen an emergent trend to develop applications that scale gracefully by exploiting the increasing number of processors provided by parallel systems, and the advent of NVIDIA's general purpose GPU architecture, commonly known as CUDA, has transformed mainstream computers into highly-parallel systems with significant computational power and considerable memory bandwidth. Since the release of CUDA's first version in 2006 there has been a continued effort to redesign algorithms in order for them to exhibit a large degree of data parallelism and benefit from the ever-increasing number of processors and bandwidth on GPUs, which are especially designed to concurrently and efficiently execute the same program over different data.

Specifically, GPUs are well suited to solve graph related problems efficiently since they have many processing units that can concurrently explore and modify a large set of nodes, and so there has been a recent boost in producing graph-processing software in CUDA. This effort has come both from NVIDIA, with its NVIDIA Graph Analytic library, or *nvGRAPH*, which includes a set of tools and parallel algorithms to perform high performance analysis on graphs, and from independent projects such as the recent *GUNROCK* library, which delivers similar performance while providing an abstract and data-centered API.

In this context, we propose an analysis of an indexing method called GRAIL, which stands for **G**raph **R**eachability Indexing via R**A**ndomized **I**nterval **L**abeling, and was proposed in article [1]. This algorithm was designed with particular emphasis on scalability, has linear index creation time and space and is able to execute reachability queries on graphs of millions of nodes and edges in times that range from constant to linear in the graph size. The core of this work is to study how to redesign this method such that it exhibits a higher degree of data parallelism and can benefit from execution on Single Instruction on Multiple Data, from now on SIMD systems, and we provide an implementation of the proposed solution developed on CUDA.

GRAIL is based upon a single depth-first traversal, which is hard to parallelize on CUDA, or on most parallel systems, and a significant part of this work regards how to exploit specific properties of Direct Acyclic Graphs (DAGs) in order to replace one iteration of a DFS with several breadth-first explorations, which by definition expose a higher degree of data parallelism.

Moreover, given that the final algorithm will be performing more traversals than its CPU based counterpart, it will be fundamental to rely on a work-efficient BFS on CUDA, and we will devote a considerable effort on adapting a state of the art BFS algorithm to suit our needs.

Chapter 2

GRAIL

2.1 Preliminary observations

In order to provide a concise representation of the problem at hand we introduce a series of definitions that will be used frequently. Let G = (V, E) be a directed graph with V being the set of vertices and E the set of directed edges. We shall refer to the cardinality of V and E, respectively, with n and m. We will also write u $\rightarrow^?$ v to indicate the reachability query of whether there is a path that goes from node u to node v, and will write u $\rightarrow$ v if such a path exists or u !$\rightarrow$ v if it does not.

Furthermore, it is well known that the problem of answering reachability queries on directed graphs can be reduced to reachability on Directed Acyclic Graphs (DAGs), since for every directed graph it is possible to construct a condensation graph in which every node corresponds to a strongly connected component of the original graph. In practice, the reachability query u $\rightarrow$? v on a directed graph can be answered by checking whether in the condensation graph the strong components of u and v, namely u' and v', coincide or whether u' $\rightarrow$ v'. Henceforth it will be assumed that all the mentioned graphs are directed and acyclic, unless explicitly stated.

2.2 Indexing approaches

As already mentioned, indices are frequently used to speed up graph exploration, there are several algorithms for creating them and the most significant metrics that set these methods apart are the index construction time, index size and the resulting query time. In fact, the index design space has at the extremes two opposing approaches: on one side reachability queries could be answered in constant time through the Full Transitive Closure, which consists in computing and storing a list of all reachable nodes from every vertex, and it is unfeasible for very large graphs since it requires quadratic space index. At the other extreme one could avoid relying on an index and could answer reachability queries by depth-first or bread-first traversals. This approach does not require neither construction time nor indexing space but implies $O(n+m)$ time for every query, which is equally unacceptable for large graphs. In practice, most reachability algorithms lie somewhere in between the index design space: they involve a pre processing phase which creates the index and use it during the query resolution to achieve efficient times for large graphs, so the trade off between indexing space and querying time constitutes the base for analyzing all indexing methods.

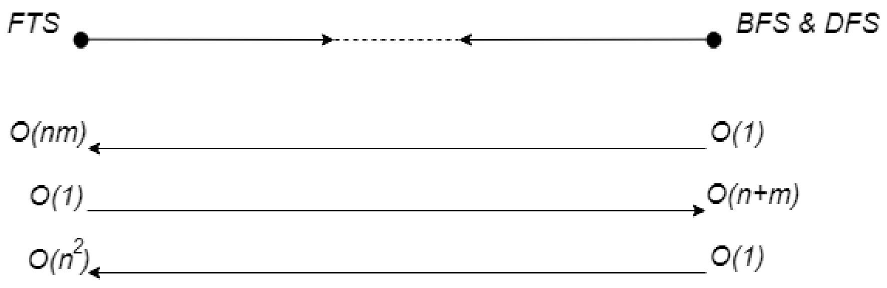


Figure 2.1: Index trade-off: query time vs index size

Major approaches can be divided into two main categories, according to whether they follow an interval labeling methodology or rely on a 2HOP indexing approach. The 2HOP indexing technique was first proposed in [2], and consists in computing for every node u two sets defined as predecessor and successor, which are respectively defined as the set of nodes that reach vertex u and that can be reached from vertex u. These sets are then used to answer reachability queries between u and v by verifying that the intersection between the predecessor set of v and the successor set of u is not empty. While this technique has an index construction time in the order of $O(n^4)$ several strategies have been proposed to improve the original algorithm such as the divide and conquer approach presented in [3], in which 2HOP indices are computed for k partitions of the original graph and then merged, which results in an $O(n^3)$ construction time. Conversely, interval labeling consists in assigning to each node a label representing an interval and, depending on the methodology used to define these intervals, there must exist a relationship between labels such that the following condition is always satisfied: given two nodes u and v and their labels, respectively L_u and L_v, u $\rightarrow$ v $\Leftrightarrow L_v \subseteq L_u$. GRAIL falls in the interval labeling category and relies on the min-post labeling method, which assigns intervals based on the post-order traversal of the graph, as will be explained afterwards.

A complete description of the main labeling approaches and their theoretical differences with GRAIL, as well as an extensive set of tests for index construction space and query times, can be found in [1], but there is one aspect worth noticing: when dealing with a DAG most of these methods create indices that cover only a sub tree of the input graph and then rely on different auxiliary indices or complementary search procedures, that cover the sub trees, that were not included in the main index to guarantee completeness. This is due to the fact that indices constructed on sub trees will obviously not cover the whole reachable set of nodes for every vertex, but will most likely include only the nodes that belong to the chosen sub tree upon which the index was built, which is equivalent of saying that the aforementioned subsumption relationship, namely u $\rightarrow$ v $\Leftrightarrow L_v \subseteq L_u$, will hold only for a subset of vertices. An example of this methodology is the dual labeling, presented in [4], which is based on defining labels for a spanning tree of the original DAG and on computing the full transitive closure for all edges that are not part of the spanning tree, or non-tree edges, which results in an index construction time of $O(n + m + t^3)$ and an index size of $O(n + t^2)$, where t is the number of non tree edges, but allows to answer queries in constant time.

This is significantly different to the GRAIL method for indexing, which computes labels based on the post-order traversal of the entire DAG: this enables to cover all reachable pairs of vertices, but generates positive answers for some non-reachable pairs, which will be henceforth referred to as false positives, or exceptions. Since false positive are possible, positive answers from label comparison cannot be relied on and, even though there are other alternatives, GRAIL will most likely perform either a DFS or a BFS traversal to determine whether u reaches v or not. While working with an index that generates false positives for reachability queries may seem counter intuitive, covering the whole DAG ensures that if the interval label of v is not subsumed by that of u, u does not reach v. Given that many real large graphs are very sparse, it is more likely that two randomly chosen nodes do not reach each other, and in this cases GRAIL is able to immediately return a negative answer in constant time. Additionally, it is possible to speed up the complete DFS/BFS traversal for positive answers by pruning the tree of nodes to visit using the index.

In the following section a complete description of the GRAIL approach will be presented, as well as the algorithms for creating the labels and for querying node reachability, but there exist several optimization techniques for improving performance on query resolution times and alternatives for dealing with false positive, such as exception lists. However, most of these go beyond the scope of this work and will be mentioned briefly. A detailed description of these techniques can be found in [1], along with their comparisons and implementation details.

2.3 Grail: algorithm and key ideas

Before delving into algorithms and implementation let us consider the core ideas behind the GRAIL approach. The foundation for this method lies on the observations that for direct trees (DTs) it is possible to build an index that occupies linear space, takes linear construction time and is able to answer reachability queries in constant time, and that a DT is a special case of a DAG where every vertex has a single parent or, conversely, that a DAG can be seen as a set of overlapping DTs.

In fact, for direct trees is possible to build an index using the min-post labeling technique, which assigns to each node u an interval label L_u such that $L_u = [s_u,\ e_u]$ where e_u, or outer rank, is defined as the rank of node u in a post order traversal and s_u, the inner rank, is the minimum e_u among the descendants of u.

$\forall u \in V, L_u = [s_u, e_u]\ s.t.\ s_u = min\left\{s_x \mid s_x \in descendants(u)\right\}, e_u = postorder(u)$

The fact that in direct trees every node has a single parent has strong implications for reachability queries: it is possible to determine whether any vertex reaches any other node in constant time by interval containment. This is due to the fact that if node u has a bigger outer rank value than v's, it will be visited later in a post-order traversal, whereas having the same inner rank implies that v is one of u's descendants. In graph 2.2 we see that vertex 4 does not reach node 9 since [2,2] is not included in [5,5], whereas node 3 does in fact reach 9, since [2,2] is included in [1,4], so a simple interval comparison is all that it takes to determine that u $\rightarrow$ v and, in general, that u$\rightarrow$v $\Leftrightarrow L_v$ € L_u. Furthermore, this index can be constructed through a simple DFS, which guarantees that its construction time is linear in the number of vertices.

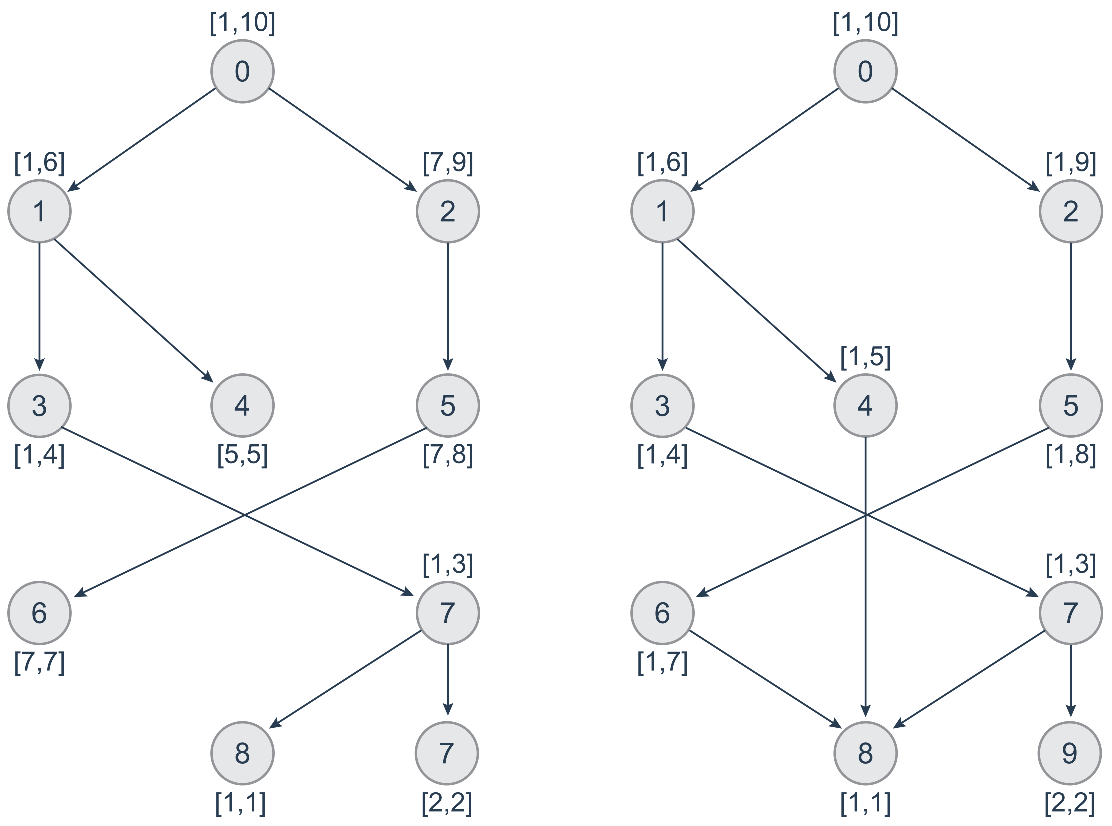


Figure 2.2: Labeling intervals for DT (left) and DAG (right)

The GRAIL method builds upon these two concepts and proposes to generalize the min-post labeling to DAGs, which allows to capture all reachable pairs while falsely marking as reachable pairs that do not, in fact, reach each other. The resulting labels can be seen in graph 2.2. Let it be noticed that this behavior is a direct consequence of nodes being able to have more than one father. As can be seen in 2.2, node 8 has vertices 4, 6 and 7 as fathers and the min-post labeling implies that node 8's inner rank will propagate to all of his parents and ancestors, such as 6, 5 and 2. This entails that two nodes that do not reach each other but have a common descendant, such as

4 and 5, could be marked as positive. And in fact, this is the case for 5 and 4: [1,5] is contained in [1,8] and this would result in concluding that 5 does reach 4, which is not true. As already stated false positives, or exceptions, must be dealt with by creating exception lists, which consists in maintaining a list of all the vertices that will result in a false positive for every node, a convenient approach for small graphs, or by resorting to a DFS/BFS query that uses interval comparison to prune the tree at every level. This method highlights that the key idea behind GRAIL is to avoid visits, or any kind of processing, for pairs of nodes for which non-reachability can be ascertained through interval comparison.

Furthermore, we note one more time that in a DAG there is more than one path from the root to the node and that there are many ways to choose the next node to visit in a DFS/BFS traversal, which constitute the basis for GRAIL's approach to reduce the number of exceptions: randomized multiple interval labeling. Instead of using a unique labeling, it is more convenient to perform many traversals following random orders, creating multiple interval for every vertex.

The basic intuition behind this is that creating more labels will minimize the impact of exceptions on performance, and it will not have a significant impact on index creation time or size. In fact, since GRAIL maintains d intervals per node, its index size will be $O(dn)$, which is still linear on the number of vertices, and the same will hold for the index construction time, which is $O(d(n+m))$, since d DFS/BFS traversals will be required to create those indices. In other words, for reachability queries GRAIL will still take constant time if the pair of nodes is immediately found as not reachable, whereas it will perform a traversal which will be pruned recursively at every level in case of positive interval containment.

In order to check for reachability we will redefine labels and rules for interval containment as follows: for a given node u and for a given number of intervals, or dimension d, the new label L_u is ${L^1}_u$, ${L^2}_u$, $\dots$ ${L^d}_u$, where $1 \leq i \leq d$ and ${L^i}_u$ is the interval label obtained from the i-th traversal of the DAG. Furthermore, given two nodes u and v we say that $L_v \subseteq L_u \Leftrightarrow \forall i \in [1,d] {L^i}_v \subseteq {L^i}_u$. If $\exists\ i\ s.t.\ L^i_v! \subseteq L^i_u\ then\ L_v! \subseteq L_u$, which implies that u $! \rightarrow$ v. In order to prove the validity of that implication we shall refer to the original paper's proof, which starts by assuming that $L_v ! \subseteq L_v$ which means

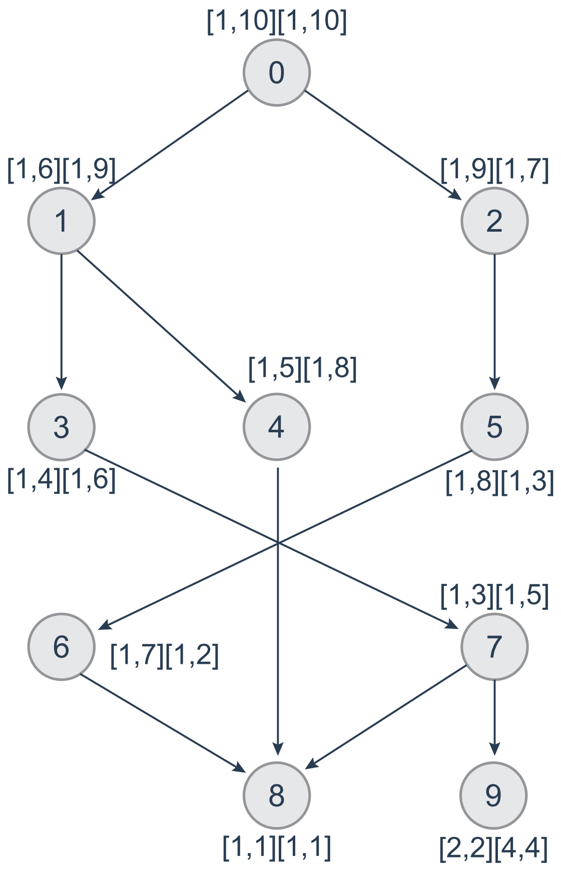


Figure 2.3: Multiple labels for DAG

that in some dimension i $L_v^i \, !\subseteq L_u^i$. Let us assume that u $\rightarrow$ v, and let us call x and y the two lowest ranked nodes that are respectively under u and v. If u $\rightarrow$ v then the following relationship must hold for the ranks of v and u in i: $r_u > r_v$ in post order, and $r_x \leq r_y$, otherwise r_x would not be u's inner rank. The contradiction is evident, since $L_v^i = [r_y, r_v] \subseteq [r_x, r_v] = L_u^i$, which leads to $L_v \, !\subseteq L_v$. Let it be noticed that usually a small number of labels is sufficient to reduce drastically the number of exceptions. By adding just one label, one can notice that the number of exceptions from labels for DT to labels for DAG in graph 2.2 decreased from 15 to just 3. Furthermore, since the number of possible labelings is exponential and there is no easy way to compute the minimum number of labels required to remove all exceptions, for bigger graphs is better to avoid creating a huge number of labels and to stop this process after a small number of dimensions has been created: usual values for d go from 2 to 5, regardless of the graph's size. As already observed, there is a variety of alternatives and modifications for optimizing the core algorithm, some of which regard the approaches for computing the labels. In [1] the authors propose some non-random methods for creating labels, and most of these are based on the idea that for the ith+1 traversal it is desirable to tackle nodes with the highest number of exceptions in the ith traversal and they propose to perform a heuristic-based guided traversal, and suggest several metrics that may be exploited by the heuristic. However, during the rest of this work, we will adopt both the lexicographic-ordered and the randomized approach for the CPU-based implementation of the algorithm, and a lexicographic-ordered traversal for the GPU version in order to have a comparison base for the two algorithms.

