Critical Links Detection using CUDA

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Informatik

eingereicht von

Thomas Schnabl
Matrikelnummer 0306109

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Univ.-Prof. Dipl.-Ing. Dr.techn. Günther Raidl
Mitwirkung: Univ.Ass. Dipl.-Ing. Dr.techn. Bin Hu

Wien, 01.03.2013

(Unterschrift Verfasserin) (Unterschrift Betreuung)

FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

Critical Links Detection using CUDA

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Informatik

by

Thomas Schnabl
Registration Number 0306109

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Univ.-Prof. Dipl.-Ing. Dr.techn. Günther Raidl
Assistance: Univ.Ass. Dipl.-Ing. Dr.techn. Bin Hu

Vienna, 01.03.2013

(Signature of Author)	(Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Thomas Schnabl
Falkenhayngasse 14, 3631 Ottenschlag

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum)	(Unterschrift Verfasserin)

Danksagung

Ich möchte an dieser Stelle meinen tiefsten Dank an meine Eltern, meine Freundin und meinen Universitätsassistenten aussprechen. Sie haben sehr viel Geduld mit mir bewiesen.

Acknowledgements

I am indebted to my university assistent, my parents and my girlfriend to have had so much patience with me.

Abstract

The *Critical Links Detection* (CLD) Problem consists of finding for the smallest set of edges in a graph to be protected so that if a given number of unprotected edges are removed the diameter does not exceed a given value. The diameter of a graph is defined as the length of the All-Pair-Shortest-Path (APSP).

This thesis presents an algorithm that takes an instance of a graph and calculates the minimal set of protected edges. For k simultaneously failing edges it checks all possible tuples of k edges if after temporarily removing them the diameter of the graph exceeds the limit. Afterwards a Integer Linear Programming (ILP) model is built up and solved, which is equivalent to choosing the minimal amount of edges from these sets to get a feasible solution.

Since the calculation of diameters takes most of the process time, the thesis presents an improvement on the performance of the algorithm. A massive parallel approach is implemented using CUDA, a framework to run calculations on graphic cards, also called GPGPU. The chosen algorithm solves one Single Start Shortest Paths (SSSP) algorithm on one GPU processor, running one thread per vertex of the graph. The performance of the algorithm shows a speedup of 30% using this approach on small instances and 70% improvement on large instances with over 1000 edges.

Abstrakt

In dieser Diplomarbeit wird ein Algorithmus entworfen, der das *Critical Links Detection* (CLD) Problem mit mehreren simultan entfernten Kanten löst. Dieses Problem ist für mehr als 2 gleichzeitig entfernten Kanten *NP-vollständig*. Im CLD Problem wird der Diameter eines Graphen untersucht, wenn eine oder mehrere Kanten entfernt werden. Gesucht wird hierbei die kleinste Menge von Kanten, die geschützt werden muss, damit der Diameter des Graphs einen bestimmten Maximaldiameter nicht überschreitet. Der Diameter eines Graphen ist definiert als die Länge des All-Pair-Shortest-Path (APSP).

Der präsentierte Algorithmus berechnet die CLD durch Aufbau und Lösens eines Integer Linear Programming (ILP) Modells. Bei k gleichzeitig möglichen Kantenausfällen werden alle Untermengen von k Kanten überprüft und all jene Gruppen von Kanten hinzugefügt, deren Entfernung aus dem Graph dessen Diameter die erlaubte Grenze überschreiten lassen. Die Lösung dieses Problems entspricht der Auswahl einer Menge Kanten aus diesen Gruppen, die möglichst klein ist.

Die Diplomarbeit präsentiert außerdem zusätzliche Verbesserungen am Algorithmus um die Laufzeit zu reduzieren. Da die Berechnung der Diameter einen großen Anteil der Prozesszeit verbraucht, wird diese mithilfe von CUDA, einem Programmier-Framework, auf der Graphikkarte berechnet. Im gewählten APSP Algorithmus werden hierzu auf jedem der GPU Prozessoren ein Single-Start-Shortest-Path (SSSP) Algorithmus durchgeführt, der intern genau einen Thread pro Knoten des Graphen behandelt. Je nach Instanzgroße rechnen so tausende oder Millionen Threads gleichzeitig auf der Graphikkarte. So konnte die Laufzeit des Algorithmusses bei kleinen Testinstanzen um 30%, bei großen Testinstanzen mit etwa 1000 Kanten um 70% reduziert werden.

Contents

CHAPTER 1

Introduction

As the Internet has become more and more important for our lives, Internet reliability has become a main issue. Companies are selling their products in online shops, employees connect to their workspace over VPN, search information on Internet or have to stay in touch via instant messaging or email, even applications rely on web services. A breakdown of connection can cause major problems and loss of income. Especially at the backbone connection, dealing with huge amount of traffic, failures have great impact on many people. However we face different risks of failure since the Internet is a very heterogeneous bunch of routers, cables, fibers and satellite connections.

On middle and high-level of the world wide Internet there may be many different routes from one computer to any other target host. “Open shortest path first (OSPF) is used as a routing protocol within a single autonomous system (AS) at the Internet.” [9] To gain shortest delay and largest throughput the OSPF chooses the shortest path. Connections over long distances increase the delay and the probability that a link included in the path could be damaged. To ensure this for any path the diameter, the maximum path length between any two nodes in a network, must be small.

In a network there are arbitrary links and some important links whose failure significantly increase the diameter or even break the network into separate unconnected parts. Those critical links must be protected against failure or have several backup links to take their duty. Surely such protection is expensive combined to normal connections, therefore the number of these links have to be small.

1.1 Problem Description

The Problem was first introduced in 2010 by Tekshi Fujimura and Hiroyoshi Miwa in “Critical Links Detection to Maintain Small Diameter against Link Failures” [9].

Given any network or graph, there are certain critical links “whose failures significantly deteriorate the performance in a network” [9]. To ensure the performance these critical links have to be protected.

At this problem the *performance in a network* is defined as the diameter of the network. The *Diameter* is given as the *length of the longest of the shortest paths between any tuples of vertices.*

A communication network is represented as a graph $G = \{V, E\}$ with a set of vertices V and set of edges E. Let $p_{u,v}$ be the shortest path in G from the vertex u to vertex v and $|p|$ be defined as the length of the path. Then the diameter is defined as

$$d = max_{u,v \in V}(|p_{u,v}|)$$

Lets call $E_P \subseteq E$ the set of (k, D)*-protected edges*. E_P is defined as any subset of E, that satisfies for any $E_K \subseteq E$ that $|E_K| \leq k$ for positive integer k and that $E_K \cap E_P = \emptyset$ the diameter of $G_K = (V, E - E_K)$ is less than or equal to positive integer D. "Even if any k or less edges except (k, D)-protected edges are deleted, the diameter of the resulting graph is no more than D " [9].

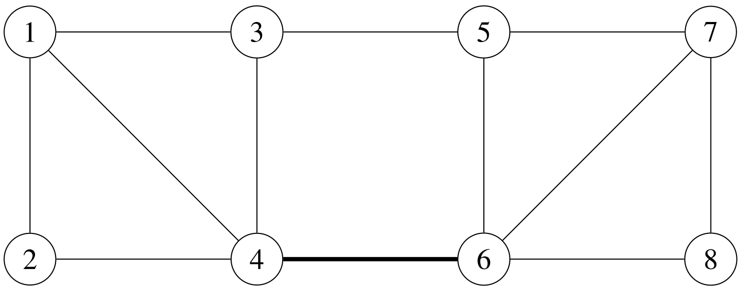


Figure 1.1: Graph with diameter 3. With one failure at $e_{4,6}$ the diameter increases to 5.

The figure 1.1 gives a small example for $k = 1$. With all edges and no failures the presented graph has a diameter of 3. With one possible failure the edge between vertices 4 and 6 may fail. The diameter of the resulting graph is now 5 because the shortest path between vertex 2 and vertex 8 has the length of 5. Protecting edge $e_{4,6}$ and the failure of any other edge the diameter is never exceeding 4. So for $k = 1$ and $D = 4$ the minimal set of protected edges is $E_K = \{e_{4,6}\}$.

Runtime Complexity

The runtime complexity of the given problem depends on the value for variable k. Multiple edges that fail at the same time increase the difficulty to find the target set of edges.

- For $k = 1$ the problem can be solved in *polynomial time*. One algorithm solving this problem is presented in section 4.2.
- For $k \geq 3$ the problem is *NP-complete*. Fujimura and Miwa present a prove in [9] by transforming 3SAT into CLD in polynomial time.
- For $k = 2$ there is *no prove* if it is NP-hard or not. It is suspected to be in NP [9].

1.2 Structure of the work

This work is structured into 6 main chapters. At first **chapter 1** I give an introduction. I explain the problem that is addressed and explain the structure of the work. At the next **chapter 2** I explain the methodology. I give some short introduction to used algorithms as Breath-First Search (BFS) and All Pair Shortest Path (APSP). I explain the principle of Linear Programming (LP), Integer programming (IP) and General Purpose Computation on Graphic Processing Units (GPGPU). I also give a short introduction to the history of graphic devices and the current state of GPGPU-APIs. In **chapter 3** I go into detail about Compute Unified Device Architecture (CUDA), the NVidia™ solution for GPU programming. Then at **chapter 4** I introduce and explain my algorithm for solving the problem, while in the **chapter 5** I present the results of my algorithm. At the last **chapter 6** I reflect about the solutions and suggest further development and investigations.

CHAPTER 2

Methodology

In this chapter I want to describe some techniques and technologies I used for my CLD algorithm. Breath First Search (BFS) and All Pair Shortest Path (APSP) are basic techniques on graphs I use to calculate the diameter. I speed up these algorithms by using GPGPU to do some preprocessing by repeatedly calculate diameters after removing some edges. These diameters are then used to build up a model (or equation system). This model is then solved using Integer Linear Programming (IP). The result of the IP solver is then transformed back to a list of edges that have to be protected against failure.

2.1 Breadth-First Search

Breadth First Search (BFS) is one way how to fully traverse trees or graphs. Starting at one vertex $s \in V$ the graph G is separated into *frontiers*. The frontier 0 is the vertex s itself. The frontier 1 is the set of those vertices and edges that are directly connected to s. And frontier N is the set of vertices and edges that are directly connected to any vertex in the frontier $N-1$, but not included in any frontier with $n < N$. An example of a graph divided into frontiers is given in figure 2.1.

The BFS traverses one frontier after the other. So all vertices in frontier i are traversed before any vertex in frontier $j > i$. The ordering of vertices inside a frontier is undefined. In most cases the algorithm uses an inner order of vertices like an ID or a name of vertices. An example for a BFS run is given in figure 2.2.

BFS can be applied to any directed or undirected graph, no matter if weighted or unweighted, directed or undirected. There are many applications for the BFS. It is mainly used to collect data about graphs. As an example for trees that are non circular graphs, the BFS is used to get depth information. Starting at the root vertex, the set of vertices with depth i equals the frontier i.

Shortest paths can be easily calculated with BFS if the graph has equal weights on all edges. The distance between any vertex u and vertex v is simply the frontier of vertex v starting at vertex u.

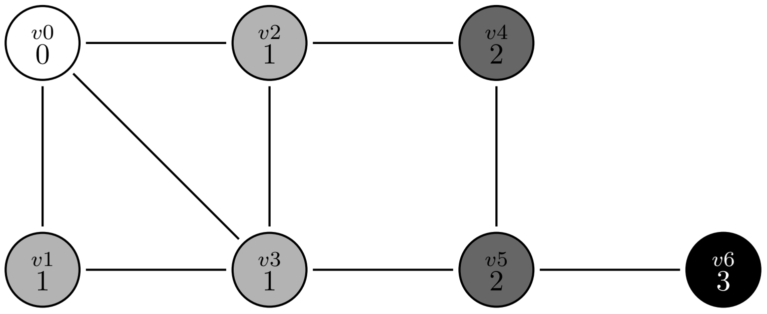


Figure 2.1: Frontiers at BFS. Vector v0 is in frontier 0; Vector v1, v2 and v3 are in frontier 2; Vector v4 and v5 are in frontier 3; Vector v6 is in frontier 4.

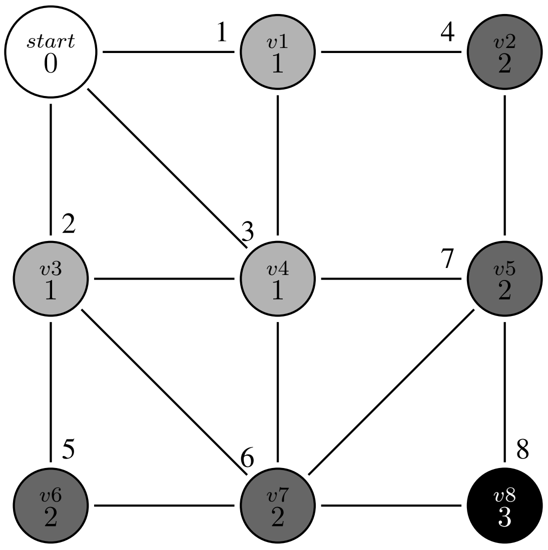


Figure 2.2: BFS run on undirected graph. The numbers on the end of the edges show the step in which the target vertex is m. Vertices at the same frontier are sorted by ascending IDs. Different ordering may result in different choice of edges but the frontiers stay in the same order.

