BREADTH-FIRST SEARCH FOR SOCIAL NETWORK GRAPHS ON HETEROGENOUS PLATFORMS

BY

LUIS CARLOS MARIA REMIS

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2016

Urbana, Illinois

Adviser:

Professor Maria Garzaran

Abstract

Breadth-First Search (BFS) is the core of many graph analysis algorithms, and it is useful in many problems including social network, computer network analysis, and data organization, but, due to its irregular behavior, its parallel implementation is very challenging. There are several approaches that implement efficient algorithms for BFS in multicore architectures and in Graphics Processors, but an efficient implementation of BFS for heterogeneous systems is even more complicated, as the task of distributing the work among the main cores and the accelerators becomes a big challenge.

As part of this work, we have assessed different heterogenous shared-memory architectures (from high-end processors to embedded mobile processors, both composed by a multi-core CPU and an integrated GPU) and implemented different approaches to perform BFS. This work introduces three heterogeneous approaches for BFS: Selective, Concurrent, and Async. Contributions of this work includes both the analysis of BFS performance on Heterogenous platforms, as well as in depth analysis of social network graphs and its implications on the BFS algorithm.

The results show that BFS is very input dependent, and that the structure of the graph is one of the prime factors to analyze in order to develop good and scalable algorithms. The results also show that heterogenous platforms can provide acceleration to even irregular algorithms, reaching speed-ups of 2.2x in the best case. It is also shown how the different system configurations and capabilities impact the performance and how the shared-memory system can reach bandwidth limitations that prevent performance improvements despite having higher utilization of the resources.

Acknowledgments

This project would not have been possible without the support of many people. Many thanks to my adviser, Maria Garzaran, who read my numerous revisions and helped make some sense of the confusion. Thanks to the University of Illinois Graduate College for providing me with the financial means to complete this program. And finally, thanks to my parents, and numerous friends who endured this long process with me, always offering support and love.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

CPU	Central Processing Unit.
GPU	Graphics Processing Unit.
GPGPU	General Purpose Graphics Processing Unit.
TEPS	Traversed Edges Per Second.
MTEPS	Millions of Traversed Edges Per Second.

Chapter 1

Introduction

1.1 Problem Description

Graphs have a very important role in representing many practical problems, such as social networks, computer networks, and data organization. Breadth-First Search (BFS) is the core of many graph analysis algorithms, and is used in many problems such as flow network analysis, shortest-path problem, and also in others graph traversal algorithms.

However, due to its irregular behavior, the parallel implementation of BFS is very challenging. When processing large graphs, it becomes necessary to deal with large data structures and lack of memory locality. In the particular case of BFS, the random locality prevents memory access optimizations of any kind, and the complexity is always O(N + E), where N is the number of vertices and E is the number of edges. From this observation, it is easy to deduct that the execution time will not only depend on the number of vertices in the graph, but also in the number of edges, and thus, in its structure.

The use of heterogeneous systems to speed up applications has become very common and has been proved to be very effective especially on data parallel algorithms with regular memory accesses [1, 2]. The use of GPUs to compute irregular algorithms, such as BFS, is very tempting because of its high throughput, but it makes the task of parallelizing it efficiently very hard.

One of the main reasons why this platforms can help to accelerate applications is the use of shared memory. In a system where a discrete GPU is present, the work flow involves transferring data from main memory to GPU memory, perform calculation in the GPU, and move the results back to main memory. This memory movement is done through the PCI express bus, which is orders of magnitude slower than making copies within the main memory. If we consider that the number of problems that can be efficiently fitted in a discrete GPU is small, the number of problems that are big enough to justify all these data movement is even smaller. What it is more, the programmer is involved in all this process of moving the data around the system. GPU architects and library developers are creating mechanisms to make all this process transparent to the programmer, and more efficient in some cases, but data movement at some point is inevitable.

Heterogenous platforms, on the other hand, offer an environment of shared memory between the CPU and the accelerators (in this case, an OpenCL capable GPU), where all the data movement can be avoided, as the memory space is shared between the CPU and the accelerator. The OpenCL specification offers the programmer the necessary APIs to both create different allocated memory blocks for both devices, as well as the allocation of a memory blocks that can be addressed from both devices. Section 4.2.1 will cover the specific details about this use cases.

The goal of this work is to implement Breadth-First Search in a heterogeneous system composed of CPUs and GPUs working cooperatively, and also achieve portability by the use of OpenCL. A performance evaluation of the implementation in two type of platforms, namely high-end and embedded has been done. Through the use of heterogeneous platforms, the deficiencies caused by the limited available parallelism can be overcome and better performance can be achieved potencially. The use of an hybrid system can bring the best of both worlds helping to achieve better results to this (and potentially others) graph application.

As part of this work, several heterogeneous approaches were implemented, using three basic approaches: one that dynamically selects the best device to run, another that uses both devices concurrently but synchronically, and the last one that uses both devices concurrently and asynchronically. For achieving this, two well-known benchmarks for GPU algorithms were used as a start point, as well as previous work related to this algorithm in both heterogeneous environment and multiple CPU versions.

A strategy for work division was designed for the BFS algorithm, and bottlenecks were identified for all the platforms. A complete analysis on how the input structure will impact in the algorithm design was performed, which lead to important conclusions. In-depth analysis was performed over social network graphs and its behavior when traversing them using BFS.

All the algorithms were implemented in C++, OpenMP, and OpenCL, portable for all the platforms we used[3], including MacOS and two different Linux distributions.

1.2 Previous Work

Several research has been made over the BFS algorithm, but just few of them were aimed to run in a heterogeneous platforms. There are algorithms specifically implemented for GPU, such as the Rodinia [4] and the Parboil [5] benchmarks, but, to the best of our knowledge, there are only a couple of implementations using both CPU and GPU for traversing the graph.

One of them is a Task-Based approach [6], whose best performing implementation only uses one of the devices at the time, depending on the number of vertices to be processed. This is because their results show that when the number of vertices to process in the new frontier is relative small, the CPU will perform better than the GPU. When the number of vertices is bigger than certain threshold, the GPU will outperform the CPU.

Hong et al. [7] is the other attempt to integrate CPUs and GPUs in a hybrid environment by choosing dynamically between three different implementations for each level of the BFS algorithm, but again, sending work to CPU or GPU, and not using both at the same time.

In both cases, the environment where the high-end GPU is connected to the system via PCIe bus, the communication cost may be too high to make it worth it. The aim of this work is to come up with algorithms that are more suitable for shared memory platforms where this communication cost are less harmful to the overall execution.

Beamer at al. [8] introduce a novel approach for performing BFS which introduces speed-ups compareds to regular implementations, but again, it is aimed to multi-core environment. Also, this work introduces valuable insight about graph processing, which was useful to perform analysis over social network graphs.

1.2.1 Rodinia Benchmark

The Rodinia Benchmark has a 2-step approach for processing each new discover frontier. This 2-step process is done until there is no more vertices in the new frontier, i.e. all vertices have been discovered and processed. The first step consist in going over all the vertices in the new frontier, discover their child, update its cost, and update a mask on each child, which will serve as input for the second step. The second step goes over each marked vertex in the first step, mark the vertex as visited, and add the vertex to the new frontier. The new frontier will now go through this process again.

The same idea is used for both the CPU and the GPU implementation in this benchmark. In the GPU, there is one thread for each vertex in the graph, for each step, and for each iteration. We believe that here there is big room for optimization since most of the threads in this case will not do any useful work, but will generate instead divergence inside the SM of the GPU, affecting the performance. By properly

giving directions to the GPU to launch fewer threads and by generating adequate masks, we can reduce the overhead of launching threads that will not do any work at all.

1.2.2 Parboil Benchmark

The Parboil benchmark has an OpenCL GPU version for the BFS problem [5]. Their implementation focus in the use of BFS for Automatic Design Automation (EDA). Each step of the BFS has two types of parallelism: one is to propagate from all the frontier vertices in parallel, and the other is to search every neighbor of a frontier vertex in parallel. Since a lot of EDA problems are usually formulated on sparse graphs and do not have many neighbors for each frontier vertex, the Parboil implementation of BFS only explores the first type of parallelism, where each thread is dedicated to one frontier vertex of the current level.

The Parboil implementation uses atomic operations to change values in the graph when propagating all the frontier vertices in parallel. The atomic operations used are *atom_min*, *atom_or*, *atom_xchg*, and *atom_add*. These atomic operations are implementation dependent and easy to use by the programmer. However, if many threads execute them over the same memory location, it will likely become a bottleneck and harm the application performance on the GPU [9].

1.3 The BFS Algorithm

Given an undirected graph $G(V, E)$ and a source vertex s_0, a breadth-first search will traverse G starting from s_0 exploring all the vertices at a given distance d from s_0 before traversing further vertices. A vertex frontier is the subset of vertices that are discovered for the first time and are enqueued in each iteration. The output will be an array of distances for every node with respect to s_0. BFS is shown in Algorithm 1.

Algorithm 1 Breadth-First Search

Input: Graph (V,E), source vertex s_0
Output: Distances from s_0 $Dist[1..|V|]$

```
for all v ∈ V do
  dist[v] = ∞
end for
dist[s_0] = 0
Q = ∅
Q.enqueue(s_0)
while Q ≠ ∅ do
  i = Q.dequeue()
  for each neighbor v of i do
    if dist[v] == ∞ then
      dist[v] = dist[i] + 1
      Q.enqueue(v)
    end if
  end for
end while
```

This algorithm allows the programer to exploit two types of parallelism: fine grain and coarse grain.

Fine grain parallelism can be exploited as each frontier is formed by a number of vertices that can be explored in parallel, and its values updated as well. When computing BFS on a graph in which each edge has an associated weight, the parallel operation of updating the distance of each vertex must be performed using atomic operations as a vertex can be reached by two or more different vertices in the frontier, each with different distances as edges are different. In this case, we need to update the distance of that vertex only if the value is smaller than the current one. For simplicity, the weight of each edge is not taken into account, meaning that a data race in this case will be benign, and will not modify the final result (all the updates for a particular vertex will be an update of the same value). Fine grain parallelism can then be better exploited by the GPU, as a thread per vertex in the frontier (or in the graph) can be launched, updating the different vertices in parallel.

On the other hand, coarse grain parallelism can be exploited by splitting the frontier in a smaller number of parts, and processing in parallel each of this parts. In this case, the use of different CPU cores for each part are more suitable than GPUs. Well-know libraries for parallel computing in CPU cores can be used,

such as OpenMP.

Algorithm 1 shows the most basic algorithm to compute BFS for a graph. There are more efficient variant of this algorithm that permits the use of parallelism using the approaches described above.

1.3.1 Processing a Frontier - Top-Down and Bottom-Up methods

A central component can be extracted from the BFS algorithm, which is the processing of a frontier, updating the distance of the vertices discovered, and its addition to the next frontier.

This part of the algorithm can be computed in many different ways, each of which having its advantages and disadvantages. It results practical to recognize two main methods to carry out this task, which are the Top-Down algorithm and the Bottom-Up algorithm.

The Top-Down algorithm is the most straightforward and intuitive way to perform the frontier exploration. For each vertex in the frontier, every neighbor is checked to determine if it has already been visited, and if not, its cost is updated and the neighbor is added to the new frontier. In this case, for every vertex in the frontier, all its neighbors will be visited and checked, which may result in redundant work as some neighbors may have already been visited, or will be visited by more than one vertex in the current frontier. As mentioned in 1.3, in terms of correctness there will be no limitation when having more than one vertex in the frontier visiting the same node, but in terms of performance, this can represent a waste of computing power.

The Bottom-Up algorithm, on the other hand, propose an exploration of the vertices that have not been explored yet in search of neighbors belonging to the frontier. This is, every vertex that has not been visited at that point in the graph traversing checked all their neighbors until it finds one that belong to the frontier, or do not find such neighbor at all. In the event of a neighbor belonging to the frontier, the distance for the vertex will be updated (with the value of its neighbor in the frontier plus 1), and will add itself to the new frontier. This method is less intuitive but lead to a correct result. The advantage of this method lay on the fact that once a vertex found a neighbor that belongs to the frontier, it does not need to keep exploring the rest of its neighbors. Such optimistic situation will only happen when an important part of the graph have been processed already.

An extensive exploration about which method is more convenient to use has been done [8]. The Top-Down method has the advantage that only the vertices in the frontier will be explored, whereas the Bottom-Up method needs to explore every single vertex and check whether it has been visited or not. However, as mentioned before, the Top-Down method needs to check all the neighbors of the vertices in the frontier, and the Bottom-Up only needs to check neighbors until it finds one that belongs to the frontier. Because of

this, using the Bottom-Up method in the first iterations of the graph traversal will result in exploring the majority of the vertices (as only a few have been visited already) and in the exploration of almost all the neighbors (as the frontier would be small, and thus, a vertex belonging to the frontier will be hard to find). But the Top-Down method during the first iterations will result in less redundant work, as only the vertices in the frontier will be explored, and the majority of their neighbors will be unvisited at that point. As the iterations go on, and most of the graph has already been visited, it is more likely that the Bottom-Up method will perform less work, whereas the Top-Down method will perform a significant amount of redundant work. This analysis, thus, is subject to the characteristics of the input graph, as will be discussed later in this work.