As can be seen in algorithm 1, which defines how to compute GRAIL labels following the random approach, the labeling procedure consists in calling the *randomizedVisit* function for every root of the graph and in using a global variable defined as r, or rank, to keep track of the number of nodes visited in the current DFS exploration. As showed in the algorithm the r variable is used to compute a node's label as follows: during the exploration of node u *randomizedVisit* is called recursively for every child, and the minimum inner rank among all of u's children, here defined as r_c^*, is used to determine u's label for the i-th iteration, or L_x^i. Notice that r_c^* is not defined for root nodes, for which L_x^i is defined as $[r,r]$, which explains why the inner label of a node is computed as the minimum between r and r_c^*.

When the label comparison is inconclusive there are different search strategies that we can use to answer reachability queries, such as BFS, DFS or Bidirectional BFS (BBFS) which consists in starting a bottom-up and a top-down BFS simultaneously. Depending on the analyzed graph's topology and on the system resources one may provide better results than the other, and some optimizations may give better results with a particular methodology rather than the others, but there is not a general-case best strategy. Figures 2 and 3 show the pseudo code for answering reachability queries through a label-guided DFS and BFS exploration and, finally, the *isReachable* function determines whether node v is reached by node u.

As can be seen, for both the GRAIL-pruned DFS and BFS versions the first thing that is checked is whether $L_v! \subseteq L_u$. If that is the case, we can immediately return that u $!\to$v, whereas if that is not the case we have to

```
Function RandomizedLabeling($G$, $d$):
    foreach $i \in [1, d]$ do
        r = 1;
        foreach $x \in Roots(G)$ do
            call randomizedVisit(x,i,G);
        end
    end
    return

Function randomizedVisit($x$, $i$, $G$):
    if $x$ already visited then
        return
    end
    foreach $y \in Children(x)$ in random order do
        call randomizedVisit(x,i,G);
    end
    $r_c^* = min(s_c^i : c \in Children(x))$
    $L_x^i = [min(r, r_c^*), r]$
    r = r+1;
    return
```

Algorithm 1: GRAIL labels computation

```
Function isReachableDFS($u$, $v$, $G$):
    if $L_v! \subseteq L_u$ then
        return false; // u does not reach v
    end
    foreach $c \in Children(u)$ s.t. $L_v \subseteq L_u$ do
        if isReachableDFS(c,v,G) then
            return true; // u reaches v
        end
    end
    return false; // u does not reach v
```

Algorithm 2: Label-guided DFS for answering reachability queries

verify that this is not a false positive by performing a full search and, in this scenario, there is no significant difference between the label guided versions of DFS and BFS: they both loop over node's u list of children and decide whether to add node c, where c belongs to children(u), to the stack/queue of nodes to visit depending on whether $L_c \subseteq L_v$, which ensures that no time is wasted exploring nodes that cannot reach v.

We conclude the chapter by considering that there are several parameters that determine whether it is preferable to use a DFS or a BFS, such as the graph's maximum depth, which is the number of levels, and the maximum diameter, which is the number of nodes of a level. Additionally the distance, both in depth and breadth, between the nodes for which reachability is queried plays an important role in determining the overall query resolution time.

2.4 Grail CPU implementations

Finally we move on to propose two GRAIL implementations that will constitute the basis for our comparison with the CUDA version. These versions include the two phases of the algorithm, namely indexing and querying, rely on the previously presented algorithms 1 and 2 and are based upon the same graph representation and data structures. They are set apart by the fact that the first version is sequential whether the other one is multithreaded, and during this discussion we will refer to these versions as *sequential* and *query parallel*. However, it is worth noticing that these two versions are developed entirely on the CPU and, as such, the *query parallel* application is characterized by parallelization strategies that revolve entirely around task parallelism, which assigns distinct tasks to different threads, or processes, running on several processing units. Conversely, parallelizing applications on NVIDIA GPUs requires them to be designed in order to exploit data parallelism, as already mentioned, and this will result in remarkable differences between the adopted parallelization strategies.

We will start our discussion by noticing that all of the developed GRAIL implementations adopt the same format for representing the graph, which is called *Compressed Sparse Row*, or CSR from now on. There are many alternatives ways to model graphs but CSR is particularly well suited for representing very large graphs since it is basically a matrix-based representation that stores only the nonzero elements of every row and, as such, is able to offer fast row access while avoiding useless overhead for very sparse matrices. Furthermore, for reasons that will be explained subsequently, this representation will be fundamental when developing the CUDA version of

GRAIL, so it makes sense to adopt it for all versions for coherence. Essentially, in the CSR format edges are represented as a concatenation of all the adjacency lists of every node and, as can be seen in figure 2.4, two additional arrays are used to store information about the cardinality of each node's set of adjacent vertices and to index the aforementioned children array. As an example, in order to iterate through node's 3 children we would have to access the 2 elements of array *children edges* starting at position 4, since *children cardinality*(3) is 2 and *children indices*(3) is 4. We also point out that, traditionally, the *children cardinalities* array is not necessary, since cardinality of node i can be computed through two accesses to the *children indices* array but, as explained in chapter 3, the CUDA architecture sets a high price for sequential accesses on consecutive array elements by the same thread, so it is preferable to retrieve cardinality using this array.

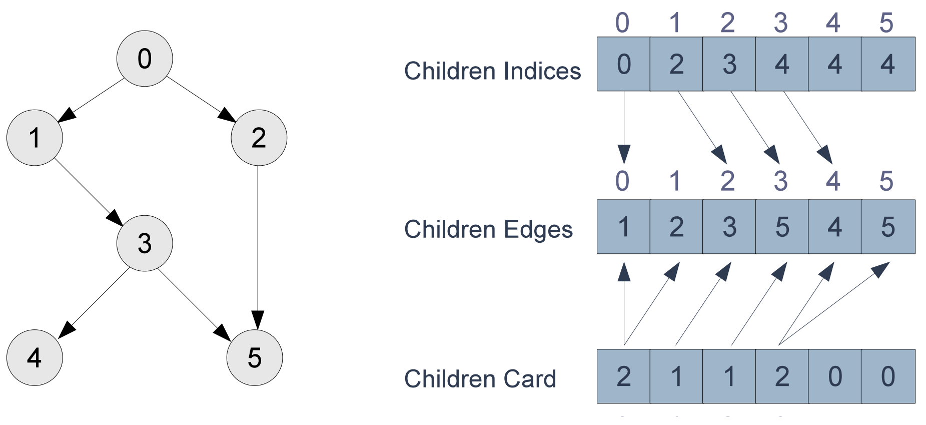


Figure 2.4: Compressed Sparse Row Format

Having explained the data structures used to model graphs we turn towards explaining the few differences between the *sequential* and *query parallel* versions, which provide a straightforward implementation of algorithm 1 for creating labels, and two different query resolution strategies can be chosen, one based on the DFS approach of algorithm 2 and an alternative based on BFS, which can be seen in algorithm 3. The only substantial difference between the two versions lies in the fact that the *query parallel* application computes d different labeling intervals by having d threads perform d concurrent DFS traversals of the graph, and then gather data about reachability by generating N random pairs of nodes and using several threads that independently solve $N/numThreads$ reachability queries each, whereas the other version does these tasks sequentially.

```
Function isReachableBFS($u$, $v$, $G$):
    if $L_v! \subseteq L_u$ then
        return false; // u does not reach v
    end
    Queue Q;
    Q.push(u);
    while Q.NotEmpty() do
        x = Q.pop();
        if $v \in Children(x)$ then
            return true;
        end
        foreach $c \in Children(x)$ do
            if $L_v \subseteq L_c$ then
                Q.push(c);
            end
        end
    end
    return false; // u does not reach v
```
Algorithm 3: Label-guided BFS for answering reachability queries

2.4.1 Benchmark

In order to have a baseline for comparison with the GPU based implementation, which will explore the graph in a BFS fashion to solve reachability queries, both the sequential and the query parallel version were tested on the same benchmark used by the GRAIL authors in [1]. All tests regarding both the CPU and the GPU implementations of the GRAIL algorithm were performed using a machine with an x64 Intel Core i7-7770HQ, which has a quad-core processor running at 2.8 GHz and 16 GB of RAM, with Ubuntu 16.04 as OS. Additionally the data set was obtained following the link in [1] and can currently be found at https://code.google.com/archive/p/grail, so all of our tests were executed on the same set of graphs that the original GRAIL algorithm was tested on. Hence we refer to the original paper for tests regarding comparisons between GRAIL and a variety of other indexing algorithms, as well as the effect of several of the aforementioned optimization techniques that the authors propose for GRAIL, whereas we will focus on explaining the main categories in which we can divide these graphs and on the results obtained by these CPU in order to provide a useful baseline for comparison against the GPU GRAIL implementation.

Finally we introduce the metrics against which we will evaluate all of our implementations, which are total query resolution time, index construction time and the number of positive answers, i.e. the nodes that reach each other, that were found, which is often correlated to the query resolution time since positive pairs require to explore the complete graph from the root to the node. On the other hand it is worth noticing that these data sets were slightly modified to impose that there is a single root node, which is necessary for the GPU based implementation, as will be explained later, and this means that, while they maintain the same topology than the original graphs, some of them have a larger edges set. All the results are averages over twenty runs, each of which consists in testing reachability for 100K randomly generated query pairs.

Table 2.1: Small sparse data set

Data set	Vertices	Edges	Avg Degree
Agrocyc	12685	13658	1.07
Amaze	3711	5505	1.48
Anthra	12500	13333	1.07
Ecoo	12621	13576	1.08
Human	38812	39817	1.03
Kegg	3618	5576	1.54
Mtbrv	9603	10439	1.08
Nasa	5606	6539	1.17
Vchocyc	9492	10346	1.09
Xmark	6081	7052	1.16

We conducted our experiments on three different data sets, all of which are generated from real applications, which can be divided depending on the cardinalities of the vertices and edges sets, as well as on their average degree which is an indicator of the graph's sparsity and is here defined as the ratio between the cardinality of the edges set against the cardinality of the vertices set. These three data sets will be referred to as *small sparse*, *small dense* and *large*. The *small sparse* data set, as can be seen in 2.1 is composed of small graphs, all with $|V| and |E| \leq O(100k)$ and with a degree less than 1.5. Among these graphs we point out that the Kegg and the Amaze data sets share a different structure than the other graphs. In fact, they both have a central node that has a very large in-degree and out-degree, which will cause them to have a higher number of reachable pairs than the others graphs in the same data set. In contrast the *small dense* data set in 2.2 contains small and dense graphs, extracted mostly from citation graph data sets, such as Citeceer and Pubmed, or ontologies like Go. Most of these graphs have more than 4 as average degree, even though the cardinalities of the vertices and edges sets remains under $O(100k)$. At last, we will have the opportunity to test GRAIL's scalability through the *large* data set which contains very large graphs, such as the Uniprot-150 which has $25M$ vertices and $46M$ of edges, as reported on 2.3. The authors of the original paper reported that these were among the largest DAGs ever considered for reachability testing, and note that most of the other reachability algorithms were unable to even run on some of these sets. As for any particularities of this data set we note that the Unitprot subset has a distinct topology: these DAGs have a very large set of roots that are all connected to a single sink through very short paths, which will have significant implications for reachability testing.

Table 2.2: Small dense data set

Data set	Vertices	Edges	Avg Degree
Citeseer	10721	48830	4.55
Go	6794	13425	1.98
Pubmed	9001	42637	4.74
Yago	6643	47568	7.16

We started our analysis by testing the 100k reachability queries with unguided, or pure, DFS in order to provide a baseline for comparison against several configurations of the GRAIL algorithm. Subsequently, we tested the sequential DFS, where queries were answered sequentially through pruned-DFS, and sequential BFS, in which we tried the pruned BFS approach that we proposed in algorithm 3. Then we compared these results against those obtained by the query parallel DFS and query parallel BFS version, which was configured to run with 16 threads. In this context, we replied all of these tests for labels with dimensions 2 and 5, in order to see if and how the label dimension exerts some influence over querying times, and we can conclude that, as the GRAIL authors explained, for most graphs there is no need to use a label dimension greater than 2, since for nearly all graphs and test configurations we found that querying times do not show any significant variation, and in most case the difference is less than 1 ms. We can also confirm that the index creation time grows linearly in the label dimension, or more precisely $O(d(n+m))$, in all of our tests. In light of these results, we will proceed to show only the results obtained for dimension 2.

Table 2.3: Large data set

Data set	Vertices	Edges	Avg Degree
Citeseer	693948	925779	1.33
Cit-Patents	3774769	17034732	4.5
Uniprot 100	16087296	30686253	1.91
Uniprot 150	25037601	46687655	1.86
Uniprot 22	1595445	3151600	1.97

2.4.2 CPU GRAIL Tests

When testing querying times, which are reported on tables 2.4, 2.5 and 2.6, all of the obtained results matched our expectations regarding the effect of being able to discard in constant time the pairs of nodes for which interval containment was not verified, as can be seen by comparison with the pure-DFS querying times, as well as the index's scalability for larger graphs. Similarly, the query parallel DFS and BFS versions outperformed consistently their sequential counterparts, being 2 to 6 times faster in most tests. Likewise it can be noted that the sequential-BFS configuration was slightly slower than its sequential DFS counterpart, even for the *small-dense* data set for which the original paper expected a possible improvement, but all querying times were within the same order of magnitude.

Additionally, we turn our attention to particular results, regarding tests on the *small sparse* and *large* data sets, that deserve further analysis. It

Table 2.4: Sparse tests (all times in ms)

Data set	Positive Queries	Pure DFS	Index CT	Sequential DFS	Parallel DFS	Sequential BFS	Parallel BFS
Agrocyc	122	28.19	0.30	6.10	2.37	5.96	2.73
Amaze	17309	172.7	0.17	26.02	5.76	33.96	7.64
Anthra	98	23	0.30	6.26	2.29	6.30	2.80
Ecoo	115	31.1	0.28	6.37	2.38	6.51	2.63
Human	19	15.54	0.74	6.54	2.39	6.71	2.33
Kegg	20214	209.85	0.16	28.08	7.24	59.90	17.20
Mtbrv	163	34.17	0.22	5.99	2.43	6.13	2.50
Nasa	547	71.33	0.26	6.47	2.46	7.45	2.76
Vchocyc	162	35.33	0.22	5.97	2.29	6.02	2.41
Xmark	1478	163.87	0.25	10.21	3.49	33.13	12.02

Table 2.5: Small dense tests (all times in ms)

Data set	Positive Queries	Pure DFS	Index CT	Sequential DFS	Parallel DFS	Sequential BFS	Parallel BFS
Citeseer	370	14290.22	1.22	20.1	4.27	28.72	11.94
Go	249	46.67	0.51	13.11	2.83	15.13	3.52
Pubmed	658	1222.1	0.87	11.2	3.53	13.60	4.46
Yago	175	30.23	0.77	7.51	2.49	8.96	2.96

was already mentioned that the Amaze and the Kegg graphs in the *small sparse* data set have a particular topology that results in a higher level of reachability among pairs of nodes than other nodes in the same set, and it can be seen in 2.4 that for these graphs 17% and 20%, respectively, of all analyzed query pairs are reachable. Consequently their aggregated querying times are higher than the average for this set but, if we analyze the number of positive queries of all the other *small sparse* graphs, it may be seen that, while the ratio between positive queries and total queries goes from 0-1% to 17-20% the resulting querying times take only 3 to 4 times longer. Similarly we observed that the Uniprot family of graphs in the *large* data set all have a large number of roots which converge to a single sink via a very short path. Since there is a single sink, we would expect all the nodes in the graph to have the same inner rank which is a known cause for false positives, and so we would expect a large amount of exceptions and, consequently, a significant aggregated querying time. However we can see in table 2.6 that this is not the case. Interestingly, the fact that the graph is has a very low depth, i.e. 6 levels, allows the queries to be answered rapidly even in a scenario that would result problematic for the GRAIL index: in fact, querying times are basically the equivalent to those of the pure DFS algorithm.

We conclude our analysis by noticing that the results obtained for our GRAIL implementation share many similarities with those presented in the

Table 2.6: Large tests (all times in ms)

Data set	Positive Queries	Pure DFS	Index CT	Sequential DFS	Parallel DFS	Sequential BFS	Parallel BFS
Citeseer	0.40	18.64	42.3	13.24	3.50	12.96	3.64
Cit-Patents	42	220458.57	1417.2	1831.41	617.80	3423.0	1576.90
Uniprotenc 100	0	30.58	1055.1	24.57	4.72	23.74	4.98
Uniprotenc 150	0	31.1	1632.2	24.96	4.83	25.20	4.82
Uniprotenc 22	0	20.4	82.2	20.50	4.13	20.70	4.10

original GRAIL paper. Given the different architectures of the two machines on which the tests were executed, our tests produced index construction and search times of the guided DFS algorithm smaller by at least one order of magnitude than those of the authors, but many similarities emerged in the comparison. In particular, for all the data set the amount of queries that resulted in a positive answer, or positive queries, were practically identical, and the graph instances that required larger processing times, both for querying and indexing, were the same in most cases. The only significant difference between the two studies regarded the times of the unguided DFS search for testing reachability. In particular, their DFS proved to be more competitive with respect to their guided DFS given that it provided similar or slightly inferior times for most instances of the small sparse and large real data sets, whereas our unguided DFS was consistently slower than our guided DFS. Moreover, our unguided DFS performed better on the small sparse data set, whereas for the most critical instances of the small dense and large real data sets, such as *Citeseer* or *cit-Patents*, their unguided DFS produced better times. Interestingly, both of the experimental results confirmed that the topography of the Uniprotenc family of graphs favors an unguided DFS exploration, given that in both studies there were no significant differences between times required by guided or unguided DFS when testing reachability in these instances.