2.2 All Pair Shortest Path

The *all-pairs shortest paths* (APSP) problem is to find shortest paths between all-pairs of vertices in a graph. it is usually equivalent to make a table of minimum distances for every pair. [5, 19]

SSSP

The *Single Source Shortest Path* (SSSP) is to find shortest path between a fixed start vertex and any other vertex. *Dijkstra's algorithm* finds the shortest path from x to any y in the order of increasing distance from x. It uses an array of distances which is initially infinite for all vertices except the start which is set to 0. Each iteration the algorithm chooses the vertex with lowest distance. Then it calculates the distances of all neighbor vertices. “It is an example of a greedy algorithm where for each iteration, the best vertex is selected” [13]. Once visited, the distance of a vertex is not changes anymore. The algorithm is the fastest known SSSP algorithm for graphs with unbounded nonnegative weights [5]. If using fibonacci heaps it solves the problem in $O(|V| * log(|V|) + |E|)$ [5].

APSP

The *Floyd-Warshall* algorithm calculates the APSP in a graph. It uses a matrix storing the distances between all pairs of vertices $d(i, j)$ where $i, j \in V$ which is initially infinite for all pairs. Each iteration it selects a vertex k and iterates over all pairs i and j. It calculates $d(i, k) + d(k, j)$ and updates the matrix if the sum is smaller than the distance already known between i and j. The Floyd-Warshall algorithm takes $O(|V|^3)$ operations [19] because of the triple nested loops and can be applied to any graph without negative circuits. The disadvantage of this algorithm is the amount of memory it needs. It needs to store, besides of the graph representation, at least two $[n \times n]$ arrays. If not only the distance of the shortest path, but the path itself is needed, two additional $[n \times n]$-arrays of so called “backlinks” have to be stored.

Another approach to solve the APSP problem is to iteratively calculate the SSSP. This is a quite naive approach, but gives the opportunity to easily introduce parallelization to the algorithm and needs less memory than the Floyd-Warshall.

2.3 Linear Programming

Linear Programming (LP) is a mathematical method to solve problems that can be presented in a linear objective function and linear inequality constraints. Let A be an $[m \times n]$-dimensional matrix where $n \geq m$. Let c be an n-dimensional vector of coefficients and b be an m-dimensional vector of coefficients. Let z be a variable scalar and x an n-vector of variables. Given an A, b and c, then the problem

$$\begin{aligned} \text{Max} z &= cx && \text{objective function} \\ Ax &\leq b && \text{constraints} \\ x &\geq 0 && \text{non negativity constraints} \end{aligned}$$

is called the *LP problem in canonical form* [7]. “If the structural constraints are $Ax = b$ rather than $Ax \leq b$, we way that the problem is in *standard form.*” [7]. LP problems in canonical form can be transformed into standard form by adding nonnegative *slack variable column* S to A [7]. The problem is now

$$\begin{aligned} \text{Max} z &= cx && \text{objective function} \\ Ax + S &= b && \text{constraints} \\ x &\geq 0 && \text{non negativity constraints} \\ S &\geq 0 && \end{aligned}$$

The feasible region of a LP is called a *polytope*. A bounded polytope is a *polyhedron*. [7] Figure 2.3 shows a polytope and figure 2.4 shows a polyhedron. Note that some authors name them differently or reverse the definitions.

A polytope is also defined as the intersection of any finite number of hyperplanes and close halfspaces. Let a_i be the ith row of matrix A and let $Ax = b$ be a given system of equations with variable vector x. Then a set of values of x where

- $a_i = b_i$ is called *hyperplane*;
- $a_i \leq b_i$ is called *closed halfspace*;
- $a_i \geq b_i$ is called *closed halfspace*;
- $a_i < b_i$ is called *open halfspace*;
- $a_i > b_i$ is called *open halfspace*.

[7]

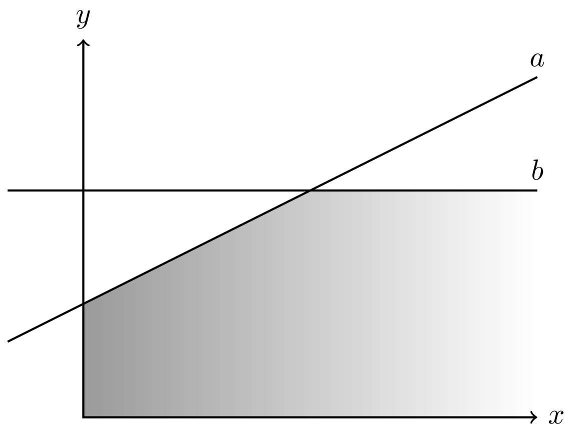


Figure 2.3: Open polytope with 2 equations. There is no equation on the "right" side to form a polyhedron.

Some methods as the Simplex Method described later use partition of A into two sub-matrix $A = [B, N]$. B consists all columns of A that are in the basis and N consists of all other columns. The *basis of A* are any collection of m *linearly independent* columns of A. The vector of variables x may also be partitioned accordingly into $x = \begin{bmatrix} x_B \\ x_N \end{bmatrix}$. The components vector x_B are called *basic variables* and the components of vector x_N are called *nonbasic variables*.

Simplex Method

Simplex is an algorithm for solving LP problems in standard form. "It does this by repeatedly transforming the coefficient matrices of the system of equations" [7]. The first step is to introduce variable z_0 and use it to transform the objective function into an equation and add it to the

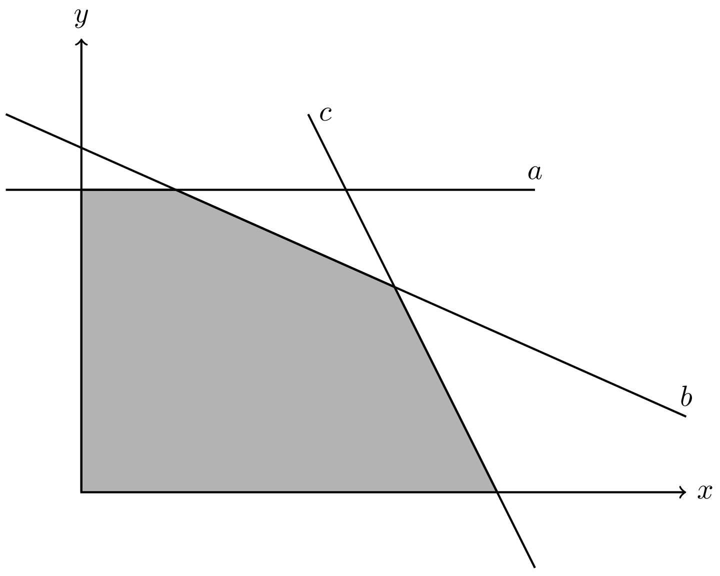


Figure 2.4: Closed polytope (polyhedron) with 3 inequations

system:

$$Ax = b$$
$$z - cx = z_0$$

where z_0 is initially equal to zero. Simplex uses tableau representation given as

z	**x**	**1**
0	A	b
1	$-c$	z_0

Simplex starts with a basic feasible solution, any solution that is inside the feasible set. Then it does tableau transformations in such a way "that $c_j = 0$ for all basic columns" [7]. Then it repeatedly select any nonbasic varibale, so that the element $c_s < 0$. The s-th column is called the *pivot column*. Then it selects the smallest value a_{rs} in the column c_s, called the *pivot element*. The r-th row is called the *pivot row*. After this selection process the tableau is transformed, so that the variable inside the *pivot row* that was in the basis now leaves the basis. This can be done by ether multiply the column with a scalar or substract a multiple of another column from this column. I will explain how this is exactly done in an example later. Further details about the algorithm can be found at [7].

There is graphical interpretation of the algorithm as well. A basic feasible solution correspond to an extreme point of the feasible polytope. You may also call them *edges* of the polytope. A tableau transformation in the simplex method corresponds "to a move between two adjacent extreme points of the feasible polytope" [7]. Figure 2.5 shows such a graphical interpretation on the polyhedron of figure 2.4.

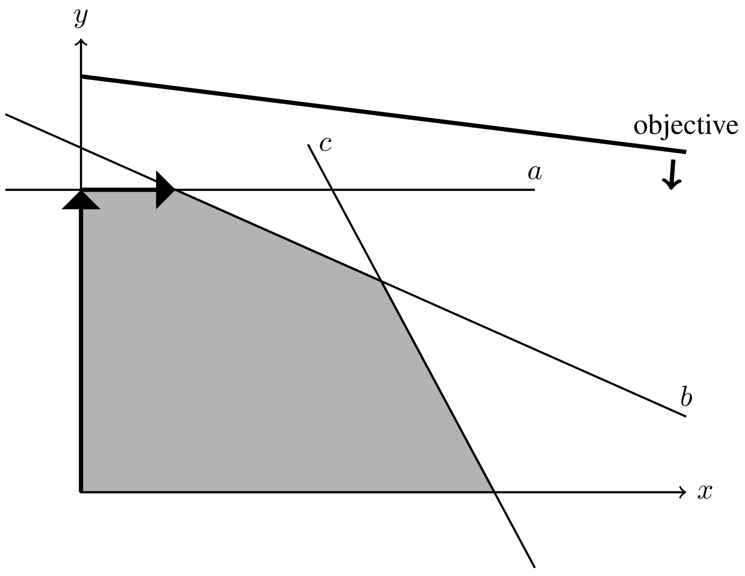


Figure 2.5: Graphical representation of Simplex method on a polyhedron with 3 inequations

Simplex Example

To give you a better impression on how Simplex works, I want to introduce it by using a simple example. Lets consider the LP problem given as

$$\begin{aligned} \text{Max} P &= 5x + 7y \\ 2x + 4y &\geq 100 \\ 3x + 3y &\geq 90 \\ x, y &\geq 0 \end{aligned}$$

where we now transform the objective function first into $0 = 5x + 7y - P$ and then into $-5x - 7y + P = 0$ to match the other rows. After introducing additional *slack variables* u and v, to eliminate the inequalities, we get the LP problem in standard form:

$$\begin{aligned} -5x - 7y + P &= 0 \\ 2x + 4y + u &= 100 \\ 3x + 3y + v &= 90 \\ x, y, u, v &\geq 0 \end{aligned}$$

which is then transformed into the LP table

x	**y**	**u**	**v**	**P**	**b**	Base
2	4	1	0	0	100	u
3	3	0	1	0	90	v
-5	-7	0	0	1	0	

where the last column shows, which row is base for this column.

Now lets select the pivots. The pivot column is found by selecting the *smallest strictly negative element* in the target row, that is in our case -7. So the pivot column is column y. Now we try to find the pivot row. For every row we divide the value in column b by the value in pivot column. So we get the values $100/4 = 25$ and $90/3 = 30$. The row with smallest value, the first one, becomes pivot and the pivot element is 4.

If there are any calculations with resulting values less or equal zero here, the problem is *unbound*. The algorithm may exit here.

After selecting pivots, we calculate the new values for the table. For the pivot column these are

$$a^1_{i,j} = a_{i,j}/\text{pivot element}$$

For any other row these are

$$a^1_{i,j} = a_{i,j} - a_{p,j} * a^1_{i,q}$$

where p is the index of the pivot row and q is the index of the pivot column. The calculations are shown in

x	**y**	**u**	**v**	**P**	**b**
$\frac{2}{4}$	$\frac{4}{4}$	$\frac{1}{4}$	$\frac{0}{4}$	$\frac{0}{4}$	$\frac{100}{4}$
$3 - 3 * \frac{2}{4}$	$3 - 3 * \frac{4}{4}$	$0 - 3 * \frac{1}{4}$	$1 - 3 * \frac{0}{4}$	$0 - 3 * \frac{0}{4}$	$90 - \frac{100}{4}$
$-5 + 7 * \frac{2}{4}$	$-7 + 7 * \frac{4}{4}$	$0 + 7 * \frac{1}{4}$	$0 + 7 * \frac{0}{4}$	$1 + 7 * \frac{0}{4}$	$0 + 7 * \frac{100}{4}$

which results into

x	**y**	**u**	**v**	**P**	**b**	Base
$1/2$	1	$1/4$	0	0	25	y
$3/2$	0	$-3/4$	1	0	15	u
$-3/2$	0	$7/4$	0	1	175	

Now we again select a pivot column, row and element. This time it is column x since it has the last strictly negative value in the target row. The pivot row is the second, since $15 * 3/2 = 22.5 < 25 * 2 = 50$, so the pivot element is $3/2$. The resulting table is

x	**y**	**u**	**v**	**P**	**b**	Base
0	1	$1/2$	$-1/3$	0	20	y
1	0	$-1/2$	$2/3$	0	10	x
0	0	1	1	1	190	

Now we are finished, there are no negative values in the target row. The values for $x = 10$, $y = 20$ and $P = 190$ can be easily read from the table.

