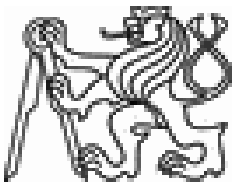
CENTER FOR
MACHINE PERCEPTION



CZECH TECHNICAL
UNIVERSITY IN PRAGUE

MASTER'S THESIS

Parallelization of Minimal Problem Solver Generator

Vojtěch Cvrček

cvrcekv@gmail.com

May 27, 2016

Thesis Advisor: Tomáš Pajdla

Center for Machine Perception, Department of Cybernetics
Faculty of Electrical Engineering, Czech Technical University
Technická 2, 166 27 Prague 6, Czech Republic
fax +420 2 2435 7385, phone +420 2 2435 7637, www: http://cmp.felk.cvut.cz

Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Cybernetics

DIPLOMA THESIS ASSIGNMENT

Student: Bc. Vojtěch C v r č e k

Study programme: Open Informatics

Specialisation: Computer Vision and Image Processing

Title of Diploma Thesis: Parallelization of Minimal Problem Solver Generator

Guidelines:

1. Review methods for constructing minimal solver generators [2,3,4], necessary basic theories [1,5] and possibility of parallel implementations of minimal solvers.
2. Suggest a parallel implementation of a generator or solvers and implement it in GPU.
3. Demonstrate the functionality of the implementation on selected examples and compare the running times between the standard and parallel GPU implementations.

Bibliography/Sources:

[1] David Cox, John Little, and Donald O'Shea. Ideals, Varieties, and Algorithms. An Introduction to Computational Algebraic Geometry and Commutative Algebra. Undergraduate Texts in Mathematics. Springer, New York, USA, 2nd edition, 1997.
[2] P. Trutman. Minimal Problem Solver Generator. Bachelor thesis. CTU in Prague, 2015.
[3] Zuzana Kukelova. Algebraic Methods in Computer Vision. PhD thesis, Department of Cybernetics, Faculty of Electrical Engineering, Czech Technical University in Prague, February 2013.
[4] Zuzana Kukelova, Martin Bujnak, and Tomas Pajdla. Automatic generator of minimal problem solvers. In Proceedings of The 10th European Conference on Computer Vision, ECCV 2008, October 12–18, 2008.
[5] D.A. Cox, J. Little, and D.O'Shea. Ideals, Varieties, and Algorithms: An Introduction to Computational Algebraic Geometry and Commutative Algebra. Undergraduate Texts in Mathematics. Springer, 2010.

Diploma Thesis Supervisor: Ing. Tomáš Pajdla, Ph.D.

Valid until: the end of the summer semester of academic year 2016/2017

L.S.

prof. Dr. Ing. Jan Kybic
Head of Department

prof. Ing. Pavel Ripka, CSc.
Dean

Prague, February 18, 2016

Author statement for undergraduate thesis:

I declare that the presented work was developed independently and that I have listed all sources of information used with it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, date May 27, 2016 Vojtěch Cvrček

Abstract

We proposed a parallelization of the minimal solvers intended for the RANSAC scheme. To utilize the GPU fully, we are solving several instances at once. We optimize the computation of the eigenvectors because it is the most time-consuming part of the solver.

We also examined the other parts of RANSAC and found that the verification process has much greater potential for parallelization.

We implemented both improvements in CUDA C/C++ and compared the results with serial implementation. The selected minimal problem was the five-point relative pose problem.

The minimal solver is often more than two times faster in a hybrid system (GPU + single-core CPU) than on a single-core CPU. The verification process is about 90 times faster on GPU than on a single-core CPU for the selected data set.

Keywords: CUDA, GPU, automatic solver, eigenvectors, RANSAC, relative pose problem

Abstrakt

Navrhli jsme paralelizaci minimalního solveru pro RANSAC. Optimální využití GPU je dosaženo řešením několika problémů zároveň. Hlavní pozornost věnujeme výpočtu vlastních čísel, neboť je to časově nejnáročnější část solveru.

Zkoumáním RANSACU jsme zjistili, že verifikace představuje potenciální zdroj zrychlení paralelizací.

Porovnali jsme implementaci v CUDA C/C++ se seriovym řešením. Pracovali jsme s pěti bodovým problémem relativní polohy.

Minimální solver je až dvakrát rychlejší na hybridním systému (GPU + jedno jádro CPU) než na jednom jádru CPU. Proces verifikace je pro zvolená data až devadesátkrát rychlejší na GPU než na jednom jádru CPU.

Klíčová slova: CUDA ,GPU,automatický solver, vlastní vektory, RANSAC, relativní poloha

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

The automatic solver is an impressive tool for solving many minimal problems in computer vision and many other areas. We observed that in practice, the minimal problem is solved for a larger number of instances. A typical application of the solutions of minimal problems is RANSAC, and we kept this in mind during construction of our contribution.

We know that the most time-consuming part of the automatic solver (the online phase) is the computation of eigenvectors. This claim is part of [20], and we reconfirmed that during the initial stage. We studied the problem [9], and came to the conclusion that if solving many instances at once, there is a good chance of dropping the runtime considerably by using parallelization. Moreover, the eigenspaces of our problems have specific properties. The properties are not used by general methods for solving the eigenvalue problem and open an opportunity to develop a more specialized eigenvalue problem solver.

We also observed that finding a consensus set during the execution of RANSAC is usually more time demanding than finding the model parameters and realized that the process could be speed up by parallelization.

1.2 Goal

The main goal is to speed up the automatic solver. The current approach computes all eigenvectors and all eigenvalues using the Matlab, the Maple [35] or the library Eigen [14]. We will develop problem-specific fast routines to solve this problem.

We will also show on a case study that we can considerably speed up the RANSAC by parallelization.

1.3 Thesis structure

Chapter 2 serves as an introduction to parallelization on GPU using CUDA. Chapter 3 is a short chapter in which we expose some properties of automatic online solvers which guided the development of methods in this thesis. Next, we summarize the necessary theory for the problem we solve in Chapter 4. The

brief Chapter 5 describes selected work concerning the eigenvalue problem and the RANSAC parallelization. Chapter 6 shows how we approach the two basic problems and explains the details.

The chapter results show performed experiments. We test the standalone routines and the automatic solver which uses these routines, and we compare the results with the previous version. We also show the properties of parallelized RANSAC with a GPU-accelerated automatic solver.

Chapter 2

CUDA

CUDA is a general purpose parallel computing platform and programming model [28]. The name CUDA originally stood for an acronym **Compute Unified Device Architecture**, but NVIDIA no longer uses CUDA as an acronym [2].

Many-core GPUs offer (far) greater raw computation power and memory bandwidth than multicore CPUs with similar power consumption (and cost) - Figure 2.1 and Figure 2.2. Moreover, the parallelization is an important branch of GPU and CPU development [31].

When programming a CPU, a programmer needs to know only the rudiments of the CPU architecture to write reasonable code. The reason is that many programming languages offer a high level of abstraction, and the compiler is often capable of shielding the user (of the programming language) from deep hardware knowledge. CUDA C API, on the other hand, offers some level of abstraction, but an understanding of the underlying hardware is a necessity for writing code which fully utilizes the given device.

Not all GPUs are supported; CUDA requires a GPU with the **SIMT** (Single-Instruction Multiple Data) architecture, which effectively means GPUs from NVIDIA.

This chapter presents chosen CUDA C API related topics relevant to this thesis and a brief description of NVIDIA GPU hardware.

The text and the terminology are based on the CUDA toolkit [26].

2.1 Programming Model

This section describes the programming model of CUDA. The programming model is designed to simplify the workflow by abstracting the underlying GPU hardware implementation while exposing the parallel structure of GPU to a programmer. The following subsections summarize the basics of CUDA C API from a developer's point of view, omitting the hardware implementation.

2.1.1 Host and device

A CPU controls the execution of a program, managing the data allocation, transfer to and from a GPU, commencing the GPU's kernel, and other issues. The CPU is hosting a GPU. Hence, a CPU is called a host in the CUDA context.A GPU is a separate device controlled by a host and is therefore called a device.

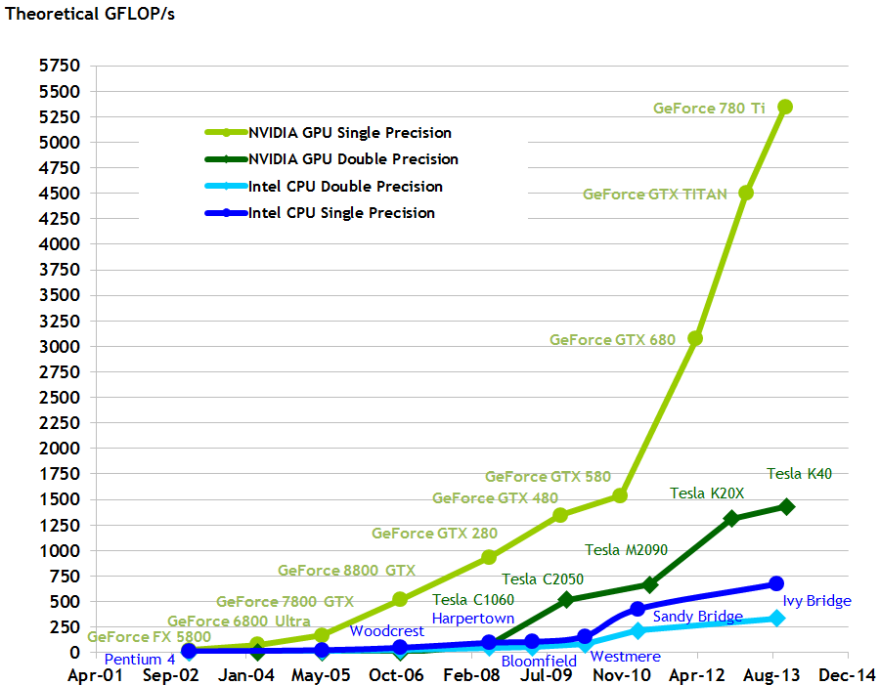


Figure 2.1: Theoretical GFLOP/s (figure taken from [28])