Chapter 3

Parallel Computing on CUDA

Before proceeding towards an in depth analysis of the strategies that were considered for parallelizing GRAIL it is useful to discuss the development environment, so as to better understand the challenges faced and the guiding principles upon which many design choices were based. Specifically this chapter will be devoted towards providing an overview of the context in which the CUDA architecture has found widespread popularity in the software development market and towards presenting the CUDA architecture and programming model, as well as the main factors that have to be taken into consideration for optimizing applications on this platform. Finally, we will conclude the chapter with a study of the main approaches to implement graph traversal on CUDA, as well as with an in detail explanation of the algorithm that we chose.

3.1 The case for GPU parallel computing

Historically speaking it can be affirmed that most of the development of software applications has been influenced in great measure by the hardware advancements, both in terms of memory capacity and processor speed, that characterized the past decades. In fact, until the early 2000 the software development market was lead by what is commonly known as *Moore's law*, which so accurately predicted the growth of the number of transistors per chip for the recent past. Confident in the accuracy of this forecast, software developers relied steadily on the hardware advances that accompanied each new generation of processors to speed up the performances of their programs, which were designed for sequential execution on CPU. Essentially it has always been possible to develop software that will simply run faster on the next-generation CPUs, without it being actively designed to benefit from future hardware advancements. Nonetheless it is now equally accepted that the number of transistors per chip is no longer growing at a rate that can satisfy market demands, which has lead both vendors and software houses towards exploring different ways to improve their applications and meet current market's demand.

Inevitably major chip vendors have moved towards offering micro-processors with an ever increasing number of completely independent cores, with constant focus on supporting hardware threads in order to speed up sequential programs. On the other hand, this has not been the only approach for obtaining better performances that has taken root in the last decade. In fact, from the first years of the 2000 there has been a constant effort towards exploiting GPU capacities for more than the usual graphic-related purposes, a practice which is now called General Purpose GPU, or GPGPU in short. Essentially during the same years in which *Moore's law* hit a saturation point, the video games industry continuously required GPU vendors to offer chips that allowed a massive number of floating-point calculations per second, or FLOPS, which are also a commonly used metric in this field for measuring both graphic unit and software performances.

As a result of this constant drive towards increasing the number of FLOPS on graphic units, which exceed by far those of a top of the line CPU, programmers started to turn towards developing software that could run on GPUs long before the release of the first CUDA APIs by NVIDIA. Software developers originally resorted to forced implementation of algorithms through APIs that were designed to paint pixels on screen, as reported on [7], so it is safe to say that when NVIDIA released CUDA in 2007 the software market was more than ready for an environment designed specifically for GPGPU. Furthermore, CUDA's rise to the most complete platform for GPGPU was aided by the fact that NVIDIA graphic units were already present in most desktop computers, which represented an amazing opportunity for software vendors to develop applications that relied on a GPU architecture that was already in most PCs in the world.

However let us point out that CPUs and GPUs cannot be thought of in terms of deciding to use one instead of the other, but rather on a complementary way: each of them is the result of a different architecture philosophy and was specifically designed to address problems that are inherently different. Historically CPUs have been designed with versatility in mind, which is the ability to run diverse algorithms, and research in this field has focused on reducing latency through sophisticated flow control and large memory caches, both of which require significant on-chip space, whereas GPUs do not need either and exploit most of the on-chip space to host a large number of ALUs.

As a matter of fact, one can argue that this is the main difference between the two architectures: GPUs do not aim to reduce the execution time of a single thread, but are rather designed to increase the throughput of a number of threads typically in the order of thousands, and do so by covering the long latencies caused by memory operations with a large number of threads, in order to always have threads ready for execution. Conversely the lack of branch prediction and flow control causes GPUs to face problems such as branch divergence, which will be discussed later, and are utterly unable to compete with CPUs for algorithms, or parts of them, that are inherently sequential. Generally speaking, while considering how to improve applications it is important to keep in mind that not all algorithms can be completely ported to GPUs, and that to achieve great performances it is often necessary to divide the program such that CPU and GPU each handle the parts that they are most suited to deal with.

3.2 CUDA Programming Model

In this context, the CUDA programming model introduces an abstraction for representing the computer as a single *host*, which represents the CPU, and possibly many *devices*, which are distinct graphic units. CUDA is a platform for writing code that will run on NVIDIA GPUs and is language independent: there are CUDA APIs for all major languages such as C, C++, Fortran and Python. However, CUDA code is always composed by regular code and special functions which typically exhibit a great amount of data parallelism and will run solely on the GPU when launched from the host. These functions are called *kernels* in the CUDA nomenclature, they get typically executed by a large number of threads on a device and are the primary source of parallel execution on CUDA, though they are not the minimum scheduling unit. The source code is compiled by a CUDA-enabled compiler such as the NVIDIA C Compiler, or NVCC, for C and C++, which divides and compiles source code depending on where does it have to run. Since it is not possible for a program to run completely on a device, all CUDA programs start execution on the host, until a kernel is launched.

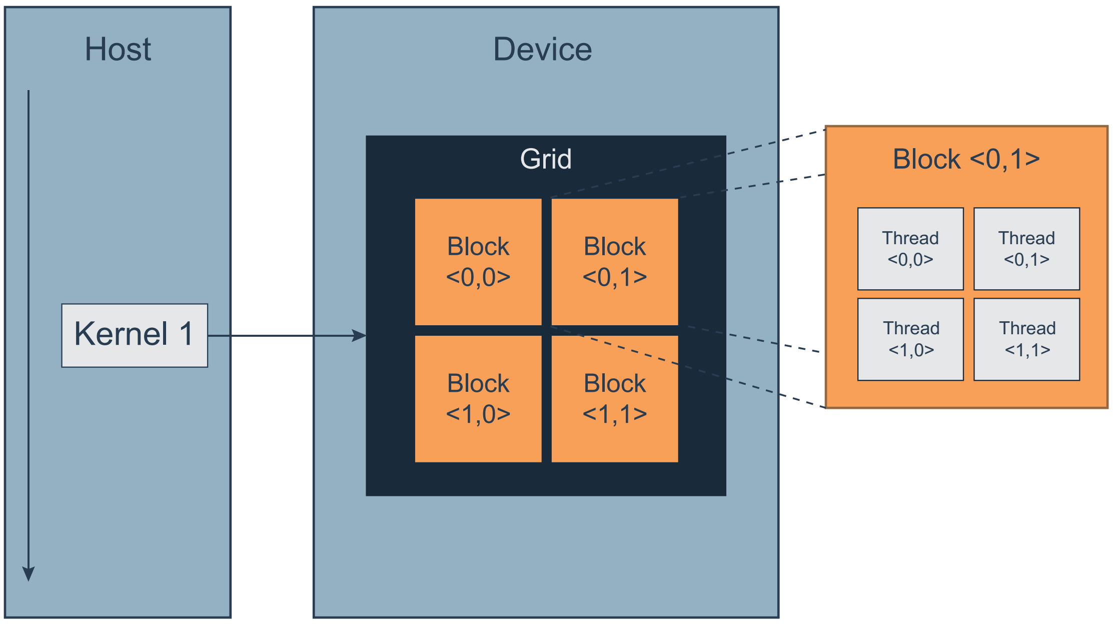


Figure 3.1: CUDA programming model

When invoked, kernels are typically launched following a hierarchic structure that is the main source for both coarse-grained and fine-grained parallelism: in fact, a kernel is run by a *grid*, which is composed of *blocks* of threads. Grids and blocks may have more than one dimension depending on the type of data that has to be processed: they may have a one-dimensional, two-dimensional or three-dimensional structure, in order to easily work with vectors, matrices or volumes, and each thread has built-in registers that assign a unique ID within its block and grid. This structure allows software developers to divide the algorithm into sub-problems that can be solved independently and in parallel by blocks of threads, thus providing both task-parallelism and coarse-grained parallelism, as noted on[9], while threads in a block can cooperate to solve these sub-problems concurrently, which also ensures data-parallelism and fine-grained parallelism. Furthermore, this is the reason for one of the hallmarks of the CUDA architecture, which is automatic scalability.

Basically, upon kernel execution a grid is created and divided into blocks, which are scheduled for execution on *Streaming Multiprocessors*, or SMs, and NVIDIA GPUs have a variable number of SMs. Since any block of threads can be scheduled in any SM, either concurrently or sequentially, in any order, a GPU with many multiprocessors will automatically execute the program in less time since more blocks are going to be executed concurrently, without the developer having to change the application. This desirable behaviour is also the main reason for which the CUDA programming model makes no guarantee on the order of execution of blocks and demands that, in order to ensure correctness, developers do not rely on blocks being executed in any order. In fact, threads within the same block can cooperate to solve the same sub-problem, as mentioned before, thanks to a shared memory and a barrier synchronization mechanism, whereas it is not expected that threads on different blocks should wait on each other, and there are few ways of achieving this without seriously compromising performances.

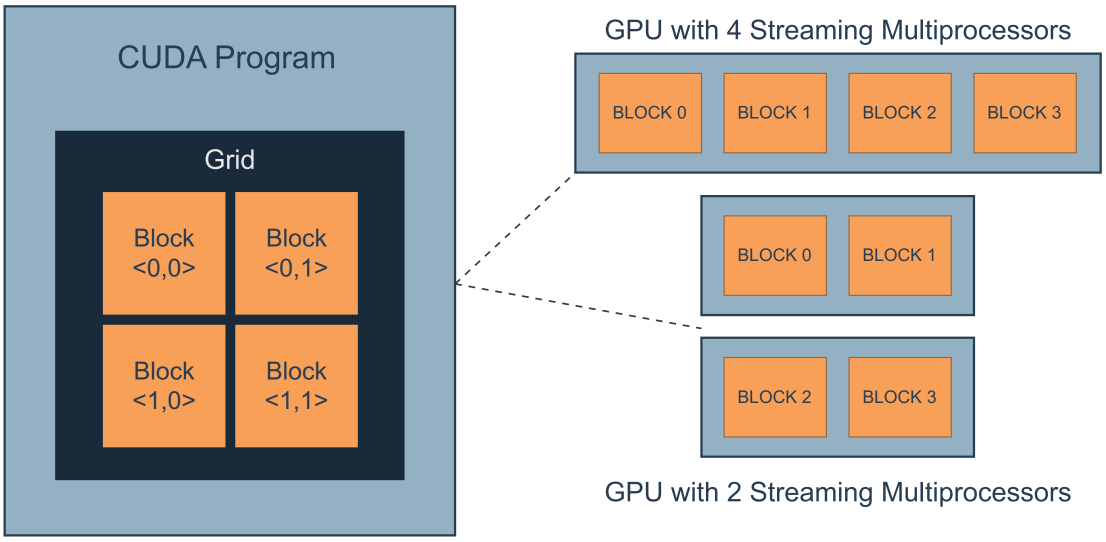


Figure 3.2: CUDA's automatic scalability

As already explained, GPUs have an array of SMs, and when a grid is created its blocks are distributed to all multiprocessors that dispose of sufficient execution resources. A Streaming Multiprocessor may execute one or more blocks concurrently, but threads within a block must be executed in the same Streaming Multiprocessor. SMs are built around the *Single Instruction, Multiple Thread* architecture, or *SIMT*, which manages a significant amount of threads at the same time by having them all execute the same instruction on different data. Interestingly when a block is scheduled for execution on a Streaming Multiprocessor, the threads in a block are divided into groups of 32 threads, which are called *warps* and are the minimum scheduling unit.

All threads in the same warp will always execute concurrently, but each of them has an individual instruction counter, which provides the ability to take different paths on a branch instruction such as an if or a for-loop. In other words, threads in a warp are expected to always execute simultaneously the same instruction, but they can take different paths depending on branches, which apparently constitutes a contradiction. However, it is time to note that, unlike CPUs, program instructions are issued in order and there is no speculative execution: if threads of a warp have to follow different paths depending on the result of a condition, the scheduler executes each branch serially, disabling at each iteration all the threads that are not on that path, and this behaviour is repeated on all possible paths, until all threads converge on the same instruction. This is called *branch divergence*, or simply divergence, which is a major factor in determining the overall execution time of a kernel and, as we will show, in many cases our parallelization strategies will focus on minimizing divergence.

Finally we conclude our discussion on the CUDA programming model by addressing the memory hierarchy, another abstraction that is key towards optimizing performances. Starting from the hardware point of view, it is worth noting that, while most memory chips are indistinctly referred to as SDRAM, NVIDIA GPUs actually dispose of a memory technology called Graphics Double Data Rate, or GDDR SDRAM, which is better suited to deal with transfers of massive amounts of data on a single clock than DDR SDRAM, the usual memory unit for CPU motherboards. The CUDA model keeps host side and device side memory as two separated memory spaces, and provides the host with functions to handle allocation, transfer and release of on device memory.

Additionally, the device memory is partitioned into several sub-memory spaces with a structure that resembles the hierarchy of threads. During its lifetime a CUDA thread has a set of private per-thread registers, and has access to a per-block *shared memory*, which is visible for threads on the same block and has the same lifetime than the block. These two memories can be used, respectively, for thread-private computation and for inter-block communication, and have fast access times since they reside in the SM. Additionally there are three other types of memory, called *global memory*, *texture memory* and *constant memory*, that can be accessed by all threads in a grid, but are not on chip and require more clock cycles for I/O.

Regarding these three types of memories, we can say that they physically reside on the same chip, even though they are optimized for different usages. A complete discussion on the specific purposes of each can be found in [9], but we will describe briefly the usages of shared and texture memories, since they will be used steadily in our implementation.

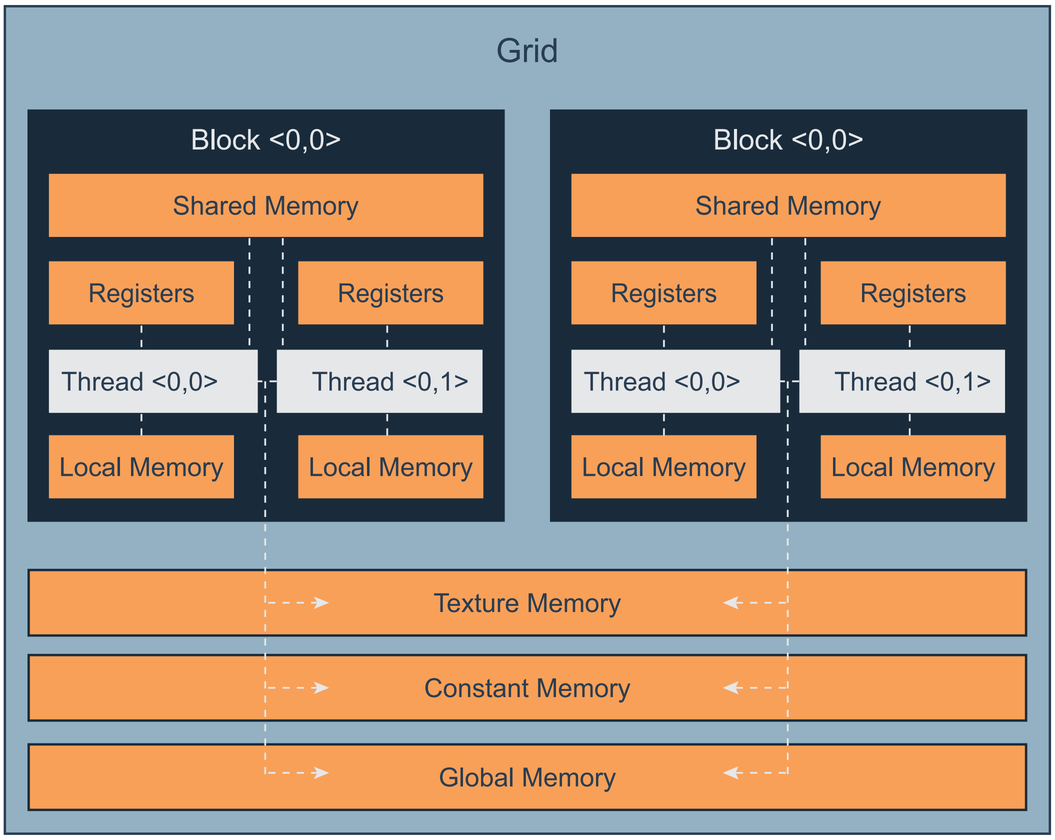


Figure 3.3: Memory Hierarchy

Shared memory is a region of memory that resides in the same chip that the Shared Multiprocessor, which implies that memory accesses to this region require lower latency and grant higher bandwidth that global memory, though to achieve peak performances certain memory access patterns must be followed. When the CUDA compiler encounters a variable labelled as shared, it allocates a private copy for every block in the grid: all threads belonging to the same block may see and modify that per-block variable, while access to the other block's variables is not allowed. As for the access patterns, shared memory is designed as an array of equally sized memory modules called banks, which are optimized to serve k simultaneous read/write requests to addresses in k distinct banks. However, if more requests to the same bank occur at the same time, they will be serialized and the operation will require longer times.

Hence programmers are expected to design code carefully to avoid limiting performances, and there are several factors that have to be taken into consideration, but here we want to address the so called *tiling pattern*, which will be used in several of our kernels, since it regards the common scenario in which an array stored in global memory has to be accessed several times by a block of threads. Usually the tiling pattern proposes to solve this by having all threads in a block load a distinct element of the array on a shared memory cache at the same time, wait on each other by synchronizing and only then perform their task on the common array data, before repeating this process if necessary. Furthermore threads with consecutive IDs are expected to load array elements with consecutive indices, since the device tries to minimize the number of transactions for consecutive store/loads by threads in the same warp, which is called *memory coalescence*, in order to maximize bandwidth usage.