To make sure we check the inequalities from the beginning

$$\begin{aligned} \text{Max}190 &= 50 + 140 \\ 20 + 80 &\geq 100 \\ 30 + 60 &\geq 90 \\ 10, 20 &\geq 0 \end{aligned}$$

which all hold true.

2.4 Integer Linear Programming

Linear programming solves problems with constraints on variables with continuous values, values in $\mathbb{R}$. Integer Linear Programming (IP) solves problems over discrete value space. Variables are restricted to values in $\mathbb{Z}$, $\mathbb{N}$ or even on a finite set such as $\{0, 1\}$. "If only parts of the variables are subject to this supplementary integrality constraint, it is called a *problem in mixed integer programming*" [1] (MP).

"One might think that integer programming is simpler, since one can exclude the points other than those with integral coordinates. Nothing can be further from the truth." [7]. There is just no simple way how to exclude the fractional points. To give an idea of the solution space, let me give a small example of the generalized assignment problem (GAP). This example is influenced by [7] and the introduction at [1].

A storage company operates 10 storage buildings for 30 customers. Customer i has a given space requirements a_ij in storage building j; every building has different shapes or adjustment skills of employees. Every storage building offers storage area of b_i square meters. The rent for storage depends on the customer, since they want to store different types of goods. Suppose that each square meter of space rented to company i will yield c_i euros per month. At last lets define zero-one decision variables y which equal 1 if company i is assigned to storage j and 0 otherwise. The problem may be then formulated as

$$\text{Max} z = \sum_{i=1}^{10} \sum_{j=1}^{30} a_ij * c_i * y_ij$$

$$\sum_{j=1}^{30} a_ij * y_ij \leq b_i \qquad \forall i = 1, \ldots, 10$$

$$\sum_{i=1}^{10} y_ij = 1 \qquad \forall j = 1, \ldots, 30$$

$$y_ij \in 0, 1 \qquad \forall i, j$$

where the objective maximizes the income for the storage company. The first set of constraints models the limitation of storage space while the second set of constraints ensure that each customer is assigned to exactly one storage building. The number of possible zero-one solutions y is 10^{30}, an impressive number for such a small problem. Even with modern equipment it is not useful to enumerate all possibilities. Suppose it is possible to evaluate 10^6 solutions in a second, it still needs 10^{24} seconds, that are roughly $3, 171 * 10^{16}$ years to test every possibility.

In fact "integer and mixed integer programming problems are *NP-hard*" [7]. In fact IP problems are also *NP-complete*. [24] But "there are problem types that frequently permit the user to solve instances of even fairly large size within a reasonable amount of time by using modern commercial solvers." [7].

Most ILP have a polyhedron as shown in figure 2.6. Several possible solution are inside the solution space. Other problems haven't any solution in the polyhedron, as shown in figure 2.7.

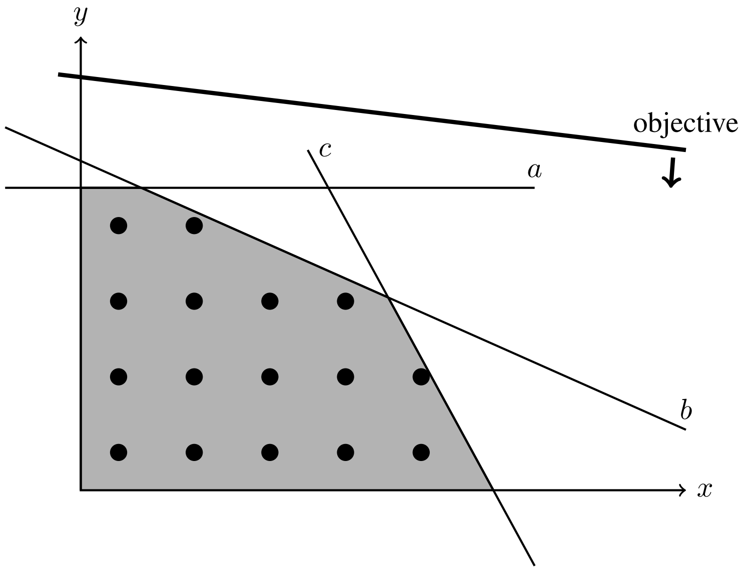


Figure 2.6: Polyhedron for an ILP. Possible solution are drawn as black dots, the gray area models the solution space for a corresponding LP.

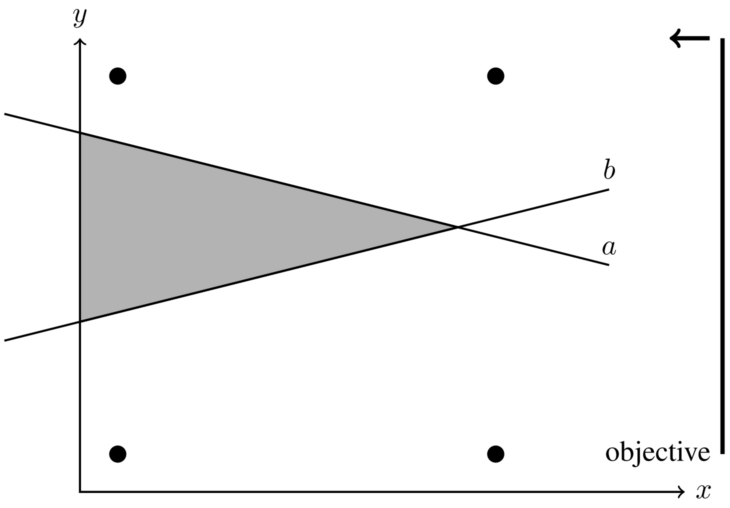


Figure 2.7: Polyhedron for an infeasible ILP.

ILP relaxation

There are two main classes of methods to solve IP problems. Since the problems are NP-complete, the usage of heuristic methods are advisable. Possible methods include Hill Climbing, Ant colony optimization and Tabu Search [24]. These methods sound most promising and give large opportunity for research. But “there are several algorithm to solve integer linear programs exactly" [7]. Since I use CPLEX®to solve the IP problem in our case and because CLPEX®uses exact methods, I want to discuss them instead.

The variety of exact algorithms to solve IP problems has been increasing in the last decades, but the basic principle has not changed since the 50s. In the following quotation I present you the words of Simonnard written in 1962, translated 1966 in [1] into english:

> "We start from a basic solution which does not satisfy all the constraints of the problem, either that the variables constrained to be integer are not, or with certain variables being negative.
> Then we add an equation (possibly several), chosen such that all integer programs necessarily satisfy this equation." [1]
> "If this does not lead to an integer program of the original problem, a new equation is added and the process is iterated. One may show that after finite number of iterations we will be led to an optimal integer program." [1]

This principle of excluding the integer constraint is now called the "Linear programming relaxation" [7]. The relaxed problem is a linear program therefore it may be solved in polynomial time by the Simplex Method.

The principle remained but the details changed in the last decades. While Simonnard used the dual simplex algorithm to generate the additional equation in [1] for a small subset of LP problems, modern approaches use ether cutting plane methods or variants of Branch and Bound to chose the new equation and are applicable on a wider set of problems.

Cutting plane

Ralph Gomory proposed the cutting plan method in 1963 in [10]. This method generates the new equations directly from the solution of the LP relaxation. Disregarding rare situations this solution contains values with fractional parts. Therefore the result is not valid for the IP problem. So this solution has to be excluded by an additional constraint. The solution is "cut" away. Geometrically spoken we introduce an additional "plane" to the polyhedron with the result point of the LP on the outside, and all feasible integer solutions on the inside.
The inequality is found by chosing any equation of the solution table of the simplex method containing fractional parts. The equations are of the form

$$x_i + \sum(a_{i,j}x_j) = b_i$$

where x_i is a basic variable and x_j are nonbasic variables. All $a_{i,j}$ are now split into the integer ($A_{i,j}$) and the fractional ($\alpha_{i,j}$) parts. b_i is split the same way into B_i and β_i. This leads to the equivalent equation

$$x_i + \sum(A_{i,j}x_j) - B_i = \beta_i - \sum(\alpha_{i,j}x_j)$$

where the right side of the equation contains all fractional parts and the left side contains the integer parts. The right side is already less than 1. For all feasible integer solution the equation has to be less than or equal 0. Adding following inequality as the new constraint to the LP generates the desired cutting plane:

$$\sum(\alpha_{i,j}x_j) \leq \beta_i$$

Of course this inequality can be again transformed into an equality by introducing an additional slack variable x_h as proposed in chapter2.3:

$$x_h + \sum(\alpha_{i,j}x_j) = \beta_i$$

Branch and Bound

The Branch and Bound was formulated by A.H. Land and A.G. Doig in 1960 in [17]. It is a general method that can be applied to a huge variety of optimization problems. The basic idea is to divide the problem into subproblems by introducing an additional boundary for one variable. To solve the subproblems they may be split up into subproblems again.

For IP problems one variable with fractional parts is chosen, lets call it x with the value v. The boundary for the branching is an additional inequality constraining the value of x. For one branch the inequality $x \leq \lfloor v \rfloor$ and for the other branch the inequality $x \geq \lceil v \rceil$ is added to the problem. The process is repeated for the subproblems until ether the subproblem becomes infeasible or the integer solution was found. Since there is just one optimal solution for feasible problems, all branches except one terminate with an infeasible problem.

There are several variants of Branch and Bound. The so called Branch and Price creates just one new branch, calculating in prior which inequality leads to infeasible subproblems. The Branch and Cut method combines Branch and Bound with Cutting Planes. Additional to bounding the solution by the LP relaxation solution cutting planes are created during the branching to reduce the fractional solutions.

CPLEX®

CPLEX®is *the* leading solver library for linear programming and integer linear programming. Developed by Robert E. Bixby in 1988 it is nowadays developed and offered by IBM.

> "IBM®ILOG®CPLEX®offers C, C++, Java, .NET, and Python libraries that solve linear programming (LP) and related problems. Specifically, it solves linearly or quadratically constrained optimization problems where the objective to be optimized can be expressed as a linear function or a convex quadratic function." [14]

It uses complex methods as Branch and Bound, Branch and Cut, Primal-Dual-Simplex and Interior Point Methods to solve offered LP and ILP models. It automatically minimizes the model by eliminating unneeded non-basic rows and columns, introducing additional slack variables and combined rows, or uses other techniques to improve the model.

Since the used techniques were implemented in decades of research by the product developers, I decided to simply use CPLEX™instead of implementing the solvers myself.

2.5 General Purpose Computation on Graphics Processing Units

The technique of *General Purpose Computation on Graphics Processing Units* (GPGPU) is a development of the last 10 years. As the name suggest it is used to run general purpose algorithms on Graphic Processing Units (GPUs), such as the normal graphic cards inside almost any computer.

The development of graphical devices and there core GPUs have increased intensively in the last decades. Starting at early 80s with first graphic hardware capable for 3D graphics, devices have got smarter, faster and cheaper every year. "In the early 1980s to the late 1990s, the leading performance graphics hardware was fixed-function pipelines that were configurable but not programmable" [16]. The fixed pipeline with fixed instructions, buffers and channels (parallel processing) made it possible to perform most graphic algorithms in little time. At the same time major graphics application programming interfaces (API) libraries came into existence. Programmers did not have to deal with basic instructions anymore, they used advanced frameworks. One such API is the proprietary DirectX™API of Microsoft with the Direct3D component. The other popular API is OpenGL®, "an open standard API supported by multiple vendors" [16].

At the start of the new millennium programmable general shaders hit the market. [16] With DirectX 8 and OpenGL vertex shader extensions it was now possible to put personalized stages into the fixed pipeline. Especially the pixel shader and the vertex shader were made customizable. A vertex is one corner of a polygon. The vertex shader reads the 3D position of a vertex and computes the screen position. Pixel shader consist of geometry shaders dealing with multiple vertices and color shaders. "For all three types of graphics shader programs, program instances can be run in parallel, because each works on independent data, produces independent results, and has no side effects" [16]

At 2005, the GeForce 8800 GPU hit the market with a changed hardware design. The separate graphics stages have been transformed to a parallel array of unified processors. "The Geforce 8800 was designed with DirectX 10 in mind." [16]. In DirectX 10 all shaders were unified, there was no difference between vertex and pixel shaders anymore. Every shader may compute anything. First time in history, GPGPU General Purpose Computing on GPU was possible. But since the only way to feed the problem into GPU was textures and polygons, GPGPU at this stage was quite awkward. "Furthermore, the only way to get a result from one pass of computation to the next was to write all parallel results to a pixel frame buffer" [16].

As the interest in such GPGPU raised, several companies and vendors stepped in to develop ways and frameworks for GPU programming aside from OpenGL®and DirectX™. Mainly the 2 biggest GPU vendors AMD (former ATI) and NVidia managed to get this done, due to their direct access to GPU specifications and influence on GPU development.

AMD Accelerated Parallel Processing (APP)

Formerly known as ATI stream, the technology was first intended to accelerate ATI's video ripping tool. The architecture is close to hardware, using explicit buffers and GPU calls. As in most older GPU programming SDKs[1] methods concentrate on textures and graphics. Additionally there are many video manipulation methods, used to transform videos quickly from one encoding to another. In late 2006 AMD bought ATI and the whole technology was transformed

[1]Software Development Kit

into a set of acceleration techniques. The latest version integrate the support for OpenCL [2], that I describe later.