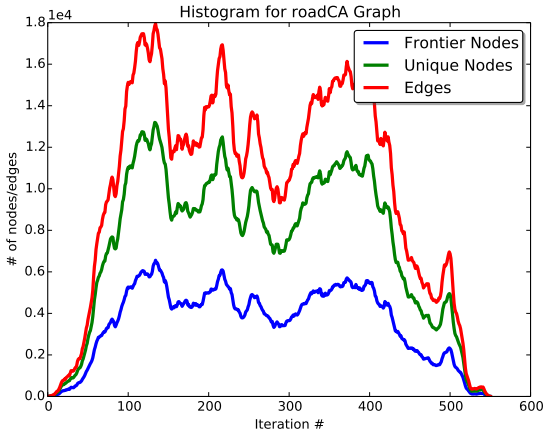
Chapter 2

Input Characterization

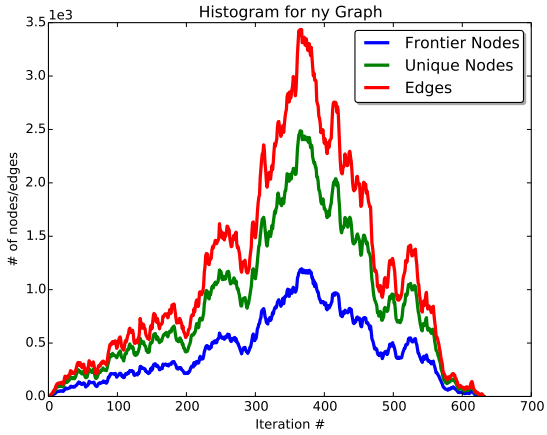
Graphs can have different shapes and characteristics. The number of vertices and the number of edges are just two of the many dimensions that can characterize a graph. Two graphs with similar number of vertices and edges but with a different distribution of the edges can be notoriously different in terms of computing its BFS. This is because graphs can represent problems of completely different natures. A typical example for using a graph as the data structure representation is a map of a city, where each intersection of 2 or more streets represents a vertex and each street itself represents an edge. A city that has a regular pattern of streets will mean that most of the vertices will have roughly the same amount of edges, and that for each vertex, around 4 edges reaching that vertex is expected. On the other hand, another typical real problem representation using graphs can be a social network, such us Facebook, Google+, or Linkedin, where each vertex correspond to a different person, and each edge represents a connection (friendship or professional relation) between to people. In this case, one can expect that a particular person is highly connected in the graph, i.e., this particular person has several connections. It can also be expected that, as a particular person is not very active in the network, or do not have many connections, he or she would be poorly connected. It is clear then in the latter case we will find a very irregular structure, with some vertices having several connections while some others having an small number of connections.

For this work, well-know sources of graphs were used, such as the SNAP collection [10], the Graph500 benchmark graphs generators[11], and synthetic generated graphs from Rodinia Benchmark[4]. These sets of inputs provide a wide range of different graphs, including scale-free and non-scale-free graphs.

After working with early implementations of the BFS algorithms, full profile of the graphs that were to be processed were computed. This helped to develop a better idea about possible bottlenecks, as well as the ability to recognize how different graph structures can present a limitation in terms of performance for Heterogeneous platforms. An application to create full profile for every graph was implemented as part of this work.



(a) Roads of California: 4M Vertices



(b) New York Map Graph: 271K Vertices

Figure 2.1: Regular Graphs: Maps of New York and California

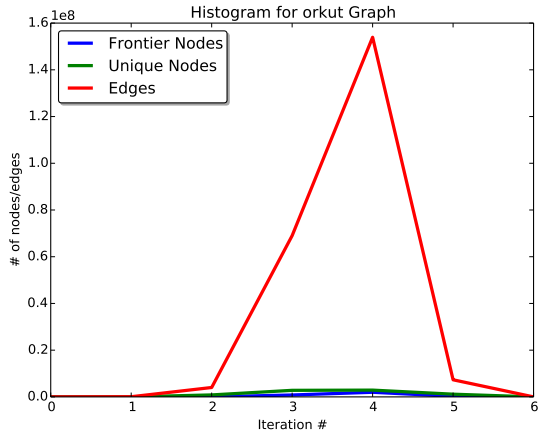
2.1 Social Network Graphs

Social Network graph started to gain relevance during the last decade due to the proliferation of Social Network services. The analysis of social networks graph becomes an important building block for the business model of several of these companies. Social network graphs have interesting characteristics when it comes to its distribution and shape. There is a notorious difference in how the BFS algorithm will perform when traversing social network graph in comparison with more regular graphs.

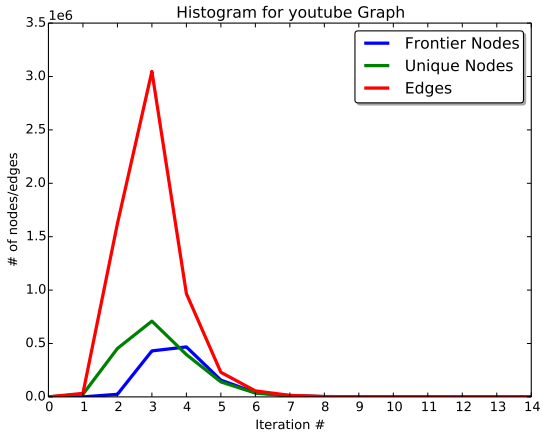
BFS is an iterative algorithm. After each iteration, a new portion of the graph will be discovered and distance values will be updated. The traversing of the graph will take as many iterations as it is necesary in order to reach every node in the graph. In a graph where all edges have a distance of one, the number of iterations will be equal to the diameter of the graph (i.e., the longest shortest path between two vertices).

In the case of very regular graphs (i.e. a graph representing a street map, where every vertex has roughly the same number of edges), several iterations will be needed in order to reach every vertex in the graph. On the other hand, social network graphs (also referred as scale-free graphs in some specific occasions) will only need around 15 or less iterations before reaching every vertex. This behavior can be explained by following the theory of six degree of separation, or more formally, an experimental study of the small-world problem, which demonstrated empirically how two people are connected in a social graph after connecting, on average, 5.2 other people [12]. One can then expect that after a small number of iterations, all the vertices in the graph will be reached. It is important to note that, however, due to incompleteness in real graphs (as not every person in the world is present in every social network), more iterations than 5 or 6 are needed in order

to reach every vertex.



(a) Orkut Graph: 3M Vertices



(b) Youtube Graph: 1.17M Vertices

Figure 2.2: Irregular Graphs: Social Networks

The need for fewer iterations when traversing the entire graph brings new opportunities to platforms that are very sensitive to synchronization barriers (implicit or explicit), such as GPUs, and thus, Heterogenous platforms my find its way as accelerators in this case. Having fewer iterations means two things when it comes to traversing a graph: fewer barriers, and relatively high amount of work per iteration. We can imagine a graph of 1 million vertices and two scenarios: a very regular graph and a social network graph. In the first case, the traversing of the graph will take several iterations, but each iteration will perform comparable amount of work. In the latter case, only a few iterations will be needed, but each iteration will entitle considerable more work than in the first case. What it is more, some iterations will only process a small number of vertices, whereas some other iteration will perform work over frontiers that may take up to 50% of the total amount of vertices. This means a very irregular work flow, as the amount of work performed by each iteration will not be comparable.

Figure 2.1 shows the regularity of traversing a graph in the case where every vertex has a similar, small number of edges. Several iterations are necessary to completely traverse the graph, and all the iterations perform a comparable amount of work. The blue line represents the number of vertices in each frontier, which is comparable across iterations. The same happens with the red line, representing the number of edges.

Figure 2.2, on the other hand, depicts the scenario on social networks. In this case, both the Orkut and Youtube Social networks are completely traversed within 6 and 14 iterations respectively. It can also be seen how there are some iterations (3 and 4 in the Orkut graph and 2 to 4 in the Youtube graph) where

Abbreviation	Description	# Vertices(M)	# Edges(M)	Source
google	Web graph from Google	0.875	5.105	SNAP
youtube	Youtube online social network	1.13	2.98	SNAP
amazon	Amazon product network	0.33	0.92	SNAP
roadCA	Road network of California	1.96	2.76	SNAP
graph1M	Rodinia Generated	1.04	6.3	Rodinia
graph2M	Rodinia Generated	2.09	12.58	Rodinia
graph4M	Rodinia Generated	4.19	25.16	Rodinia
graph8M	Rodinia Generated	8.38	50.34	Rodinia
graph16M	Rodinia Generated	16.77	100.62	Rodinia
NY	Syntetic Graph: New York Roads	0.27	0.74	Parboil
SF	Syntetic Graph: San Francisco Roads	1.25	7.23	Parboil
rmat	Graph500 Rmat	2.13	81.16	Graph500
kron	Graph500 Kroneker	1.75	41.10	Graph500

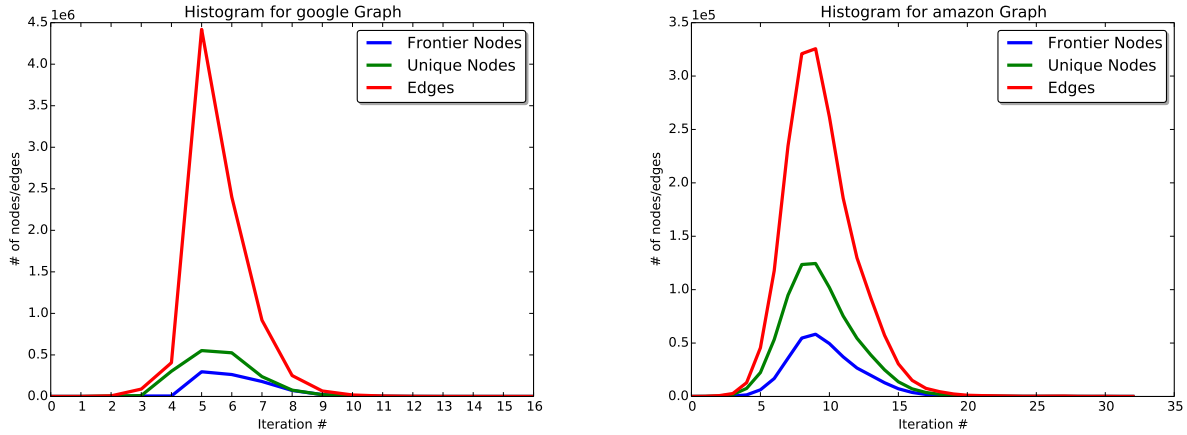
Table 2.1: List of Graphs

the amount of edges explored is one order of magnitude higher than in the rest of the iterations. This presents both an advantage and challenge for Heterogenous platforms, as the number of iterations (and thus, synchronization) is small, but the load is not evenly distributed among the iterations.

Graphs generated by the Rodinia Generator are fully traversed in less than 15 iterations, even though they have very different number of nodes (2M, 4M, and 8M). Besides, in only a small part of the iterations most of these graphs are traversed (from 7th to 12th iteration). On the other hand, both NY and SF graphs are traversed in a very different way, taking around 600 and 1000 iterations respectively. It is important to notice that NY graphs have only 271K vertices, and takes several iterations to complete compared to the Rodinia Generated 8M graph. As mentioned before, more iterations will result in more points for synchronization, which will affect the performance on implementations that have considerable overhead at the end of each iteration.

2.2 Redundant Work Analysis

When implementing parallel versions of BFS for processing a frontier, some inefficiencies can result from the effort of designing fully parallel algorithms. Consider the discovery of vertices in the frontier. For each vertex in the frontier, it is necessary to explore all the neighbors in the case of the Top-Down method. This means that vertices that have already been discovered will be explored again. This vertices could have been visited under two conditions: either they were discovered on a previous iteration, or either they were discovered on the current iteration by some other vertex that is in the frontier. In any case, there is redundant work involved. As part of this work, an analysis of the amount of redundant work was performed.