Additionally, we will use the texture memory during graph exploration, so it is interesting to discuss some of its main features and uses. Traditionally designed for *openGL* and *DirectX* rendering pipelines and optimized for access patterns whose addresses exhibit a great deal of spatial proximity, it has recently gained traction in non graphic GPU computations since it possesses some useful features which, if nothing else, do not impact on bandwidth from global memory. Among its features, the first element worth noticing is that it is a read-only cached-on-chip memory, so it costs an access from global memory only upon a cache miss. Furthermore, it has a dedicated unit for address computation, which would otherwise be carried out by the kernel's assigned ALU unit, and it is especially well suited to broadcast data to separate variables during a single operation, as reported on section 3.5 of [9].

3.3 Graph Exploration on CUDA

We will conclude this chapter by making the argument for the use of BFS instead of DFS when it comes to graph traversal algorithms on CUDA. Since the next chapter is devoted to explaining how to port the GRAIL labeling phase, which is entirely DFS dependent, we will summarize here the most frequent techniques for CUDA based BFS, and we will explain the algorithm that we selected as well as its core ideas and implementation choices.

3.3.1 Challenges of parallel Depth First Search

At first glance it may not be clear why should DFS be harder to parallelize than BFS, since the core difference between them is the order in which they visit the same nodes and edges. However, if one looks at these algorithms under a different light, the underlying problem is clearer: BFS revolves around exploring the current frontier, which is the set of all unvisited nodes among the current vertex's adjacent nodes, whereas the DFS commits towards exploring a path until all its leaves have been reached and explored before going back. In other words, BFS tries to explore a set of nodes, and has no restrictions against these vertices being processed concurrently, whereas DFS imposes to visit a precise node at any time and, as such, is inherently sequential. There is also no approach that would allow us to organize the graph data in such a way that it exhibits data locality and so the main features that are traditionally used to speed up performances in parallel implementations are of no use.

Nonetheless note that this does not mean that there cannot exist a version of the DFS algorithm that makes use of the multi-threading capacities of a processor: usual approaches, however, tend to use more cores to perform concurrently many DFS, but this strategy is not valid for our algorithm, since during the labeling phase we have to assign to each node a ranking value that is strictly dependent on the number and order of previously visited nodes. Any strategy to parallelize the labeling DFS must take into consideration that a global variable must be read and modified every time a node is visited

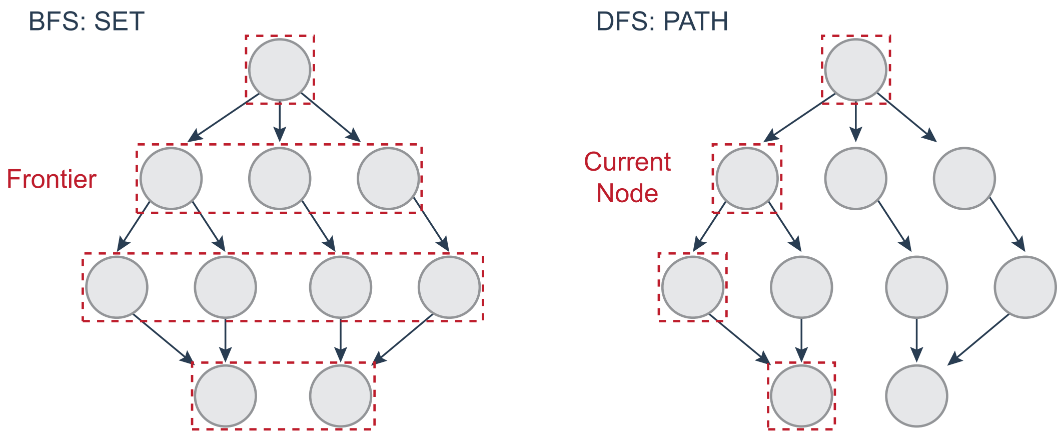


Figure 3.4: BFS vs DFS node processing

and that these visits must occur in a precise order, which is a major limit for parallelization, at least on the CUDA architecture.

3.3.2 Breadth First Search on CUDA

Most sequential BFS implementations have as a core data structure a queue to which they add, at every iteration, the set of nodes that have to be explored which is called frontier. Usually CPU based parallel versions rely on a unique queue which is complemented by a synchronization mechanism such as a lock or a semaphore. Nonetheless, this approach is of no use for porting BFS on CUDA: a lock and a queue may work fine for threads in the order of hundreds, but will never guarantee acceptable performances for fifty or sixty thousand threads competing for access and, even if we simply did not care about efficiency, CUDA does not provide lock or sophisticated atomic operations that go beyond atomic add, exchanges and similar.

In the past decade several CUDA implementations of BFS have been proposed, but they usually agree on replacing the queue with what is called a status or a frontier array: an array which contains one element for every node in V and represents whether vertex i will be part of the frontier on the next iteration or not. Henceforth we will use the term queue to refer to a conceptual queue, which will be actually implemented through a status array and several mechanisms built on top. Initial versions of the BFS on CUDA were based on the work in [10], which basically went little further than parallel exploration of the array: at every iteration one thread would be assigned to every node, the thread would process the vertex depending on its value on the status array and would finally update the status array for all of the vertex's children. While this version was important, since several papers cite it as the first BFS on CUDA, it has several performance shortcomings that make it practically unusable. In fact, even a preliminary analysis helps us understand why this algorithm fails to deliver reasonable performances: it may happen that, depending on the topology of the graph, only a subset of nodes belongs to the frontier, whereas this method requires a full array scan at every iteration, which entails huge levels of divergence. Furthermore, divergence is also caused by the different number of children that each node has, or its out-degree, since some threads will process thousands of children vertices while others none.

Several improvements have been put forward to address these shortcomings and an in depth discussion of the major approaches for parallelizing BFS on CUDA can be found in [8], along with their source code, but for brevity we will focus here on the ideas that are of most interest to our work. The principal methodologies exposed in the paper deal with postponing the processing of nodes with a high out-degree until all other nodes have been analyzed, which aims to reduce divergence by concurrently processing nodes that have similar out-degrees and is usually done as follows: at every iteration the number of children that the current node has is checked before proceeding and, if that number is higher than a predefined threshold, that node is added to a shared memory queue, private to each block, and its processing is deferred until all other nodes have been visited. Other approaches target the queue generation mechanism in order to avoid the full frontier array exploration at every turn: this is usually achieved through an in-block queue, kept in shared memory, on which threads store the children that belong to the frontier. Threads synchronize accesses to this queue through atomic adds on an index counter, also kept in shared memory, since atomic operations on shared memory do not add significant overhead, and at the end of the iteration all of these queues are merged into a global array through a very limited number of global atomic adds.

Ultimately all of these ideas suffer from limits imposed by the shared memory, in the order of kilobytes for each block on most architectures, and never allows the expected scalability, but we insist upon noting that the major approaches for improving CUDA BFS are based upon two strategies: optimizing the queue generation mechanism and limiting the overhead caused by divergence when simultaneously analyzing nodes with very different out-degree.

We conclude this section by analyzing an alternative approach for generating the queue that has gained significant approval in recent times and is based on parallel prefix sums. We recall here that prefix sum of an array is an algorithm that takes as input an array A, produces as output an array B of equal dimension and that the two are bound by the following relation: $B[x] = \sum_{i<x} A[i] \; if \; x > 0$ and $B[x] = 0 \; if \; x = 0$. Prefix sum is currently used by a great variety of algorithms in distributed or parallel environments: usual scenarios involve a set of k globally distributed processes, each of which has to write i_k elements in a shared memory without synchronizing. This is usually done by having a prefix sum performed on an array with the sizes of the i_k elements to be written, which produces the starting addresses of the partitions in which each process can then write independently.

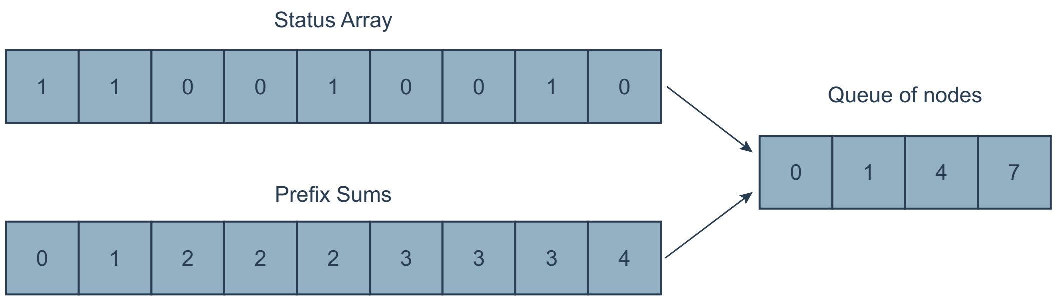


Figure 3.5: Prefix sum queue generation

Similarly this concept has been used to generate the queues as follows: given a status array of size n, where n is the number of vertices, we will use k threads, with $k \leq n$, to scan the status array and store the ids of the frontiers nodes in private per thread arrays, from now on called *bins*. Then a prefix sum on the lengths of these bins is done and each thread will use the result to write its part of the frontier on a global array, as can be seen in 3.5. This procedure will be used by the Enterprise BFS, and it is interesting to note that this approach divides frontier exploration from its generation which will be crucial in tackling divergence.

3.4 Enterprise BFS

In this context we chose to base our BFS implementations on a recently developed algorithm specifically designed for CUDA which is called Enterprise BFS, was proposed in [6] and is based upon many of the ideas previously exposed. Enterprise is designed to reduce divergence by exploiting both coarse and fine grained parallelism when each is required, and achieves this through three fundamental optimization strategies: it divides the frontier into three distinct queues which are generated through the previously explained prefix sum mechanism, the criteria for dividing the frontier depends on the out-degrees of each vertex and the BFS switches direction when certain nodes are found. We will address all of these points, but we immediately want to note that this algorithm has been used as a basis for our implementations, since we had to develop several versions of BFS, and, as such, there are some differences between the original implementation, whose code is available online, and ours.

In fact, we did not use the direction switching technique that they suggest since we use the BFS to create GRAIL indices and, as will be explained in the following chapter, this requires four distinct BFS traversals that are to be performed solely bottom up or top down. However it is worth explaining the rationale behind this idea, since it is very efficient: Bidirectional BFS, or BBFS, is widely used in certain scenarios to improve performance, and it might be the subject of future work for the query search phase. Essentially, Enterprise develops a heuristic approach for deciding if it is convenient to switch direction when a node with a large out-degree is found, in order to postpone or to avoid a visit to such a node.

Turning our attention to the ideas upon which we developed our BFS, as already stated Enterprise proposes to use prefix sum to generate three separate queues, which are set apart by the out-degree of their nodes. The main concept here is to divide the nodes that have to be processed depending on the number of children that each vertex has. In fact, nodes are inserted in the small queue if they have less than 32 children, in the large queue if they have more than 256 or in the medium queue for values in between. Subsequently these nodes will be processed by a number of threads befitting the amount of work that has to be performed: nodes belonging to the small queue will be handled by a single thread, whereas a warp will be assigned to each node in the medium queue and a block to each vertex of the large one.

Obviously three different kernels will be launched, each optimized to handle its queue, which will result in approximately the same amount of work for each thread in each kernel, as well as in assigning resources proportionally to each node's processing requirements.

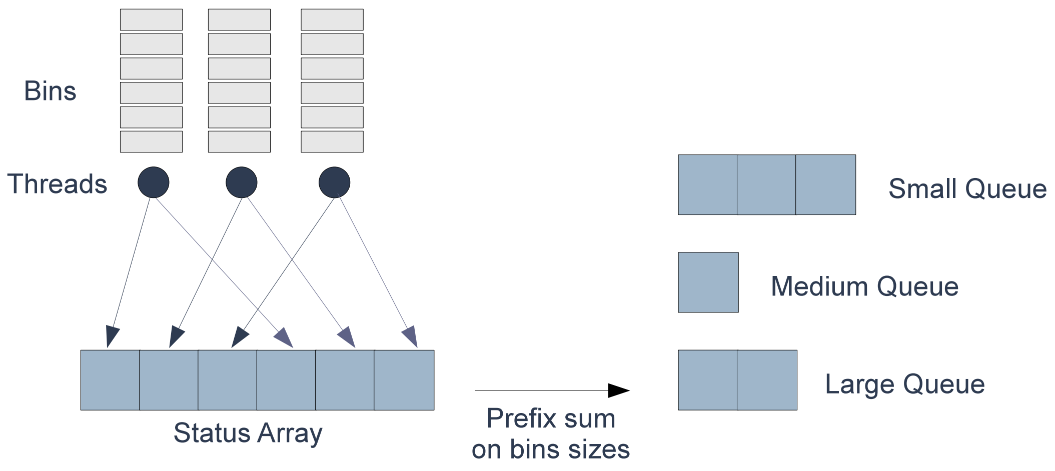


Figure 3.6: Enterprise queue generation

The Enterprise algorithm is divided in two phases: queue generation and exploration. In particular, during the first phase the status array is scanned by a kernel launched with 256 blocks of 256 threads each, for a total of 65536 threads. Prior to the kernel launch three arrays of $256 \times 256 \times 512$ elements each are allocated, which can be seen as each thread having three bins of 512 elements for the small, medium and large queue. These kernels iterate over the frontier array with an access pattern that guarantees coalescence as can be seen in figure 3.6, since consecutive threads will be assigned consecutive array elements, and will store in their internal bins the ids of the nodes that are to be visited, depending on the node's out degree.

After the bin generation has ended, all threads write the actual number of elements in each of its bins into three arrays in global memory, on which distinct prefix sums are then performed. These three prefix sums are executed in parallel by a different set of kernels and, given that in [6] there is no reference to the parallel prefix sum algorithm used, we decided to implement it following the Kogge Stone's strategy, as described in Chapter 8 of[7], which basically applies the well known parallel reduction pattern, as can be seen in algorithm 4. Afterwards a third set of kernels concludes the queue generation process by using the values computed by the prefix sums to copy the nodes ids stored in the internal bins into the respective global queue.

```
Function prefixSumKoggeStone(int *src, int *dest, int size):
    __shared__ int T[blockDim.x]
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    int stride;
    if tid < size then
        T[threadIdx.x] = source[tid];
    else
        T[threadIdx.x] = 0;
    end
    for stride = 1; stride < blockDim.x; stride *= 2 do
        __syncthreads();
        if threadIdx.x >= stride then
            tmp = T[threadIdx.x-stride];
            __syncthreads();
            T[threadIdx.x] += tmp;
        end
    end
    __syncthreads();
    dest[tid] = T[threadIdx.x];
```

Algorithm 4: Kogge-Stone kernel for prefix sum

The queue exploration phase has to be executed by three different kernels, one for each queue, since each thread has to compute the indices for accessing queue elements depending on the queue: we require threads that handle the small queue to take a node each and to process all of its children, which are less than 32, sequentially. Threads that deal with the medium queue nodes have to cooperate at warp level on the same node, and each thread in the warp is assigned one of its children, which are between 32 and 255. Finally the large queue requires cooperation at block level: each thread in a given block has to process a children of the same node.

We recall here that each thread has a block index, called threadIdx, and a grid index called blockIdx, which represent the thread's number in the block and the block's position in the grid, respectively, and are used to determine the element assigned to each thread. Specifically small queue threads compute their initial index with the following formula $vertex\_id = blockIdx * blockDim + threadIdx$, then execute two nested loops. In the inner loop a thread processes the current node's children sequentially whereas in the outer loop they select another small queue vertex by adding the number of nodes that have been processed at the current iteration: $vertex\_id += blockDim * gridDim$. Similarly, large and medium queue kernels adopt the same nested loop structure to iterate over their respective queues, but the formulas for computing the indices and iterating over the children vary since many threads have to cooperate on the same node by choosing a distinct children each. For the large queue kernel this is easily achieved by having $vertex\_id = blockIdx$ and $children\_id = children\_indices[vertex\_id] + threadIdx$. We recall that $children\_indices[x]$ contains the starting point of node x's children in array $children\_edges$, so we see that this access pattern enhances coalesced memory accesses to the $children\_edges$ array.

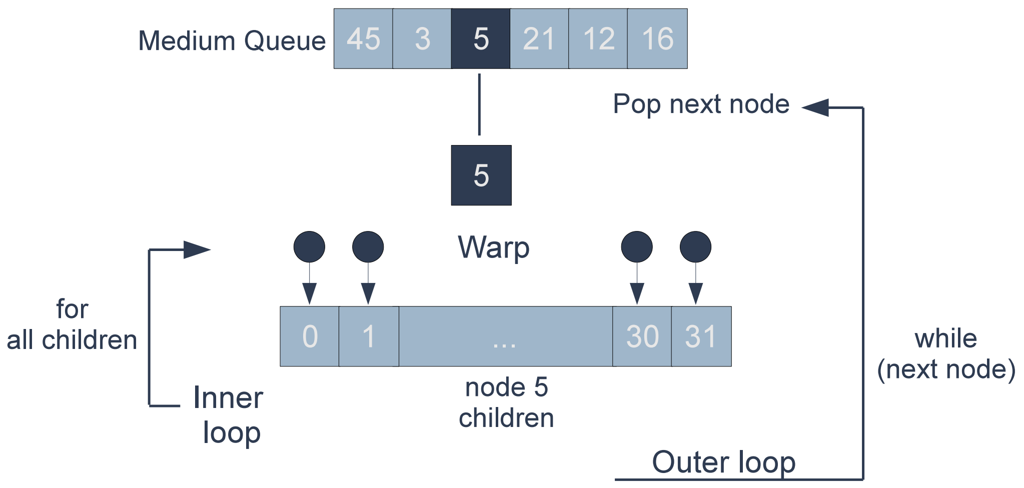


Figure 3.7: Enterprise medium queue exploration

Conversely the medium queue kernel has a less intuitive indexing approach since there is no explicit warp id and, in general, warp level cooperation has to be customized on the particular kernel's task. In Enterprise they propose to compute the node id with the following formulas $tid = blockIdx * blockDim + threadIdx$ which is used in $node\_id = tid/32$ to grant that every thread in the warp chooses the same node. Obviously the next node on the queue, if any, will be chosen by taking into consideration that one node is processed by 32 threads, so $node\_id + = blockDim * gridDim/32$. Furthermore all threads in a warp have an additional id for choosing the children that they have to process, which is given by $lane = tid$ & 31 which produces thread dependent values in range [0,31] and are used to ensure coalesced accesses to $children\_edges$ through $children\_id = children\_indices[vertex\_id]+lane$.