Compute Unified Device Architecture (CUDA)

In November 2006, NVIDIA introduced CUDA™, a general purpose parallel computing architecture [20]. It uses extended C syntax to call Kernel functions on NVidia's graphical devices. Due to the close bound to NVidia devices, it is not portable to products of any other vendors. As an advantage the internal methods could be automatically optimized to specific hardware characteristics. Further information and details I will be describes at Chapter 3.

OpenCL

"OpenCL™ is the first open, royalty-free standard for cross-platform, parallel programming" [11]. OpenCL is a open standard for GPU programming [4]. Originally developed by Apple, it was finally released by a group of companies including NVidia, AMD, Intel and IBM. It is now maintained by the Khronos Group, a consortium of companies.[2]
OpenCL's strength is its portability. It works on almost every modern graphics device, as thow of NVidia, Intel, AMD [2] and VIA [15]. The syntax is close to OpenGL, that is used for graphic programming.

Other frameworks

Beside these three frameworks there are many others. I will mention only the two most widely used. Beside of these there also exist Frameworks and Graphic Programming Languages developed for special purposes as Fast Fourier Transformation or Statistics. And others are developed as wrappers for the large frameworks, to make them easier to understand or use.

OpenAAC

OpenAAC is a framework developed 2011 by PGI, Cray, NVidia and Caps. Is purpose is to *"enable the millions of scientific and technical Fortran and C programmers to easily take advantage of the transformative power of heterogeneous CPU/GPU computing systems."* [23]. It uses some simple hints called *directives* inside common C or Fortran programs. These hints tell the compiler to transform the code in between to methods and codes running on GPU [23].

C++AMP and DirectCompute

Microsoft has released a SDK for GPGPU as part of DirectX 11. Starting with DirectX 10.1, Microsoft introduced the so called "Compute Shaders". [6]. Previously shaders were used for graphic calculations like textures, lighting or surface manipulations. Direct X 10.1 made it possible to use Shaders for all purposes.

[2]Khronos Group was founded in January 2000. It is a consortium of companies including AMD, Intel, IBM, Samsung, Accenture, Creative, HP, EA, Opera, Sony, Nokia and many others.

C++ Accelerated Massive Parallelism (C++ AMP) is an open framework by Microsoft. It enables C++ developer to use DirectCompute for GPGPU. It simplifies DirectCompute techniques, capsules the initialization and copy methods to and from the graphics device [6].

CHAPTER 3

Compute Unified Device Architecture

The Compute Unified Device Architecture (CUDA$^{\text{TM}}$) is developed by the NVidia Corparation. It was presented 2006 as the first GPGPU API on the market [8].

3.1 Introduction and History of CUDA

Before 2006 all efforts to compute algorithms on GPUs had to be done by using graphic engines with textures and shaders. Starting with CUDA$^{\text{TM}}$ 1.0 developers were now able to directly pass variables from and to the GPU memory and perform C code directly on the GPU. Since still separated from any calculation on the CPU, to a CUDA developer the computing system now consists of two parts, the *host*, which is a traditional CPU such as a microprocessor in personal computers today, and one or more *devices*, the GPUs. Programming a host method is standard, CUDA makes it possible to call code on devices as normal c functions. Programming a device is much different.

Device Functions (or kernel function, or simply: kernels) typically use a large number of threads. In contrast to CPU threads that typically require ”thousands of clock cycles to generate and schedule” [16], GPU threads are operated with few cycles of overhead due to efficient hardware support. Another difference is, that all threads execute the same kernel, the same code. There are just some predefined constants to identify the thread among the others. I will describe this in section 3.2.

Devices have no access to main memory. All data has to be transferred from main memory (or host memory) to device memory and back. I explain this in section 3.3.

While those facts stay solid over all the different CUDA versions, some other changes and improvements have been done. All versions of the framework are backward compatible, so an application for Cuda 1 will also work on Cuda 4.

3.2 CUDA Threads organization

Threads are organized in a two-level hierarchy of threads and blocks [16]. As mentioned in section 3.4 all threads execute the same code, they have to “rely on unique coordinates to distinguish themselves from each other” [16]. In CUDA there is one coordinate for the thread inside the block `threadIdx`, and one coordinate for the block inside the grid `blockIdx`. These coordinates are built in variables automatically assigned by the CUDA runtime system accessible inside any kernel function.

The variable `threadIdx` is a 3-component vector with index `x`, `y` and `z` [20]. Thats the reason why the index is called *coordinate*. The dimension of the vector is declared at kernel function definition inside the `<<<...>>>` syntax. The parameters given can be of type `int` or `dim3`. [20] A short example is given in algorithm 3.1. Inside kernel functions the dimension can be accessed via the built-in variables `blockDim`, also a 3-component vector. So the final 1-dimensional thread ID can be computed using the following function:

```
int threadIDz = threadIdx.z;
int threadIDy = threadIdx.y + blockDim.y * threadIDz;
int threadID = threadIdx.x + blockDim.x * threadIDy;
```

The maximal block dimensions are fixed on CUDA. There are at most $512 * 512 * 512$ threads in one block.

Blocks are organized the same way. The built-in 3-component vectors `blockIdx` and `gridDim` give the coordinate of the block inside the grid and the dimension of the block vectors.

On the physical layer, one cuda GPU contains several Multi Processor Units (MPU), with multiple cores. The exact number depends on the GPU model. The NVidia GTX 680 contains 1536 cores [21]. Several blocks can be assigned to each MPU, also known as streaming multiprocessor (SM) as long as there are enough resources. The number can be dynamically changed by the CUDA runtime. So the maximum amount of blocks assigned to MPUs is limited, but some grids may contain many more blocks. And some MPUs may be used for graphics at the same time it performs the kernel. The runtime system maintains a list of blocks that are suspended, as on CPU, until other blocks have completed and new blocks may be assigned to MPUs.

Thread Snychronization

Threads inside a block do not depend on each other. It is unpredictable in which order they are executed. Their is a possibility that one thread finishes its computation before another thread has even started. For many problems some control over thread execution is needed. Especially if some thread has to read some information other threads have calculated before.

One way would be to end the kernel function and switch back to host. The calculation waits until all threads and all blocks have finished. Then the computation may switch back to device again. Of course such a switch costs time, especially because it is impossible to store data on devices between different runs. But this method is the only way to synchronize blocks.

The second method uses `__syncthreads();` function introduced in Cuda 1.1. It is a barrier synchronization function for all threads inside one block. Calls to this function blocks the current thread, until all threads inside the same block have reached this line of code. It is a barrier that ensures that no thread passes the line of code until all threads have completed the code before. Two phase iterations can be realized this way, one phase to read values and one phase to write new values back.
It is important that *all* threads have to reach this exact line of code. If the synchronization is used inside an if-clause some threads never reach, the other threads will be blocked forever. If the synchronization is used inside an if-clause in the if-path and the else-path as well, all threads will wait for each other forever on different barrier synchronization points.

Atomic Operations

Atomic Operations were introduced in CUDA 2.0. ”An atomic function performs a read-modify-write atomic operation on one 32-bit or 64-bit word residing in global or shared memory.” [20] For example, `atomicInc()` reads a variable, increment it by one and writes it back. The operation is done in an atomic way, so it is garanteed, that the variable was incremented by exactly one. At non atomic functions, it is possible that two concurrent threads read the variable, get the same value x, and write the same value $x + 1$ back. At the end the variable has the value $x + 1$, while it should have $x + 2$. A full list of atomic functions can be found in the ”NVIDIA CUDA C Programming Guide” [20]

3.3 CUDA Memory organization

Since the GPU has no access to the main memory, all the data for computation has to be stored on the device itself. Besides the local memory and registers of each thread, there are 3 different types of memory on a GPU. The types are shown in figure 3.1.

Shared Memory

Shared memory is a small memory directly at the multiprocessors. It is on-chip memory and is common to all streaming processors inside multiprocessors. All threads in one block share the same shared memory. The access to shared memory takes only 2 clock cycles compared with approximately 300 clock cycles for global memory [18]. Shared memory cannot be addressed by the host and is automatically cleared after its block has finished execution. It can be compared with L2-cache on CPU and best fits for intermediate results and shared control variables.

Global Memory

Global memory on graphic devices have experienced the same fast development as graphic processors. Modern graphic cards as the Geforce GTX 680 have 2048MB of GDDR5 chips on board [21]. Global memory can be accessed from host. The Communication of parameters and results is done by copying data from and to global memory. All threads in all blocks can access the whole global memory. Accessing it takes much more time than on shared memory,

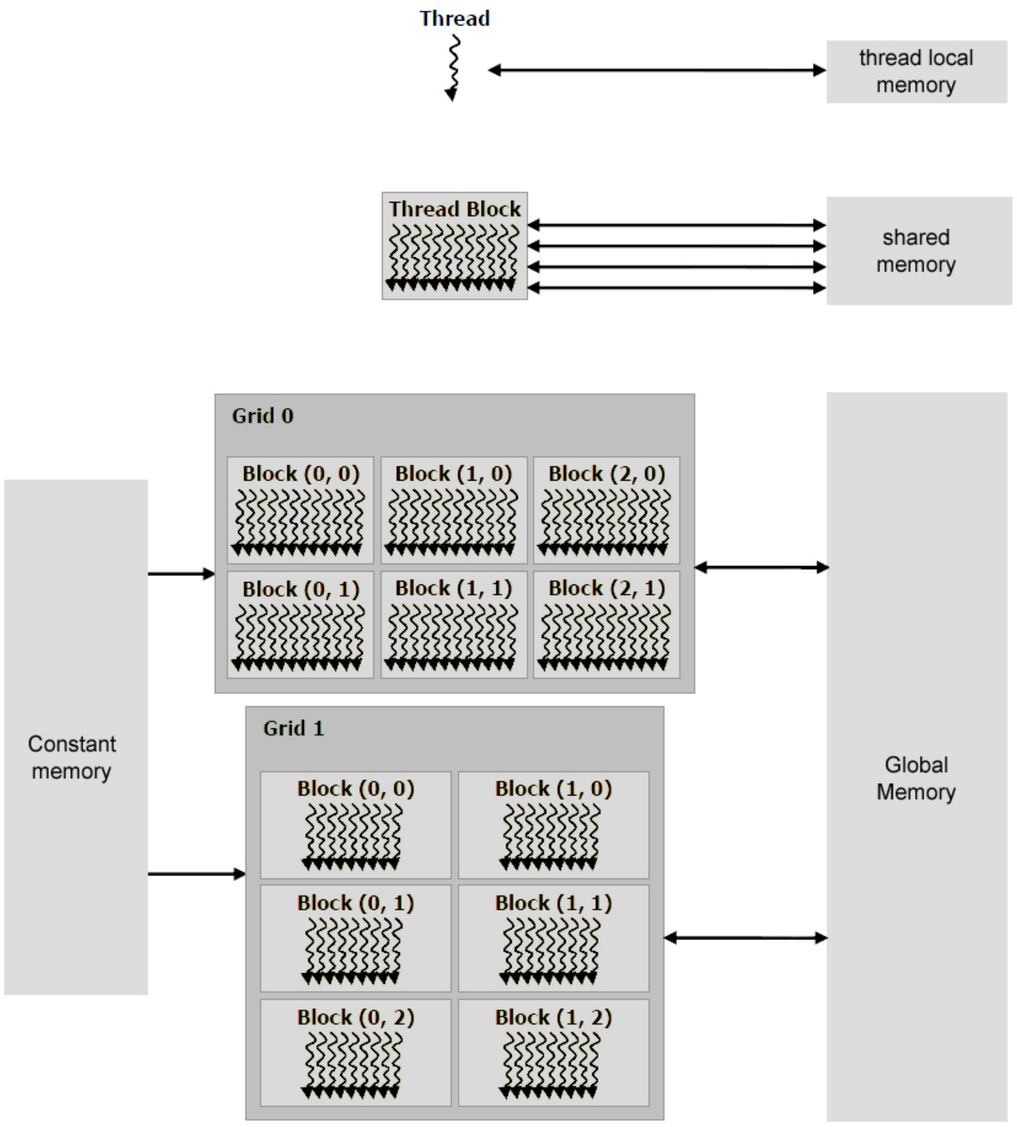


Figure 3.1: Memory Hierarchy on CUDA devices
Source: NVIDIA CUDA C Programming Guide [20]; Page 23

approximately 300 clocks. The device memory is accessed via 32-, 64-, or 128-byte memory transactions [20], the order of memory accessing is important to achieve *memory coalescing*. Adjacent blocks shall access adjacent memory at the same time, so different data can be loaded at once. For devices of compute capability 2.x and higher, the memory transactions are cached to reduce the need for manual access ordering.

Constant Memory

Constant memory is a read-only memory common for the whole device. It stores a limited amount of data, once written through host method. It is cached and since there is no need for write synchronization access it is faster than global memory. It best fits for large sets of parameters or constants common for all threads.