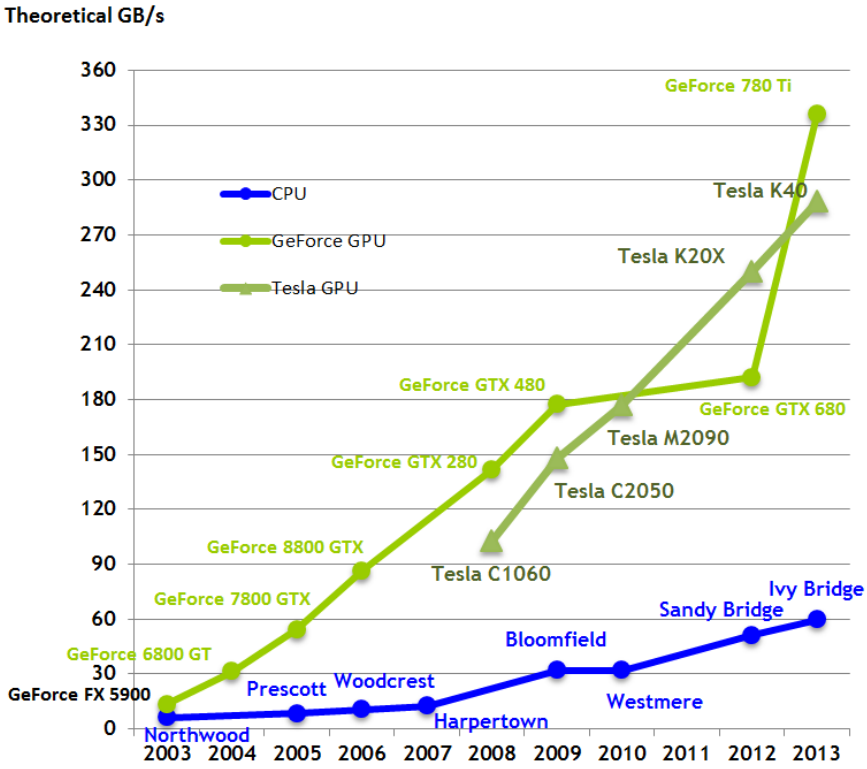


Figure 2.2: Theoretical GB/s (figure taken from [28])

2.1.2 Kernels

A kernel is a special function, an extension of C/C++, which allows a programmer to access a device from a host. Therefore, a kernel acts as a gateway between a host and a device. Unlike a standard C/C++ code, a kernel function runs on

many threads in parallel. A programmer has to define a kernel function with a declaration specifier __**global**__, and when calling the function, a programmer is required to provide the number of threads on which the function will run using the C extended syntax "$\langle\langle\langle ... \rangle\rangle\rangle$". Each thread has a specific ID assigned. Many thousands of kernels can reside on a device during a single kernel call. Examples can be found in the official documentation [28].

2.1.3 Thread hierarchy

Threads are grouped into blocks. The position of a thread in a block is given by its ID, which is a three-dimensional vector. The size of a block is limited, based on the Compute Capability of a device (section 2.2). Blocks can be further organized into grids. The block position is also determined by its ID, which is a three-dimensional vector as well, and its size limits are given by the Compute Capability of a device as well. There is one common general rule for all CUDA supporting devices. A block can hold at most 1024 threads.

The three-dimensional vector is denoted by x,y,z, and its primary purpose is to simplify work with multidimensional data. A programmer is required to provide block and grid dimensions when calling a kernel.

The reason for the two layer grouping of threads is the synchronization at blocks level (threads inside a block can be synchronized). Threads in a block can also access a shared memory (subsection 2.1.4), but the shared memory can't be accessed by a thread from another block.

The size of a grid is usually determined by the amount of data. A programmer is free to choose the size of a block (only upholding limits imposed by Compute Capability) as he wishes. We have to consider the possibilities of the shared memory; an in-depth description of different compute capabilities (section 2.2) is in the appendix of [28]. For the optimal choice, knowledge of the underlying GPU is necessary.

Kernel has to by written such that thread blocks can be executed in any order (either in parallel or series). This requirement is attributed to scalability. The device usually doesn't run all blocks on the grid at once, but rather in groups; an example of such situation is shown in Figure 2.3.

2.1.4 Memory Hierarchy

There are three basic types of memory. A global memory is accessible both by the host and device. A shared memory is shared among threads in a thread block. Registers belong to a thread and cannot be shared among threads. Examples of various variables declaration and their scope and lifetime are given in Table 2.1.

The table depicts the type of memory in which a variable *usually* resides. Notably, if there are not enough registers, even a simple variable is placed into the local memory. And similarly, if we declare a small static array, the compiler might put it in registers. The only way to be sure is to inspect SASS; more information can be found in subsection 2.4.2.

Different memory types offer different access speeds. Registers are the fastest. The shared memory resides in a cache and offers much quicker access than the global memory. The local memory is essentially the global memory with a reduced

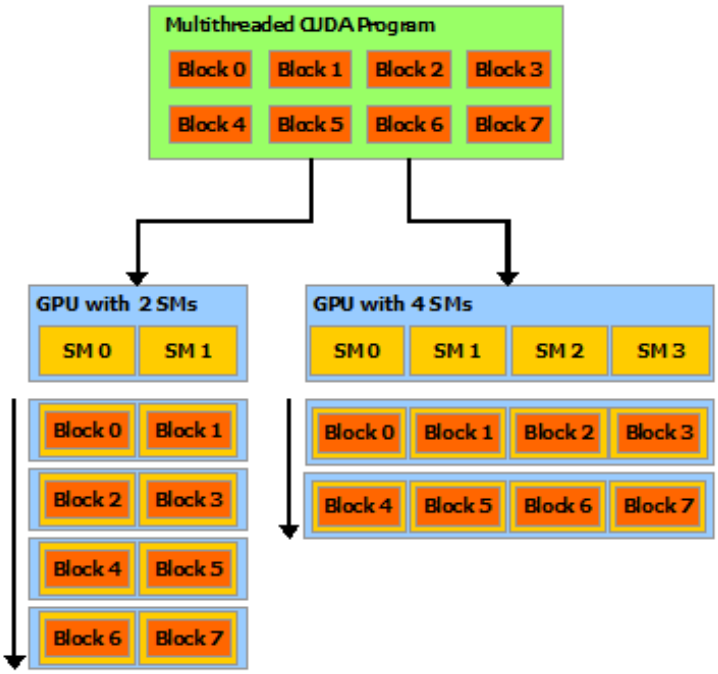


Figure 2.3: Automatic scalability (figure taken from [28])

Variable declaration	Memory	Scope	Lifetime
int LocalVar;	register	thread	thread
int LocalArray[10];	local	thread	thread
[__device__] __shared__ int SharedVar;	shared	block	block
__device__ int GlobalVar;	global	grid	application
[__device__] __constant__ int ConstantVar;	constant	grid	application

Table 2.1: Different memory types [40]

scope and lifetime, so the access speed is the same as global memory. The constant memory is cached global memory. Access speed decreases with the number of requests and can be as slow as the global memory. Table 2.2 offers raw insight into CUDA memory management, and we will return to it in more detail later.

We summarize this brief introduction to the CUDA memory hierarchy with Figure 2.4.

2.2 Compute Capability

The compute capability (CC) of a device identifies features supported by GPU hardware. It's accessible during runtime and allows an application to know which features/instructions are at our disposal. The name itself comes from [28].

The CC consist of two numbers, X.Y, where X is a major revision number, and Y is a minor revision number. The major revision number indicates core device architecture. Between devices with different major revision numbers is a notable difference. Devices with the same major revision numbers but different minor revision numbers differ less; improvements may consist of added CUDA

Variable declaration	Memory	Performance penalty
int LocalVar;	register	1x
int LocalArray[10];	local	100x
[__device__] __shared__ int SharedVar;	shared	1x
__device__ int GlobalVar;	global	100x
[__device__] __constant__ int ConstantVar;	constant	1x

Table 2.2: CUDA type qualifiers, performance penalty (source: [40])

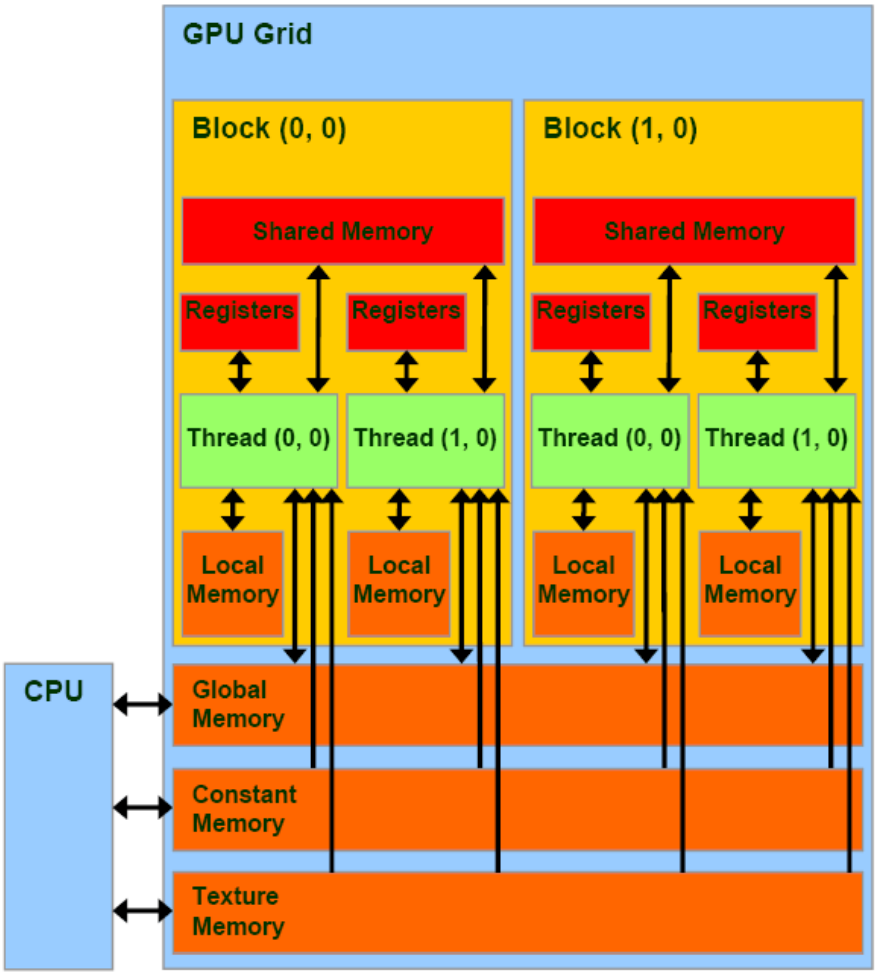


Figure 2.4: Memory model (figure taken from [36])

cores, better warp schedulers, or slightly different cache. This makes the minor revision number still quite important.