Part II

GRAIL Implementation

Chapter 4

Labeling implementation on CUDA

4.1 Introduction

This chapter aims to explain the approach followed to implement the GRAIL labeling method on CUDA. We recall here that GRAIL has labels defined as follows: $\forall u \in V, L_u = [s_u, e_u]\ s.t.\ s_u = min\left\{s_x \mid s_x \in descendants(u)\right\}, e_u = postorder(u)$, which underlines that GRAIL labels are strongly dependent on the postorder rank computed by a DFS. As explained in the previous chapter, it is not possible to implement a DFS on CUDA, and thus we will devote this chapter to explaining that for DAGs a single DFS graph exploration can be replaced by several BFS traversals, as reported on [5]. This strategy was adopted given that BFS traversals expose a higher degree of data parallelism and, in order to develop it, we tailored the Enterprise BFS to meet each of these BFS's requirements, as we will show in this chapter. Moreover, the authors of [1] suggest that the min-postorder labeling methodology that they propose is not the only possible technique that can be used to build the index and that there exist different labeling strategies. This work conducted a study on the parallelization of the GRAIL algorithm based on the original min-postorder labeling technique, but the algorithms that we developed could constitute the basis for future work regarding distinct indexing approaches, such as labels based on the preorder rank which were already proposed as future work in the original GRAIL study. In particular, we believe that it could be interesting to compare a CUDA implementation of the preorder based labels with our implementation based on postorder labels, since computing the preorder ranks requires a smaller number of BFSs graph traversals, as suggested in [5].

4.2 Replacing DFS with BFS for DAGs

If we analyze the objectives of performing a DFS traversal on a DAG we find that the main reason to do so is to determine a node's unique father, preorder rank (which can be seen as a node's discovery time), postorder rank (a node's finish time) or all of the above. Moreover, the key to substitute the DFS traversal is to realize that *"in a DT finding preorder and postorder of a node is equivalent to computing an offset based on the number of nodes to the left and below yourself"*, as reported on [5]. In other words we will focus on algorithms that rely on the number of nodes that every vertex can reach in order to compute the graph's preorder and postorder sets.

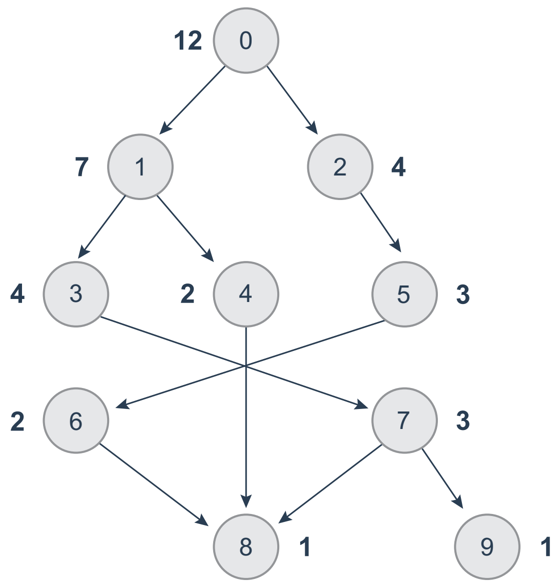


Figure 4.1: Sub graph sizes

We introduce here the following definitions, as they are presented in [5], which are valid for any graph $G = (V, E)$ where E is a set composed of directed edges with no cycles. Let ς_p and ζ_p denote the number of nodes reachable under and including node p: if a certain sub graph is reachable from k multiple parents then its nodes will be counted once in ς_p and k times for ζ_p. It is easy to understand that, as a consequence, $\varsigma_p = \zeta_p$ for any DT and $\varsigma_p \leq \zeta_p$ for any DAG. It is also important to notice that the following recursive relationship can be used to compute ζ_p for any node p: $\zeta_p = 1 + \sum_{i \in C_p} \zeta_i$, where C_p is the set of children of node p. In other words ζ_p is the cardinality of the subgraph of nodes reachable from vertex p, and from now on we will refer to ζ_p with the term *sub graph size.*

Additionally, given a node p, whose set of children C_p has an ordering relationship defined, we define $\tilde{\zeta_l} = \sum_{i<l, i \in C_p} \zeta_i \ \forall \ l \in C_p$ and we notice right away that $\tilde{\zeta_l}$ is the cardinality of the set of vertices that are reached by, or reachable from, all the preceding siblings of node l, excluding l itself since i is less than l. In other words, if node p has a set of children C_p and there is an ordering relationship among these children, $\tilde{\zeta_l}$ indicates how many nodes can be reached from all the children of parent p before l itself. It is also important to notice that for any node p we can compute both $\zeta_p - 1$ and $\tilde{\zeta_l}$ for all children $l \in C_p$ by doing a prefix sum of ζ_p starting from 0, as can be seen in figure 4.2. Generally speaking, these definitions were adopted as they were proposed in [5] but we note that in a DAG nodes may have more than one father, which implies that node l has a $\tilde{\zeta_l}$ for each father, so using $\tilde{\zeta_{l,p}}$ to indicate $\tilde{\zeta_l}$ for father p would result in a more precise notation. In fact, figure 4.2 contains two sub graphs of the graph introduced in figure4.1 and the reader may verify what do the sub graphs sizes represent and how does the CSR format help in the computation of $\tilde{\zeta_l}$: the first sub graph has its root in node 1 and as reported in figure4.1 node 3 has sub graph size 4 since from 3 we can reach 3,7,8 and 9, whereas node 4 has sub graph size of 2 since it reaches 4 and 8. The CSR format is particularly suited to allow the $\tilde{\zeta_l}$ calculation since the sub graphs sizes of a node's children can be stored in the same positions used by the *children_edges* array to store a node's children, and a simple prefix sum on the sub graphs would produce $\tilde{\zeta_l}$ for every node, as can be seen for node 1, which has children 3 and 4. Nodes 3 and 4 have sub graph sizes of 4 and 2, respectively, and a 0-started prefix sum of these values yields $\tilde{\zeta_3} = 0$ since node 3 has no preceding siblings and $\tilde{\zeta_l}$ is the sum of the sub graphs sizes of the preceding siblings, $\tilde{\zeta_4} = 4$ since node 3 has sub graph size of 4, and $\zeta_1 - 1 = 6$, since ζ_p is the sum of all of P's children sub graph sizes plus one. Similarly, the second sub graph analyzed in figure 4.2 shows that if a node has more than one parent, it has more than one $\tilde{\zeta_l}$.

The pseudo-code for algorithm 5 was introduced in the original paper and will be used to compute the sub graph sizes and their prefix sums for every

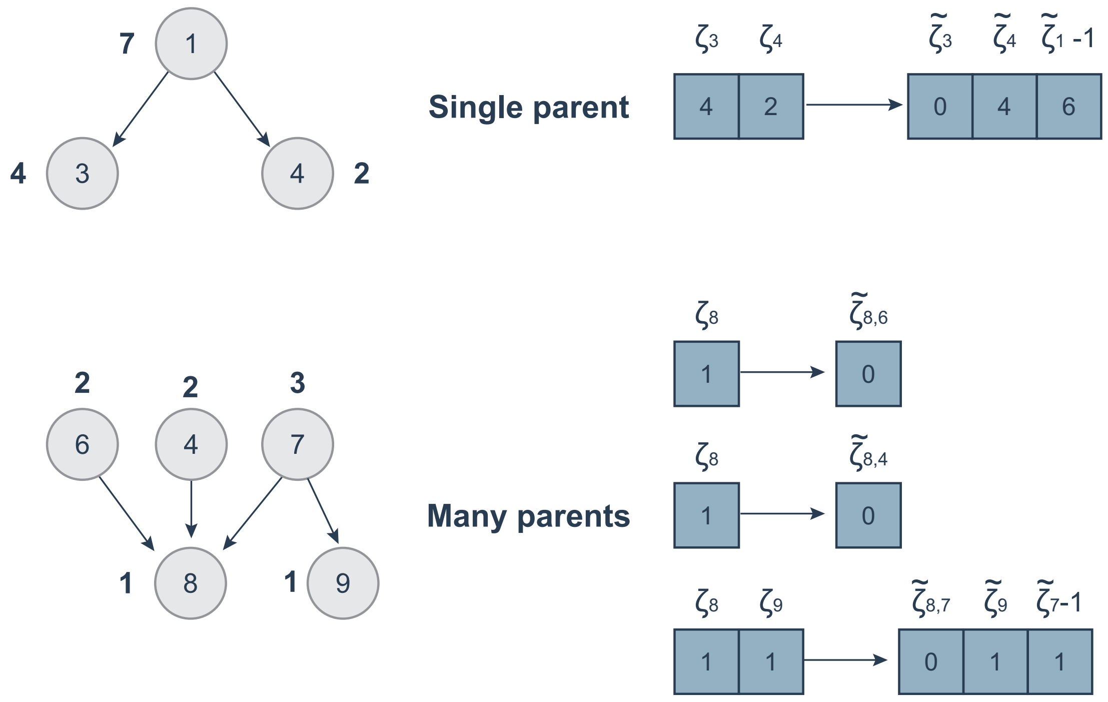


Figure 4.2: Prefix sums

node of a DAG. It basically explores the set of nodes following a bottom up traversal in which every time a node i is encountered the node's sub graph size is propagated to its parent p. At every iteration a node i is extracted from the queue of nodes to visit, and the edge <p,i>, where p is i's parent, is marked as already visited. The algorithm then checks if parent p has been visited by all of its children and, if so, it is inserted on a secondary queue of nodes which will be visited during the next iteration, after a prefix sum on the children sub graph sizes is performed. Notice that, since the algorithm proceeds bottom up it is necessary to wait until a parent has been visited by all of its children in order to compute the prefix sum, given that the children's sub graph sizes would not be available otherwise. As a result, the algorithm produces the sub graph size for every node and the prefix sum of the sub graph sizes for every pair <node i, parent p>.

After having introduced the sub graph sizes and their prefix sums and having proposed an algorithm for computing them on any DAG, it will be showed that for any DT the preorder and postorder can be built using the sub graph sizes ζ_p. In fact, preorder and postorder are entirely dependent

```
Function compute Subgraph Sizes():
    Initialize all subgraph sizes to 0
    Insert all leaves into queue Q
    while Q.NotEmpty() do
        foreach node $i \in Q$ in parallel do
            Let $P_i$ be a set of parents of i
            Initialize queue C as empty
            foreach parent $p \in P_i$ in parallel do
                Mark edge (p, i) as visited;
                if all outgoing edges of p are visited then
                    C.push(p);
                end
            end
        end
        foreach node $i \in C$ in parallel do
            Let $C_p$ be an ordered set of children of p
            Compute prefix sum on $C_p$, obtaining $\zeta_p$
        end
        Q = C;
    end
```

Algorithm 5: Subgraph size computation with bottom up BFS

on the number of nodes that precede the vertex p throughout various depth levels given any topological order. Particularly in a DT there is a unique path that goes from root r to node p, $P_{r,p} = \{r,i_1,\ldots,i_{k-1},\text{p}\}$, where k is the depth of node p, and if we define τ_p as the sum of all the $\tilde{\zeta}_l$ in the path from root r to node p, or $\tau_p = \sum_{l \in P_{r,p}} \tilde{\zeta}_l$ we can compute the preorder and postorder as follows:

$$preorder(p) = k + \tau_p \tag{4.1}$$

$$postorder(p) = (\zeta_p - 1) + \tau_p \tag{4.2}$$

As a matter of fact, we can observe in figure 4.3 that τ_p represents the cardinality of the set of nodes that precede node p at all levels of depth, since it is the sum of all the $\tilde{\zeta}_l$ in the path from root to p and $\tilde{\zeta}_l$ represents the number of nodes that are visited by l's siblings before vertex l is visited. In other words, it is the number of nodes that would be discovered before p in a DFS traversal, which implies that the preorder of p is at least τ_p.

Since $\tilde{\zeta}_l$ does not include the node l in the count it is necessary to include all the nodes in the path $P_{r,p}$, so we add the depth k of node p. Instead a DFS traversal computing the postorder would not process the k nodes on the path from root to p, but would process the sub graph of node p before analyzing p, which explains the $\zeta_p - 1$ factor for the post order formula.

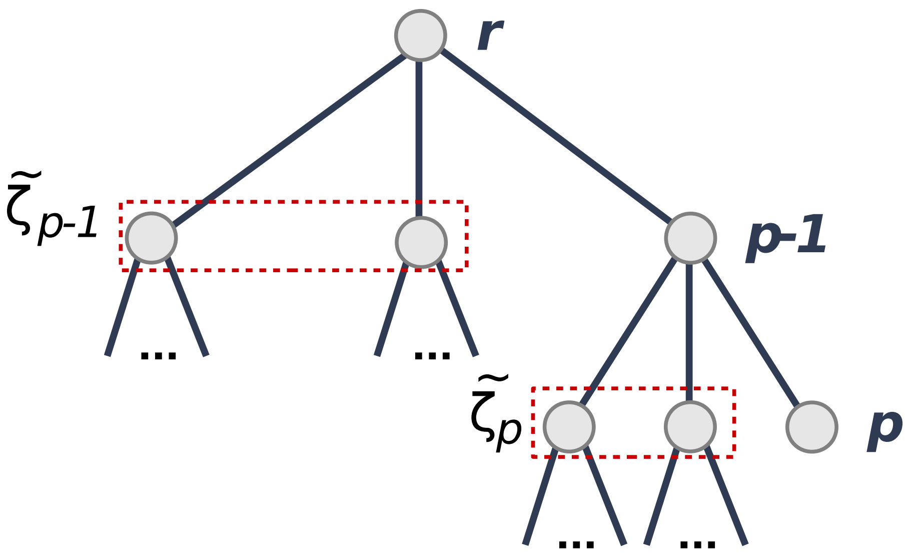


Figure 4.3: Weights based on sub graph size

Algorithm 6 was introduced in [5] and proposes a method for computing preorder and postorder of a DT following a parallel top down BFS traversal of the graph. Essentially, the algorithm has the same structure that algorithm 5 but explores the graph top down rather than bottom up, and so it iterates from root to leaves and processes a vertex p by propagating its τ_p to all of its children i and by marking the edges <p,i> as visited, until all of the children's incoming edges have been marked which causes the children to be added to the next iteration queue. The algorithm is based on the inherent recursive structure given by $\tau_p = \tau_{i_{k-1}} + \tilde{\zeta}_p$, and updates node p's preorder and postorder by adding the depth and the sub graph size after that τ_p has been propagated to all its children.

In summary, we have showed that pre and postorder can be computed on any Direct Tree by performing two separate BFS traversals but, unfortunately, algorithm 6 cannot be used as it is on any DAG because it is possible to encounter nodes that have more than one parent, for which distinct $\tilde{\zeta}$ are

```
Function computePrePostOrder():
    Initialize pre and post orders to 0 for every node
    Insert all roots into queue Q
    while Q.NotEmpty() do
        foreach node $p \in Q$ in parallel do
            Let pre = pre-order(p)
            Let post = post-order(p)
            Let $C_p$ be a set of children of p
            Initialize queue P as empty
            foreach node $i \in C_p$ in parallel do
                Set pre-order(i) = pre + $\tilde{\zeta}_i$;
                Set post-order(i) = post + $\tilde{\zeta}_i$;
                Mark edge (p, i) as visited;
                if all incoming edges of i are visited then
                    P.push(i);
                end
            end
            Set pre-order(p) = pre + depth(p);
            Set post-order(p) = post + $\zeta_p$;
        end
        Q = P;
    end
```

Algorithm 6: Pre and post order computation with top down BFS

defined, so we cannot define a unique τ for the node since distinct paths are possible. In order to extend our procedure to DAGs it will be necessary to assign a unique parent to every node, which will result in a unique path and unique τ for all vertices and we will continue our discussion by explaining our approach to do so.

4.2.1 Single parent computation

Generally speaking there are many methods that can be followed to select a parent for each vertex and transform a DAG into a DT, and the authors of [5] presented two approaches, one based on path comparison and an alternative based on weights comparison. We chose the weight comparison method, but we refer to the aforementioned paper for a complete discussion on the path comparison method's implementation, as well as for an in depth comparison of their performances. Specifically the path based method follows an intuitive approach: it chooses a unique path for every vertex mirroring the same behaviour that a DFS traversal is expected to follow. In fact it iterates over the graph's nodes through a top down BFS and assigns to each children its parent's path, unless the children has already been provided with a path. In that case a node-by-node comparison of the two paths takes place, until a decision point is found and solved by selecting the path with the "smaller" node according to the ordering relationship in the graph. It is also worth noticing that this algorithm is presented within an in depth analysis of the data structures that the authors suggest for representing the path, as well as with several optimization techniques for avoiding full-path representation when possible, which will not be covered here.

$$w_{P_{r,p}} = \sum_{l \in P_{r,p}} \overline{\zeta_l} \tag{4.3}$$

Regardless of the difference with which this is achieved, the weight based method is also based on computing the same path that a DFS with a given ordering relationship would follow. Specifically, what this approach proposes is to constructs a DT from a DAG by performing a Single Source Shortest Path algorithm, or SSSP from now on, in which every vertex has a positive weight that is dependent on its sub graph size, or number of reachable nodes from that vertex. In fact, weights are defined as $\overline{\zeta_l} = \sum_{i<l, i \in C_p} \zeta_i = 1 + \tilde{\zeta_l}$, with C_p set of children of node p and we immediately notice that these weights can be obtained through the same algorithm proposed for computing the sub graph sizes and its prefix sums: as a matter of fact, both $\overline{\zeta_l}$ and ζ_p are the result of the same prefix sum used to compute ζ_p with the difference that it now starts at one instead of zero.