3.4 Starting a Kernel Function

Listing 3.1: Simple CUDA example with matrix addition

```
// Device code
__global__ void VecAdd(float* A, float* B, float* C, int N)
{
        int i = blockDim.x * blockIdx.x + threadIdx.x;
        if (i < N)
                C[i] = A[i] + B[i];
}

// Host code
int main()
{
        int N = ...;
        size_t size = N * sizeof(float);
        // Allocate input vectors h_A and h_B
        // in host memory
        float* h_A = (float*)malloc(size);
        float* h_B = (float*)malloc(size);

        // ...
        // DO Initialize input vectors
        // ...

        // Allocate vectors in device memory
        float* d_A;
        cudaMalloc(&d_A, size);
        float* d_B;
        cudaMalloc(&d_B, size);
        float* d_C;
        cudaMalloc(&d_C, size);

        // Copy vectors from host memory to device memory
        cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
        cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
        // Invoke kernel (threadsPerBlock, blockPerGrid)
        int tpb = 256;
        int bpg = (N + threadsPerBlock - 1) / threadsPerBlock;
        VecAdd<<<bpg, tpb>>>(d_A, d_B, d_C, N);

        // Copy result from device memory to host memory
        // h_C contains the result in host memory
        cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

        // ...
        //  DO something with result
        // ...

        // Free device memory
        cudaFree(d_A);
        cudaFree(d_B);
        cudaFree(d_C);
        // Free host memory ...
        free(h_A);
        free(h_B);
        free(h_C);
}
```

A *Kernel function* is a function, that is run on the GPU. Listing 3.1 gives an example with an algorithm that stores the sum of elements of two arrays into a third array. It is designed to give an easy to understand introduction to kernel function, although doing nothing for our problem.

The function *MatAdd* is the kernel function, it is executed by GPU. This is indicated by the keyword `__global__` at the function header. This keyword also indicates that the function can be called from host, and its parameters are stored in global memory. (See section 3.3). As described in section 3.2 one thread does one single addition, first calculating his index with `int i = blockDim.x * blockIdx.x + threadIdx.x;`.

The invokations of the kernel function is done by the `<<<...>>>` syntax [20]. Inside the triple bracket two variables or constants give the amount of blocks and threads calculating the kernel function. In GPGPU it is a common practice to create more threads than needed. At the example there are a multiple of 512 threads created. Even if `N` equals 513, there are 2 blocks used with 1024 threads. Unnecessary threads just do nothing.

The allocation of variables in memory of cuda devices is done with `cudaMalloc()` and freed with `cudaFree()`. Variables passed to the device have to be explicitly copied to device memory before and copied back afterwards. This is done with the `cudaMemcpy()` function. Notice: Their is no "Segmentation fault" exception on devices. Copying *too much* data into a device variable or reading from outside the range of an array causes to undefined behavior!

Listing 3.2: CUDA example calculating maximum value with reduction code

```
// just calculates the maximum value
__device__ int maxReduce(int *array, int n, int tn)
{
        __shared__ int max[N];
        int arrayIndex = threadIdx.y * blockDim.x + threadIdx.x;
        if (arrayIndex < n)
        {
                max[arrayIndex] = array[arrayIndex];
        }
        __syncthreads();

        int nTotalThreads = tn;

        while(nTotalThreads > 1)
        {
                // divide by two
                int halfPoint = (nTotalThreads >> 1);
                int otherIndex = arrayIndex + halfPoint;

                // only the first half of the threads is active.
                if (arrayIndex < halfPoint && otherIndex < n)
                {
                        // Get the value stored by another thread
                        // and update own value if needed
                        int next = max[arrayIndex + halfPoint];
```

```
                    if (next > max[arrayIndex])
                    {
                            max[arrayIndex] = next;
                    }
            }
            __syncthreads();

            nTotalThreads = halfPoint; //divide by two.
    }
    return max[0];
}
```

Reduction

A further common method to calculate values out of arrays is the *Reduction* method. Examples for such values are the maximum and minimum and the average of all elements in an array. The calculation uses synchronized threads. The method starts with $n/2$ threads, where n is the size of the array; It should be a exponent of 2. ($n = 2^z; z \in Z$) Every thread calculates the amount for the value at his index i and at the index $i + n/2$. Then the threads are synchronized, n is divided by 2 and the calculation is repeated. The method ends when $n = 1$. The value at index 0 is the result.

At listing 3.2 I show example code that calculates the maximum value out of an array. This example first copies the array to shared memory to speedup the calculation. Then it iterates the calculation and splits the amount of working threads into half until the last thread 0 returns the result.

3.5 Decision for CUDA

It seamed a good choice to use OpenCL for my application. The comparison of OpenCL and CUDA made by Jianbin Fang, A.L Varbanescu and H. Sips [8] also mentions, that there is no real disadvantage of OpenCL. Nether less at last I decided against it and for CUDA.
CUDA offered a easy-to-understand syntax, many tutorials and a large community at the website of NVidia. There is no bondage to graphic methods and textures and an easy-to-use compiler with proper debug information.
The disadvantage to be bound to NVidia devices was irrelevant as all computers at my working place, the institute and at home already contained NVidia Geforce™ devices.

CHAPTER 4

Algorithm

In this chapter I present my algorithm for solving the CLD problem for arbitrary k. The algorithm was developed in C using CUDA and CPLEX$^{\text{TM}}$. It uses the methods described in chapter 2.

First of all the algorithm reads the problem instance containing the graph from file. The graph is stored as adjacency matrix. While using more storage as adjacency vectors and lists, the matrix is more efficient on adding new edges or removing edges. As another advantage, the matrix can be easily passed to GPU.

After setting up and loading the graph, the algorithm starts with its 2-step approach to solve the problem. A very coarse-grained overview is given in Algorithm 4.1.

```
Input: matrix G: containing the Adjacency Matrix
Input: num k: Amount of maximal concurrently failing edges
Input: num D: maximal diameter that shall not be exceeded
Result: Set of critical Edges
1 begin
2     edges ⟵ findAllTuplesOfCriticalEdges(G, D, k);
3     result ⟵ solveILP(buildUpModel(G, edges));
4 end
```

Algorithm 4.1: Short overview. After loading the graph, the algorithm solves the problem in 2-steps.

The first step operates on the matrix and searches for all tuples of edges that increase the diameter over the given border D. The second step builds up an ILP model, solves it and returns the set of edges that have to be protected.

4.1 Finding critical edges

The search for set of possibly critical edges is done in several steps. The next subsections describe these steps and some ideas to reduce the complexity of the problem.

Preprocessing

Removing edges on a graph always increases the shortest path between some of the graph's vertices. As presented in section 1.1 removing some single edges may also increase the diameter of the whole graph. If the diameter exceeds the given parameter D the removed single edge has to be protected. The algorithm is based on the paper [9] written by Fujimura and Miwa.

```
Input: graph G(V,E) with vertices V and edges E
Input: num D: maximal diameter that shall not be exceeded
Result: Set of critical Edges
1  Function SearchCriticalEdges(G,D)
2  begin
3      $E_p \longleftarrow \emptyset$;
4      foreach $e \in E$ do
5          $d_e \longleftarrow$ diameter($G(V, E \setminus \{e\})$);
6          if $d_e > D$ then
7              $E_p \longleftarrow E_p \cap \{e\}$;
8          end
9      end
10     return $E_p$.
11 end
```

Algorithm 4.2: Solving the problem for k=1 in polynomial time.

The algorithm for $k = 1$ presented in algorithm 4.2 is solved in linear iterations and polynomial time [9]. They also prove that the problem for $k > 2$ is *NP-complete*. So I run the presented algorithm as preprocessing task, reducing the problem difficulty of the ILP model while extending the runtime with just reasonable small amount of time.

Excluding tuples of edges

Allowing several edges to be removed at the same time leads to significant increase of complexity. Iterating over all single edges is possible in linear time. Testing all combinations of k edges of a graph with m edges needs $\binom{m}{k}$ checks. So the question arises if all tuples of edges have to be visited. There are several ideas that suggest a reduction of iterations.

1. Vertices with *low degree*. As shown in the example in figure 4.1 vertices with degree less or equal k are disconnected if all edges around fail at the same time.

2. Edges in *fully connected subgraphs* as triangles form redundant paths for each other. Removing k edges from a fully connected subgraph with $k + 2$ nodes may never increase the diameter by more than 1.

3. Edges that are *most visited during diameter calculation*. Calculating the APSP (see section 2.2) and considering the visited edges, the edges with the most "traffic" are likely to increase the diameter on removal.

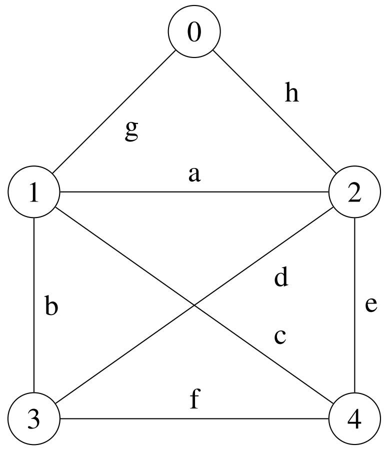


Figure 4.1: Graph with diameter of 2. For $k = 1$ the diameter remains 2. For $k = 2$ the graph may split into two disconnected subgraphs.

The first idea is helpful on some vertices. At least one edges of a vertex with low degree less or equal to k has to be protected in any case. These Tuples of edges are directly included, without the need to check the diameter of the remaining graph. But in practice this method decreases the performance of the overall algorithm. All tuples have to be checked if they have one vertex in common and the percentage of positive tuples in an average graph is far to low.

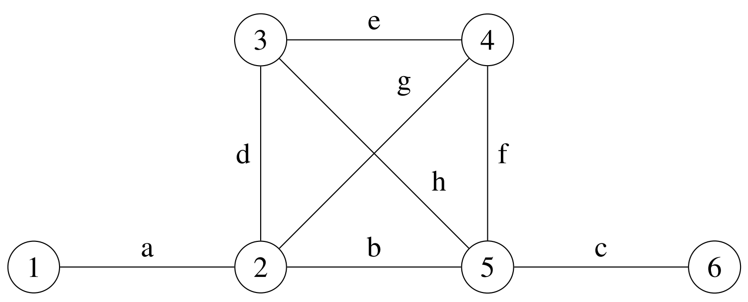


Figure 4.2: Graph with diameter of 3. For $k = 1$ the graph becomes disconnected. After protecting edge a and c the diameter is still 4. For $D = 3$ the edges $\{a, b, c\}$ must be protected.

The second idea exclude tuples of edges that just contain edges from the subgraphs and none from outside. Other tuples may already increase the diameter to D after removing the edges outside the subgraph and then send the diameter to $D + 1$ when removing the edge(s) from inside. Figure 4.2 demonstrate a simple example, showing a complete graph with 4 vertices extended by two edges leading to single connected vertices. One edge of the fully connected graph needs to be included to the set of protected edges if D is set to 3.

Testing tuples of edges

All possible combinations of tuples of edges have to be checked if the diameter exceeds D if they are completely removed from the graph. This is done the same way as in the preprocessing step, by removing those edges, calculating and comparing the diameter of the remaining graph. Algorithm 4.3 shows this process for any Graph and $k = 2$.

```
Input: graph $G(V, E)$ with vertices $V$ and edges $E$
Input: num $D$: maximal diameter that shall not be exceeded
Result: Set of critical Edges
Function Search2TuplesForProtectedSets($G, D$)
begin
    $E_p \longleftarrow \emptyset$;
    foreach $a \in E$ do
        $E_a \longleftarrow E \setminus \{a\}$;
        foreach $b \in E_a$ do
            $E_b \longleftarrow E_a \setminus b$;
            $d \longleftarrow$ diameter($G(V, E_b)$);
            if $d > D$ then
                $E_p \longleftarrow E_p \cup \{\{a, b\}\}$;
            end
        end
    end
    return $E_p$.
end
```

Algorithm 4.3: Iterating over all tuples of 2 edges and checking the diameters.

Since k may be set to any number less than D, writing such procedures for every k is impossible, at least insufficient. So the algorithm is rewritten to use recursion to iterate from 1 to k. The result is presented in 4.4. This algorithm may be transformed back into an iterative form using an explicit stack for the current state of the algorithm including removed edges and the reduced graph.

Calculating the diameter (on CPU)

As mentioned before the diameter is nothing than the APSP. Using the CPU one of the fastest and yet simplest method is to use the algorithm by Floyd and Warshall, described in chapter 2.2. In algorithm 4.5 I show pseudocode for the implementation used in my algorithm.