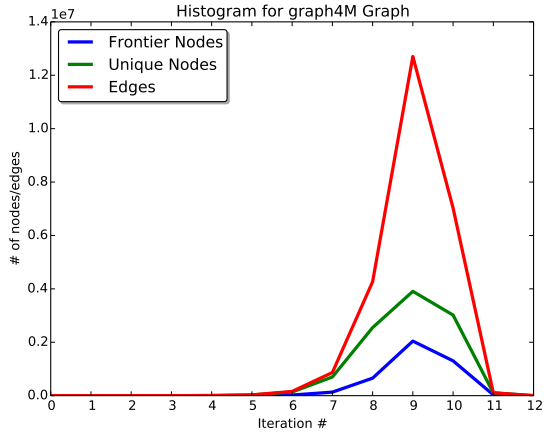


(a) Google Graph: 875K Vertices

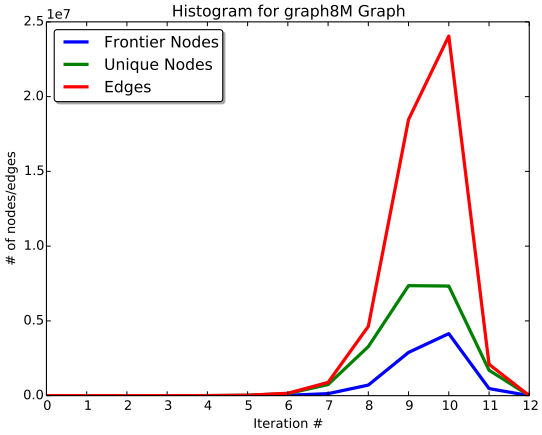
(b) Amazon Graph: 224K Vertices

Figure 2.3: Redundant work analysis on BFS

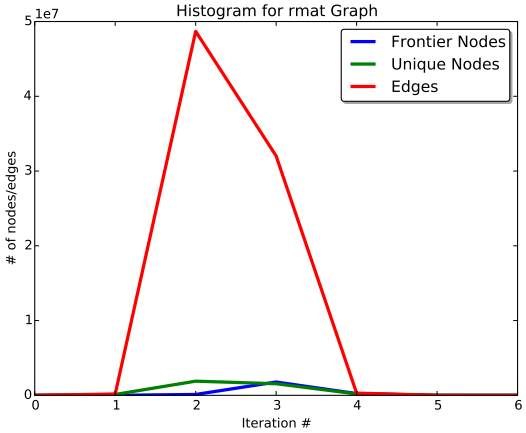
Figure 2.3 depicts this phenomenon. The red line represents the number of edges that will be explored during each frontier, whereas the red line represents only those edges that will result in the discovery of an unvisited vertex. This behavior affects social network graphs more than regular graphs. In the case of the Google Graph, there is a difference of one order of magnitude between the explored edges versus the edges that represent new discoveries. In the case of regular graph, this number of edges explored and new discoveries is comparable, as it can be interpreted from Figure 2.1. This analysis would help improve the prediction whether when it is convenient to use the Bottom-Up method. The Bottom-Up method only needs one of the neighbors of a vertex being processed to belong to the frontier. Once this occurrence happen, there is no need to keep exploring other neighbors, thus, reducing the amount of redundant work performed. For this particular part of the algorithm, an heuristic proposed by Beamer at al [8] is used.



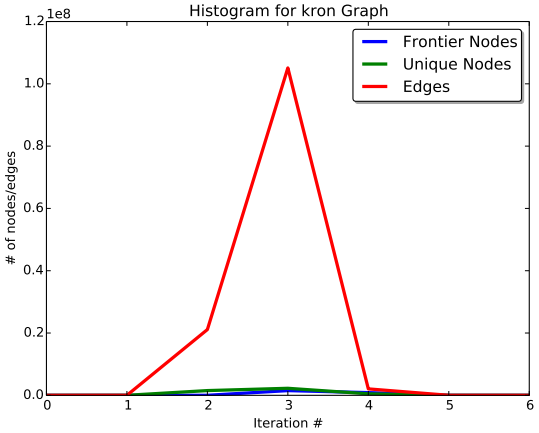
(a) Rodinia Generated: 4M Vertices



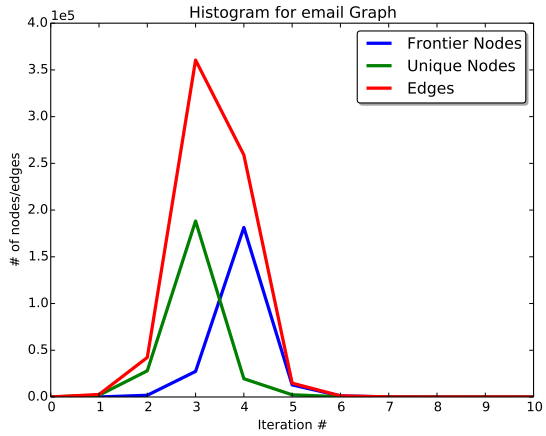
(b) Rodinia Generated: 8M Vertices



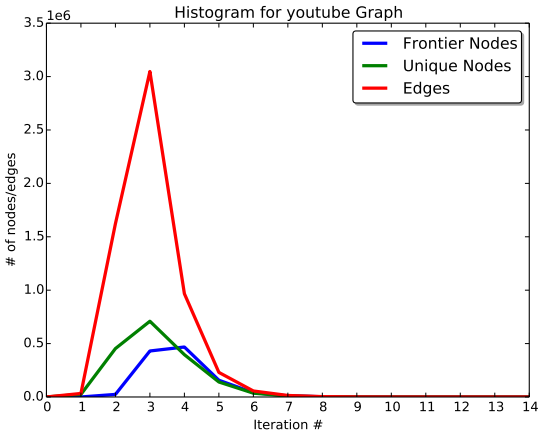
(c) Graph500 Rmat: 2.13M Vertices



(d) Graph500 Kronecker: 1.75M Vertices



(e) Email Graph: 265K Vertices



(f) Youtube Graph: 1.17M Vertices

Figure 2.4: Social network graphs

Chapter 3

BFS Approaches

As part of this work, three approaches for performing BFS were designed and implemented. The first approach was the proof of concept of the idea of using both devices (CPU and GPU) to computer BFS by selecting the best device to run each iteration, based on the size of the frontier. This approach is referred as *Selective*. The second approach is called *Concurrent*. In this case, both devices are used at the same time in a specific iteration. The frontier is splitted between the devices and they traverse the graph concurrently. Lastly, a third approach, called *Asynchronous*, was developed in order to exploit fundamental characteristics associated with scale-free graphs, as described in section 5.3.

The initial version of the *Selective* approach was based on the Rodinia and Parboil Benchmarks implementations for CPU and GPU, in C and OpenCL.

3.1 Selective

The strategy for the *Selective* approach is to select between CPU and GPU based on the size of the new frontier. The idea is to dynamically decide where to process each iteration by using a function with the new frontier as input. For instance, one heuristic implemented for this function is to use a threshold to decide the device to be used based on the number of vertices in the new frontier. If the number of vertices in the new frontier is below the threshold, the frontier is processed in the CPU, otherwise, it is processed in the GPU. In other words, the frontier is to be executed in the device which fits best. This approach has been proposed previously by Munguia et al [6], but from the perspective of a task-based framework using discrete GPUs. The *Selective* approach is based on the observation that CPU can process a given frontier faster than the GPU when the number of vertices in the frontier is smaller than certain threshold. When the frontier is big enough, then the processing will be faster in the GPU, and thus, all the work will be moved to the GPU. It is important to note that the threshold mentioned above will depende on several factors, including the platform (what is the ratio between TEPS on CPU and GPU), as well as the approach used for calculating the new frontier and updating the values. In some cases, the use of the TD algorithm will be faster than

the BU algorithm in CPU, and sometimes the opposite will happen in the GPU.

This approach can be implemented using different combinations of method (i.e. Top-Down and Bottom-Up methods): The firsts iterations can be explored by using Top-Down method, then, when the frontier is bigger than a threshold, process some iterations in the GPU using Top-Down or Bottom-Up method, and during the final iterations the Bottom-Up method can be used to complete the exploration of the graph. The threshold value, as well as the method to be used in each iteration will be further detailed when discussing the experimental evaluation.

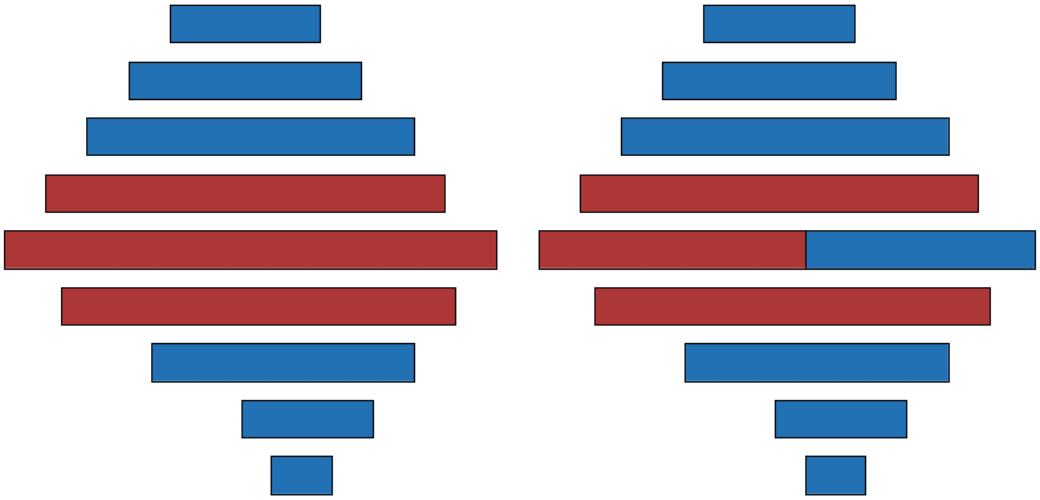


Figure 3.1: Selective approach (left) and Concurrent approach (right). The red part represents part of the frontier executed on the GPU and the blue part represents part of the frontier executed on the CPU cores

3.2 Concurrent

The second approach is called *Concurrent*. In this approach both devices are used at the same time in the iterations where the frontier is bigger than a threshold. The frontier is splitted between the devices and each device traverses the graph concurrently.

At the end of each iteration, it is necessary to make sure that the information in both CPU and GPU is consistent before starting a new one. Thus, in both cases, the graph bitmask specifying the vertices discovered in the current iteration and the vertices' distances to the source are exchanged between the devices. This data exchange increases proportionally with the number of vertices in the graph and can have a considerable impact on performance as the size of the graph increases.

This approach introduces the need for synchronization, since after each level we need to make sure that the information in both the CPU and GPU is the same, before starting the new level. This means, making a copy of the graph data structures from CPU to GPU and vice versa. The specific procedure to synchronize after each iteration will depend on the method use for exploring the frontier (Top-Down or Bottom-Up), and will be further discussed on section 4.2, were implementation details are discussed.

3.3 Asynchronous

The *Asynchronous* approach attempts to maximize the utilization of both the CPU and GPU during the graph traversing. The idea is to set to work both devices in different parts of the graph from an early stage in the iteration process. This would mean both devices working independently, each at its own pace, until all the vertices are explored in the graph. Once this exploration is completed is it is necessary to join the results of both devices, an operation that would result cheap in terms of processing work as it would only involve a *min* operation between the two results. The appealing aspect of these approach is that both devices would be working independently without expensive synchronizations in the middle, as it is the case for both the *Selective* and the *Concurrent* approaches.

However, there is a limitation in this approach imposed by the potential repetition of work performed in both devices. It is easy to see this potential hazard by analyzing a simple example. Lets suppose we start traversing a graph. The first iteration will always contain a single vertex, which is the source vertex, referred in this work as *s0*. Now, lets assume that the source vertex has only two edges, meaning that during the first iteration the distance for those two vertices will be updated (with a value of one) and those two vertices will be added to the next frontier. Lets then assume that we want to split this frontier of two vertices, making the CPUs work on one vertex and the GPU work on the other vertex, letting each device update its own distance vector, and adding each discovered vertex to its own new frontier. The evident problem here is that each device may include in its frontier at some iteration a vertex that have already been discovered in a previous frontier by the other device. This will not affect the correctness of the final result, as the join process at the end will fix the values to its minimum, which is the correct distance for every vertex. However, the implications of this behavior will harm the performance as more work is performed than in regular, synchronous approaches. It will be discussed further in this work (section 5.3) whether the effect of avoiding synchronization will overcome the redundant work.

Chapter 4

Implementation Details

In this chapter, specific implementation details will be discussed, including the underlying hardware where the tests were performed (platforms), as well as details regarding the OpenCL specific APIs used and the different approaches used with its correspondent data structures.

4.1 Platforms

Embedded systems have today a huge market in cell phones and other carry-on devices. This high demand makes them cheaper and, consequently, attractive to be used in other fields. Also, because of a constraint in energy consumption, they have high performance/watt ratio that makes them interesting to be used in other fields like high performance computing. Therefore, embedded platforms are a very interesting architecture for implementing important algorithms, such as BFS.

It is also interesting to explore processors for high-end workstations, as the computing power provided by this platform is considerably bigger than the case of embedded systems, and the analysis of the results over this platforms may contribute to a better understanding of systems as a whole. The limitation imposed in this high-end workstation platforms resides in the fact that these platforms are usually assumed to live in an environment where a discrete GPU (a high-end GPU connected through PCI express) is present, and thus, the integrated GPU (the one living inside the same package with the CPU cores) is small.

The above reasoning led to choose embedded or mobile platforms. It would be intelligent to argue that there is no interest in the use of such platforms for the processing of large amounts of data, or big graphs in this case. But the key point is that these platforms usually provide an equilibrated environment in terms of the processing power of CPU cores and GPU. In other words, since embedded and mobile processors usually must condense all the processing power in a small area, and in the system there is no room for discrete GPU, the package will have CPU cores and a GPU which are similar in terms of area and processing power.

Component	Core i5	Core i7	SnapDragon	Odroid
Cores	2	4	2	4/4
RAM	4GB	16GB	2GB	2GB
GPU	Intel HD 5000	Intel HD 3000	Adreno 330	Mali MP6
OpenCL CU	40	20	8	16

Table 4.1: Platform Comparison

4.1.1 SnapDragon

The SnapDragon Board is composed by a Qualcomm Krait processor with ARM ISA and a Qualcomm Adreno GPU. The Qualcomm Krait processors has 2 cores and the Adreno GPU has 16 OpenCL compute units. The systems has 2 GB of RAM. The CPU and GPU share the physical address space, and with the use of OpenCL API we have a mechanism to share the virtual space, which may be beneficial to the overall execution speed.