What makes the difference is a discontinuity in a binary compatibility. A binary code is generated for a specific device with a compute capability X.Y. The binary compatibility isn't guaranteed between different major revision numbers. The binary compatibility is guaranteed for the next minor revisions (higher, i.e. Z $\rangle$ Y) with the same major revision number.

There are currently four major revision numbers. Number 1 is outdated Tesla architecture (don't confuse with Tesla series, intended for high-performance computing), number 2 is Fermi architecture, number 3 is Kepler, and number 5 is

Maxwell. The Tesla architecture is no longer supported (starting with CUDA 7.0). A CUDA version shouldn't be confused with the compute capability. The CUDA version specifies the version of used CUDA platform.

2.3 Hardware implementation of the GPU

We present this section as an introduction to an architecture of devices.

2.3.1 Streaming Multiprocessor

Each GPU consists of several Streaming Multiprocessors (SMs). When a kernel function is called, groups of blocks of a grid are assigned to multiprocessors of a device. Each SM is capable of hosting several thread blocks and execute them (and threads inside blocks) concurrently. CUDA monitors finished blocks and assign new blocks to SMs as illustrated in Figure 2.3.

2.3.2 SIMT architecture

Single-Instruction, Multiple-Thread (SIMT) is a unique architecture associated with NVIDIA devices. It refers to the basic grouping of threads to a **warp**. The warp is a group of 32 threads (common across all CUDA devices). Each thread in warp starts at the same program address, but branching is possible at the cost of the speed.

Each block is partitioned into groups of warps and executed by the SM's warp scheduler. There might be many warps, but the warp scheduler usually executes a fraction of them at once.

The terminology is a little confusing. A device can reside thousands of threads, but most of them don't run concurrently. The number of simultaneously running threads is determined by the warp scheduler, numbers of SM and branching. With the optimal branching (i.e. no branching at all) and fully occupied device, the number of concurrently running threads is given by Equation 2.1.

$$\begin{aligned} &\text{Concurrent threads} = \\ &\text{(number of warps warp scheduler is capable of running) x} \\ &\text{(number of SMs) x} \\ &\text{(size of warp, usually 32)} \end{aligned} \tag{2.1}$$

A warp can only execute one instruction at a time, although the instruction can be shared among warp threads. The optimal situation is when each thread executes the same set of instructions. If however, there is some data dependent on a conditional branch, the warp diverges.

A divergence means that some threads in a warp execute a different instruction from the rest. The warp executes one branch at a time until different execution paths converge.

The divergence occurs only on warp level; blocks or grids can't diverge. A typical example is in Figure 2.5. The peak performance is related to no branching versus full branching, but the performed instruction matters a lot. Each SM can

perform a limited number of native arithmetic instructions per clock. 32-bit floating-point add, multiply, multiply-add instruction capabilities are related to the name **CUDA cores**. Single precision units are numerous on basic GPU. There are far less double precision units per SM unless we are using the Tesla series (GPUs from Tesla series are in Figure 2.1), which is designed for precise computation. We can find out the precise numbers in [28].

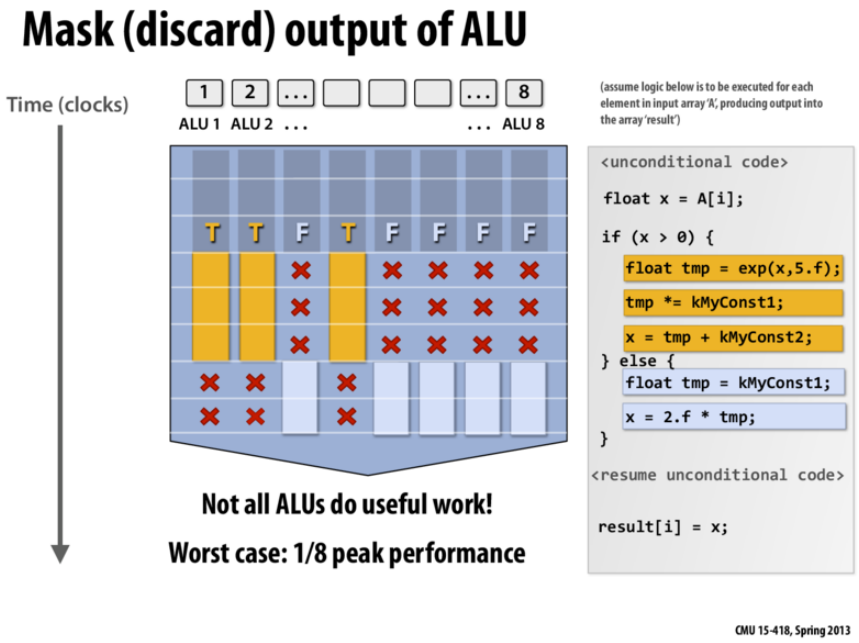


Figure 2.5: Divergence in a warp (figure taken from [32])

We'll address this problem in the next chapter.

SIMT can be perceived as an enriched vectorization. We can either work with independent threads (warps for optimal performance) with independent data or solve problems typical for vectorization. Both cases can gain from the parallelization.

Threads in a block whose instructions are executed by the current warp are called active threads. If divergence occurs, or the size of the threads block isn't aligned to a multiple of the warp size, there are inactive threads. These threads wait until the end of the kernel (if they finished early or if the block wasn't properly aligned) or for other branch(es) to finish. It should be noted that a programming model doesn't know warps. A programmer who chooses to ignore their existence may or may not experience the divergence.

2.3.3 Execution model

When the host calls a kernel with given partitioning into grid and blocks, CUDA assigns blocks to SMs. CUDA runtime checks if the assignment is correct (i.e. the size of a block is according to the CC of the device and the memory request of shared memory can be satisfied). If more than one block can run on the SM, CUDA will assign more blocks to the SM, up to the limits imposed by CC of the device.

In the case when not even one block can be supported by any SM, the kernel launch will fail. The number of warps per block is given by Equation 2.2 from [26].

$$(\text{warps per block}) = \text{ceil}\left(\frac{\text{size of a block}}{\text{size of the warp}}\right) \tag{2.2}$$

From Equation 2.2 it is clear why the optimal size of a block is multiple of a size of the warp. This computation among others is part of CUDA occupancy calculator.

2.4 Compilation

The next issue is a compilation of the source code. It's clear that the extension of C/C++ can't be compiled by a standard C/C++ compiler. We have to handle the host and the device code merged into one source file with another tool. NVIDIA distributes a CUDA compiler driver as part of CUDA. This compiler is called nvcc.

A programmer of a CUDA device should have a good understanding what the nvcc compiler offers for the best performance of her code.

2.4.1 Overview

A source code intended for a heterogeneous (host+device) runtime is expected to have suffix .cu, which indicates a combination of host and device code. If there are no device functions in the source, nvcc will still comply with the .cu suffix (by just calling the host compiler). If, however, there are some device functions, then the host compiler will quite expectedly fail.

The nvcc compilation trajectory is a complicated process that consists of several steps. A source is split (on device and host code), compiled (host functions are compiled with native compiler for C++, and device functions are compiled with CUDA compiler) and merged. Figure 2.6 illustrates the full process. We encountered some simplified figures, but since we ran into problems that require the knowledge of the full compilation trajectory, we decided to present an unaltered depiction.

2.4.2 PTX

The parallel thread execution (PTX) instruction set is not directly executed. It can be viewed as an assembly for virtual GPU. The generation of PTX is an intermediate step during compilation and serves for the final step, which is the creation of a CUDA execution binaries (cubin files).

A programmer can use the PTX assembly directly and similarly to the CUDA C. We also don't have to bother with the target device specification too much. If we want to see details about execution binaries in a human readable form, we can use SASS.

The instruction set for PTX is determined by a virtual architecture (-arch option), which decides for which lowest CC of a device is the code applicable. The generation of binaries can be carried later on (subsection 2.4.3) (-code option).

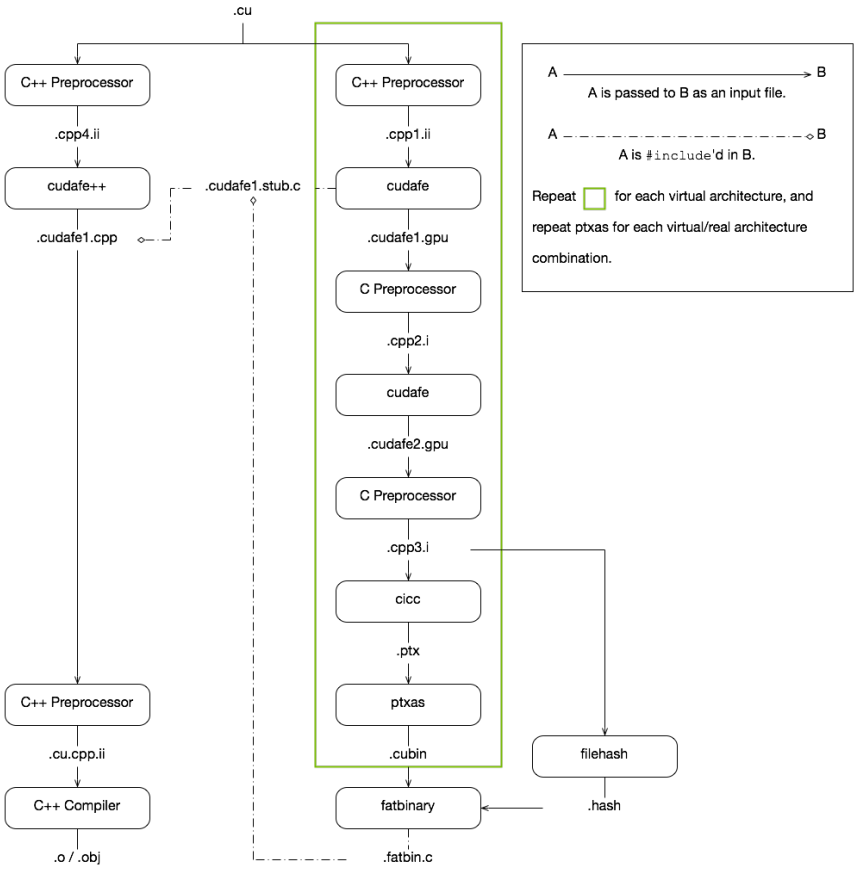


Figure 2.6: CUDA whole program compilation trajectory (figure taken from [29])

PTX is necessary for a Just-in-Time compilation (subsection 2.4.3) and is required for use of kernels by the MATLAB parallelization toolbox.