4.2 Build up and solve the model

Now that we have found all the tuples of critical edges, we have to select the edges that we have to protect. Lets take a look on figure 4.1 and the graph shown in this example. Using the algorithms of section 4.1 with parameters $k = 2$ and $D = 2$ we get to know that if we remove

```
Input: graph G(V,E) with vertices V and edges E
Input: num D: maximal diameter that shall not be exceeded
Input: num k: number of edges that are removed
Input: set R: set of edges that are already removed, initially $\emptyset$
Result: Set of sets of critical edges
1  Function RecursiveSearchProtectedSets($G, D, k, R$)
2  begin
3      $E_p \longleftarrow \emptyset$;
4      foreach $e \in E$ do
5          $E' \longleftarrow E \setminus \{e\}$;
6          $R' \longleftarrow R \cup \{e\}$;
7          if $k > 1$ then
8              $E_p \longleftarrow E_p \cup$ RecursiveSearchProtectedSets($G(V, E'), D, k - 1, R'$);
9          else
10             $d \longleftarrow$ diameter($G(V, E')$);
11             if $d > D$ then
12                 $E_p \longleftarrow E_p \cup \{R'\}$;
13             end
14         end
15     end
16     return $E_p$.
17 end
```

Algorithm 4.4: Iterating over all tuples of k edges using recursion.

the tuple $\{(0, 1), (0, 2)\}$ the diameter of the reduced graph exceeds D. But it is enough to protect just *one* edge of this group. Protecting for example edge $(0, 1)$ prevents the graph from splitting and all other vertices may be reached again from vertex 0 in 2 steps or less.
That holds true for this example, at other graphs it may be necessary to protect all 2 edges of the tuple. On problems with k simultaneously failing edges, the range of necessarily protected edges in a tuple may vary between 1 and k.
This selection problem is solved by building up and solving an ILP model. This I will present in the next two subsections.

Building up the model

In fact given the list of all tuples of critical edges the creation of the model is rather straight forward. Such a model can be presented by the following table:

```
Input: graph $G(V, E)$ with vertices $V$ and edges $E$
Data: matrix $D$: matrix with $|V| \times |V|$ elements
Result: Diameter of the graph $G$
Function FloydWarhsall($G$, $D$)
begin
    foreach $v \in V$ do
        $D_{v,v} \longleftarrow 0$;
        foreach $u \in V \setminus \{v\}$ do
            $D_{v,u} \longleftarrow \infty$;
        end
    end
    foreach $k \in V$ do
        foreach $u \in V$ do
            foreach $v \in V$ do
                if $D_{u,k} + D_{k,v} < D_{u,v}$ then
                    $D_{u,v} \longleftarrow D_{u,k} + D_{k,v}$;
                end
            end
        end
    end
    return maximum($D$).
end
```

Algorithm 4.5: Solving APSP using Floyd-Warshall.

$$\begin{aligned}
\text{Min } z = \sum_{a \in E} p_a & \\
p_b = 1 & \qquad \forall b \in E_s \\
\sum_{c \in T} p_c \geq 1 & \qquad \forall T \in E_p \\
p_d \in \{0, 1\} & \qquad \forall d \in E
\end{aligned}$$

where E_s is the set of critical edges found at preprocessing for $k = 1$ and E_p is the set of set of edges that are found for $k > 1$.

Every edge i is given a binary variable p_i that is one if the edge has to be protected, otherwise it is 0. As discussed in section 2.4 that makes the problem an ILP problem. The objective function is to minimize the size of the solution set that is the amount of protected edges. The tuples with single edges resulting from the preprocessing have to protected for sure. For every those edges in P_s an equation is added to the model fixing the p_i at the constant value one. For all tuples with multiple edges at least one edge has to be protected. The sum over all p of every

$T \in P_t$ has to be at least one.

Solving the model

The model is solved using CPLEX®which is explained in section 2.4. It uses some advanced versions of the methods explained in subsection 2.4. I don't want to explain that in detail.

More important is the interpretation of the solution. After a optimal solution is found CPLEX®returns the objective value and the list of final variable assignments. The objective value equals the amount of protected edges. The variable assignments shows us which edges have to be protected.

If there are several optimal solutions, as shown in the example for figure 4.1, CPLEX®shows us the first solution it finds. Explicitly excluding the solution leads to the next optimal solution, until the model gets infeasible. To exclude a solution S the inequality

$$\sum_{e \in S} p_e \leq 1$$

is added to the model.

4.3 Acceleration with diameter calculation on CUDA

As Fujimura and Miwa proved, the problem of solving the CLD for $k > 2$ is *NP-complete* [9]. After implementing the first version of the algorithm I noticed the impact on concrete running time of the algorithm. Profiling the process it was soon clear that most of the time is consumed at the preparation of the model. For example the overall runtime for an instance with 100 vertices and 200 edges (inst100x200) was roughly *50 seconds* excluding loading the instance from file. Solving the model with CPLEX took only around one second, thats just 2% of the overall runtime. Therefore the biggest speedup can be achieved on preparation. As I already mentioned at the previous section some ideas for changing the model building algorithm lead nowhere. So I concentrated on these parts of the algorithm.

For the given example of inst100x200 the algorithm does 17793 diameter calculations consuming almost 90% of calculation time. The idea now is to accelerate the diameter calculation. According to *Amdahl's law* and Annaratone's interpretation in [3], parallize the algorithm will improve the performance significately. Annaratone specialized Amdahl's law for massive parallel processors (MPP). Such MPP's are current Graphical processing units, as described at chaper 2.5.

Floyd Warshall on GPU

The Floyd-Warshall as presented in chapter 2.2 and chapter 17 is successful on single processors finding the APSP in short time. The algorithm uses three lopps. One loop can be replaced by concurrent calculation. Nevertheless its adaptability to multiple processors is limited.

The problem is the high memory consumption of the algorithm. Every iteration the algorithm updates the distance value for some of its edges. So besides the adjacent matrix to represent the matrix the algorithm needs another matrix for current distance values. As presented in section 3.3 memory on GPU is limited. The adjacent matrix may be stored in constant memory. The distance matrix is written by every thread at the same time, so it needs to be stored in global memory. Global memory is slow. Running several Floyd-Warshall at once increases the amount of needed memory space.

APSP using SSSP on GPU

Another approach uses a concept proposed by Tomohiro Okuyama, Fumihiko Ino and Kenichi Hagihara in 2008. [22] The algorithm uses the Single Start Shortest Path (SSSP)-based iterative method [12]. It computes "an SSSP $|V|$ with varying the source vertex $s \in V$" [22]. To solve SSSP the authors implement an iterative algorithm [12] using CUDA.

> "This algorithm associates every vertex $v \in V$ with cost c_v, which represents the cost of the current shortest path from the source s to the destination v. The algorithm then minimizes every cost until converging to the optimal state." [22]

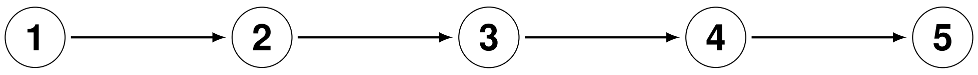


Figure 4.3: Highly degenerated graph. Diameter equals the amount of vertices minus 1.

The authors use one thread per vertex, calculating the costs for all neighbor vertices and update them, if lower costs are found.

My algorithm also uses one thread per vertex, but calculates the cost c_v for the vertex itself starting at all neighbor vertices, and updates it, if lower cost is found. This is done in a two phase iteration as described in section 3.2.

- *Initialization Phase*: All variables are initialized: `running` is set to `true` and the costs array is set to special value `NOT_VISITED`.
- *Phase 1* is used to visit all neighbor vertices and calculates the new c_v. If the new cost is lower than the old, `running` is set to indicate that updates where found.
- *Phase 2* is used to write the new costs to shared costs array. If there were no updates in this iteration, the algorithm exits the loop.
- *Exit Phase* The threads calculate the maximum cost with the reduction method described in section 3.4.

The algorithm runs for all vertices in $O(n^2)$ where n is the number of vertices in the graph. The maximum function needs $O(n^2)$ iterations. The methods inside the main loop travel all neighbors in $O(n)$ and set the local cost in $O(1)$. The main loop itself runs at most n times. This can be shown by considering the worst case: a run in highly degenerated graphs as shown in figure 4.3. Only one thread can update the cost every iteration: $n + 1$ iterations are needed until the algorithm exits the loop. The overall algorithm for one SSSP needs $O(n^2+(n+1)*(n+1)) = O(n^2)$ iterations.

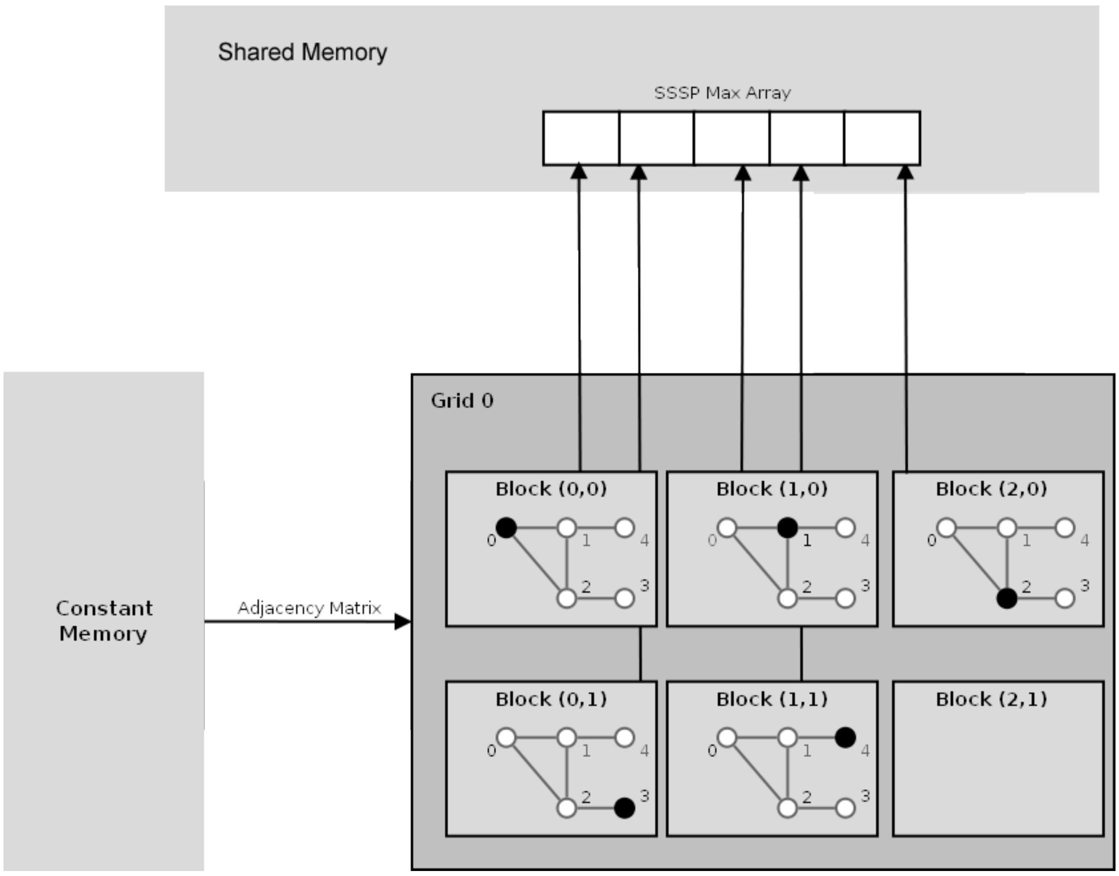


Figure 4.4: The APSP is computed by n blocks parallel by the GPU

A complete code for the diameter calculation is given in algorithm A.2 in appendix A.

For graphs with no or equal edge weights, the algorithm shows relations to the BFS algorithm of chapter 2.1. It can be shown, that the main loop runs exactly $f + 1$ times for every graph, where f is the number of frontiers in the BFS algorithm.

As Floyd-Warshall does, this method uses constant memory for the adjacent matrix. But the frontier array may be stored at shared memory which is by far faster than global memory. Comparing the two algorithms shows a speedup of almost 20% for medium sized graphs up to 3000 edges. The example chart of some experiments is presented in table 4.1.

vertices	edges	runtime APSP [s]	runtime FW [s]
20	40	0.06	0.08
40	80	1.02	1.12
75	150	14.15	15.21
100	400	110.40	118.64

Table 4.1: My implementation of APSP compared to Flowd-Warshall

Memory Coalescence

The performance on graphic devices raises and falls with memory access speed. The fastest numeric processors perform badly if they have to wait for their data to be read. Programming programs on CPUs, registers, caches and fast main memory with high potential optimizers have made modern programmers unaware of the core principles of I/O optimization. On graphic devices they have become essential again. One of this principles is explained in section 3.3 called *Memory Coalescence*.

In the algorithm's implementation every thread access the adjacent vertices of its neighbors. Since I use an adjacency matrix in constant memory these information for vertex i are stored twice. Once at column i and once at row i. Constant memory is read in buckets that contain several values. All threads read at approximately the same time. Arrays are stored in adjacent memory cells, and I have decided to store the rows of the adjacency matrix one after another in one large array. Combining these observations it is the better choice to let threads read their values from columns. So that adjacent threads read from adjacent memory cells.

Synchronization

Parallel computation depends heavily on synchronization. Besides the implicit locking on I/O operations that may be influenced only indirectly, explicit synchronization is completely in programmer's responsibility. Here I want to mention again that one single core of GPU is far slower than one core of CPU but the GPU has more of them.