4.1.2 Intel Core i Family

The Intel Core i Family is the latest family of processors for mid-range to high-end, including both mobile and desktop machines. This is commodity hardware that can be found in most of the machines sold nowadays, including Apple Macbook computers. More specifically, for all the experiments run on Intel Core i platforms, it was used the 4th generation microarchitecture, knows as it code name *Haswell*.

Core i5 - Macbook Air

One of the platforms used for the tests was a Macbook Air with an Intel Core i5, with 2 cores at 1.3 GHz, 4 GB 1600 MHz DDR3, and an integrated GPU Intel HD Graphics 5000. This GPU has 40 OpenCL compute units. It runs MacOS X El Capitan.

Core i7 - Desktop Machine

A high-end desktop machine running Linux CenOS 7.0 was used to test the case of high-end machines. This systems has an Intel(R) Core(TM) i7-4790 CPU wit 4 cores at 3.60GHz, 16 GB 1666 MHz DDR3, and integrated GPU Intel HD Graphics 3000. This GPU has only 20 OpenCL compute units, as this processor is expected to belong to a system with discrete GPU present.

It is important to note how different the two Intel Core processors are. The Core i7 has 4 cores running at higher frequency compared to the Core i5. On the other hand, the Core i5 has a bigger GPU in terms of OpenCL compute units.

4.1.3 ODROID-XU4

This embedded/mobile platform has a Samsung Exynos5422 Cortex-A15 2Ghz and Cortex-A7 Octa CPU cores, a Mali-T628 MP6 GPU, and 2GB of LPDDR3 RAM. This is a development platform for mobile, as it has similar characteristic as most modern mobile phones.

4.2 Implementation Details

As a starting point, two well-known implementations were selected for the BFS algorithm. One implementation is from Rodinia and the other is from Parboil. These implementations use different approaches for BFS on GPU. These algorithms are implemented using C and OpenCL. A C++ application was developed adapting these algorithms and adding optimizations for multicore systems using OpenMP.

A common representation for graphs is the Compressed Sparse Representation (CSR). This representation is used by the Rodinia bechmark, as well as the Graph500 benchmark, and consists mainly of two vectors. CSR is easy to implement and use during the graph traversal. The first vector will have N + 1 elements, where N is the number of vertices in the Graph. We can refer to this vector as *vertices*. The second vector will have E elements, where E is the number of edges in the graph. We can refer to this vector as *edges*.

In order to understand this representation, we can explore the example in Figure 4.1, where an Adjacency Matrix is shown together with its respective CSR representation. The Adjacency Matrix will have a 1 in the position i, j if there is an edge between the vertices i and j.

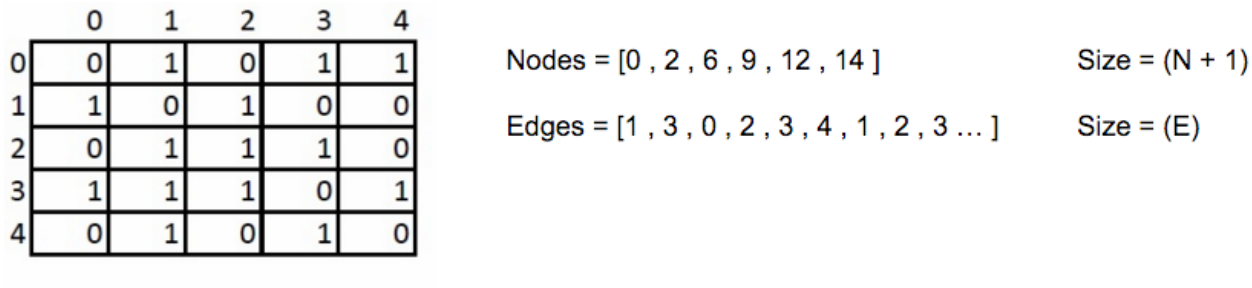


Figure 4.1: Example of an Adjacency Matrix for a Graph and its CSR representation, where N is the number of vertices, E is the number of edges. The Cost vector is originally initialized to predefined valid

The *vertices* vector will contain the indexes of the first edge in the *edges* vector for each vertex. The *edges* vector contains vertex indexes indicating an edge between a vertex in the *vertices* vector and a referenced vertex. In other words, if we want to discover the vertices connected to vertex i, we check the position i in the *vertices* vector. The *ith* element in *vertices* will contain an index e in the *edges* vector, which is the first

edge that the vertex i has. If we substract the element the $i+1$ in *vertices* with the element i in *vertices*, we will obtain the number of edges for vertex i. With this information (the index e in the *edges* vector and the number of edges for vertex i), we know which are the vertices connected to vertex i by exploring the elements from e to $e + number\_of\_edges\_for\_i$ in *edges*.

It is also common to have a third vector of size E, which contains the weight of each edge in the *edges*. Since for the purpose of our analysis, which involved social network graphs (usually undirected and with no weighted edges), and since we are interested in the implications of the use of Heterogenous platforms rather than a particular graph application, we always assume undirected graphs where every edges represents a distance of 1. Because of this, we can ignore the third vector and only used the first two, described above.

Along with the two vectors (*vertices* and *edges*), two more vectors were defined: *cost*, of size N, holding the distance value with respect to the source node s_0, and *mask*, also of size N which represents a bit mask to indicate which vertices are present in the current frontier. Some other auxiliar vectors were used depending on specific details of the implementation for the *Selective*, *Concurrent*, and *Asynchronous* approaches.

4.2.1 Shared Memory in OpenCL

For the use of shared memory, it is necessary to do some specific OpenCL API calls. First, the memory must be allocated through the OpenCL API using special flags, indicating that such memory buffer might be read by the CPU. After this allocation is done, the memory buffer can be accessed by the GPU code (also referred to as kernel). But if this memory buffer is to be read from the CPU, another API call must be made to indicate the driver that CPU will do some operation over the data, and thus, the GPU cannot perform operations on that memory block during this time. This is done by mapping to the CPU virtual space the physical memory space of the buffer.

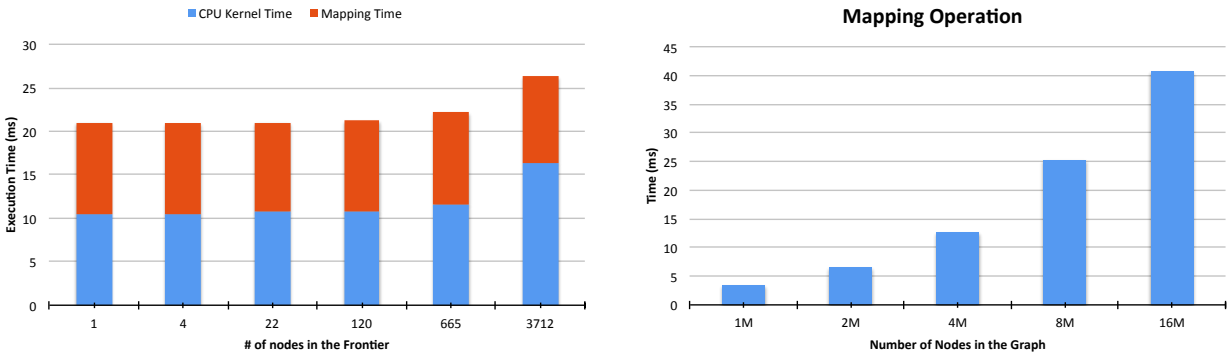
A pseudo code for this procedure is the following:

```
cl_mem d_mem;    //Pointer for the GPU
char* map_mem:   //Pointer for the CPU

d_mem = clCreateBuffer(context, special_flags, size, ... );

/* The memory is ready to be used by the GPU */
char* map_mem = clEnqueueMapBuffer(queue,d_mem,flags,size, ...);

/* Memory is used by the CPU */
clEnqueueUnmapMemObject(queue, d_mem, map_em, ... );
```

```
/* The memory is ready to be used by the GPU again*/
```

In this case, we are avoiding a copy from one memory space to another, which is necessary in the case of a discrete GPU connected through a PICe bus. The downside of this capability is the overhead introduced by the mapping/unmapping operation, and, what it is more, the limitation that only one device at a time can be performing operation over that memory block.

The overhead of the mapping/unmapping operation was considered to be negligible, as, supposedly, it would only involved pointer and flags operations. Remember that all the management within the map and unmap process is handled by the OpenCL driver, provided by the vendor of the platform. After running some initial tests on the Snapdragon platform, and discovering that the execution time in both the *Selective* and *Concurrent* approaches was longer that the graph traversing executed in CPU only, the overhead of the map/unmap operations was measured.



(a) Mapping Overhead

(b) Mapping time cost

Figure 4.2: SnapDragon Analysis. Mapping overhead for a 4M graph for different iterations (a) and Mapping cost for different graph sizes (b)

For this particular platform, the overhead is proportional to the size of the graph (Figure 4.2(b)), and independent from the size of the new frontier, as can be seen in Figure 4.2(a), where approximately 11ms are added to the execution time of the CPU version independently of the size of the each frontier. This means that, once the map is done, the memory access time for the CPU is the same as if it was accessing a piece of data allocated by itself. The problem is that the time that it takes to perform the map operation is not only linear with the memory block size, but also similar to performing a copy of the same memory size. This experiment was carried out to verify that the mapping operation is the only factor that adds time, and there is no on-demand address translation, which would be harmful when the size of the frontier is big, but it ended up showing an unexpected source of latency. After this incident, we performed analysis on all the

platforms, but this issue was not found. All the platforms show similar behavior as the one in Figure 4.3, where, even if the time it takes for each map operation increases as the memory block size mapped increases, the time is negligible (less and a millisecond).

Because of this, the idea of heterogeneous approaches was dismissed in the case of the SnapDragon Platform, and carried out in the other platforms. However, some important findings are associated this behavior in order to compare the different performance implications in all the platforms, as it will be discussed in section 5

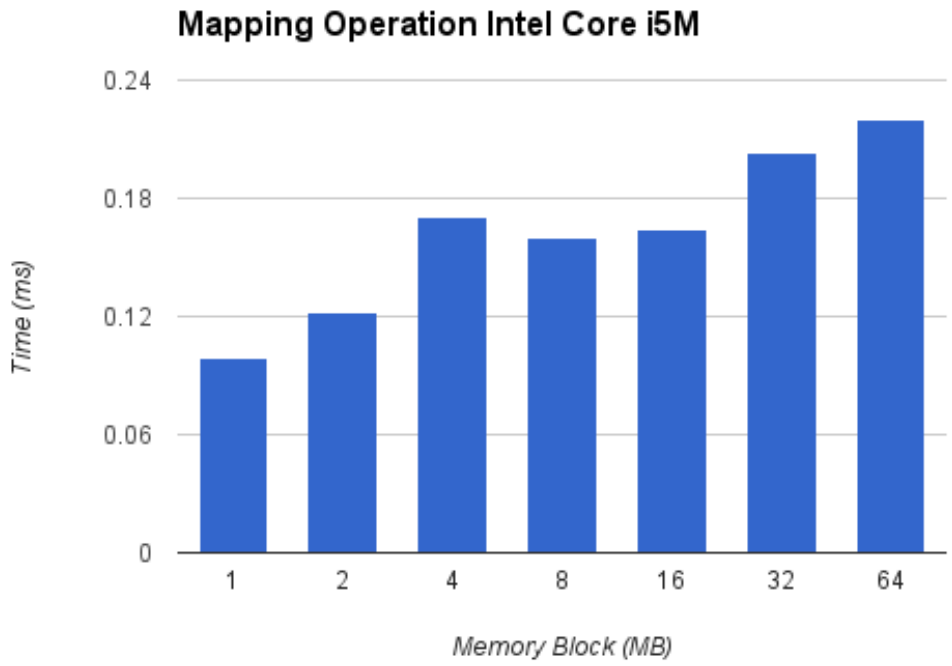


Figure 4.3: Mapping overhead for memory blocks of different sizes in Core i5

OpenCL 1.2 was used for all implementations. It is important to note that this version does not provide an APIs that allows the use to read/write to the same memory locations from both GPU and CPU. The *Concurrent* approach works around this limitation, and performs reads and writes after in both CPU and GPU after a map call. Some problems due to the lack of cache coherence between CPU and GPU were found on the Odroid platform, but on Intel platforms no problems were found even though the this technique is not allowed by the standard.

4.3 Optimizations

The Rodinia benchmark kernel originally iterates over all graph's vertices and checks a bitmask to know which vertices on the graph are in the current frontier to be processed. However, if the frontier is small, this incurs in a big overhead. Hence, we optimized the code to generate a frontier vector, at the beginning of each iteration, only with the vertices indexes that are part of the new frontier. With the frontier vector, the kernels only need to iterate through the vertices that are going to be processed in the current iteration. Furthermore, since the GPU has fine-grain threads, iterating only over the vertices that are going to be processed allowed us to have less threads and avoid scheduling overheads, but introducing more irregularity in the memory access.

The Rodinia implementation on GPU dispatches one thread for each vertex in the graph, and as the graph increases it might become a bottleneck for small GPU architectures like the SnapDragon. Thus, we limited the number of threads in GPU and the same GPU thread goes over multiple vertices in the current frontier.