2.4.3 Just-in-Time compilation

During the compilation the PTX code is generated. The PTX can be immediately used for the generation of executable binaries (.cubin file), specific for an intended target device. If however we use another device or device driver is updated, and we don't have valid binaries. CUDA offers a feature called Just-in-Time compilation.

Just-in-Time compilation is a compilation during the runtime. The CUDA runtime generates binaries from PTX and caches (a compute cache) them for later use. The compute cache is invalidated when a driver is updated and new binaries are generated and cached in the compute cache during the next execution.

The just-in-Time compilation prolongs the execution (when binaries aren't cached). It's a tradeoff for using the most recent instruction set.

The general rule is to use the lowest possible virtual architecture for PTX and highest possible architecture for GPU. Such compilation maximizes devices span, and just-in-time compilation produces optimal binaries for the newest devices (non-existent during code development).

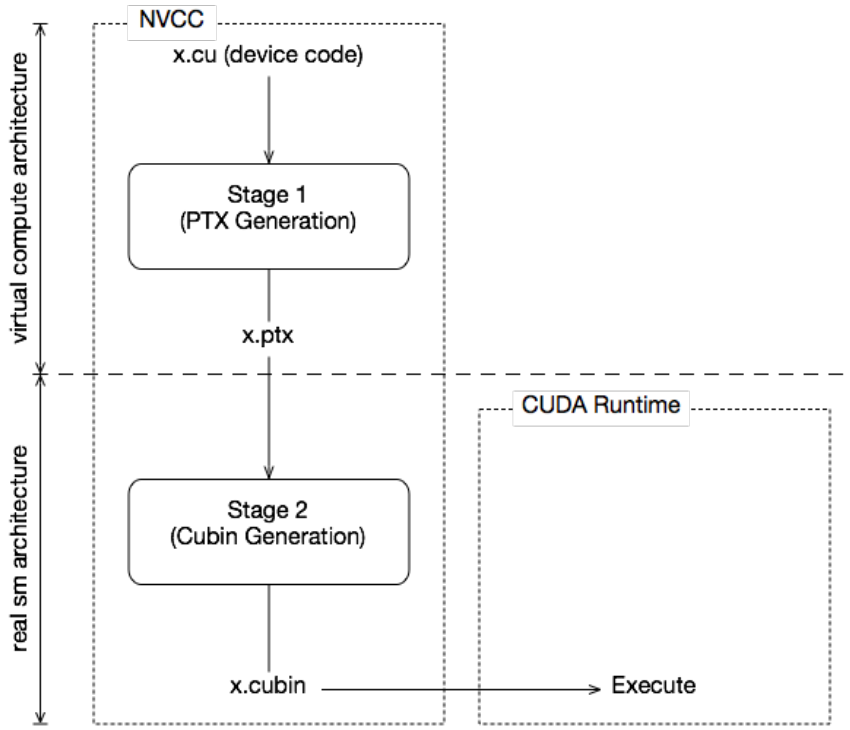


Figure 2.7: Two-Staged compilation with virtual and real architectures (figure taken from [29])

2.4.4 options

We will name two options: the -arch option for the virtual architecture and the -code option for the target gpu. To our knowledge, the official documentation [29] offers the best source to learn all the options and their impact on the product of the compilation.

If a programmer uses nvcc in some IDE (the VS with the nsight plugin, the Nsight Eclipse), options are usually reasonably preset. The best practice is to consult the official documentation with any modifications to preset options. The Visual Studio profiling tool offers a good guide which provides inspection of Source/PTX/SASS and their cross-comparison as well as memory management inspection.

Chapter 3

Automatic generator

The automatic generator is a tool for solving selected minimal problems. The problem is stated as a system of polynomial equations. The system describes a general instance of the problem. The result is a generic online template that can solve concrete instances. Details are in [20].

The theoretical background can be obtained in [8], [9]. [20] explains the automatic generator, including selection of proper minimal problems.

This chapter summarizes selected facts about the online phase and forms the base for the optimization in Chapter 6. Since the solvers of minimal problems are good generators of hypotheses for the RANSAC paradigm, we incorporated the RANSAC into this chapter.

3.1 Online solver

The online solver is a special case of the online phase. The problems for which we can find the online solver are specified in [20].

The input is a system of polynomial equations $f_1 = f_2 = \cdots = f_n = 0$ with concrete coefficients. Online solvers comprise three steps:

- The solver first generates necessary polynomials q_i for the construction of a multiplication matrix.
- The next step is the construction of the multiplication matrix from coefficients of polynomials q_i.
- The final step is the extraction of the solutions of the input system from the eigenvectors of the multiplication matrix.

Details are in [20]. We would like to emphasize, according to [20]: *"Eigenvalue computation is usually the most time-consuming part of final solvers."*

3.2 Multiplication Matrix

The multiplication matrix has several properties that can greatly simplify finding eigenvalues/eigenvectors. First, we can assume the multiplication matrix is nonderogatory (i.e., all eigenspaces have dimension one).

The next observation is that the multiplication matrix represents multiplication by some function f:

$$f = c_1 x_1 + \cdots + c_n x_n \tag{3.1}$$

where x_i is a variable and c_i is a randomly chosen integer (such that f will have a distinct value for distinct solutions of the input system). [9] shows eigenvalues of the multiplication matrix generated by f are equal to values of the function f for x_i which solves the input system. [9] further shows how to obtain x_i directly from eigenvectors of such multiplication matrix.

3.3 RANSAC

The random sample consensus (RANSAC) is a paradigm for model fitting [10]. Its basic form can be summarized in following steps:

- Select a sample of m points (assume these m points are inliers).
- Calculate model parameters from the sample points.
- Test the whole set of points against the calculated model and collect points within some predetermined threshold (these points are called a consensus set or inliers). This step is sometimes called the verification.
- Compare the current model with the best model encountered so far(compare the number of inliers). Choose the model with more inliers as the next best model.

[10] also includes an optimization step (using only the consensus set). The stopping criterion is a part of [10].

Since we assume that a selected sample of m points is a subset of inliers, it's a good idea to have the size of the sample as small as possible [20] and it is the reason for using minimal problems.

Chapter 4

The Unsymmetric Eigenvalue Problem

Searching real eigenvectors is the most time-consuming part of the automatic solver online template. This chapter will present the theoretical background, necessary for computing real eigenvectors. We also present some algorithms which we will parallelize later.

4.1 Eigenvalue problem

[12] defines the **eigenvalues** and the **spectrum** as follow:

Definition 1. *Let there be a square matrix $A \in \mathbb{C}^{n\times n}$, the **eigenvalues** of the matrix A are the n roots of its characteristic polynomial p(z) = det(zI - A). The set of these roots is called **spectrum** and is denoted by $\lambda(A)$.*

The definition of the **eigenvectors** [12]:

Definition 2. *If $\lambda \in \lambda(A)$, then the nonzero vectors $x \in \mathbb{C}^n$ that satisfy*

$$Ax = \lambda x$$

*are referred to as **eigenvectors**.*

Pair (λ,x) is often called an **eigenpair** of A [3]. Our goal is to find the real subset of the spectrum and its eigenvectors.

4.1.1 The real Schur decomposition

Real matrices can have complex eigenvalues. But since the characteristic polynomial has real coefficients, the complex eigenvalues are always in pairs [12]. Computations with complex numbers are costly, following theorem from [3] is the key to avoid complex computation:

Theorem 1. ***The Real Schur decomposition*** *If $A \in \mathbb{R}^{n\times n}$, then there is an orthogonal matrix $Q \in \mathbb{R}^{n\times n}$ such that*

$$Q^T AQ = \begin{bmatrix} R_{1,1} & R_{1,2} & \cdots & R_{1,m} \\ & R_{2,2} & \cdots & R_{2,m} \\ & & \ddots & \vdots \\ & & & R_{m,m} \end{bmatrix} \tag{4.1}$$

is an upper quasi triangular. The upper quasi triangular matrix is a matrix which has blocks on diagonal and zeros below diagonal. In this case, the blocks $R_{i,i}$ are of size 1×1(a real eigenvalue) or 2×2(a pair of complex conjugate eigenvalues).

The proof can be found in [3].

4.2 The QR iteration

We first introduce a following algorithm 1 from [12]. The Input is a matrix $A \in \mathbb{C}^{n \times n}$ and an unitary matrix $U_0 \in \mathbb{C}^{n \times n}$. It's easy to show that T_k is

Algorithm 1 QR iteration

$T_0 = U_0^T A U_0$
for k $= 1, 2, \ldots$ **do**
  $T_{k-1} = U_k R_k$ ▷ QR decomposition
  $T_{k-1} = R_k U_k$
end for

unitarily similar to A.

$$T_k = R_k U_k = U_k^T (U_k R_k) U_k = U_k^T T_{k-1} U_k \tag{4.2}$$

And by induction [12].

$$T_k = R_k U_k = (U_0 U_1 \cdots U_k)^T A (U_0 U_1 \cdots U_k) \tag{4.3}$$

A nice summarization of the convergence property is in [30]:

Property 1. ***Convergence of QR iteration*** *Let $A \in \mathbb{C}^{n \times n}$ be a matrix with following magnitudes of eigenvalues:*

$$|\lambda_1| > |\lambda_2| > \cdots > |\lambda_n|$$

then

$$\lim_{k \to \infty} T^{(k)} = \begin{bmatrix} \lambda_1 & t_{1,2} & \cdots & t_{1,n} \\ 0 & \lambda_2 & t_{2,3} & \cdots \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \lambda_n \end{bmatrix} \tag{4.4}$$

Convergence rate is:

$$|t_{i,i-1}^{(k)}| = \mathcal{O}\left(\left| \frac{\lambda_i}{\lambda_{i-1}} \right|^k \right), \quad i = 2, \ldots, n, \quad \textit{for } k \to +\infty. \tag{4.5}$$

The property 1 assumes the complex representation. In the case of the real representation, as mentioned in subsection 4.1.1, matrix T_k converge to upper quasitriangular. The proof can be found in [39].