A complete proof of correctness of the SSSP based parent computation can be found in the original paper, but it is useful to notice that, given an ordering relationship, every node will have a greater weight than its preceding siblings since the sub graph sizes of the siblings will contribute to its weight. In fact $w_i > w_{i-1}$ since $\overline{\zeta_i} = \overline{\zeta_{i-1}} + \zeta_{i-1}$ and ζ_{i-1} is at least one. Intuitively this pattern can be applied at every level of the tree to realize that selecting the path with the smaller weight is equivalent to selecting the path that the DFS would follow under the same ordering of vertices. As can be seen in algorithm 7 the SSSP algorithm follows the same structure that algorithm 6: it iterates over the graph in a top down BFS traversal updating a node's parent and current cost by comparing costs computed on the node's different

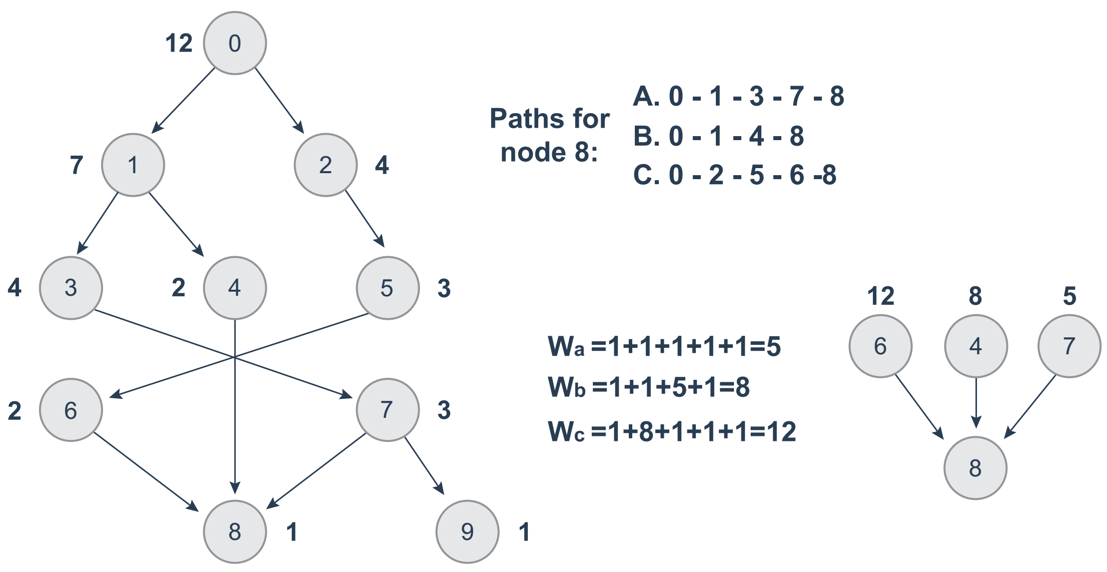


Figure 4.4: Weights for node 8

weights for the distinct parents. In figure 4.4 the reader may notice that node 8 has three different parents, which are 4, 6 and 7, and that each parent is associated to a distinct path from the root. In fact, vertex 8 has three distinct weights that depend on the sub graph sizes of the nodes of each distinct path: in this scenario the SSSP algorithm would assign 7 as node 8's parent since path A has the minimum weight, given that most nodes on this path have no preceding siblings and, as such, coincides with the path followed by a DFS visit.

4.2.2 Final observations

In summary we have introduced a method that consists in performing several BFS traversals for computing the pre and post order ranks that would be produced by a single DFS, and this method is based on computing the cardinality of the sub graphs that are reachable from all vertices. These procedures are valid for any DAG, but require that we transform the graph into a DT by computing a single parent for every node, which is done through a BFS based SSSP algorithm that has been presented in this chapter. In other words, this procedure replaces a single DFS traversal with several BFS visits: one to compute the sub graph sizes of the DAG, which are used to transform the DAG in a DT via a BFS based SSSP. Then one BFS is required to compute the sub graph sizes on the DT, which allows the last BFS to compute

```
Function computeParentBySSSP():
    Initialize cost to $\infty$ and parent to -1 for every node;
    For all roots set cost to 0 and insert them into queue Q
    while Q.NotEmpty() do
        foreach node $p \in Q$ in parallel do
            Let $C_p$ be a set of children of p
            Initialize queue P as empty
            foreach node $i \in C_p$ in parallel do
                Let cost(i) be the current cost for node i
                Let new cost $\alpha = cost(p) + \overline{\zeta_i}$
                if $\alpha < cost(i)$ then
                    Set $cost(i) = \alpha$
                    Set $parent(i) = p$
                end
                Mark incoming edge (p,i) as visited
                if all incoming edges of i are visited then
                    P.push(i);
                end
            end
        end
        Q = P;
    end
```
Algorithm 7: Compute DFS-Parent through SSSP

the post orders used by the GRAIL labels. The main steps are highlighted in algorithm 8, and the implementations proposed for these algorithms will be analyzed in the following sections.

```
Function computeLabels():
    computeSubgraphSizes(DAG)
    computeParentBySSSP()
    computeSubgraphSizes(DT)
    computePrePostOrder()
    computeMinPostOrder()
```
Algorithm 8: GRAIL labels computation through BFSs

4.3 Subgraph sizes computation

In order to compute the sub graph sizes we modified the Enterprise BFS as follows: since we need to process the graph in a bottom up traversal the *parent_edges*, *parent_cardinalities* and *parent_indices* will be used instead of their children counterpart and the initialization phase will consist in setting to one the status array elements of the leaves rather than the roots. This function will also work on two additional global memory arrays, called subgraph_sizes and prefix_sums. The subgraph_sizes array has the same cardinality of the set of nodes, since every node has one unique sub graph size whereas the prefix_sum has one element for every edge to convey that one vertex has a $\tilde{\zeta_l}$ for every parent. More importantly, in order to allow concurrent prefix sum and sub graph size computation, the prefix_sum array is accessed through the children_index vector, similarly to the children_edges array: to access $\widetilde{\zeta_{l,p}}$, which is node l's prefix sum for parent p, two accesses have to be made: children_indices(p) = k and prefix_sum(k) = $\widetilde{\zeta_{l,p}}$. This mechanism ensures that during the bottom up exploration many of node p's children may concurrently write their sub graph sizes in the prefix sum array at the starting point of p, which is indicated by children_indices(p). Subsequently, during the processing phase, the sub graph sizes are accessed by many threads that cooperate in computing the prefix sum of the sub graph sizes of p's children, which is then used to store the parent vertex's sub graph sum. In figure 4.5 the reader can observe that during the queue expansion phase the graph is being traversed bottom up using the parent indices and edges arrays. During this phase, threads processing nodes 8 and 9 will write 8 and 9's sub graph sizes in 7's starting address on the prefix sum array, whereas during the queue processing phase a zero-started prefix sum of 8 and 9 sub graph sizes is performed and used to produce 7's sub graph size according to the algorithm that has been presented in the previous chapters. In fact, $\tilde{\zeta_8} = 0$, since 8 has no preceding siblings, $\tilde{\zeta_9} = 1$ since 8's sub graph size is 1 and $\zeta_7 - 1 = 2$, which is the result of the prefix sum between 8 and 9's sub graph sizes.

Notice that since this is a bottom up traversal all the queues will have to be classified according to the number of fathers that each node has, rather than their children, to limit divergence, as explained in the previous chapter. However this function's performances are also dependent on the number of children that each node has, because it has to compute the prefix sums on every vertex's set of children. In fact, if we classified queues depending

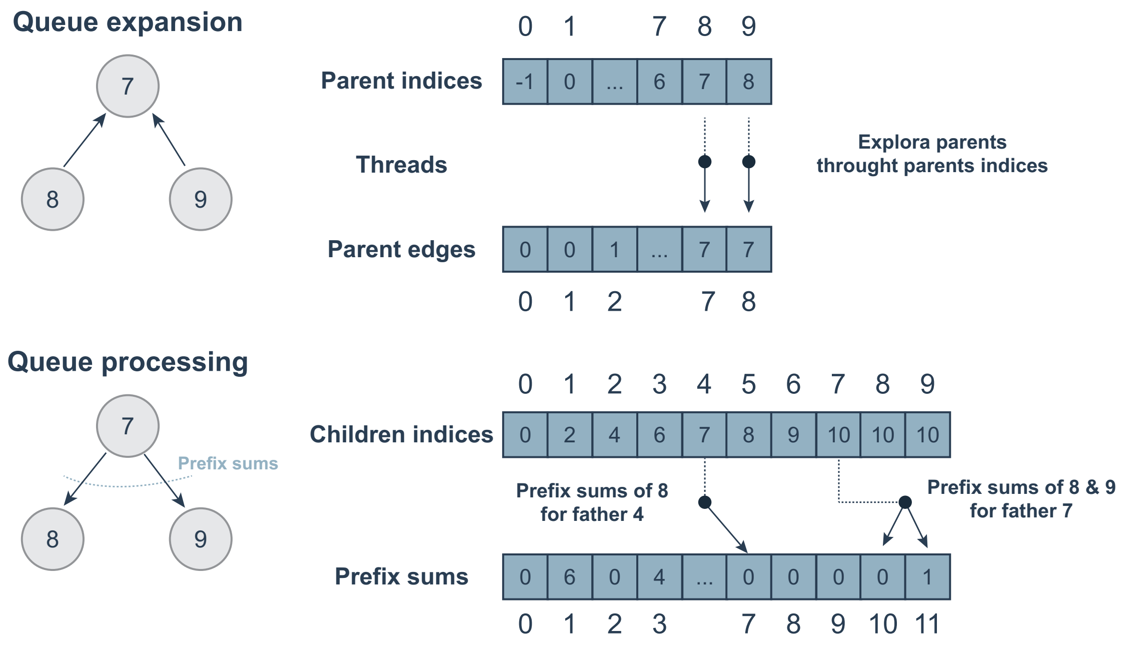


Figure 4.5: Prefix sum computation during bottom up navigation

on the number of parents and then proceeded to do a prefix sum on the node's children we could have high divergence for vertices with one parent and ten thousand children, which would be handled by a single thread. To this purpose we leverage on the separation between queue exploration and expansion and add an intermediate step during which the queue will be regenerated according to the children cardinality to perform both prefix sum and sub graph size calculation. In other words, at every level of the bottom up BFS the same set of nodes will be sorted on children cardinality to process and then on parent cardinality to expand, as showed in 9.

```
Function computeLabels():
    while Q.NotEmpty() do
        sortQueueOnChildrenCard()
        updateSubgraphSizes()
        sortQueueOnParentCard()
        expandQueue()
    end
```

Algorithm 9: GRAIL labels computation through BFSs

Algorithm 9 explains the basic steps of this function, where by sortQueueBy we intend the entire process of generating the queues classified by children or parent cardinality, which is constituted by bin generation, prefix sum on bin sizes and queue generation. If algorithm 9 is observed closely, it can be noted that its steps are mapped to the two phases presented on the pseudo code of algorithm 5: during the queue expansion all of the node's parent are visited and the edges <node i, parent p> are marked, which is represented by *sortQueueOnParentCard* and *expandQueue*, then if a parent has been visited by all of its children it is added to the queue of nodes that are to be processed for prefix sum and sub graph size computation, as seen by *sortQueueOnChildrenCard* and *updateSubgraphSizes*.

In particular, this algorithm requires to track whether a node has already been visited by all of its children and uses a global array to count the number of children that have visited their parents, and to use this array to check if a node has been visited by its children and needs to be expanded during the next iteration. This array is updated through atomic adds, which are expensive operations executed concurrently by many threads, but we expect them to regard different nodes and hence to be on distinct addresses which partially limits the amounts of extra clock cycles required. Furthermore, the sub graph size computation gets calculated by a distinct kernel for every queue: small, medium and large queues are given thread, warp or block granularity according to the out degree of their children. The small queue is processed by a kernel that processes a vertex by iterating sequentially over all of its children's sub graph sizes, computing the prefix sum and storing the final sub graph size before extracting the next node on the small queue, as can be seen in algorithm 10.

Conversely, the medium and large queue are handled by two similar kernels which differ only on the indices used to access the queue of nodes to explore, given that threads collaborate at a warp level for the medium queue and at block level for the large queue, so we will analyze the large queue kernel's process to compute the prefix sum without loss of generality. The large kernel has a shared memory array on which all threads load concurrently a children's sub graph size, synchronize in order to be sure that all the sub graph sizes have been loaded and then cooperate to compute the prefix sum according to the Kogge Stone's algorithm. During this process the kernel computes the prefix sum of *block size* children's sub graph sizes at a time, and performs as many iterations as needed, should the node have more than *block size* children, and then uses the result of the prefix sum to compute and store the node's sub graph size in a global memory array and extracts the next node in the large queue.

```
Function smallQueueProcessing():
    const int queueSize = queueSmallDim;
    const int granularity = gridDim.x * blockDim.x;
    int tid = blockDim.x * blockIdx.x + threadIdx.x;
    int subgraphSize = 1;
    int prefixSum = 0;
    if tid < queueSize then
        currVertex = queueSmall[tid];
        currCard = childrenCards[currVertex];
        currStartPos = childrenIndices[currVertex];
    end
    while tid < queueSize do
        lane = currStartPos;
        currCard += currStartPos;
        currChildren = edges[lane];
        currChildrenSubgraphSize = subgraphSizes[currChildren];
        while (lane < currCard) do
            prevLane = lane++;
            subgraphSize += currChildrenSubgraphSize;
            globalPrefixSums[prevLane] = prefixSum;
            prefixSum += currChildrenSubgraphSize;
            // load next children's sub graph size
        end
        globalSubgraphSizes[currVertex] = subgraphSize;
        // load next vertex
    end
```

Algorithm 10: Simplified kernel for small queue processing

4.4 SSSP based parent assignment

Having explained the first algorithm necessary for computing GRAIL labels, we move onto the implementation of the second BFS, which transforms a DAG into a DT, as explained in the previous sections. The general structure of the algorithm mirrors closely algorithm 9, even though in this case the graph is visited following a top down navigation which implies that the data structures used for the exploration phase are once again the *children_edges*, *children_cardinalities* and *children_indices*. Similarly, at every level of the traversal the same frontier of nodes is generated twice: during the processing phase three queues of nodes are created depending on the node's parent cardinalities, since in this phase is necessary to perform the parent assignment, which requires to find the path with the minimum weight among all of the node's parents. Conversely, during the expansion phase the same queues are generated depending on the children cardinality to propagate a node's weight to all of its children during the top down exploration.

In order to assign a unique parent to each node, the *costs*, *visited_counter* and *candidate_parents* global arrays are allocated and accessed as follows: costs and candidate_parents have one element for every parent edge since all vertices have one cost for every parent, whereas *visited_counter* has the same cardinality that the set of vertices since it is used to count how many of a children's parent have already visited it, which is used to mark edges <parent i, node i> as visited, similarly to what was done in algorithm 7. Essentially, these arrays are used to compare path weights as follows: during the expansion phase we take node i from the queue, load its weight w and process a children c by performing an atomic add on $visited\_counter[c]$, which returns the number of parents that has already visited c, referred to as k, and is used as index for the costs and candidate_parents arrays. In other words, $costs[parent\_indices[c] + k]$ contains the cost of parent i for node c, which is then updated by adding $\tilde{\zeta}_c$ to it, whereas $candidate\_parents[parent\_indices[c] + k] = i$ and contains the parent to whom that cost is associated. Obviously, if k is equal to the number of parents that the vertex has, the node's status array element is set to one in order for it to be visited in the next iteration. Notice that this scheme is based on the CSR format to store the partial path costs and that it allows several threads that are simultaneously expanding parents $p_1, p_2, ...p_n$ to concurrently write their cost for their common children c .

Let us address here the implementation details of the processing phase that decides a node's parent by comparing all of its parent's costs. As for the previous algorithm, there is an identical code structure between large and medium queue processing, so for simplicity we will analyze only the small and large queue with the understanding that these principles also apply

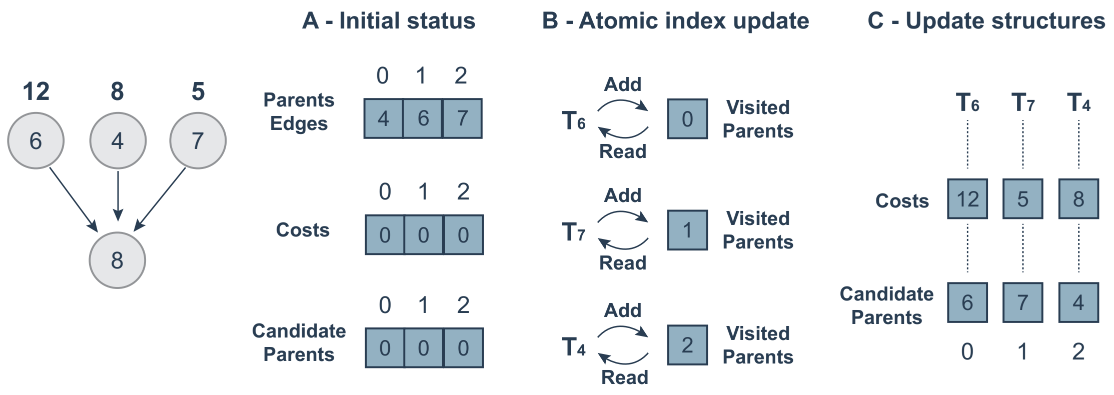


Figure 4.6: Queue expansion during SSSP parent computation algorithm

for the medium queue. Once again the implementation of the small queue kernel is straightforward, given that it simply iterates over a node's parents while keeping track of the path with lower cost and the parent to whom it is associated. The large queue uses a pair of integers which are shared among all threads in the block to compare the minimum costs and to keep track of the parents to which they are associated. The weight comparison is performed by having each thread perform an atomic minimum operation with these shared integers, but is implemented in such a way that only a fixed amount of atomic operations is performed, regardless of how many weights have to be compared. Notice that if the blocksize is fixed at 256 threads and a node with 10000 parents has to be processed, each thread processes 40 weights by keeping track of the minimum weight among these candidates in a private per-thread register, without performing any atomic operations on shared memory. Once each thread has determined the minimum weight, and the parent to whom it is associated, all threads in the block synchronize, determine the final parent through the atomic minimum operations and then synchronize again to allow each thread to check whether its value was chosen as min and, if so, to update the global array of parents and costs.