Another optimization is the the parallelization of the CPU version, both Top-Down and Bottom-Up methods, which was implemented through the use of OpenMP *parallel for* in the loop that goes over the frontier vector, enabling coarse grain parallelism by the use of all the CPU cores.

The last optimization is related to the generation of the frontier vector. Even if generating the frontier vector *a priori* for our implementations would signify launching several less number of threads for a particular iteration, it was a costly operation since it involved checking each vertex in the bitmask and adding it to a vector sequentially. This operation cannot be done simply with a OpenMP *parallel for* since each thread does not know how many vertices will appear, and thus, does not know in which location to place it. A solution for this is to apply a prefix sum after each thread has discovered the vertices that will be part of the new frontier. For this, each thread would have a private vector with the vertices for the new frontier, and a private counter. After all threads have discovered the vertices, a synchronization is done and a prefix sum is performed using the counters, so that each thread know where to place its private frontier vector in a global frontier vector. Then, this copy can be done by each thread in parallel. We found out that when the number of vertices in the graph is small (up to 2M), this prefix sum method does not show any advantages over the sequential operation. This is expected since we are introducing the overheads of thread creation and synchronization. In the cases where the number of vertices in the graph is big (more than 2M), the prefix sum method outperform the sequential operation, and in the case of the graph of 16M vertices, it performs almost 3.5 times faster, and thus, this optimization was used in all the approaches implemented.

Chapter 5

Experimental Evaluation

5.1 Execution Time Histograms

In order to have a good understanding of how each of the methods (Top-Down or Bottom-Up), each of the devices (GPU or CPU), and each platform would perform, an application for profiling the graphs was developed. This application creates histograms for each graph, providing information about its structure and how it is processed differently in different scenarios (i.e., different platforms, approached, and method).

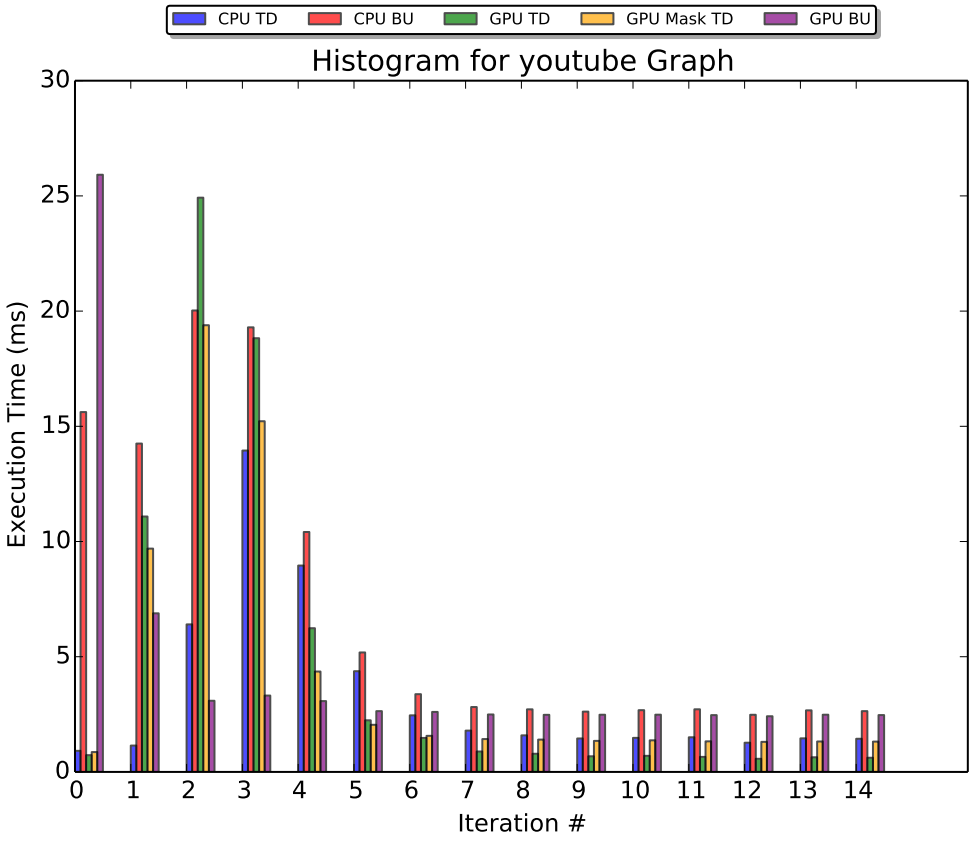


Figure 5.1: Histogram for Youtube Graph on Intel Core i5

It is important to note that this work introduces a comprehensive study of a combination of techniques in order to improve execution time for BFS, and there are several factors that must be analyzed in order to get the best out of the underlying hardware. Because of this, it was important to understand the internals (iteration by iteration) of the BFS algorithm.

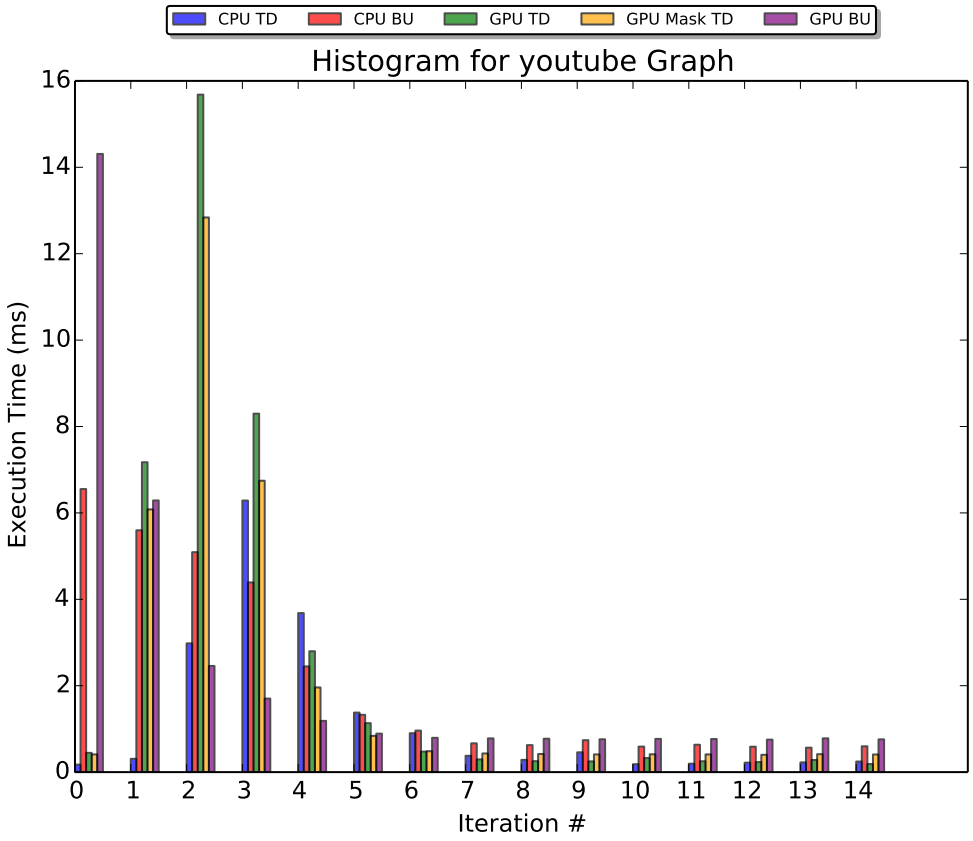


Figure 5.2: Histogram for Youtube Graph on Intel Core i7

Figure 5.1, for instance, shows histograms for the Youtube graph on the Core i5 platform. The different bars represent a combination of a method (Top-Down or Bottom-Up) and a device (CPU or GPU), as specified in the legend of the figure. The difference between the green bar (GPU TD), and the yellow bar (GPU Mask TD) is the way in which the frontier is processed.

In the GPU TD case, a vector with all the vertices in the frontier is pre-processed in the CPU, using a prefix sum, whereas in the case of the GPU Mask, the GPU will process all the vertices and only perform the exploration if the vertex has a 1 in the mask, indicating that that vertex belongs to the frontier. These two different implementations were designed in order to explore how the regularity or irregularity of memory accesses would affect the processing time in the GPU. In all platforms, except from SnapDragon, the GPU

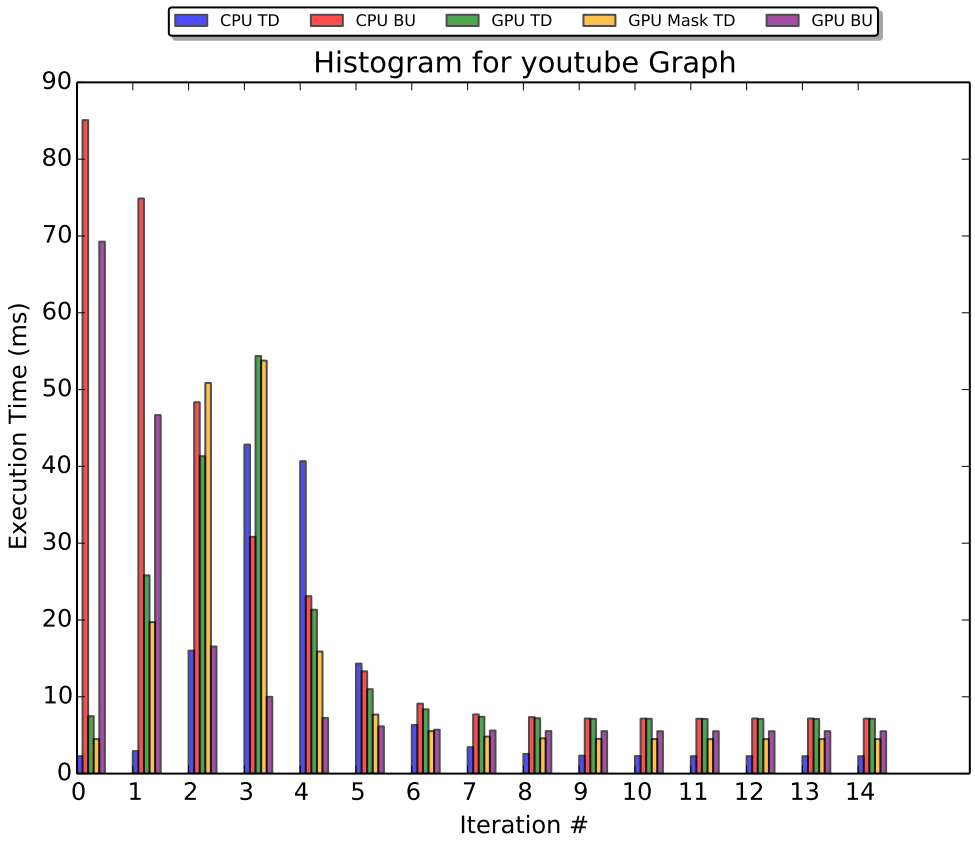


Figure 5.3: Histogram for Youtube Graph on Odroid

Mask TD (yellow) proved to be better performant than the GPU TD (green). This may be attributed to the regularity of the memory access in the GPU. Each GPU thread will be accessing contiguous memory locations when threads have contiguous ids. The case of SnapDragon behaving differently from this expected case will be further explained in section 5.4.

5.2 The Effects of the GPU Computing Power

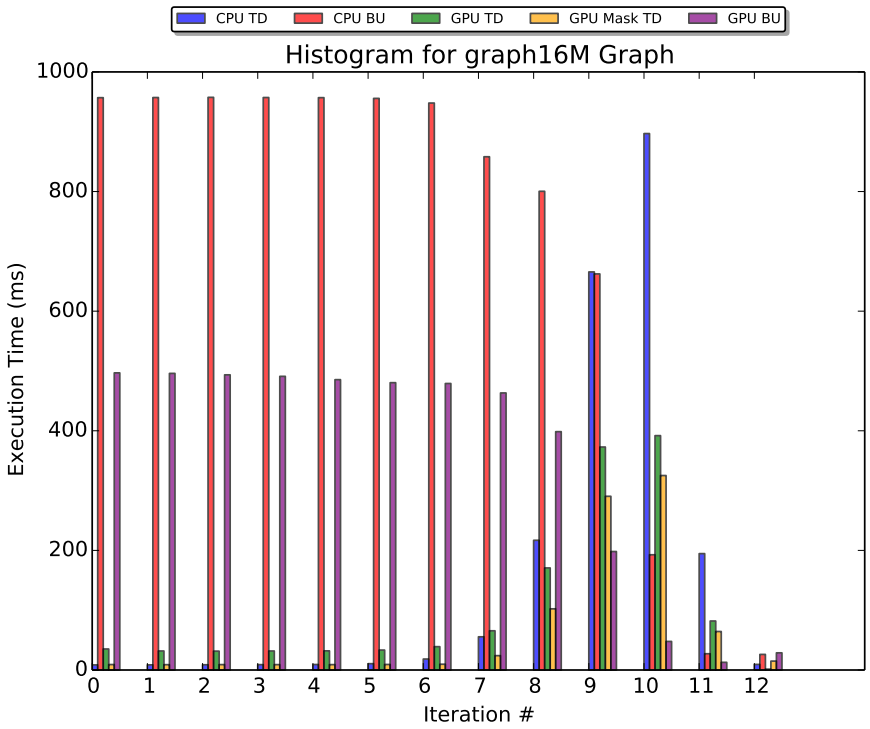
As it was mentioned in the introduction of this work, the motivation to use embedded and mobile platforms lays in the observation that in these environments a System-on-Chip (SoC) is used due to the need of small area and better energy efficiency. Since the option of having an external GPU to perform graphics processing is not viable, the SoC tends to sacrifice CPU computing power in order to have space for a GPU with enough capabilities. This results in an environment in which the computer power of the CPU is comparable to the one in the GPU. This is not the case in high-end processors, where the integrated GPU is relatively small because an external GPU is assumed to exist.