4.3 Spectral shift

Shifting eigenvalues of a matrix A means subtracting a matrix μI. If the original matrix had magnitudes of eigenvalues:

$$|\lambda_1|, |\lambda_2|, \ldots, |\lambda_n|$$

After the spectral shift:

$$|\lambda_1 - \mu|, |\lambda_2 - \mu|, \ldots, |\lambda_n - \mu|$$

This changes the order of magnitudes of eigenvalues and the convergence rate of QR iterations. New convergence rate is:

$$|t_{i,i-1}^{(k)}| = \mathcal{O}\left(\left|\frac{\lambda_i - \mu}{\lambda_{i-1} - \mu}\right|^k\right) \tag{4.6}$$

In the extreme example when we know the eigenvalue exactly, we need to do only one QR iteration [3],[12].

4.4 Decoupling

Decoupling is the key to divide the problem of finding eigenvalues to subproblems [12]:

Theorem 2. *If $T \in \mathbb{C}^{n\times n}$ is partitioned as follows:*

$$T = \begin{bmatrix} T_{11} & T_{12} \\ 0 & T_{22} \end{bmatrix} \tag{4.7}$$

then $\lambda(T) = \lambda(T_{11}) \cup \lambda(T_{22})$.

Hence, we can solve two smaller matrices T_{11}, T_{22} and ignore the matrix T_{12}. Notice that this is only possible if we are interested only in eigenvalues. Otherwise, the T_{12} can not be ignored.

4.5 Deflation

The deflation is a part of QR algorithm. It's a direct application of theorem 2. We apply the deflation when an element on the subdiagonal is sufficiently small. There are many different strategies. Details of the standard deflation are in [12],[3] and [11]. The modern approach is to use the aggressive early deflation [6].

4.6 Givens rotation

The Givens rotation is an orthogonal matrix $G(i, j, \theta)$ in the form [12]:

$$G(i,j,\theta) = \begin{pmatrix} & & i & & j & & \\ 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \cdots & c & \cdots & s & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \cdots & -s & \cdots & c & \cdots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{pmatrix} \begin{matrix} \\ \\ i \\ \\ j \\ \\ \\ \end{matrix} \tag{4.8}$$

where $c = cos(\theta)$ and $s = sin(\theta)$.

For $x \in \mathbb{R}^n$ and $y = G(i, j, \theta)^T x$:

$$y_k = \begin{cases} cx_i - sx_j, & k = i \\ sx_i + cx_j, & k = j \\ x_k, & k \neq i, j \end{cases} \tag{4.9}$$

The Givens rotations are usually used to zero a particular element. We can set the y_j to be zero by setting:

$$c = \frac{x_i}{\sqrt{|x_i|^2 + |x_j|^2}}, \quad s = \frac{-x_j}{\sqrt{|x_i|^2 + |x_j|^2}}. \tag{4.10}$$

4.7 Householder reflectors

Householder reflectors are unitary matrices which allow reflecting/mirroring vectors.

Definition 3. *(**Householder reflectors**). Let $\boldsymbol{q} \in \mathbb{R}^n$ and $||\boldsymbol{q}|| = 1$. A matrix*

$$H(\boldsymbol{q}) = I - 2\boldsymbol{q}\boldsymbol{q}^T \in \mathbb{R}^{n \times n}$$

is called a Householder reflector [33].

Householder reflectors can zero many elements of a vector at once. An important observation is, that we only require $4n$ flops [3] to perform $H\mathbf{x} = \mathbf{x} - \mathbf{q}(2\mathbf{u}^T\mathbf{x})$.

For vector $\mathbf{x} \in \mathbb{R}^n$, we search a Householder reflector H which transforms vector $\mathbf{x}$ to $\pm||\mathbf{x}||\mathbf{e}_1$ ($\mathbf{e}_1$ has usual meaning, i.e. $\mathbf{e}_1 = [1, 0, ..., 0]^T$).

The numericaly stable solution is [33]:

$$\mathbf{q} = \frac{\mathbf{x} + sign(x_1)||\mathbf{x}||\mathbf{e}_1}{||\mathbf{x} + sign(x_1)||\mathbf{x}||\mathbf{e}_1||} \tag{4.11}$$

We use them to obtain the Hessenberg form (section 4.8).

4.8 Hessenberg form

The **Hessenberg form** is defined in the following way [3]:

Definition 4. *A matrix H is a Hessenberg matrix if its elements below the lower off-diagonal are zero,*

$$h_{ij} = 0,\ i > j + 1.$$

The Hessenberg form's attractivity comes from its properties, namely more effective QR decomposition.

The QR decomposition time complexity is $\mathcal{O}(n^3)$ [3] in the general case. The precise number of arithmetic operations for different methods can be found in [33]. It's easy to see that with a matrix in the Hessenberg form we have to zero far less elements.

The overall time complexity would remain the same if we had to compute the Hessenberg form repeatedly since its complexity is also $\mathcal{O}(n^3)$ [3]. Fortunately, theorem 3 shows why it's sufficient to compute the Hessenberg form just once.

Theorem 3. *The Hessenberg form is preserved by the QR algorithms.*

The constructive proof is in [3] and a trivial corollary is a reduction of time complexity of QR decomposition for a matrix in Hessenberg form to $\mathcal{O}(n^2)$.

4.9 The single shift QR algorithm

The spectral shift from section 4.3 allows us to speed up the convergence rate. The following algorithm from [12] shows how. This variation of the algorithm

Algorithm 2 QR single shift

```
for k = 1, 2, ... do
    μ = H_{n,n}
    H − μI = UR                    ▷ QR decomposition
    H = RU + μI
end for
```

algorithm 1 is likely to have the quadratic convergence rate (proof in [12]).

4.10 The double shift QR algorithm

In the section 4.9, we presented a modification of the algorithm algorithm 1 with better convergence rate. Unfortunately, if the lower diagonal 2x2 submatrix of the matrix H is:

$$G = \begin{bmatrix} h_{mm} & h_{mn} \\ h_{nm} & h_{nn} \end{bmatrix}, \quad m = n - 1 \tag{4.12}$$

and the algorithm algorithm 2 converges to a pair of complex eigenvalues λ_1, λ_2. Then h_{nn} approximates only the real part of the eigenvalue.

[11] firstly approaches the problem with a complex representation. [11] later came up with a more elegant solution which is explained in the following section.

But we first examine a similar method which helps us to understand the next section [12].

The idea is to perform two single-shift QR steps at once:

$$H - \lambda_1 I = U_1 R_1 \tag{4.13}$$
$$H_1 = R_1 U_1 + \lambda_1 I \tag{4.14}$$
$$H_1 - \lambda_2 I = U_2 R_2 \tag{4.15}$$
$$H_2 = R_2 U_2 + \lambda_2 I \tag{4.16}$$

U_1, U_2, R_1, R_2 are complex if λ_1, λ_2 are complex.

We can show that [12]:

$$(U_1 U_2)(R_2 R_1) = (H - \lambda_1 I)(H - \lambda_2 I) = M \tag{4.17}$$

where M is a real matrix (even if λ_1, λ_2 are complex) in the following form:

$$M = H^2 - sH + tI \tag{4.18}$$

where

$$s = trace(G) = \lambda_1 + \lambda_2, \quad t = det(G) = \lambda_1 \lambda_2. \tag{4.19}$$

We may choose U_1 and U_2 such that $Z = U_1, U_2$ is a real orthogonal matrix [12]:

$$H_2 = U_2^H H_1 U_2 = U_2^H (U_1^H H U_1) U_2 = (U_1 U_2)^H H (U_1 U_2) = Z^T H Z. \tag{4.20}$$

where A^H is a Hermitian transpose. Due to the roundoff error the matrix H_2 will almost always be in the complex field [12]. According to [12], obtaining the real matrix at this point is too computationally demanding ($\mathcal{O}(n^3)$).

4.11 The double implicit shift QR algorithm

There is a much better approach, which does not require complex computations. But first we introduce an essential theorem from [12]:

Theorem 4. ***(Implicit Q Theorem)*** *Let $Q = [q_1, \ldots, q_n]$ and $V = [v_1, \ldots, v_n]$ be orthogonal matrices with the property that both $Q^T A Q = H$ and $V^T A V = G$ are upper Hessenberg where $A \in \mathbb{R}^{n \times n}$. Let k denote the smallest positive integer for which $h_{k+1,k} = 0$, with the convention that $k = n$ if H is unreduced. If $q_1 = v_1$, then $q_i = \pm v_i$ and $|h_{i,i-1}| = |g_{i,i-1}|$ for $i = 2 : k$. Moreover, if $k < n$, then $g_{k+1,k} = 0$.*

The proof is in [12].

Now, we can significantly simplify the double shift. For any transformation matrix Z_1 such that:

- Z_1 transforms H_1 to an upper Hessenberg matrix
- the first column of Z_1 is the same as the first column of Z

- both $Z^T HZ$ and $Z_1^T HZ_1$ are unreduced upper Hessenberg matrices

We can conclude from theorem 4 that $Z^T HZ$ and $Z_1^T HZ_1$ are essentially equal [12].

The task of finding the Z is therefore much simplified [12]:

- find the first column of the M
- determine the Householder reflector P_0, which transforms the first column of the M to a multiple of e_1
- Apply P_0 to H and compute Householder reflectors $P_1, \ldots, P_{n-2}$, such that $Z_1 = P_0 \cdots P_{n-2}$ and $Z_1^T HZ_1$ is an upper Hessenberg.