4.4.1 Labeling phase conclusion

We close this chapter by recalling that, having assigned a parent to every node we require three additional BFS to finish the GRAIL labeling phase: we need to produce the DT's sub graph sizes, to compute the post orders and to assign to each node the minimum of all of its children's post order. These BFSs are all simplified versions of the two BFSs that have already been explained into great detail, where by simplified we intend that they follow the same ideas that have been exposed so far but do not generate multiple times the queue at every level, but rather have a queue generation mechanism similar to the enterprise's, so it will suffice to remark here the main differences.

The first BFS that has to be performed is to recompute sub graph sizes for the DT, which could theoretically be executed by the same function already developed. In practice, however, we implemented a more efficient function that exploits the fact that nodes have a single parent to avoid generating three distinct queues during the exploration phase, but the general data structures and code organization remain the same. Notice also that there is an additional reason that will result in this function running faster than its DAG counterpart, as will be showed when analyzing the running times of these procedures: while the two functions are equivalent they iterate over different amounts of data since the DT requires a single parent to be processed for every node. Furthermore notice that before calling this function it is necessary to re-compute most of the data structures that characterize the graph, and that this re-computation has to be performed on CPU, which inevitably causes overhead in copying data back and forth.

As for the last BFS implementations we will not analyze in depth the post order computation since it is a straightforward adaptation of the BFS that we have used so far and follows algorithm 6, in which the graph is traversed top down and the processing phase consists in loading the current node post order rank and updating each of its children's post order with a sum of the parent's rank and the children's $\tilde{\zeta}_l$. In this context the only difference reported between our implementation and the aforementioned algorithm is that in our case the pre order is not actually needed, and thus its calculation is avoided. Concluding our discussion we move to the last BFS, which basically traverses the graph bottom up and propagates the minimum post order rank among a node's set of children through the use of atomic min operations, adopting techniques to contain the overhead similar to those analyzed when computing the minimum cost for every children in the SSSP BFS implementation.

Chapter 5

Search Implementation on CUDA

5.1 Introduction

As the reader may recall, two CPU based implementations of the GRAIL algorithm were proposed and analyzed in the previous chapters, which we referred to as the sequential and the query parallel versions. These two implementations shared the main algorithms and data structures, but differed mostly in the number of threads that executed the two phases of the algorithm: in fact, the query parallel version computed labels by using one thread for each dimension and solved a large number of reachability queries by assigning them to many threads, whereas the sequential version performed all of these tasks sequentially. However it is useful to notice that, despite the different number of threads devoted to the research phase, both versions compute the answers for N reachability queries by launching sequentially many DFS traversals: as a matter of fact the sequential version's search simply launches N distinct DFS explorations, while the query parallel version uses Q distinct threads, each of which performs N/Q visits sequentially.

Nonetheless one has to consider that this approach does not look as promising when porting the search phase on the GPU, since launching a distinct graph traversal for every query requires to continuously switch between CPU and GPU which entails significant overhead, especially when executing a large number of queries. Moreover, considering the difficulties of implementing an efficient DFS traversal on CUDA and given that the search phase can be based on either BFS or DFS, according to [1], with performances that depend entirely on the graph's topography, it is clear that our CUDA implementation will be based on the Enterprise BFS. Since this algorithm is explicitly designed to explore concurrently all the nodes of the current frontier, it makes sense to avoid performing a traversal for every query and to focus on developing a search phase that conducts several concurrent graph explorations to maximize the number of queries that can be answered in parallel.

However, while the structure of the Enterprise algorithm is particularly suited to enhance concurrent explorations of many nodes during a visit the reader has to notice that is was designed to compute the depth of every node and that during this task a large number of threads visit several nodes at the same time, but these nodes are all logically associated, or belong, to the same traversal. Interestingly, this property remains valid for the Enterprise version developed to compute the GRAIL indices whereas, on the contrary, our query search strategy will solve several queries in parallel by performing distinct traversals concurrently. In general, this implies that a significant number of threads will visit a large number of nodes at the same time, but that these nodes will be logically associated to distinct explorations and these concurrent traversals may overlap at any time, since a node may belong to more than one of the paths explored to solve a query, and it will be necessary to develop a strategy for understanding to which query search is a node exploration associated. In this context, this chapter aims to explain our approach to associating nodes to queries, as well as to analyze how does this impact on the performances of the suggested implementation.

5.2 Bitmask based search implementation

Let us analyze the proposed implementation for the search procedure by introducing the general program's organization, the main data structures and the guiding principles on major design choices before delving into the modified Enterprise BFS and the kernels that implement the actual search phase. The developed implementation completes the index creation procedure by computing the labels on the CPU rather than the on the GPU given that, as showed in the next chapter, the CPU version is several times faster and, as a consequence, most of the data structures for the labeling phase has been reused and coincide with the previously introduced data structures on the CPU side. As the reader will recall, the GRAIL authors suggest that the optimal label dimension is between two and five and for some graphs increasing the number of labels may result in faster search times. However, for this implementation it has been chosen to rely on two-dimensional labels given that every dimension imposes four accesses into global memory when performing the label comparison, given that each comparison regards two nodes and each node has two ranks, inner and outer, for every dimension. It has already been explained why having thousands of threads concurrently request data that does not allow a coalesced access pattern can severely limit performances on the CUDA architecture, but this particular scenario will be studied in greater depth detail when analyzing the queue exploration kernels of the search algorithm. Furthermore, notice that while it makes sense to store the entire set of indices on a multi-dimensional matrix in the CPU side, which allows to access the i-th node's label of dimension j through $labels[i][j]$, it has been separated into j distinct pairs of arrays for the GPU side.

As for the program's structure it follows a straightforward implementation in which, after having read the graph and having computed its labels, a set of random queries in the form of $< source, destination >$ pairs of nodes is generated. As soon as the query generation process is completed, the labels are transformed in the aforementioned pairs of arrays, all the data regarding labels and queries is transferred to the GPU, and the GPU starts processing queries in groups of k queries at a time. In fact, k represents the number of concurrent BFS searches, has a significant impact on the search phase's performances, and is currently set at 64 as showed in the following section. Before explaining the rationale of answering for this choice, it is necessary to address the previously mentioned issue of having overlapping sets of nodes that are visited by many BFSs simultaneously and to explain how does this affect performances.

When processing a query $< s_1, n_1 >$ the algorithm starts a BFS on the sub tree with root s_1 and uses labels to prune the set of nodes that have to be visited in order to discover whether n_1 is reachable or not. In this context each BFS exploration can be seen as a set of nodes such as $[s_1, a_1, a_2, ..., a_q]$ where a_q may correspond to n_1, which are visited concurrently according to the various levels of depth on which every node resides. Let us recall, however, that the Enterprise BFS tracks the set of nodes belonging to the current level through a boolean status array which only determines whether each node has

```
Function gpuConcurrentSearch():
    batch = 0;
    totBatch = (QueryNum / SearchDim);
    // with SearchDim = 64
    while $batch \neq totBatch$ do
        cacheDestinationLabels();
        initStatusArray();
        while $Q.NotEmpty()$ do
            generateBins(statusArray);
            prefixSums(bins);
            generateQueue(bins);
            queueExploration(Q);
        end
        prefixSum(results);
        batch++;
    end
```
Algorithm 11: Main loop of the GPU based search phase

to be explored during this iteration or not, and assume that another of the k concurrent BFS searches also happens to visit node a_2 at the same level of depth that the first query. Since at every iteration a test of equality between the searched node n_1 and the currently explored node a_i is performed, it is clear that it is necessary to implement a mechanism to identify the possibly many queries to which node a_i is associated at every level and that the status array mechanism, as it is, cannot provide this functionality.

In order to associate a node with more than one traversal we propose to use the status array's elements as bit masks rather than as boolean values, and to rely on bitwise operators to both set and test to which searches is a node associated to. In other words, the set of queries is divided in groups of sixty-four queries which are processed one at a time, and within the group queries are identified through an index comprised between 1 and 64. Therefore, during the bin generation phase of the Enterprise BFS the status array is scanned and a node is inserted into a bin if its value is different than zero, whereas in the expansion phase the status array's value is modified through a bitwise atomic *or* operation. This function, exposed in the CUDA API as *atomicOr(address, value)*, allows to set the i-th bit of node n to represent that node n has to be explored during the next iteration and that its exploration

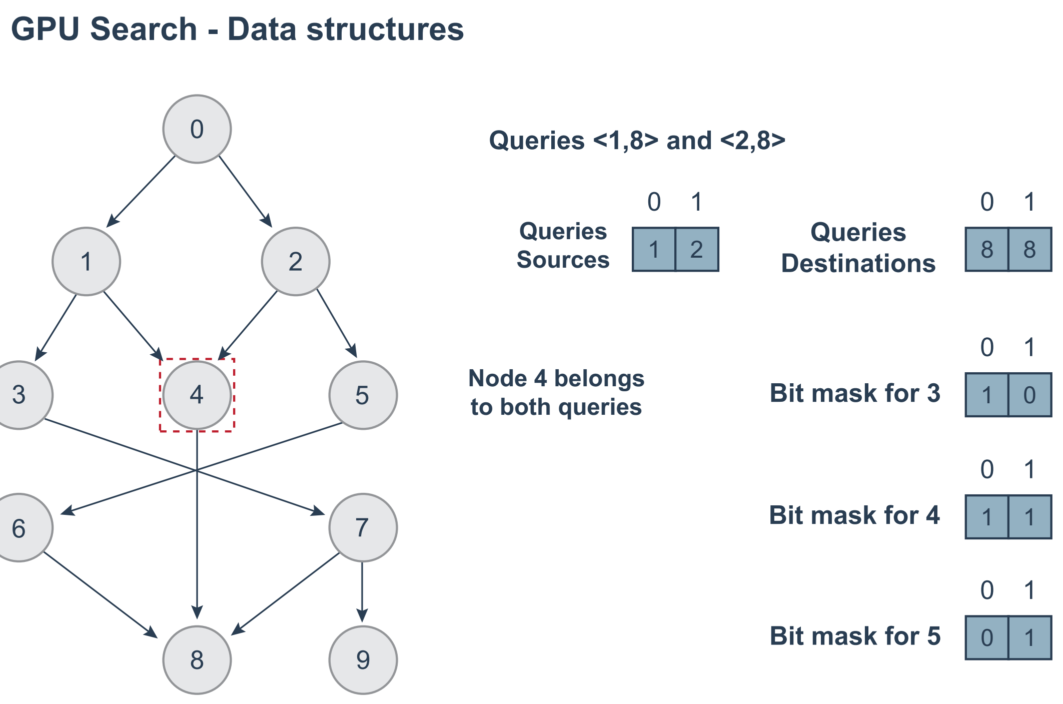


Figure 5.1: Overlapping set of nodes during search phase

is associated with the i-th search. Similarly to the atomic operations in the prefix sum computation of the CUDA labeling phase, the atomicity of the operations will require additional clock cycles for these memory accesses, but these overheads are expected to have a limited impact on the overall performances given that most of them will not be executed on the same addresses and therefore will not require serialization.

Before analyzing the updates to the Enterprise BFS it is useful to notice that this solution cannot be implemented using only the status array: during each iteration it is necessary to reset the values of the status array in order to avoid visiting nodes that belong to the previous level and the Enterprise BFS does this at the end of the bin generation phase. Therefore the status array is paired with a similar array, namely the *search_indices* array, that is used to transfer each node's bit mask to the queue exploration phase in order for that kernel to associate the node to the distinct queries to which it belongs, which is done by extracting the positions of all the set bits and using these positions as modulo-64 indices for the queries.

Besides the additional data structures the primary modifications to the Enterprise algorithm can be found in the main loop and in the exploration kernels, as can be seen in algorithm 11. In the main loop an array with the labels of the destination nodes of the sixty-four queries is created and passed to the kernels in order to allow a coalesced access of these data structures. Furthermore, given the high number of expected label comparisons, this array will be accessed continuously and it is important to limit the number of global accesses related to these requests, so at the beginning of the kernel execution all threads cooperatively load the labels of the searched nodes from this global memory array into a shared memory cache. As the reader may notice, at the end of every sixty-four queries batch an additional prefix sum is executed on the *results* array. This array resides in main memory and has sixty-four elements which are used to store whether a given search resulted in a positive outcome, so this final prefix sum provides the number of reachable queries in the recently processed group.

As for the exploration kernels, this discussion will be conducted in broad terms since the only significant difference between the functions that explore the small, medium or large queue resides in the amount of threads assigned to each node: as for the standard Enterprise algorithm, each node's children are processed by a thread, a warp or a block depending on the queue to which the node belongs. As already explained, at the beginning of the execution a per block cache that contains the labels of the sixty-four destination nodes is created, referred to as *label_dim* in algorithm 12. Notice that this array is loaded cooperatively even for the small queue exploration kernel, in which children processing is implemented without inter-block thread cooperation, given that these labels will be frequently accessed by every thread regardless of whether they are collaborating on a query or not. As soon as the labels are loaded, the kernel has to load node p, extract from p's bit mask the indices of the n queries to which it is associated, load the destination nodes of these queries and test for equality between p's children and the destination nodes in order to determine whether the destination node was found or not. If the destination node coincides with one of p's children, the query index is used to store this positive outcome on the global result array, otherwise a label comparison between the destination node and children c is executed to decide whether c has to be inserted into the next iteration queue. Notice that each thread accesses a node's *status flag* from the search_indices array which, as already explained, is a copy of the status array's values on that iteration. The status flag contains the indices of the destination vertices to which a

```
Function queueExplorationKernel():
    SearchDim = 64
    granularity = $gridDim * blockDim$;
    _shared_ label_dim_1[SearchDim];
    _shared_ label_dim_2[SearchDim];
    if (threadId < SearchDim) then
        label_dim_1[threadId] = destination_labels_1[threadId];
        label_dim_2[threadId] = destination_labels_2[threadId];
    end
    _syncthreads();
    while (threadId < queueSize) do
        vertex = queueSmall[threadId];
        foreach (children c $\in$ children[vertex]) do
            load labels of children c;
            statusFlag = research_indices[vertex];
            while (statusFlag != 0) do
                searchIndex = extractLSBit(statusFlag)-1;
                searchedNode = destinationNodes[searchIndex +
                  (batch*SearchDim)];
                if (searchedNode = c) then
                    results[searchIndex] = 1;
                end
                if (destinationLabels $\in$ childrenLabels) then
                    atomicOr(statusArray + c, 1ULL « research_index);
                end
                statusFlag &=  (1ULL « search_index);
            end
        end
        threadId += granularity;
    end
```

Algorithm 12: Small queue exploration kernel

node's exploration is associated and is manipulated through CUDA bitwise functions, such as the *___ffsll()* function that returns the position of a long's least significant bit, and has been reported in algorithm 12 *extractLSBit()* as for comprehensibility.

Finally it is important to address a noteworthy modification to the search algorithm regarding the amount of queries that are assigned to the GPU for resolution. As explained, the proposed search structure imposes to answer reachability queries in groups of sixty-four queries, which entails significant overhead when solving a high number of queries. Moreover, it is clear that this overhead cannot be entirely eliminated given that each group of queries requires, at the very least, a data structure initialization phase and a final prefix-sum operation on the results array. Nonetheless, during the testing phase of the CPU based algorithms, we noticed that the vast majority of the queries that yielded a negative result had been immediately discarded at the first label comparison, which can be used to obtain a consequential speedup on the GPU based version of the search process. In fact, as subsequently showed, the GRAIL algorithm discards between 75 and 90%, depending on the graph's density, of the queries during the first label comparison and, given the small percentage of the queries that require further analysis, the search algorithm was modified in order to avoid assigning GPU resources to those for which an answer can be provided with a single label comparison. In conclusion, the GPU search algorithm was modified in order to generate a fixed number of queries, which are instantly tested for reachability in order to decide whether to assign their resolution to the GPU and this strategy resulted in a speedup of an order of magnitude compared to indistinctly analyzing all queries on the GPU.

Chapter 6

Results

In this chapter have been reported experimental results of the GPU based labeling and search algorithms which are then followed by a comparison with their CPU based counterparts. As the reader will recall, the adopted benchmark divides its graphs into three classes according to their size and density, and our analysis will emphasize which topographic characteristics have a greater impact on the algorithm's performances. Additionally, this discussion addresses several aspects of GRAIL that are inherently challenging to parallelize following a data parallel approach, as well as the major drawbacks of our implementation. We propose distinct parameters for tests regarding the labeling and search phases, and our results have been computed as averages over twenty runs and are reported here as structured tables.

6.1 GPU Labeling

As the reader will recall, the labeling algorithm has been designed as a sequence of BFS explorations that generate the sub graph sizes for the DAG, assign a unique parent to each node and compute the sub graph sizes on the DT, which are then used to compute the labels via two additional traversals. Considering that these sub algorithms have different weights on the final running time our metrics for analyzing the labeling phase's results include the times required for each sub procedure, and a significant part of this analysis is dedicated towards studying how do these weights vary depending on the graph's size and density. Furthermore, our GPU algorithms were carefully designed to avoid constant calls to memory allocation and transfer functions in order to maximize continuous GPU execution time, and we exploited the fact that each algorithm's input is the output of the preceding one to pre-allocate and keep most of the data structures in the GPU's global memory until no further use was required. However, given the limited size of the GPU's global memory, which is 4 GB, and the dimension of the larger graphs that we analyzed, it was not possible to pre allocate all the data structures beforehand and several structures were freed, reallocated and initialized only when needed. This factor, along with the time required to transfer the data structures to and from the CPU in order to recompute the DT edges added an overhead to the total time that was also measured and analyzed. In the subsequent tables the reader may observe the times of each sub procedure, which are referred to as follows: *Subsizes DAG* and *DT* represent the time required to compute sub graph sizes for DAG and DT respectively, *ParentSSSP* regards the time required by the unique parent computation procedure, *Edges RC* stands for edges recomputation, and measures the time taken to create the data structures regarding the DT representation, *Post-Order* shows the time required to compute the post order in the computePrePostOrders procedure, whereas the *Min P.O.* column refer to the time taken by the computeMinPostOrder algorithm to compute the minimum post order for every vertex, which defines a node's actual labels. Additionally, the *overhead* time was defined as the difference between the total execution time and the sum of all the other procedure's computation times, and includes the time required to allocate intermediate data structures and to transfer data between CPU and GPU, which has a significant impact on the total labeling computation time. Moreover, we included the *CPU ICT* column, which stands for CPU Index Construction Time and reports the time required by the CPU to create a one-dimensional label, in order to make an accurate comparison with the GPU labeling procedure, which also creates a one-dimensional label.