Figure 5.4 provides a clear overview of this expected behavior. Figure 5.4(a) corresponds to the case of the Core i5, where the only GPU present in the system is the integrated GPU within the processor. It can be seen that, for instance, during iterations 1 to 7, when using the Bottom-Up method in CPU (red) and in GPU (purple), the GPU can perform the traversing of a frontier in almost half the time that it takes to the CPU cores. However, in the case of Figure 5.4(b), where the system is less equilibrated in favor of CPU cores, the GPU implementation (purple) is slower than the corresponding CPU implementation (red).

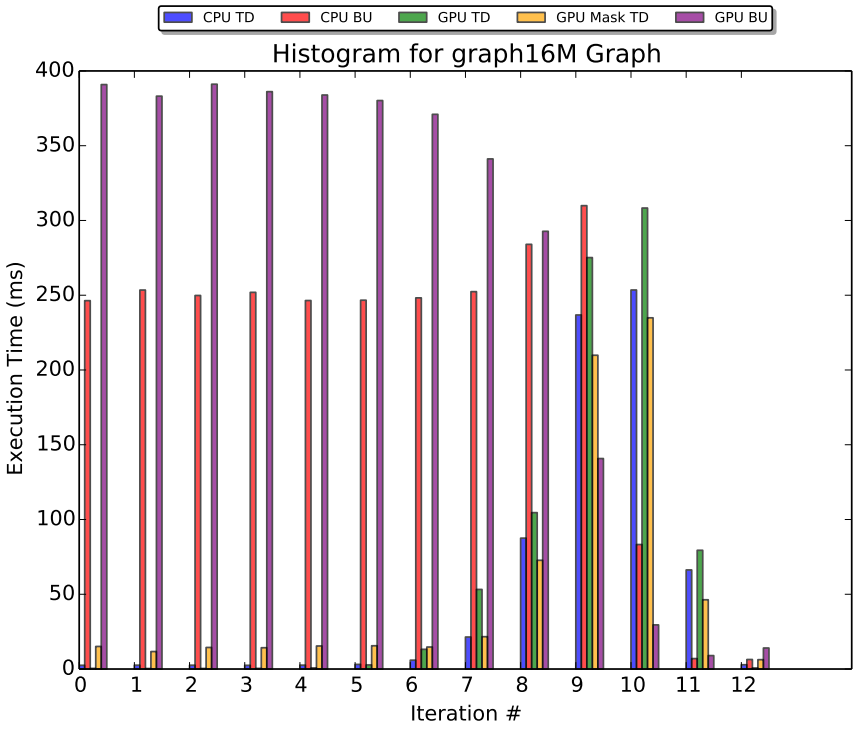
Another similar analysis can be made over iteration 9. In the case of figure 5.4(a), the GPU implementations (green, yellow, and purple) are notoriously better than the CPU implementations. One can consider the best case for CPU (blue), and the best case for GPU (purple), and will realize that the GPU outperforms the CPU by more than 3 times. This means that there is a solid opportunity to gain in performance. On the other hand, in the high-end Core i7 (figure 5.4(b)), the gain in performance is smaller as the GPU is not big enough to introduce a considerable gain compared to the CPU.

In this frontier, the amount of work to be performed is several times compared to any of the iterations 1 to 7, and since more parallelism can be exploited, the GPU is more suitable to traverse the frontier. In the case of the mobile platform, this opportunity for improvement can be exploited, but in the high-end, any improvement will be smaller.

This experiment shows how important is for the overall performance the computing power of the GPU and its ratio with the computer power of the CPU cores.



(a) Histogram for Rodinia Graph 16M on i5



(b) Histogram for Rodinia Graph 16M on i7

Figure 5.4: Histograms: Comparison between high-end (Core i7) and mobile (Core i5) platforms

5.3 Repeated Work Analysis for Social Network Graphs

Social network graphs have the particular characteristic of being traversed in a small number of iterations (less than 15 for the studied graphs, some of them having as few as 6 iterations). Also, the amount of work performed in each iteration (i.e., the number of edges traversed per iteration) may differ in orders of magnitude. This is an interesting characteristic that can give room for intelligent work division between GPU and CPU when analyzed in detail. In a seek for the most efficient use of the resources, the *Asynchronous* approach was initially though to exploit as much parallelism as possible, including the use of both devices working at the same time in different parts of the graph. The problem of this technique is that it may inccur in work repetition, as it was described in section 3.3.

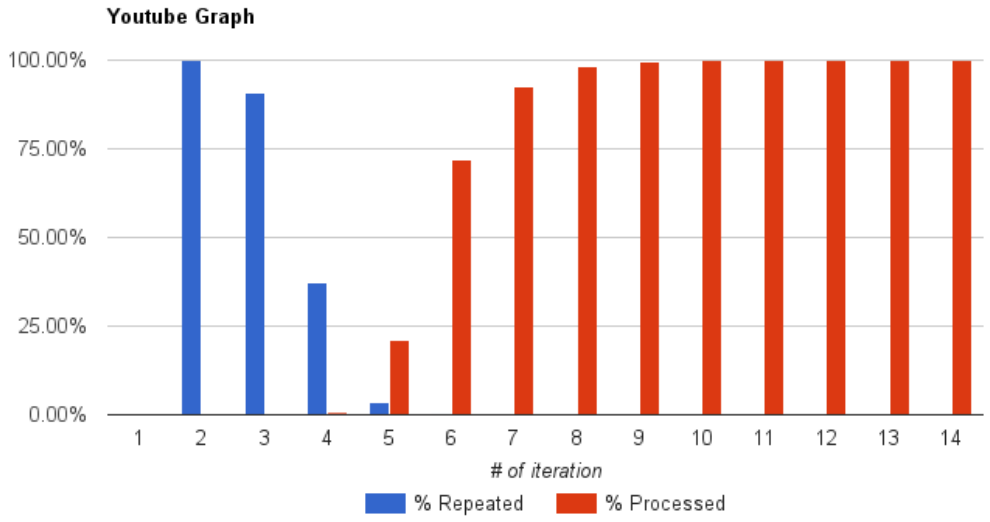


Figure 5.5: Repeated work if splitting the frontier at each iteration for the Youtube Graph

A profiling application was implemented in order to understand this behavior and discuss whether this approach will be worth or not.

Figure 5.5 shows how much of the vertices would be revisited if the frontier was split across both devices at the given iteration. It is not possible to split the frontier in the first iteration, as it only contains a single vertex, which is the source. For instance, if the frontier for the second iteration is split, almost all the vertices will be visited by both CPU and GPU.

On the other hand, as the graph is traversed, vertices are less likely to be visited twice, as less vertices will be left to explore. The extreme case will be when the frontier is split in the last iteration. In this case, only those vertices left to visit that have a connection with vertices that are in both the frontiers explored

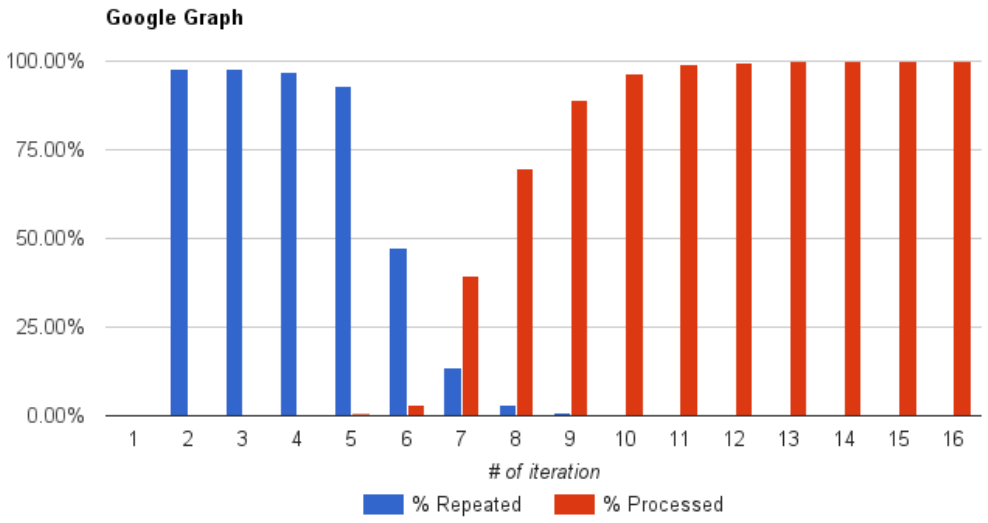


Figure 5.6: Repeated work if splitting the frontier at each iteration for the Google Graph

by the GPU and CPU will be repeated.

But if we consider the case of iteration 5 in Figure 5.5, if the frontier is split after only traversing less than 25% of the graph, less than 5% of the vertices will be visited by both devices. This characteristic can be considerer as an opportunity for performance gain, as the less work is done before splitting a frontier, the more work that will be done asynchronously in both devices, making use of all the resources. In the case of the Youtube graph, for instance, it is possible to split the frontier among the devices, and each device can run independently, at its own pace, on a different part of the graph. The same can be applied to other social network graphs that we studied: The frontier can be split at iteration 7 in the Google graph (Figure 5.6), and at iteration 11 in the Rodinia 8M Graph (Figure 5.7), and in both cases, more than 50% of the graph can be processed in parallel, and asynchronously.

It is also important to note that the worst case is being analyzed here, which is when the frontier is split in half. Assuming that, for instance, the GPU can perform twice the work than the CPU cores in the same amount of time, it would make sense to split the frontier 33% to CPU and 66% to the GPU, in which case, the number of repeated vertices will decrease.

However, when testing this approach, an unexpected behavior was found. When splitting the frontier among the devices, each device will, at the beginning, process a frontier that will be half the size of what it would be if the frontier was not split. This means that half of the vertices corresponding to that frontier will not be explored, and the vertices connected to these unexplored nodes will be left to discover later in some other iterations. As a result, more iteration will be needed in each device. It is important to remember that