An algorithm executing one step of the Francis double shift QR algorithm is the algorithm algorithm 3. This algorithm is also used in [3]. The overall process

Algorithm 3 The Francis QR step (from [12])

Require: An unreduced upper Hessenberg matrix H
Ensure: An upper Hessenberg matric H_2 (double shifted)

$m = n - 1$;
$s = H(m,m) + H(n,n)$;
$t = H(m,m)H(n,n) - H(m,n)H(n,m)$;
$x = H(1,1)H(1,1) + H(1,2)H(2,1) - sH(1,1) + t$;
$y = H(2,1)(H(1,1) + H(2,2) - s)$;
$z = H(2,1)H(3,2)$;
for k $= 0 : n-3$ **do**
  $[v, \beta] = house([x,y,z]^T)$
  $q = max\{1, k\}$
  $H(k+1 : k+3, q : n) = (I - \beta vv^T)H(k+1 : k+3, q : n)$
  $r = min\{k+4, n\}$
  $H(1 : r, k+1 : k+3) = H(1 : r, k+1 : k+3)(I - \beta vv^T)$
  $x = H(k+2, k+1)$
  $y = H(k+3, k+1)$
  if
    then$z = H(k+4, k+1)$
  end if
end for
$[v, \beta] = house([x,y]^T)$
$H(n-1 : n, n-2 : n) = (I - \beta vv^T)H(n-1 : n, n-2 : n)$
$H(1 : n, n-1 : n) = H(1 : n, n-1 : n)(I - \beta vv^T)$

can differ. [3] assumes that we deflate the matrix from the bottom right corner (continualy shrinking the matrix), [12] on the other hand assumes we split the matrix in three parts (one of undetermined properties, one upper quasi triangular matrix and one unreduced upper Hessenberg matrix) and work with only one part (the unreduced upper Hessenberg matrix).

4.12 Eigenvectors

The QR algorithm can be modified to produce the real Schur form. Not only the quasi upper triangular matrix. The necessary modifications are described in [3].

The QR algorithm can, therefore, be used to find eigenvectors [38]. One possibility is to run the QR algorithm without accumulating transformation matrix and then perform the second pass with exact shifts and the accumulation of the transformation matrix. We'll use another method, which is described both in [38] and [12].

A method called inverse iteration (with an exact shift) can be employed to find specific eigenvector (algorithm algorithm 4). The stopping criterion and error

Algorithm 4 Inverse Iteration

q_0 = random unit 2-norm vector
for k = $1, 2, \ldots$ **do**
  Solve $(A - \lambda I)z^(k) = q^(k-1)$
  $q^(k) = \frac{z^(k)}{|z^(k)|_2}$
end for

analyze can be found in [38]. [38] also describes why it is not important whether $A - \lambda I$ is ill conditioned or not.

[12] notes that the upper Hessenberg form can be used to perform the inverse iteration in $\mathcal{O}(n^2)$. Details for solving Hessenberg systems are in [21]. This is particularly interesting, since the upper Hessenberg form is the input of the QR algorithm. [12] summarizes the approach in following steps:

- *compute the Hessenberg decomposition* $U_0^T A U_0 = H$
- *apply the double implicit shift Francis iteration to* H *without accumulating transformation*
- *for each computed eigenvalue* λ *whose corresponding eigenvector* x *is sought, apply* algorithm 4 *with* $A = H$ *to produce a vector* z *such that* $Hz \approx \lambda z$
- *set* $x = U_0 z$

[12] further claims that: "*A widely followed rule of thumb for deciding upon a suitable eigenvector method is to use inverse iteration whenever fewer than 25 % of the eigenvectors are desired.*"

Chapter 5

State of the art

We examined several articles and dozens of other resources. We did not find any paper solving the same problem. This chapter summarizes the state of the art of the unsymmetric eigenvalue problem, its parallelization, software tools and the RANSAC parallelization.

5.1 QR algorithm

The parallelization of the QR algorithm is a well-studied problem. It is often used in practice when many eigenvalues/eigenvectors are needed, or when we have no prior knowledge about eigenvalues (most iteration methods require some guess about the searched eigenvalue). Unfortunately, all the research we found is either concerned with large dense unsymmetric matrices or small symmetric/Hermitian matrices.

The state of the art represents LAPACK for serial implementation and ScaLAPACK for parallel implementation [19], [13]. The three common strategies for speeding up the QR algorithm are chains of bulges [19], aggressive early deflation [6] and bigger bulges (multishift) [4]. There are also some more robust deflation criterions [1].

The state of the art, implemented in libraries, is aimed at large problems, usually with different computation architecture in mind.

Closer to our problem is [7], author search eigenvalues and eigenvectors of a large number of small Hermitian matrices on a GPU. Small matrices are, in the context of this article, matrices of sizes 128, 256, 512 or 1024.

Our problem is rather specific. We are interested only in real eigenvectors. Our matrices are smaller than 128, unsymmetric and nonderogatory.

5.1.1 Software resources

Serial implementations use the library Eigen. Eigen is a very popular [14] C++ library, which offers linear algebra routines. It is fast [15], reliable and easy to use.

We are familiar with the library cuBLAS, which offers several batched functions. Unfortunately, none of them are appropriate (the Hessenberg form allows us to create our own more effective batched function).

There are some GPU-accelerated libraries which solve eigenvalue problem for dense unsymmetric matrices. We are aware of the library CULA [18], which is built upon the cuBLAS [23]. The next library is MAGMA [5], which is the successor of LAPACK/ScaLAPACK.

We are currently unaware of any software which would solve the dense eigenvalue problem for the small matrix in batches. Creating the solution from CULA/MAGMA routines is likely to be underperforming. MAGMA is abstracting the device, so we do not work with device memory directly. Since [27] encourages the minimizing of data transfers between the host and the device, the MAGMA is probably not the right choice.

CULA offers an interface that follows the standards from NVIDIA cuBLAS [23]. Moreover, CULA could be a reasonable choice for larger matrices because we have direct access to CUDA streams and could pipeline our kernels with CULA functions. We did not pursue this idea since we would have been forced to use the CULA memory model and CULA is non-free.

The library cuSOLVER [24] documentation has a chapter entitled "Dense Eigenvalue Solver", but the chapter contains only functions for bidiagonalization and SVD.

We conclude that none of the freely available libraries provides tools ideal for our goal.

5.2 RANSAC

We searched articles concerning GPUs and RANSAC. [34] describes parallelization of MC-RANSAC by testing several hypotheses at once. Other materials mentioned the computation of homography using CUDA, but without much details.

The closes work we managed to locate is [17]. The article compares CUDA, OpenMP and POSIX threads on the problem of fitting a plane to a set of observations in 3D space. We choose to concentrate on different problem and comparision is therefore impossible. Moreover, the authors did not published the implementation and the data (or we were unable to locate them) and the describtion is very vague.

Chapter 6

Contribution

,

6.1 Parallelization of the model testing

Minimal problems are mostly part of the RANSAC scheme. We worked with the relative pose estimation in the RANSAC scheme. Preliminary results showed that most of the time is spent with model testing (approximatly 94.8%).

The solver, while relatively complex, works with a small subset of input data. The model testing, on the other hand, works with full dataset and performs simpler operations independently. This simple observation explains preliminary results from the preceding paragraph and leads to the conclusion that speeding up the solver is meaningless without significantly improving the runtime of the model testing. We realized that model testing can often be parallelized. This chapter describes the method we arrived to and its limitations.

6.2 Generall approach

The model testing might be performed by numerous error functions. For an effective CUDA implementation, an error function should have the following properties.

- An error computation should be expressible using CUDA C (i.e. most of the C++ standard library and other libraries without CUDA support cannot be used)
- Computations should be memory and time independent (e.g. computation of different errors should not require each other's results)

The following points are relevant for the optimal performance.

- follow the efficient memory alignment as described in [28] or better yet [27],
- design a branch free code,

- split the code into blocks which size is a multiple of the warp size,

- prefer floating point computations with single precision.

6.3 Case study: Relative pose problem

The five-point relative pose problem is described in [37]. We used a template generated by the automatic solver in the RANSAC scheme. The following subsection describes the implementation of the model testing.

6.3.1 Model testing

The process of model testing is described in [16]. Here we show how we proceeded with the parallelization. We keep the process of the essential matrix decomposition [37] on a CPU, remaining substeps (i.e. triangulation, Sampson error, thresholding inliers) were carried on a GPU. Each substep has its kernel and the communication is done through the global memory.

Each thread is responsible for one pair of points. The implementation uses registers extensively. We do not use the constant memory for the data distributed between all threads (a rotation matrix, a translation vector, a fundamental matrix) and rely on compiler and cache. The Figure 6.1 shows the flow of the input data in the model testing.

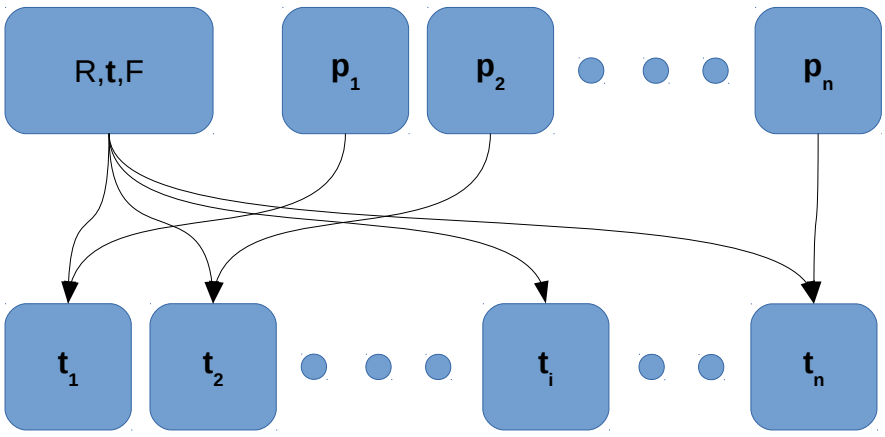


Figure 6.1: Data model, $\mathbf{p}_1$,...,$\mathbf{p}_n$ are input point pairs, $\mathbf{t}_1$,...,$\mathbf{t}_n$ are threads (one thread per one pair). GPU broadcast data common to all threads (rotation matrix R, translation vector $\mathbf{t}$ and fundamental matrix F)