As mentioned in section 3.2 there are three ways to synchronize threads using the CUDA framework. The algorithm uses all three of them for different purposes. First of all the algorithm synchronizes all blocks on all grids after the algorithm is done. There is no explicit call every time a kernel function exits the program waits for all threads on the device to finish. The other two possibilities are explicite synchronization of threads.

One early version of the algorithm used just explicit calls to `__snycThreads()`. It used a two phase approach: read values and write values. The algorithm finished if none of the threads had been written new values. This exit criteria is set in phase 1 and read in phase 2. In each iteration every thread calculates the lowest level to its corresponding vertex regarding the current levels of its neighbors. Setting this value via an atomicMin function removes the need for the second synchronize. The exit criteria is finally realized using 3 variables. On a rotating basis on is set to default value “false”, one is set to true, if any write was done and one is read. Atomic functions serializes the threads accessing the variable. Just one thread is allowed to enter the atomic function at once. The function `__snycThreads()` blocks all threads, until every thread has reached a certain line of code. Therefore the new approach with less explicit `__snycThreads()` is faster.

Calculating the max level

The result of the algorithm so far was an array of levels of the vertices. The level represents the length of the shortest path from the set start vertex. We do not need the whole array, we just need the maximal value. We could calculate this value on CPU, but we are already on the graphic device. With the algorithm in listing 3.2 mentioned in section 3.4 we possess an excellent tool to calculate the maximum value.

Memory limitations

The presented algorithm uses all kinds of memory CUDA can offer. We use shared memory for the level array, we read the adjacency matrix from constant memory and we communicate the result back to CPU by using the global memory. But for huge instances this does not work anymore. The special memories have limited sizes. The shared memory is with 16384 bytes the smallest. Storing 4-byte integers this is sufficient space to save one array with 4096 values. Constant memory is larger, it can store 65536 bytes or 16384 4-byte integers. Since I use this memory for the adjacency matrix, this border is reached at a graph with only 128 vertices.

For larger instances data has to be stored in the global memory. For adjacency matrix the impact on runtime is rather small. Global memory does not use as efficient caching techniques as constant memory, but it still performs well on reading. Calculating the current level at global memory has much greater impact. The runtime of any write operation is increased up to the factor of 30! While shared memory performs at the same level as registers, using less than 10 clocks to read, global memory requires around 300 clocks or more. In current practice graphs with more than 128 vertices are quite normal, and graphs with more than 5000 vertices are unusual. One cause that this will hold true for some time is that the overall runtime of the algorithm on such instances is measured in days and weeks, as the next chapter 5 shows.

CHAPTER 5

Benchmarks

In this chapter I provide information about the test environment, the used test cases and finally the test results.

5.1 Test Environment

The tests were run and measured on an Ubuntu Linux 3.0.0-20.34 lucid1-generic 3.0.30 operated computer. The system provided following computing devices:

- The **CPU** was a Intel®Core™i7-2600 with 4 cores on 3.40 GHz.
- The **GPU** was a single *GeForce GTX 560 Ti* with CUDA Capability 2.1, 1023 Megabytes global memory and 384 CUDA cores on 1.66 GHz.

The program was compiled using gcc version 4.4.3 and nvcc version 4.1. The GPU was operated using NVidia device driver 290.10.

Test Instances

To be able to run tests on graphs of any size and density, the test instances were automatically generated. The generator takes two parameters, first the *amount of vertices* and second the *amount of edges* in the graph. The edges are selected with standard C++ pseudo-random function by selecting one edge out of a vector of all possible edges. Finally the graph is checked to be fully connected, otherwise the whole generator starts again. The generator is far from optimal in runtime and space, but it produces any edge with equal probability, depending only on the distribution of the pseudo-random number generator.

Verifier

The results of the algorithm are sets of edges that have to be protected. For very small examples the results may be validated by humans. They look at the graphs, delete some edges and try to find any longer paths. Of course this becomes unrealistic for larger and more complicated instances. Nether the less the results of the algorithm must be verified.

Therefore the results are validated by a program. This program is very similar to the algorithm discussed in this master thesis. It load the graph, removes up to k edges and checks, if the diameter of the resulting graph exceed the given D. If the diameter exceeds D and no protected edges have been removed then one of the removed edges must be protected as well and therefore the result is invalid.

If the diameter becomes larger than D and a protected edge has been removed, this protected edge is marked. At the end, if the algorithm has checked all combinations, the result is not invalid and all protected edges are marked, then the result is optimal too. Otherwise the result is valid, but at least one edge may be removed from the protected set, so the result is not optimal.

The algorithm was implemented in a different programming language, in Java, prior to the development of the CLD algorithm. This was done to ensure that the program does not implement the same errors as the algorithm.

5.2 Test Results

I want to present the results for the runs with $k = 2$ on the test instances with custom D. The number of protected edges are presented at figure 5.1. The runtime is presented at figure 5.1 and figure 5.2.

Figure 5.1: Runtime of calculation at linear scale

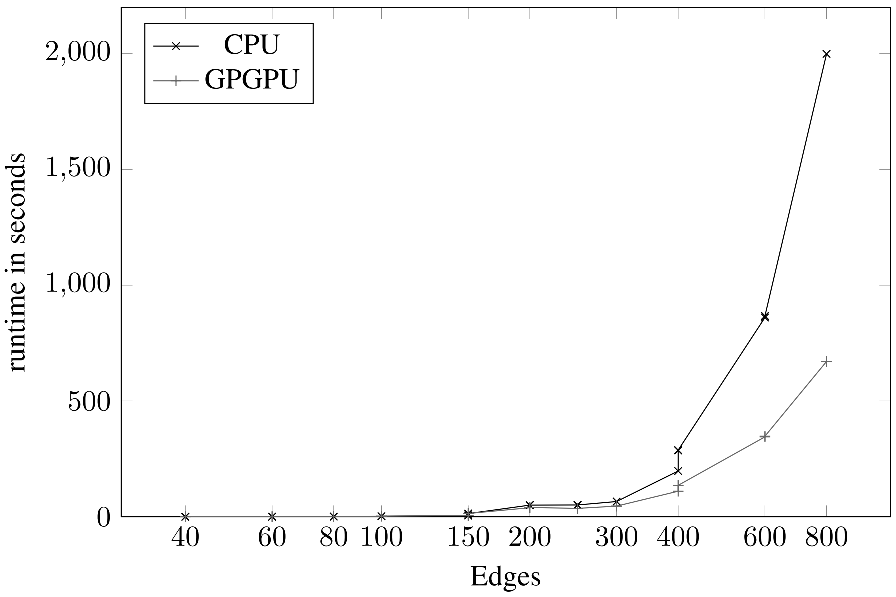


Figure 5.1 shows best the complexity of the problem. Calculating an instance with double the edges, the algorithm needs up to 10 times the runtime. On logarithmic scale figure 5.2 shows that for small instances the GPGPU implementation needs more time to compute and it

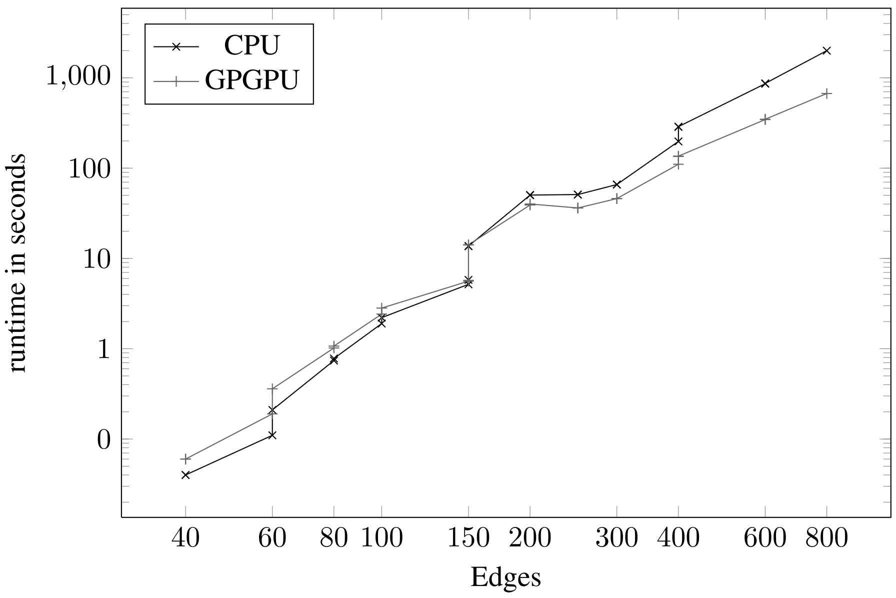
Table 5.1: The results for $k = 2$

Vertices	Edges	D	Protected	Time CPU [s]	Time GPGPU [s]	Difference [%]
20	40	4	9	0.04	0.06	50
20	40	5	4	0.04	0.06	61.72
30	60	4	32	0.11	0.19	81.05
30	60	5	8	0.21	0.36	76.14
40	80	5	16	0.74	1.02	38.42
40	80	6	7	0.78	1.07	37.32
50	100	6	24	1.9	2.42	27.03
50	100	7	13	2.21	2.82	27.56
50	150	4	13	5.17	5.6	8.32
50	150	5	4	5.79	5.72	−1.17
75	150	8	18	13.67	14.15	3.52
75	150	9	17	13.71	14.15	3.19
100	200	8	24	50.24	39.05	−22.27
125	200	7	23	50.32	39.51	−21.48
130	200	7	23	50.33	40.1	−20.35
80	250	7	2	50.99	36.21	−28.99
80	250	6	2	50.97	36.19	−29.01
75	300	4	8	65.8	46	−30.1
75	300	6	3	65.86	46	−30.15
100	400	4	16	197.57	110.4	−44.12
200	400	8	8	286.47	134.68	−52.99
200	400	9	6	287.9	135.76	−52.84
300	600	8	21	860.47	345.05	−59.9
300	600	9	10	867.45	348.01	−59.88
400	800	9	9	1,998.75	670.5	−66.45

needs less time for larger instances. The reason is the overhead of the memory initialization on CUDA devices. The whole adjacency matrix has to be loaded to device memory prior to any calculation. And after the diameter is calculated, the result has to be copied back to host memory again. If the algorithm were using the implementation of Floyd Warshall on CPU, it would finish the whole calculation in that amount of time. Tests have shown that for any instance with less than 80 vertices, the CPU is faster than the GPU. At the problem instance with 30 vertices and 60 edges the Floyd Warshall implementation on CPU was faster than the APSP algorithm on GPU (181% runtime). At 80 vertices and 250 edges the GPU was already more than 20% faster than the CPU. At 400 vertices and 800 edges the GPU finished at 670 seconds, while the CPU implementation took 1999 seconds. That is a speedup by factor three.

As explained at section 4.3, the constant memory becomes too small for graphs with more than 128 vertices. For larger graphs the adjacency matrix has to be stored in global memory. The algorithm was tested with two additional test instances, one has 125 vertices and the other has 130 vertices. Both have the same amount of edges and the same basic diameter. The change

Figure 5.2: Runtime of calculation at logarithmic scale



of runtime of the CPU algorithm is less than 0.02%. The increase of runtime of the GPGPU algorithm is about 1.4%. This shows that the change of memory has an impact on runtime, but the change is rather small.

The change of storing the frontiers-array in global memory instead of shared memory will cost more performance of the algorithm. For tested both test instances by manually switching the used memory and the algorithm took almost 10% more time for computation. But the problems only exists for instances with more than 4096 vertices. These instances already take days and weeks computing time. For such huge instances it may be wiser to use approximation methods that present an feasible solution in hours than to use the exact method to wait for the optimal solution.

Increasing the value of k increases the complexity of the problem and therefore the runtime of the program significantly. This is shown in figure 5.2 and at the diagram in figure 5.3. The change of runtime is quite impressive. For small instances the algorithm is 10 times faster for small k. The difference is not linear, so it increases for larger instances. While the problems for $k = 3$ and $k = 4$ are both NP-complete, the difference is large. As figure 5.2 shows the increase is even larger as between the problem for $k = 2$ and $k = 3$.

Figure 5.3: Runtime of calculation at logarithmic scale for $k = 2$,$k = 3$ and $k = 4$

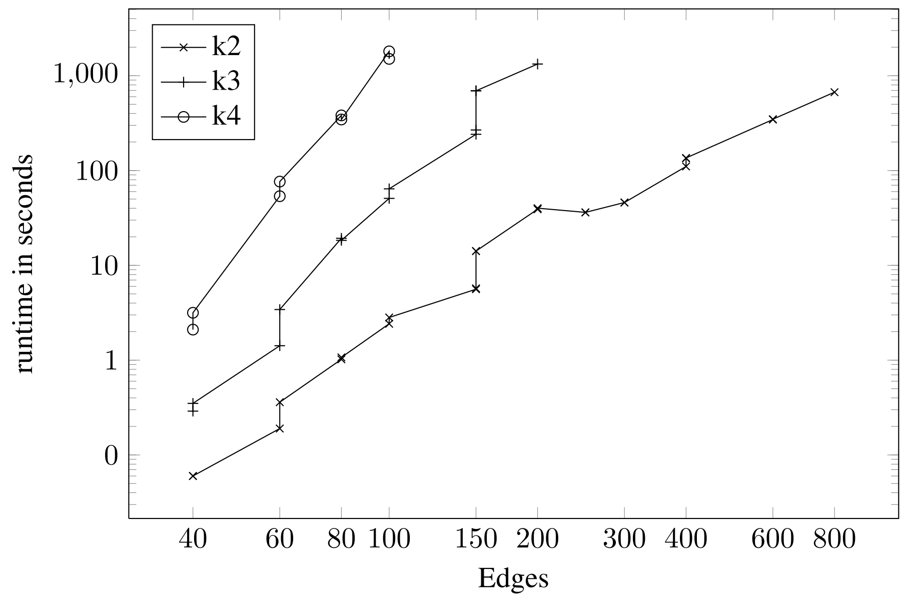


Table 5.2: The results for $k = 2$, $k = 3$ and $k = 4$. The value - indicates that this run has taken too long to complete within the timeout of 30 minutes.