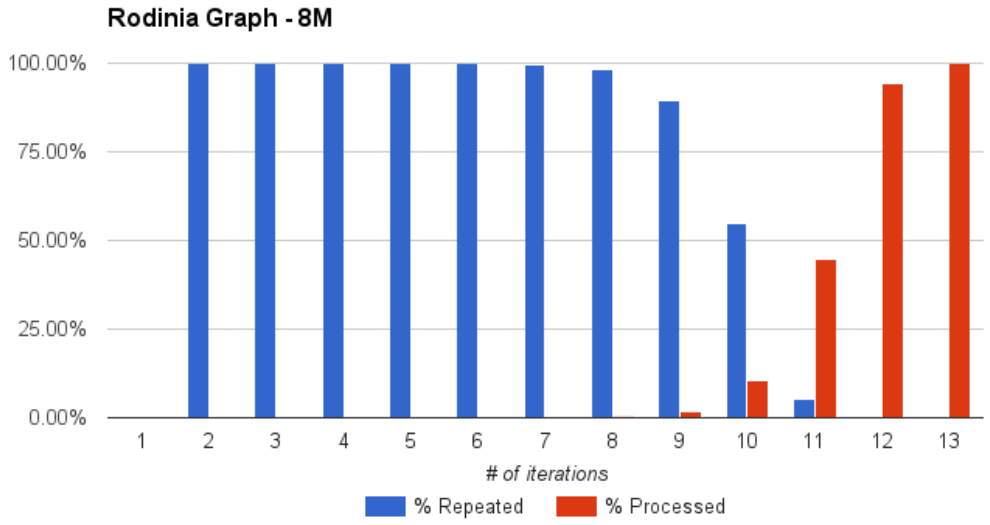


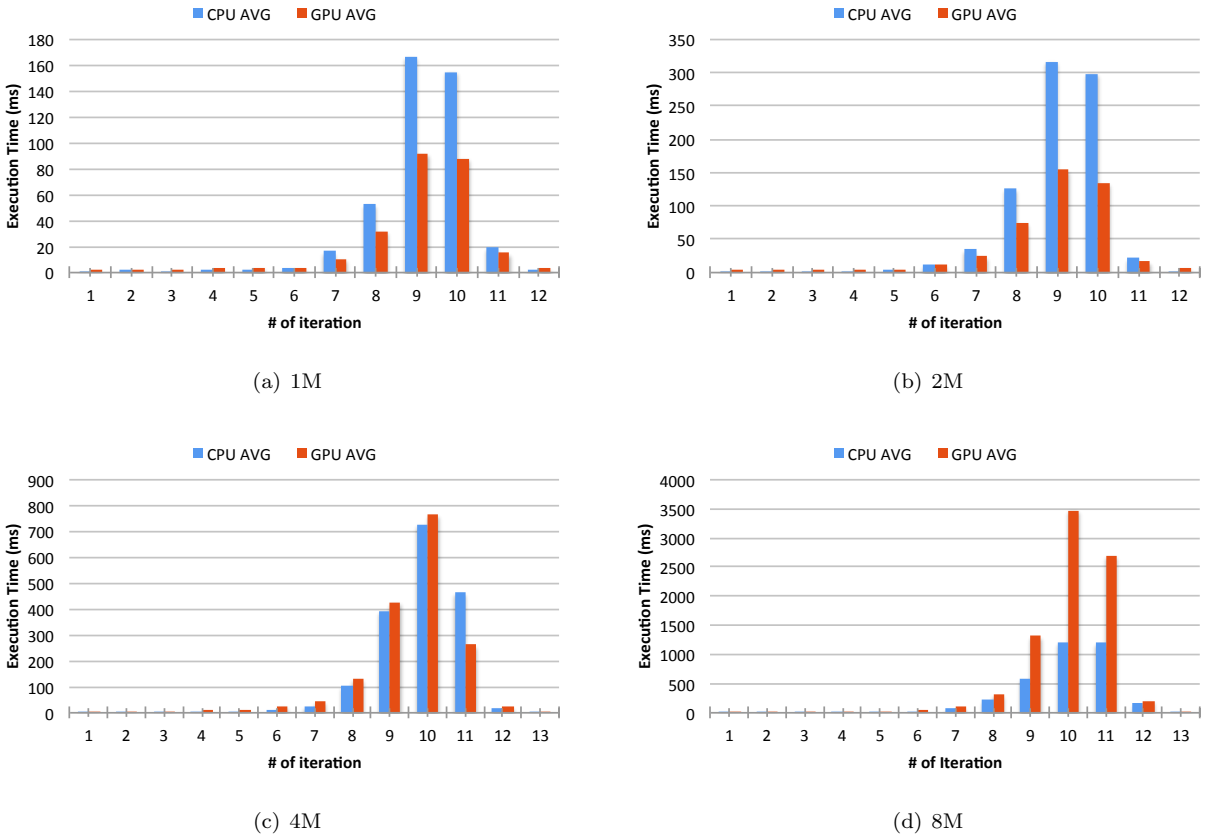
Figure 5.7: Repeated work if splitting the frontier at each iteration for the Rodina Generated 8M Graph

a new iteration represents an implicit point of synchronization, that will badly harm the performance of the GPU. Even if splitting the frontier and making each device work at its own pace without synchronization among the devices, the effects of introducing more iteration proved to harm the performance more than the speedups obtained by the use of this technique. It is also important to note that, even if the number of iteration will be larger, the final result will be correct as, after both devices are done traversing its portion of the graph, the results will be joined using a *min* operation.

The *Asynchronous* approach was implemented then to exploit this property, but after several testing with different graphs, the execution time was bigger than the other 2 heterogenous approaches implemented, and this approach was dismissed.

5.4 Experimental Evaluation on SnapDragon

SnapDragon was the first platform to test the initial version of the approaches, and was useful to define the path of this research. The problem with this platform is that the computing power of both the CPU and the GPU is very limited. The GPU has only 4 OpenCL compute units, meaning that the amount of parallelism that can be exploited is limited and will only make sense on very regular memory access.



(a) 1M

(b) 2M

(c) 4M

(d) 8M

Figure 5.8: Average execution time for SnapDragon in Both CPU and GPU for each iteration of Rodinia Generated Graphs, using 100

In order to understand the capability of SnapDragon, Rodinia graphs were profiled for this platform using the mentioned histogram application. Since the first approach for a heterogeneous implementation involves the selection of a threshold to decide in which device to run at any given iteration, it was necessary to know under how much load the CPU or GPU began to perform better or worst. The result of this experiment in the SnapDragon architecture can be seen in Figure 5.8. It can be seen that when processing frontiers of the size of 2M vertices or less (from Figure 5.8(c) and Figure 2.4(a)), the GPU performs better (or almost equal, in the case of Figure 5.8(c)) than the CPU. When the number of vertices in the frontier is bigger

than 2M, the GPU will start to perform badly, and CPU will be the best option. This can be attributed to over-scheduling in the GPU because of the launching of a big number of threads. This behavior led the research to focus on platform with more capabilities in terms of processing power, as having such a little computing power would not be a suitable environment for graph processing.

Another important information from the figures is that when the number of vertices in the frontier is very small (less than 15K vertices), the CPU outperforms the GPU (this is an expectable behavior since launching little work to the GPU incurs in overhead).

After understanding the above results, it become evident that no much improvement (if any at all) would be found after implementing heterogenous versions. The results of the execution time for different graphs in the SnapDragon board are shown in Figure 5.9.

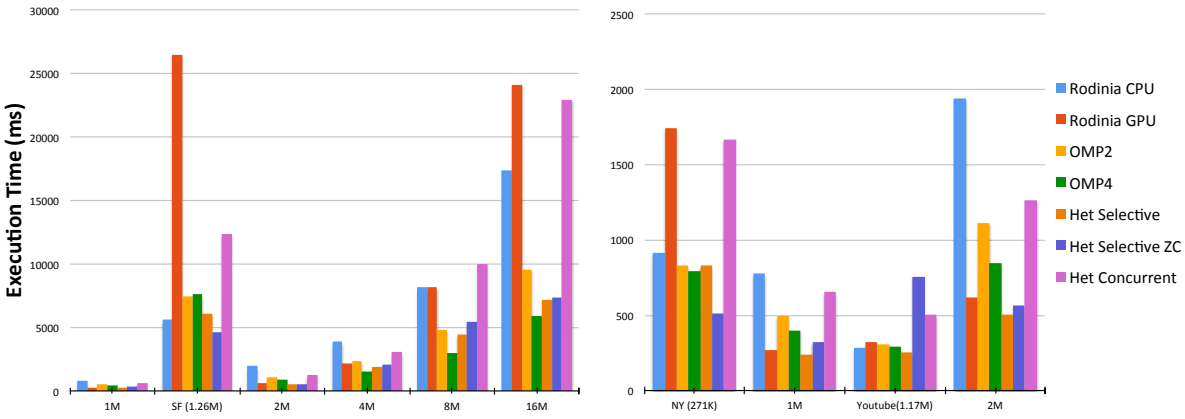
In SnapDragon, the GPU version is not always the best after certain threshold. After 8 million vertices, the CPU performs better than any other implementation, while in graphs with less than 8 million vertices the GPU version has a better performance. This is contrary to the common knowledge about performance in GPU, when contrasting to CPU. In the presence of large degree of TLP and relative simple work to do in each thread, the GPU architecture is supposed to perform better. But this is not the case when having a small GPU in an environment with limited resources. One reason for this effect is that Rodinia GPU version is launching a big number of threads, and with very limited hardware resources, the overhead of scheduling and handling such load becomes very large. In the case of the CPU, this is implemented as a for loop, and no overhead is introduced when the number of vertices increase (the execution time of each loop will increase linearly with the size of the input).

The *Selective* approach proved to be best in small graphs (up to 4M), as can be seen in Figure 5.9(a). It also performs better than the CPU only sequential version and the GPU only version for all the cases, but does not work better than the CPU OpenMP version with 4 threads, which is best for the cases of bigger graphs. There are two reasons for this. First, in big graphs, the cost of synchronization after each iteration becomes substantial as the number of vertices increase. Second, when the number of threads launched in the GPU (which is the size of the frontier) is bigger that 2M, a notorious scheduling overhead is introduced since the number of compute units available is small for this platform.

Another important analysis that can be made from Figure 5.9 is how the *Selective* approach with zero copy had almost no performance improvement. This is because, even if all the copies are avoided, the overhead introduced by the mapping/unmapping operation makes the overall execution in the CPU slower. However, there are two particular cases where the zero copy implementation works better: in the SF and NY graphs. As can be seen in 2.4, these graphs need several iterations to be completed. In the case of the

NY graph, iterations in the middle will be executed in the GPU, whereas iterations at the beginning and at the end will be executed in the CPU. Since this happen for several iterations, there is no need to map and unmap multiple times during the execution of the graph traversal, and having zero copy will mean less synchronization overhead. In the case of the SF graph, most of the time the execution is done in CPU, and some iterations at the beginning and at the end are executed at the GPU (this is not displayed in the figures), and again the synchronization cost is less.

Finally, Figure 5.9 also depicts how the Heterogeneous Concurrent approach brought no improvement, and even performs poorly for all the cases. This is because a the map/unmap operation is expensive in this platform. This synchronization, as described early, involves merging the bitmask of the new discovered vertices as well as the vector of the distances of each vertex to the source and can be done in parallel (using OpenMP and 4 threads), so it is not introduce any considerable overhead.



(a) SnapDragon Big Graphs

(b) SnapDragon Small Graphs

Figure 5.9: Execution Time for SnapDragon

5.5 Performance Evaluation

In this section, a performance evaluation is made over different platforms in order to compare the results and understand the reasons for different performance gains across different graphs and GPU-CPU configurations.

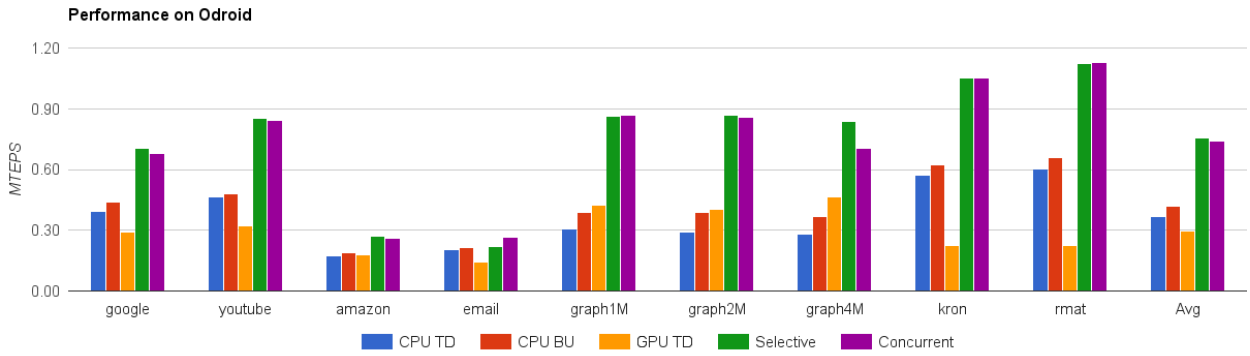
Figure 5.10(a) shows the results obtained after running different graphs in the Odroid platform. Both the *Selective* and the *Concurrent* approaches proved to work better than any other implementation using a single device at the time. This image depicts the Bottom-Up and Top-Down methods implemented for CPU cores (in red and blue, respectively), a Top-Down approach for GPU only (in yellow), and finally, both the *Selective* and the *Concurrent* approaches (green and purple). The *Asynchronous* method proved to perform poorly compared with the other two heterogenous approaches and was not included in the analysis. The cause for this is explained in section 5.3.

All the results are shown using the metric Millions of Traversed Edges Per Second (MTEPS), which is a very common metric in the graph literature. This metric provides a normalization of performance across graphs with different number of vertices and edges, and thus, different running times. It is important to note that, even if this metric is very common, it can introduce some confusion when interpreting the results. Strictly speaking, the number of traversed edges can change even when running the same graph, as it will depend on the starting point, or source vertex. It is possible to traverse the graph from a starting point and traverse certain number of edges, which will very likely contain repeated edges, and then traverse the graph again from a different start point an exploring edges different from the ones before. Even more confusion will come when interpreting the results obtained from Bottom-Up and Top-Down methods, as the way in which these approaches explore the graph is completely different. Many other works in regards to graph use this metric considering just the number of edges in the graph, and not looking into the detail of how many edges are, in fact, traversed. In other words, the metric MTEPS in this work, as in many other, results from dividing the number of edges in a graph by the execution time, and does not consider any of the scenarios above mentioned.

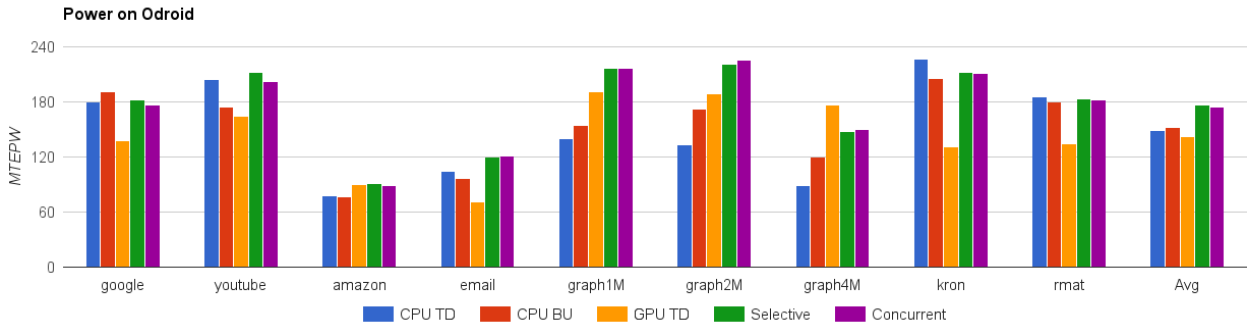
For performance testing, 500 BFS were run, starting from different sources, and the average was calculated. It is important to perform this average as running time might change slightly depending on the source vertex.