An output for different kernels differs. The triangulation and the Sampson error are passed to the kernel which checks if triangulated points lay in front of the camera and the Sampson errors are below the threshold. The kernel then increases the counter of inliers by one if the pair of points passes the tests. A scheme of the process is in Figure 6.2

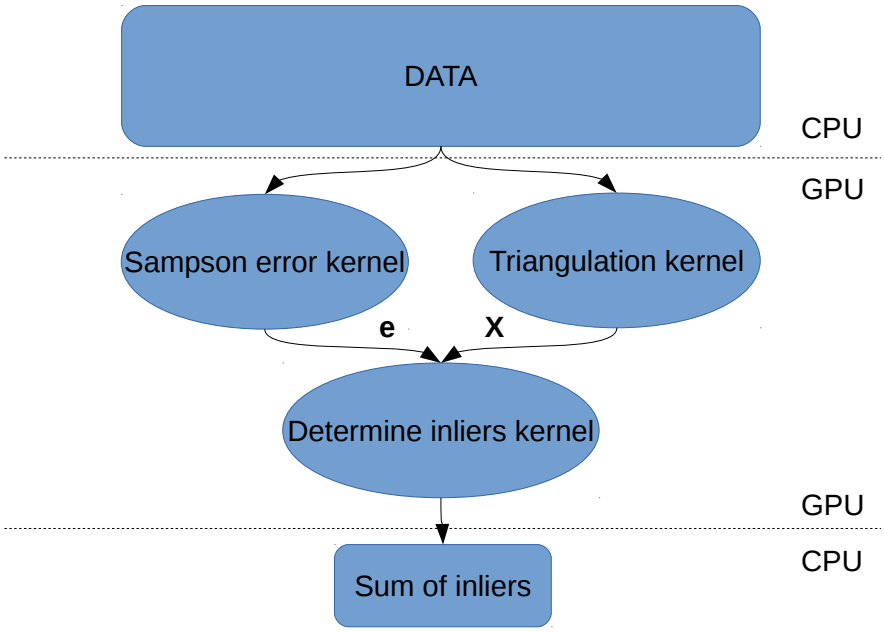


Figure 6.2: Simplified process model

Triangulation

Triangulation is as much simplified as possible. Algorithm excepts the projection matrix P_1 to be a canonical projection matrix, and the P_2 is described only by a rotation and a translation. The simplification means smaller data transfer and a reduction of the computation. The solution of small linear 3x3 systems is done by the Cramer's rule. All arrays are linear.

Sampson error

The input data are in a linear array, besides that, the implementation doesn't differ.

6.3.2 Inliers computation

Each thread determines if the point pair is inliers. Now we need a sum of inliers. We can either use an atomic function and let the device take care about synchrounous access to one variable by multiple threads or we can use the reduction (the reduction is standard algorithm and can be found in [25].

6.4 Simplification of eigenvectors search in the online solver

In Chapter 3 we have showed special properties of the eigenspaces of the multiplication matrix in section 3.2. Therefore, we are certain that each eigenvalue is asocieted with exactly one eigenvector, hence the situation is much simpler then in the general case.

Moreover, we are interested only in real solutions. That implies that the only eigenvectors we are interested in are the real one.

6.5 QR algorithm parallelization

We parallelized the standard for dense eigenvalue problem. Since the main source of parallelization is the batched form, our main objective was to optimize the data transfer and the control logic.

The resulting implementation is based on the serial code from Chapter 4, and as such, they inhereted they properties (including accuracy).

We created several kernels. Each of them performing one part of the QR algorithm:

- transform a real unsymmetric matrix A to the Hessenberg form H
- extract real eigenvalues from the Schur form of the H
- find all real eigenpairs associated with the Hessenberg form
- transform eigenvectors to eigenvectors associated with the input matrix A

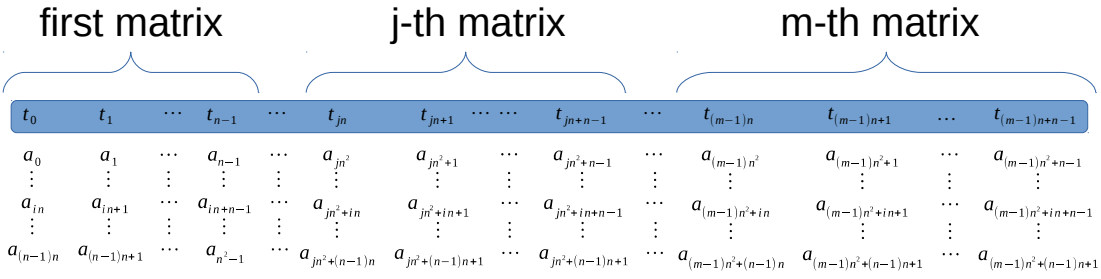


Figure 6.3: **the structure of data and threads**, each thread is associated with one column/row of one matrix. Threads are denoted as t in the figure. Matrices are saved in linear array. Elements of matrices are denoted as a.

6.6 Data and thread structure

The input data are stored in a linear array. The position of each matrix in the linear array is depicted in Figure 6.3. This representation is used in the global memory and the shared memory. We can use a macro for accessing elements in a 2D fashion.

It's a common practise to use one thread per one element of a matrix (e.g. matrix multiplication [28]). We choose a different organization of threads. The Figure 6.3 shows that each thread works with one column (or row).

It's clear from Chapter 4 that the algorithms use increasingly smaller portion of matrix. This mean that less and less threads will participate in computation. But since threads are not associated with elements of matrix but columns, the remaining threads have less work to do.

6.6.1 Hessenberg form

Obtaining the Hessenberg form is an essential preprocessing step. We used householders reflectors as described in Chapter 4. We also collected the transformation matrix U, which is necessary for obtaining eigenvectors.

We based our implementation on the one in [3]. We store the transformation vectors in zero elements of constructed Hessenberg matrix and one extra vector. And we use the thread structure to work at the vector level, we modify whole vectors at once, rather than per element.

The Euclidean norm of vectors is computed using the reduction [25].

block size

Optimal decomposition is dependent on the thread and data structure. We prefer threads to be aligned on warps. The thread's block size is computed using Equation 6.1.

$$T = min(lcm(W_{size}, n), MAX(device)) \tag{6.1}$$

where T is number of threads in one block, W_{size} is the size of the warp and n is the size of a square input matrix. MAX is a function of a device and represents either limitations by shared memory (for larger matrix) or device limits for threads per block .

The minimization is desirable since synchronization is done on the block level. With more blocks per SM, we have lower penalization from synchronization (as mentioned in Chapter 2 there are many threads but only few warps work simultaneously).

The usage of lcm is given by properties of warps, the idea is to have all threads working. Additional details are in the section 2.3.

6.6.2 Real eigenvalues

As mentioned in subsection 4.1.1 the real Schur decomposition allows us to extract eigenvalues. The used algorithm to obtain the upper quasi triangular matrix is Francis double shift qr algorithm [12].

The algorithm had to be modified in order to allow the synchronization. We add a counter of finished instances and stop the block once every instance has finished. This step is necessary but might harm the overall performance, since all instances in a block are limited by the slowest one. We can control this behaviour by creating smaller blocks.

The data structure and thread structure are identical with those in .

The control of threads is sketched in Figure 6.4.

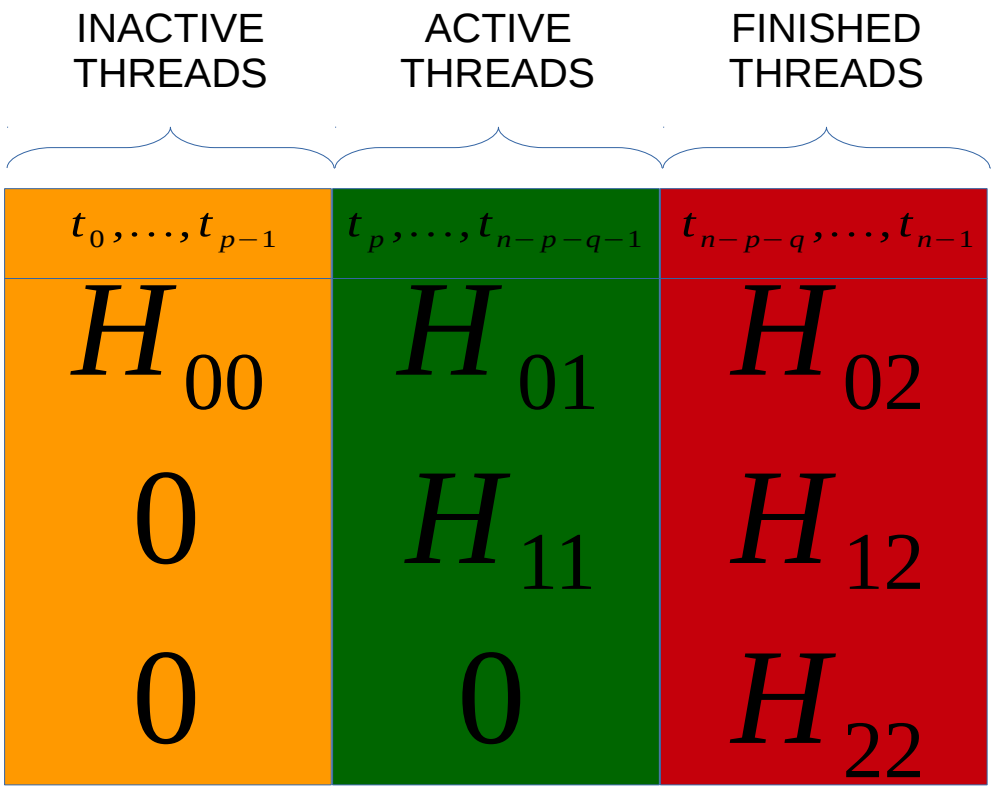


Figure 6.4: **QR algorithm parallelization**, the algorithm performs the QR iterations only with the matrix H_{11}. This would normaly mean creating a loop. We instead activate/deactivate appropriate threads.

We observed that matrix produced by the algorithm is indeed a quasi upper triangular with a block 1x1 and 2x2 on the diagonal. Unfortunately, due to the convergence property and the deflation strategy, some blocks of 2x2 elements represent pair of real eigenvalues.