Table 6.1: Small sparse labeling times (all times in ms)

Data set	Subsizes DAG	Parent SSSP	Edges RC	Subsizes DT	Post-order	Min P.O.	Overhead	Total	CPU ICT
Agrocyc	59.2	64.9	0.2	50.9	38.5	30.0	41.5	285.5	0.14
Amaze	58.8	62.3	0.09	47.4	36.1	28.4	41.9	275	0.82
Anthra	61.6	67.4	0.2	53.3	39.5	31.9	41.9	295.8	0.18
Ecoo	80.5	84.8	0.2	54.0	41.1	39.5	41.2	341	0.21
Human	66.1	74.4	0.75	50.2	41.1	35.2	51.3	319.1	0.48
Kegg	94.0	97.2	0.1	63.8	45.3	48.6	45.5	394.5	0.09
Mtbrv	79.2	82.4	0.18	67.4	47.2	38.8	41.0	356.1	0.11
Nasa	103.1	111.8	0.2	72.6	48.5	50.3	40.0	426.7	0.16
Vchocyc	73.4	76.0	0.1	49.6	38.2	35.2	40.5	313.1	0.14
Xmark	112.0	119.0	0.1	81.9	53.6	56.2	40.4	463.4	0.14

As reported in table 6.1 the labeling algorithm completes the indexing process for most of the sparse graphs in times that range from 270 to 450 milliseconds, and spends almost half of that time in the first two phases, which are the DAG sub graph sizes and the parent computation. Additionally, given the graph's size and low density degree, the edge recomputation is always under one millisecond and amounts to less than the 1% of the total execution time, which is also reflected on the ratio between the times for computing the DT and DAG sub graph sizes, which is greater than 80-85% and suggests that the two algorithms require similar times since few edges have been discarded. Moreover, the reader can notice that for small graphs the overhead constitutes up to 16% of the execution time for some graphs, such as the Human, and in most cases is greater than the time required by the CPU to complete the labeling procedure.

Table 6.2: Small dense labeling times (all times in ms)

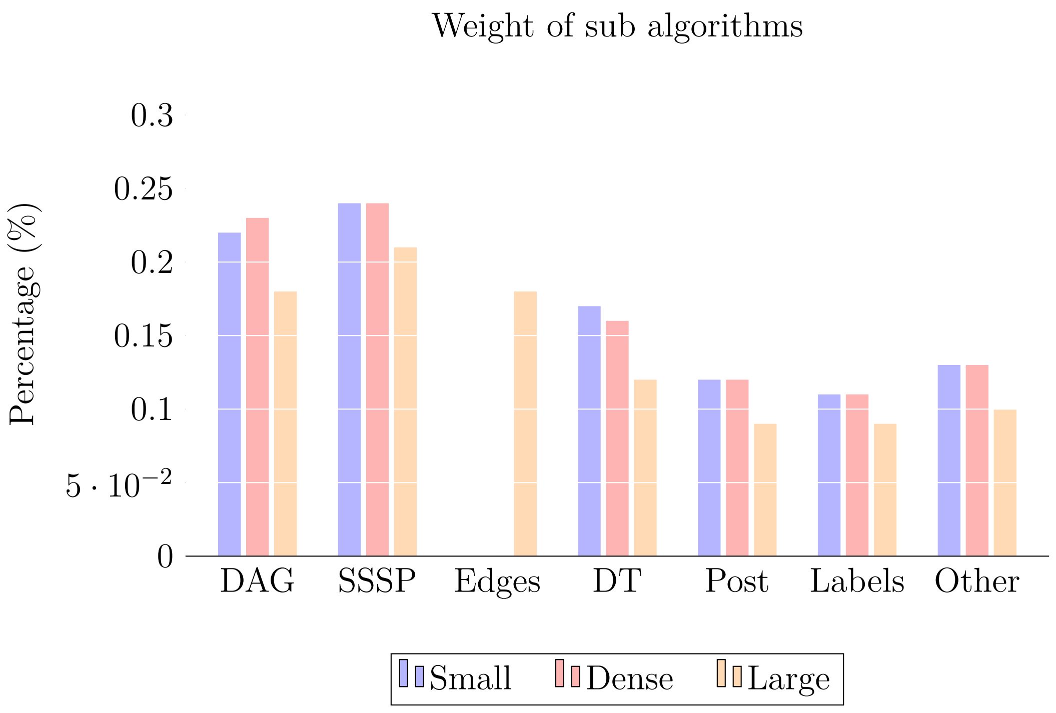
Data set	Subsizes DAG	Parent SSSP	Edges RC	Subsizes DT	Post-order	Min P.O.	Overhead	Total	CPU ICT
Citeseer	108.9	110.3	0.7	51.6	38.8	53.8	41.6	405.7	0.61
Go	65.4	70.1	0.4	60.8	44.1	31.3	41.8	313.8	0.24
Pubmed	68.8	72.8	0.5	38.7	32.2	32.9	42.2	288.2	0.47
Yago	54.9	59.7	0.4	50.9	37.8	26.9	42.6	273.4	0.39

Interestingly, the results for the small dense set outlined in table 6.2 show that most of the observed parameters resemble closely times obtained for the small sparse data set, which is consistent with the results obtained for the CPU based implementation of the indexing algorithm. In fact, the size of the graphs in the dense data set still determines that the total time is within the 270-400 milliseconds range, that the edge recomputation times never amount to more than 1% of the total time and that the overhead still amounts to 16% of the total execution time for some instances. Similarly, the algorithm spends from 40 to 50% of the time in the first two phases and the BFS that computes the minimum post order, which is referred to as *Min P.O.* in the table since it actually completes the label creation process, falls in the same range, 10 to 13%, than in the small sparse data set. The only significant difference regards the ratio between times for DT and DAG sub graph sizes computation for the *citeseer* and *pubmed* instances, which drops to 50%, signaling that a large number of edges has been discarded in these cases. However, given that both of these graphs represent citation data sets it is reasonable to assume that this contrast depends on the particular topography rather than on either size or density.

Table 6.3: Large real labeling times (all times in ms)

Data set	Subsizes DAG	Parent SSSP	Edges RC	Subsizes DT	Post-order	Min P.O.	Overhead	Total	CPU ICT
Citeseer	61.5	66.8	23.3	45.8	34.0	28.8	58.3	318.5	21.92
cit-Patents	270.3	362.4	551.2	133.7	96.3	192.9	87.2	1694.1	698.42
uniprotenc_22	60.5	60.6	34.4	45.5	37.0	26.4	65.7	330.2	514.31
uniprotenc_100	479.9	525.9	490.4	270.1	234.9	229.3	147.4	2378.7	846.26
uniprotenc_150	730.5	836.9	726.9	421.9	361.9	356.2	176.6	3611.1	44.18

On the contrary, the analysis of the large real graphs yields contrasting results on several parameters, reported in table 6.3, and requires a further sub classification given that there are considerable differences in the data set, both in topology and dimension, that have considerable impact on times and behaviour of the algorithm. In fact, the data set includes the *citeseer* graph, which is significantly smaller, the Uniprot family of graphs, which range from relatively large to huge and have a distinct structure composed by many roots converging to a single sink through a short path, and the *cit-Patents*, which is a large dense graph with no similarities in this data set. Generally speaking, the indexing times range from 300 milliseconds for the smaller graphs in the data set, such as *citeseer* and *Uniprotenc_22*, to 3600 milliseconds for the *Uniprotenc_150*. As the reader may see in 6.1, which contains the averages of the weights for every data set, the first observation regards the weight of the edge recomputation procedure, which goes from being negligible in the small data sets to being even the most expensive phase of the procedure for the *cit-Patents*, for which it constitutes over one third of the total time. Furthermore, while the weights of the first two phases decrease from almost 50% to only 36% of the time for *uniprotenc_22* and *cit-Patents*, the algorithm still spends considerable time on the DAG sub sizes computation and on the SSSP. In fact, in the larger instances of the large real data set these two phases and the edge recomputation procedure require two thirds of the total running time. Conversely, in these instances the weights of all the subsequent procedures decrease and constitute one third of the time when combined, whereas the weight of the overhead which amounts to only 5% of the total execution time. This can be explained by considering the large amount of edges that are discarded in the first procedures, as confirmed by a similar decrease in the DT to DAG sub sizes computation ratio, which is around 50% whereas in the previous sets was over 85-90%.



As a final note it is useful to reflect on two of the major disadvantages of the suggested approach for designing a data parallel version of the indexing procedure, which are the interdependence among phases and the CPU based redefinition of the data structures regarding the DT graph. As already reported, the developed approach is inherently sequential in the sense that each phase produces the data structures required by the following sub procedure, which inevitably prevents CPU and GPU from concurrently working on different parts of the algorithm.

6.2 GPU Search

As previously explained, the search procedure performs an on CPU preliminary filtering of all queries, which consists in a single label comparison that decides whether the query is negative or whether it is necessary to solve it on the GPU, if the first comparison was inconclusive. Considering this initial CPU side screening, the metrics for measuring the search algorithm's performances include the time spent to perform the first label comparison of every query on CPU, the time spent analyzing queries on GPU, which are referred to as CPU filter and GPU Search, the number of queries for which reachability was determined, also addressed as Positive Queries, and the number of queries that were assigned to the GPU for testing reachability, defined as Queries on GPU, which represent the amount of queries that were not answered by the CPU side screening. Additionally, the search times of both the DFS and BFS versions of the sequential CPU algorithms, defined as *Seq DFS* and *Seq BFS*, were conveniently reported in all tables.

Table 6.4: Small sparse search times (all times in ms)

Data set	CPU Filter	GPU Search	Queries on GPU	Positive Queries	Sequential DFS	Sequential BFS
Agrocyc	2.3	23.8	255.0	97.5	6.10	5.96
Amaze	2.5	773.0	22008.8	17235.0	26.02	33.96
Anthra	2.3	21.5	222.8	91.9	6.26	6.30
Ecoo	2.3	68.5	1157.0	121.0	6.37	6.51
Human	2.6	16.1	61.3	11.6	6.54	6.71
Kegg	2.7	1347.2	26178.9	20158.0	28.08	59.90
Mtbrv	2.5	42.3	498.7	154.7	5.99	6.13
Nasa	2.5	221.5	2343.8	543.1	6.47	7.45
Vchocyc	2.3	32.1	357.0	154.7	5.97	6.02
Xmark	2.3	684.8	7430.2	1460.9	10.21	33.13

As the reader may observe in table 6.4 answering 100k queries following the dual CPU-GPU approach requires less than 100 milliseconds for most of the instances with the exception of the *Amaze* and *Kegg* data sets, both of which have a unique structure with a central node that has a high number of incoming and outgoing edges and requires the GPU to handle almost 20% of the queries, most of which are effectively reachable and have need of a full exploration. However, the comparison between these results and the times obtained by both CPU versions allows to conclude that the GPU implementation is significantly outperformed by both of its CPU counterparts, as can be seen by noting that in the most favorable instances the GPU implementation is in the same order of magnitude than the pure DFS approach.

Table 6.5: Small dense search times (all times in ms)

Data set	CPU Filter	GPU Search	Queries on GPU	Positive Queries	Sequential DFS	Sequential BFS
Citeseer	2.3	1559.3	17765.9	372.3	20.10	28.72
Go	2.4	324.0	7334.1	244.6	13.11	15.13
Pubmed	2.6	757.6	13556.7	650.8	11.20	13.60
Yago	2.7	756.5	16919.7	166.6	7.51	8.96

Continuing with the results for the small dense set of graphs, the reader may notice in table 6.5 that even though only a small fraction of the tested queries actually resulted in reachable pairs, a considerable amount of queries had to be assigned to the GPU for a full exploration, which resulted in query resolution times within the 300-700 milliseconds range for all the graphs in this data set, with the exception of the *Citeseer*, for which the algorithm required more than 1500 milliseconds. As for the small sparse data set, the comparison with the CPU versions confirms that the GPU search is consistently outperformed by the CPU GRAIL implementation, whereas it is significantly faster than the pure DFS approach for the *Pubmed* and *Citeseer* graphs, which require two to ten times more, respectively.

Table 6.6: Large real search times (all times in ms)

Data set	CPU Filter	GPU Search	Queries on GPU	Positive Queries	Sequential DFS	Sequential BFS
Citeseer	8.5	221.4	7324.0	0.2	13.24	12.96
cit-Patents	12.1	4376.0	19112.1	40.6	1831.41	3423
uniprotenc_22	14.9	5709.4	26511.7	0.0	24.57	23.74
uniprotenc_100	15.4	8806.8	25030.7	0.0	24.96	25.20
uniprotenc_150	12.4	1418.2	48582.5	0.2	20.50	20.70

For the large real set of graphs a significant amount of pairs have to be explored by the GPU and only a negligible percentage of them regards nodes that reach each other, as the reader may observe in table 6.6. As an example, for the uniprotenc_150 graph almost 50% of the queries are analyzed by the GPU, but in average none of them contains reachable pairs. In this context, it can be noted that the CPU versions continue to outperform the GPU implementation when running on the *Uniprotenc* family of graphs, presumably because of their particular topography, which is characterized by short paths that connect many roots to a single sink and allows an efficient DFS exploration. However, it is important to observe that for the *cit-Patents* graph the gap between the performances of the CPU sequential DFS implementation and its GPU counterpart has significantly reduced, and that the times are within the same order of magnitude. Furthermore, the GPU implementation is slightly faster than the CPU sequential BFS implementation and significantly outperforms the pure DFS exploration.

Chapter 7

Conclusions and future work

The main objective of this work was to design a data parallel version of the GRAIL algorithm and to develop it on the CUDA architecture in order to compare its performances with the sequential and task parallel versions implemented on a traditional processor. Experimental results for both the labeling and the search procedure show that the sequential and task parallel implementations performed decisively better than our data parallel version for almost all the graphs in our data set, independently of their size or density. As previously discussed, the obtained results allows us to conclude that both of the CPU implementations are to be preferred when answering reachability queries on graphs with small sets of nodes and edges. Interestingly, while in several tests on the largest graphs in our benchmark the data-parallel version is outperformed by the task-parallel versions, there are several facts worth discussing before concluding that these implementations are to be preferred when indexing or testing for reachability in large graphs extracted from a real data set.

In particular, results for the data parallel labeling procedure on the largest graphs of our data set highlight that index creation times are within the same order of magnitude than those of the task parallel implementations. In this context, we believe that the main limitation of our work regarding the labeling procedure lies in the amount of large graphs tested and that it would be useful to test the behaviour of the data parallel algorithm on a set of larger and more variegated graphs, given that our large graph benchmark is composed by few graphs, most of which characterized by an unusual topography that favors sequential DFS, and that our most promising results were obtained for the cit-Patent graph, which is unique within the large set. Furthermore, future work regarding the labeling algorithm could explore an alternative approach for dividing a DFS into several BFS that was proposed in [5], which could allow a limited amount of concurrent CPU and GPU execution during the final phases. In fact, the authors prove that the order of the nodes based on the weights used during the SSSP unique parent computation coincide with the node's pre order and suggest that one can also compute the post order given the pre order and each node's depth. In order to exploit concurrently CPU and GPU, a future implementation may test whether having the GPU perform a top down BFS to compute each node's depth while the CPU is sorting nodes based on their SSSP weights leads to a significant improvement for the labeling procedure.

As for the search algorithm, our tests confirm that the data parallel version is consistently outperformed by all the task parallel versions and results advocate that the maximum number of parallel searches that can be concurrently performed is the main cause for its limited performances when compared to the CPU based implementations. In fact, we developed an alternative search algorithm based on a 32-bit bit mask and we compared it with the proposed implementation, which is based on a 64-bit bit mask, and results suggest that there is a linear dependence between the required time and the amount of queries executed in parallel. Given that the current CUDA API does not support atomic operations on any data type defined over more than 64 bits, we believe that future work should be based on abandoning the bit mask array to explore distinct methodologies for increasing the amount of parallel queries answered at each run, and on testing these improvements on larger and more heterogeneous graphs, given that the same considerations regarding the large graph data set are valid for the search procedure.

Bibliography

[1] H. Yildirim, V. Chaoji, M.J. Zaki,*GRAIL: a scalable index for reachability queries in very large graphs*, The VLDB Journal (2012).

[2] E. Cohen, E. Halperin, H. Kaplan, U. Zwick*Reachability and distance queries via 2-hop labels*, SIAM Journal of Computing (2003).

[3] R. Schenkel, . Theobald, G. Weikum *HOPI: an efficient connection index for complex XML document collections*, EBDT (2004).

[4] H. Wang, H. He, J. Yang,P. Yu and J.X. Yu *Dual labeling: answering graph reachability queries in constant time*, ICDE (2006).

[5] M. Naumov, A. Vrielink, M. Garland,*Parallel Depth-First Search for Directed Acyclic Graphs*, Nvidia Technical Report NVR-2017-001.

[6] H. Liu, H.H. Huang,*Enterprise: breadth-first graph traversal on GPUs*, George Washington University

[7] D.B. Kirk, W.W. Hwu,*Programming Massively Parallel Processors, A Hands-on Approach*, Third Edition, Morgan Kaufman

[8] M. Springer, *Breadth-First Search in CUDA*, Tokyo Institute of Technology

[9] NVIDIA CUDA, NVIDIA CUDA C Programming Guide, version 4.2

[10] P. Harish, P.J. Narayanan,*Accelerating large graph algorithms on the GPU using CUDA*, International Conference on High-Performance Computing 2007