For the Odroid platform, an average speedup of 1.7x is obtained, and in Rodinia graphs speedups up to 2.25x are reached. The speedups are attributed mainly to a better use of the resources, since for the case of the *Selective* approach, only the best device to perform the work is used, and in the case of the *Concurrent*, there are certain iteration where sufficient work must be performed to justify the use of both devices at the same time, having all the resources fully utilized at that point.

Another important factor to consider in the Odroid platform is the energy consumption. This platform provides a library that allows power measurements, which was used as part of this work. Figure 5.10(b) shows how both the *Selective* and the *Concurrent* approaches are more energy efficient compared to any other approach, using the metric Millions of Traverse Edges per Watt (MTEPW). This energy efficiency can be attributed to two main characteristics. First, these approaches traverse the graph in less time than any other approaches, thus consuming energy for less time. Second, GPU at peak performance consumes less energy than the CPU cores under similar stress, and since these approaches make use of the GPU, a reduction in the total power consumption is expected to happen. It is also expected to see a slightly less energy efficiency when comparing the *Selective* and the *Concurrent* approaches, as the latter will be using both devices at the same time during some iterations. But it is not expected to find a big difference, as even if both are working at the same time, the amount of work that each device is performing is less than in the case of the *Selective*, which will put a single device to work to its full capacity. On average, energy efficiency improvement of 1.14x was achieved, which is an important factor considering that both execution time and the power consumption are optimized.



(a) Performance evaluation on Odroid



(b) Power Evaluation on Odroid

Figure 5.10: Performance evaluation on Odroid

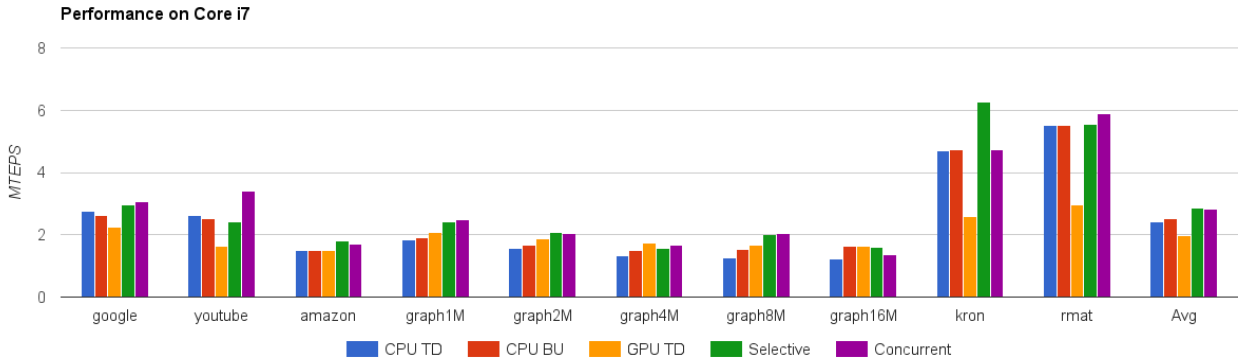


Figure 5.11: Performance evaluation on Core i7

In the case of the Core i7, the results are not as good as the ones obtained for Odroid, but this is a expected outcome as the size of the GPU is not comparable to the computing power of the CPU cores. For this particular platform, an average speedup of 1.17x is achieved. The cause for this results is explained by following the results presented in 5.2.

Finally, the middle point between the Odroid platform and the Core i7 platform can be the Core i5 processor, as main processor of a Macbook air. In this platform, both the CPU and GPU capabilities are bigger than in the Odroid. The GPU is also bigger in this case compared with the Core i7 (Core i5 has 40 OpenCL compute units, whereas the Core i7 has only 20), but the Core i5 has smaller cores compared to the 4 high-end cores in the Core i7.

The results for this platform are shown in Figure 5.5. An average of 1.3x is achieved in this case, but the interesting result here comes with the performance gain when using only the GPU TD approach (yellow bar). This implementation, which does not show any gain in any of the other platforms, is in this case the better performant approach. This can be attributed to the nature of the platform. The Core i5 is the only processor present in the system, as no discrete GPU is attached. But as this systems corresponds to a laptop where users expect high quality graphics, an important part of the chip is devoted to a graphics processor. In the Odroid platform, which is more oriented to hand-held devices, high graphics performance is not expected, and the GPU does is not require to have high performance.

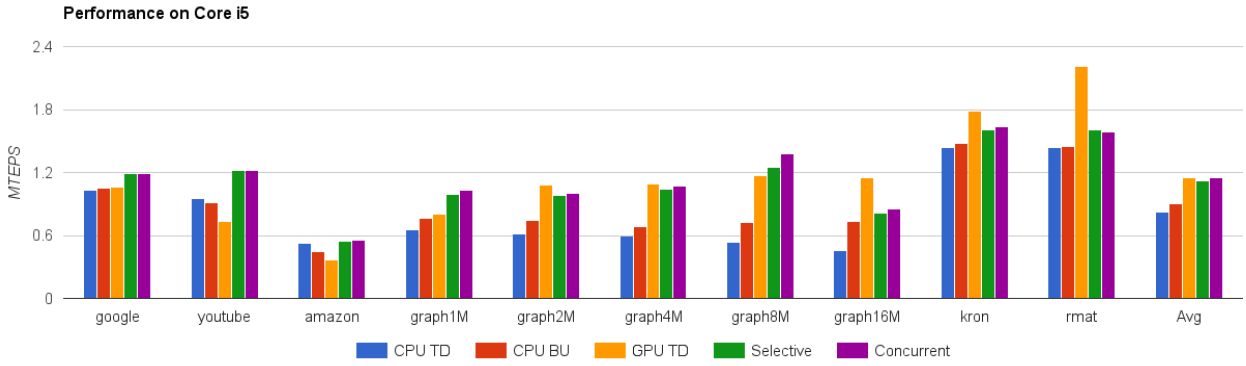


Figure 5.12: Performance evaluation on Core i5

5.5.1 Memory Bandwidth Limitations

Both the *Concurrent* and the *Asynchronous* approaches were initially design to maximize the use of the resources. Since heterogenous platforms provide a shared-memory environment, the data in memory can be used by both devices at the same time without requiring memory copies. This is why it is so appealing to explore algorithms that can traverse the graph in parallel. But after implementing the approaches aimed to exploit this characteristic and finding no improvement, a more comprehensive study was perform around this matter.

Since the *Asynchronous* approach was already proven to have difficulties in achieving better performance because of redundant work that introduce new iterations, the study focus on understanding why the *Concurrent* approach showed no improvement with respect to the *Selective* approach.

Figure 5.5.1 shows how the system reach a memory bandwidth limitation when both devices are working at the same time. This experiment consisted on running the iteration with the heaviest work load in both devices, in a similar manner that the *Concurrent* is implemented, but also timing how long does it take to each device to finish its part if it were running alone in the system, and it was run over the Rodinia 16M graph. The blue bar corresponds to the execution time of the CPU working on its part alone, the red bar corresponds to the execution time of the GPU working on its part alone, and finally, the yellow bar shows what is the execution time when both devices are working at the same. In a best case, the yellow bar is expected to be as big as the max of the red and blue bars. In other words, when both the CPU and the GPU are working, it is expected that all the work will be finished after the slowest device is done. But this is not the case, and the when both devices work at the same time, the memory system cannot provide the devices with the sufficient memory bandwidth.

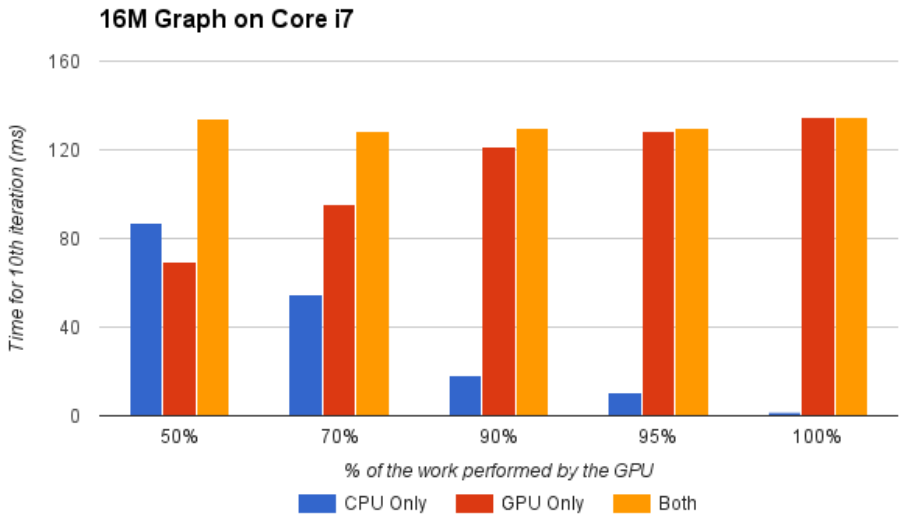


Figure 5.13: Memory bandwidth limitation analysis on Core i7

This experiment also shows how, in theory, an optimal distribution of the work for this platform would be around 50-60%, as this is the point where the amount of time it takes for each device is similar, and once the fastest device is done it will have to wait the minimum possible time idle. The case of splitting the frontier 50-50 shows how the system experiment memory bandwidth limitation, as both devices work at the same time, compared to having a single device issuing memory request. The number of memory requests per time will increase when both devices are working at the same time, preventing the devices to work at their peak performance. It is important to note that this behavior is due to nature of the BFS processing, which is completely memory bound. Each explored vertex only perform reads and writes to memory, and little to zero computation.

Chapter 6

Conclusions and Future Work

The use of Heterogenous systems is becoming increasingly important as we approach the ends of the Moore's Law. It becomes important then to explore algorithms that make use of all the resources present in a system. The characteristics of the shared-memory environment makes the use of a GPU as a co-processor very promising, as expensive data transfer are no longer required.

This study focused on the implementation and evaluation of heterogenous approaches on the BFS algorithm for graph processing, achieving speedups of 1.7x on average and 1.3x improvements in energy efficiency.

A comprehensive comparison among different platforms was performed in order to understand the implications of the system characteristics when traversing graphs of different characteristics. These studies led to the conclusion that the best performance improvement can be obtained by having CPU cores and GPU with similar computing capabilities, which is an environment mostly present in hand-held devices and mobile platforms.

This work also explored characteristics of social network graphs and attempted to transform these characteristics into opportunities for speedups in graph traversing, finding fundamental limitations on heterogenous systems when performing memory bound tasks.

The use of shared-memory systems creates unique opportunity to a more efficient use of the resources, but effort must be dedicated to the design of hardware and software that can handle each of the components working in close memory locations at the same time without limiting each others capabilities. With the advent of more complex memory systems which better interface with the processor, this limitation can partially disappear, creating an environment where heterogenous approaches will bring performance improvements even in the more irregular and bad-behaved applications.

The exploration of other irregular algorithms for graphs and other applications is left as future work, as well as the implications of this results in well-behaved regular algorithms usually only delegated to the GPU.

References

[1] T. Hiragushi and D. Takahashi, "Efficient hybrid breadth-first search on gpus," in *Proceedings of the 13th International Conference on Algorithms and Architectures for Parallel Processing - Volume 8286*, ser. ICA3PP 2013. New York, NY, USA: Springer-Verlag New York, Inc., 2013, pp. 40–50. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-03889-6_5

[2] D. G. Merrill and A. S. Grimshaw, "Revisiting sorting for gpgpu stream architectures," in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '10. New York, NY, USA: ACM, 2010, pp. 545–546. [Online]. Available: http://doi.acm.org/10.1145/1854273.1854344

[3] A. Munshi., "The opencl specification," 2012. [Online]. Available: http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf

[4] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, Oct 2009, pp. 44–54.

[5] L. Luo, M. Wong, and W.-m. Hwu, "An effective gpu implementation of breadth-first search," in *Proceedings of the 47th Design Automation Conference*, ser. DAC '10. New York, NY, USA: ACM, 2010, pp. 52–55. [Online]. Available: http://doi.acm.org/10.1145/1837274.1837289

[6] L. Munguia, D. Bader, and E. Ayguade, "Task-based parallel breadth-first search in heterogeneous environments," in *High Performance Computing (HiPC), 2012 19th International Conference on*, Dec 2012, pp. 1–10.

[7] S. Hong, T. Oguntebi, and K. Olukotun, "Efficient parallel graph exploration on multi-core cpu and gpu," in *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, Oct 2011, pp. 78–88.

[8] S. Beamer, K. Asanović, and D. Patterson, "Direction-optimizing breadth-first search," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 12:1–12:10. [Online]. Available: http://dl.acm.org/citation.cfm?id=2388996.2389013

[9] D. Merrill, M. Garland, and A. Grimshaw, "Scalable gpu graph traversal," in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '12. New York, NY, USA: ACM, 2012, pp. 117–128. [Online]. Available: http://doi.acm.org/10.1145/2145816.2145832

[10] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.

[11] Graph500, "http://www.graph500.org/."

[12] J. Travers and S. Milgram, "An experimental study of the small world problem," *Sociometry*, pp. 425–443, 1969.