Vertices	Edges	D	runtime k 2 [s]	runtime k 3 [s]	runtime k 4 [s]
20	40	4	0.06	0.29	2.1
20	40	5	0.06	0.35	3.16
30	60	42	0.19	1.42	53.62
30	60	5	0.36	3.42	76.49
40	80	5	1.02	19.3	381.55
40	80	6	1.07	18.34	345.58
50	100	6	2.42	50.85	1807.98
50	100	7	2.82	64.13	1505.93
50	150	4	5.60	241.25	-
50	150	5	5.72	267.88	-
75	150	8	14.15	694.56	-
75	150	9	14.15	693.07	-
100	200	8	39.05	1329.03	-
125	200	7	39.51	-	-
130	200	7	40.1	-	-
80	250	7	36.21	-	-
80	250	6	36.19	-	-
75	300	4	46.00	-	-
75	300	6	46.00	-	-
100	400	4	110.40	-	-
200	400	8	134.68	-	-
200	400	9	135.76	-	-
300	600	8	345.05	-	-
300	600	9	348.01	-	-
400	800	9	670.50	-	-

CHAPTER 6

Conclusion

The CLD is easy to solve for small values of $k = 1$, there is an algorithm [9] that gives a result in polynomial time. For larger k the problem's complexity raises, for $k \geq 3$ it becomes NP-complete. For those values the use of approximation method seems to be a wise choice. For $k = 2$ there is no prove yet if it's ether P or NP but it is assumed to be NP. I introduced an algorithm that solves the problem by exact methods. Using APSP and diameter calculation it constructs an ILP model that is then solved to select the set of protected edges. For $k = 2$ and small problem graph instances results are calculated within a few seconds. On larger instances the pure CPU algorithm runs a very long time.

Therefore I show that with the use of GPGPU on CUDA devices the diameter calculation, an essential part of the algorithm, can be speed up. Furthermore I use some advanced techniques such as memory coalescence, map/reduce and reducing thread synchronization points. For small instances this speedup is swept away by the overhead for CUDA kernel methods. The larger the problem instance the faster the GPGPU algorithm is compared to CPU implementation. While for the smallest instance with 60 edges the GPU needed 50% more time than the CPU, it took over at 200 edges and at 800 edges it was already finished in 33%.

When increasing k the complexity of the problem rises. While solving a problem with 100 of edges for $k = 2$ in 2.8 seconds, the algorithm solves the same instance for $k = 3$ in 64.1 seconds and in 1505.9 seconds for $k = 4$. An instance with 200 edges is solved in 39 seconds for $k = 2$, for $k = 3$ it calculates 25 minutes and needs more than 2 hours for $k = 4$.

While the algorithm may be used in practice to solve real world problems for $k = 2$, further improvements and researches have to be done for larger k. Other parts of the algorithm may be adopted to use GPGPU as well. Other APSP algorithms have to be tested for using them on GPU. Future releases of CUDA and other frameworks will offer new methods and techniques to be used on graphs. And finally complete different approaches as to use heuristics and approximation techniques have to be regarded to get results for large instances and higher k in minutes and hours.

APPENDIX A

Cuda Diameter

The following code is used to calculate the diameter of a graph via GPGPU. The code is written in *C* with some extensions to handle the CUDA specific issues. Further information about the code can be found in chapter 4.3.
The following code A.1 sets up the device memory and copies the adjacency matrix of the graph. It reserves the given amount of threads and blocks and starts the calculation.

Listing A.1: CUDA example calculating maximum value with reduction code

```
int cuda_apsp_wrapper(int *adj, int vertices) {
  assert(vertices <= MAX_VERTICES);

  int *h_levels;
  int *d_levels;
  int i;

  h_levels = (int *)malloc(sizeof(int) * vertices);

  cudaMalloc( (void **)&d_levels, sizeof(int) * vertices) ;

  // Prepare GPU...
  // Init dimensions
  int gridDimX = (vertices>BPGX) ? BPGX : vertices;
  int gridDimY = (vertices/BPGX) + 1;
  dim3 blockGridRows(gridDimX, gridDimY);
  int blockDimX = (vertices>TPBX) ? TPBX : vertices;
  int blockDimY = (vertices/TPBX) + 1;
  dim3 threadBlockRows(blockDimX, blockDimY);

  // Do the multiplication on the GPU
  cudaMemcpyToSymbol(adjazenz, adj,
    sizeof(int)*vertices*vertices);

  cuda_bfs_const<<<blockGridRows, threadBlockRows>>>
          (d_levels, vertices);

  cudaThreadSynchronize();

  // Copy the data back to the host
  cudaMemcpy(h_levels, d_levels, sizeof(int) * vertices,
          cudaMemcpyDeviceToHost) ;
  // Calculate the maximum
  int cost = INFINITE;
  for (i=0; i<vertices; i++) {
    if (h_levels[i]==INFINITE)
    {
      cost = INFINITE;
      break;
    }
    else if (cost == INFINITE || cost < h_levels[i])
    {
      cost = h_levels[i];
    }
  }
  cudaFree(d_levels );
  free(h_levels);

  return cost;
}
```

The calculation itself is done by another method presented in A.2. The code is completely executed on the Graphic Device. It uses a map-reduce method for calculating the maximum that is listed in algorithm 3.2 at section 3.4.

Listing A.2: CUDA example calculating maximum value with reduction code

```
#define INFINITE 99999
// Threads per Block X
#define TPBX 512
// Threads per Block Y
#define TPBY 512
// Threads per Block
#define TPB (TPBX * TPBY)
// Blocks per Grid X
#define BPGX 65535
 // Blocks per Grid Y
#define BPGY 65535
// Blocks per Grid
#define BPG (BPGX * BPGY)

// Maximal shared memory: 16384 / 4 bytes (int) - system memory
#define MAX_VERTICES 4096-16
// Maximal constant memory: 65536 / 4 bytes (int)
// 16384 = approx 181x181 (so 128 vertices)
#define MAX_CONST_MEM 16384

#define indexXYN(x,y,n) (y*n+x)

__constant__ int adjazenz[MAX_CONST_MEM];



// BFS from each vertex at same time
// block=SSSP (single start shortest path,
//              thread=vertex in SSSP)
__global__ void cuda_bfs_const(int *level, int vertices)
{
  // which vertex is the starting vertex in this block
  int start = blockIdx.y*gridDim.x + blockIdx.x;
  // which element this thread is reading/writing in memory
  int arrayIndex = threadIdx.y * blockDim.x + threadIdx.x;

  __shared__ int lvl[MAX_VERTICES];

  int i;
  int run = 0;
  // >1 vertex have updated last iteration
  __shared__ bool running[3];
  // init shared and global memories
  if (arrayIndex < vertices) {
    if (arrayIndex==start) {
      lvl[arrayIndex] = 0;
    } else {
      lvl[arrayIndex] = INFINITE;
    }
    if (arrayIndex==0) {
      running[0] = true;
      running[1] = false;
      running[2] = false;
    }
  }
  __syncthreads();

  while (running[run%3] && run<vertices*2) {
    if (arrayIndex < vertices) {
      for (i=0; i<vertices; i++) {
        int link =
                  adjazenz[indexXYN(arrayIndex,i,vertices)];
        int nValue = lvl[i];
        int newValue = nValue+link;
        if (link!=INFINITE && nValue !=INFINITE) {
          int oldValue =
                atomicMin(&lvl[arrayIndex], newValue);
          // actually did something
          if (newValue < oldValue)
          {
            running[(run+1)%3] = true;
          }
        }
      }
      if (arrayIndex==0)
        running[(run+2)%3] = false;
    }
    // Wait to avoid race-conditions between old/new iteration
    __syncthreads();
    run++;
  }

  int cost = maxReduce(lvl, vertices);
  if (arrayIndex==0)
  {
    level[0] = cost;
  }
}
```

Bibliography

[1] *Linear Programming*. Prentice-Hall, INC., Englewood Cliffs, N.J., 1966.

[2] Advanced Micro Devices, Inc., One AMD Place P.O. Box 3453 Sunnyvale, California, USA. *AMD APP SDK developer release notes*, 2.7 edition, june 2012.

[3] M. Annaratone. Mpps, amdahl's law, and comparing computers. In *Frontiers of Massively Parallel Computation, 1992., Fourth Symposium on the*, pages 465 –470, oct 1992.

[4] Slo-Li Chu and Chih-Chieh Hsiao. OpenCL: Make ubiquitous supercomputing possible. In *High Performance Computing and Communications (HPCC), 2010 12th IEEE International Conference on*, pages 556 –561, sept. 2010.

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. Introduction to algorithms. *The MIT Press*, 3rd ed, 2009.

[6] Microsoft Corporation. Compute shader overview, july 2012. `http://msdn.microsoft.com/en-us/library/ff476331.aspx`; [Online; accessed 29-August-2012].

[7] H.A. Eiselt and C.-L. Sandblom. *Integer Programming and Network Models*. Springer, 2000.

[8] Jianbin Fang, A.L. Varbanescu, and H. Sips. A comprehensive performance comparison of CUDA and OpenCL. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 216 –225, sept. 2011.

[9] T. Fujimura and H. Miwa. Critical links detection to maintain small diameter against link failures. In *2010 International Conference on Intelligent Networking and Collaborative Systems*, pages 339–343, 2010.

[10] R.G. Gomory. An algorithm for integer solutions to linear programs. *Recent Advances in Mathematical Programming*, pages 269–302, 1963.

[11] Khronos Group. OpenCL - the open standard for parallel programming of heterogeneous systems, aug. 2012. `http://www.khronos.org/opencl/`;[Online; accessed 29-August-2012].

[12] P. Harish and P.J. Narayanan. Accelerating large graph algorithms on the GPU using CUDA. In *14th International Conference High Performance Computing (HiPC 07)*, pages 197 – 208, dec. 2007.

[13] B.S. Hasan, M.A. Khamees, and A.S.H. Mahmoud. A heuristic genetic algorithm for the single source shortest path problem. In *Computer Systems and Applications, 2007. AICCSA '07. IEEE/ACS International Conference on*, pages 187 – 194, may. 2007.

[14] IBM. *User's Manual for CPLEX*, 2009. `http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/`;[Online; accessed 15-September-2012].

[15] VIA Technologies Inc. VIA brings enhanced windows 7 desktop to life with worlds most power efficient DX10.1 chipset. Press Release. `http://www.via.com.tw/en/resources/pressroom/pressrelease.jsp?press_release_no=4327`[Online; accessed 12-January-2012].

[16] David B. Kirk and Wen mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach (Applications of GPU Computing Series)*. Morgan Kaufmann, 2010.

[17] A.H. Land and A.G. Doig. An automatic model of solving discrete programming problems. *Econometrica*, 28(3):497–520, july 1960.

[18] Lijuan Luo, M. Wong, and Wen mei Hwu. An effective GPU implementation of breadth-first search. In *Design Automation Conference (DAC), 2010 47th ACM/IEEE*, pages 52 –55, june 2010.

[19] K. Matsumoto, N. Nakasato, and S.G. Sedukhin. Blocked all-pairs shortest paths algorithm for hybrid cpu-gpu system. In *High Performance Computing and Communications (HPCC), 2011 IEEE 13th International Conference on*, pages 145 –152, sept. 2011.

[20] NVidia. *NVidia CUDA C Programming Guide*, april 2012. `http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf`; [Online; accessed 04-September-2012].

[21] NVidia. *NVidia GTX 280 Specifications*, 2012. `http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-680/specifications`;[Online; accessed 14-September-2012].

[22] T. Okuyama, F. Ino, and K. Hagihara. A task parallel algorithm for computing the costs of all-pairs shortest paths on the cuda-compatible gpu. In *Parallel and Distributed Processing with Applications, 2008. ISPA '08. International Symposium on*, pages 284 –291, dec. 2008.

[23] OpenACC-Standard.org. *The OpenACC^{TM} Application Programming Interface*, nov. 2011. `http://www.openacc.org/sites/default/files/OpenACC.1.0_0.pdf` [Online; accessed 12-August-2012].

[24] Vijay V. Vazirani. *Approximation Algorithms*. Springer, 2004.