Because of decoupling (section 4.4). We can simply determine if the eigenvalues are real or complex from the 2x2 blocks.

6.6.3 real eigenvectors

We have shown how to use inverse iteration with exact shift to find eigenvectors section 4.12. And once again, we didn't need to change data and threads structure.

6.7 Latency hiding

One problem with the GPU is the costly transfer of data from a host to a device and vice versa. We address this problem by using a page-locked memory (details in [28]). The Page-locked memory prevent the allocated host memory from swaping. Hence the transfer is faster and asynchronous. The asynchrounous transfer allows us to use streams and perform computations both on the device and the host simultaneously. The hybrid computation model is in Figure 6.5.

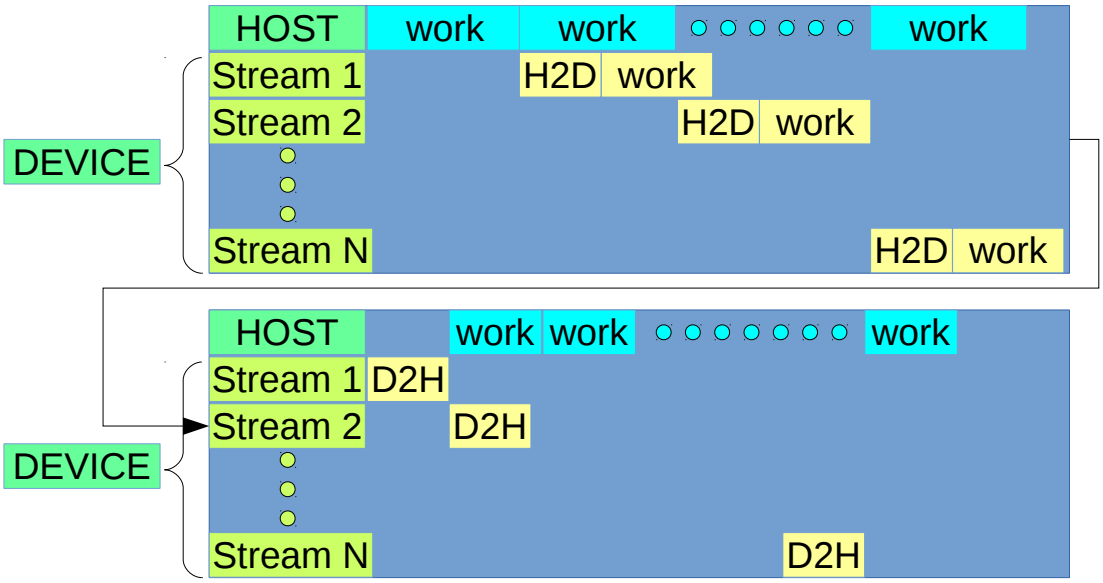


Figure 6.5: **Asynchrounous execution**, H2D and D2H are functions performing data transfer between a host and a device. "Work" has different meaning for host and device. "Work" in the top rectacle means computation of multiplication matrix for the host and computation of solutions for the device. "Work" in the bottom rectangle means appling the results of the solution.

Chapter 7

Results

We will show performance of implemented algorithms based on the input data and in-depth analysis of selected kernels.

7.1 Hardware

All test were performed on the GPU NVIDIA GeForce GTX 750 Ti and a single-core of the Processor Intel(R) Core(TM) i7-5820K CPU @ 3.30GHz.

7.2 The Unsymmetric Eigenvalue Problem

The results are shown for several sizes of matrices. The input matrices are random nonderogatory matrices. Output are real eigenvectors. Notice periodicity of results in Figure 7.1 and Figure 7.3. We will address this phenomen in next chapter. Figure 7.2 shows the impact of memory allocation and memory transfer.

7.2.1 Input Data

Input are random nonderogatory matrices. We generated random diagonal matrices with unique eigenvalues and multiply it by random orthogonal matrix.

The random orthogonal matrix is the orthogonal matrix Q from the QR decomposition of random matrix. There exist more solid way of finding random orthogonal matrix [22], but the convergence properties are given by eigenvalues and we can therefore relax on the distribution of our orthogonal matrices.

7.3 Hybrid solver

The results of hybrid solver are in Figure 7.4, Figure 7.5.

7.4 case study:Relative pose problem

The five-point relative pose problem was tested on two datasets(Figure 7.6,Figure 7.7). The relative speed up is in Figure 7.8.

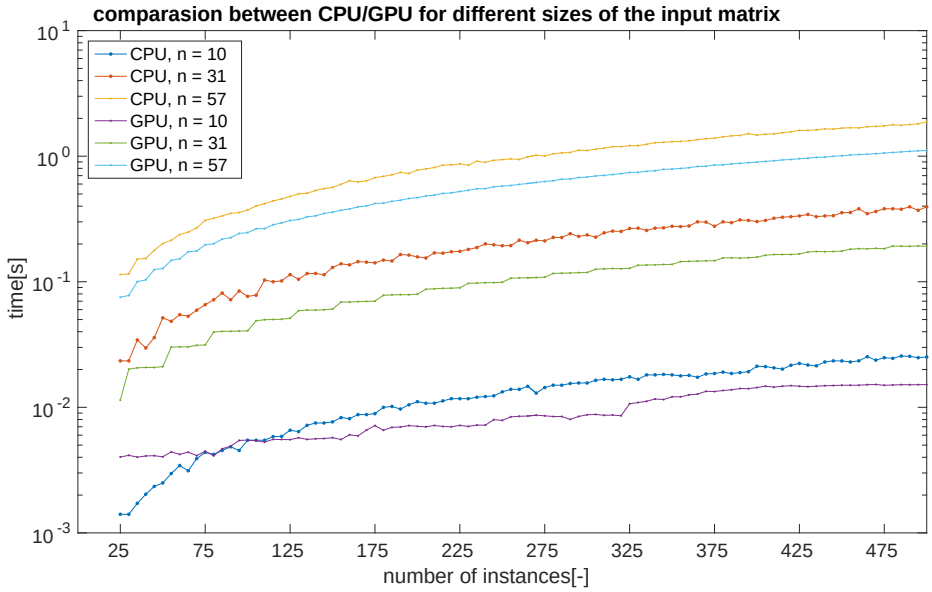


Figure 7.1:

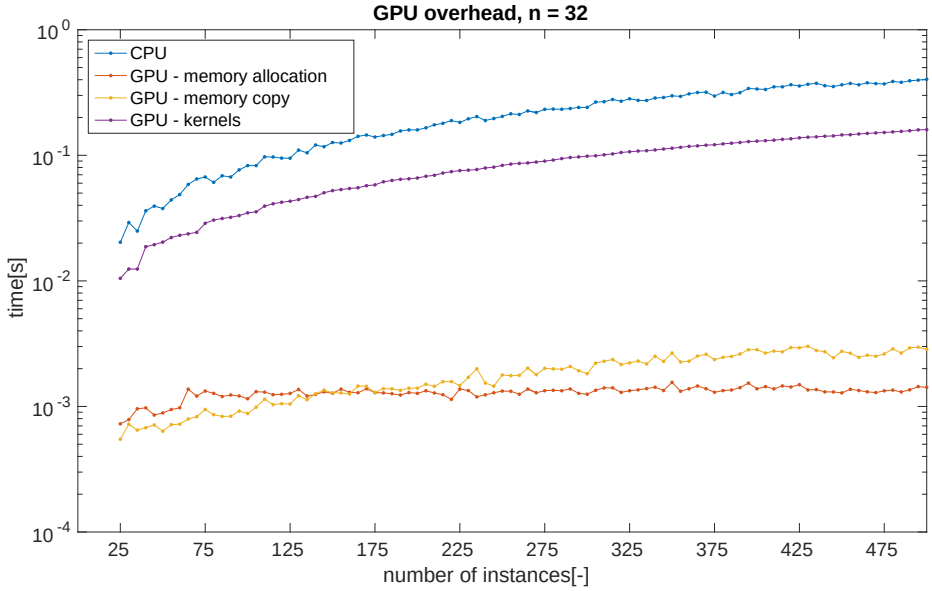


Figure 7.2:

7.5 Selected profiler results

7.5.1 The eigenvalue kernel

We used matrix of size 10 and run the kernel 500 times. Selected results are in Figure 7.9 and Figure 7.10.

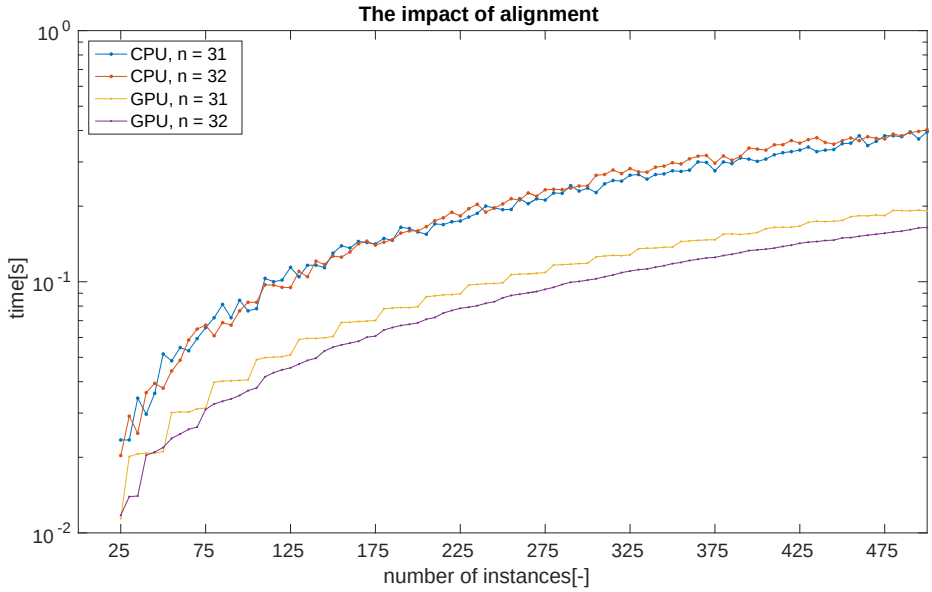


Figure 7.3:

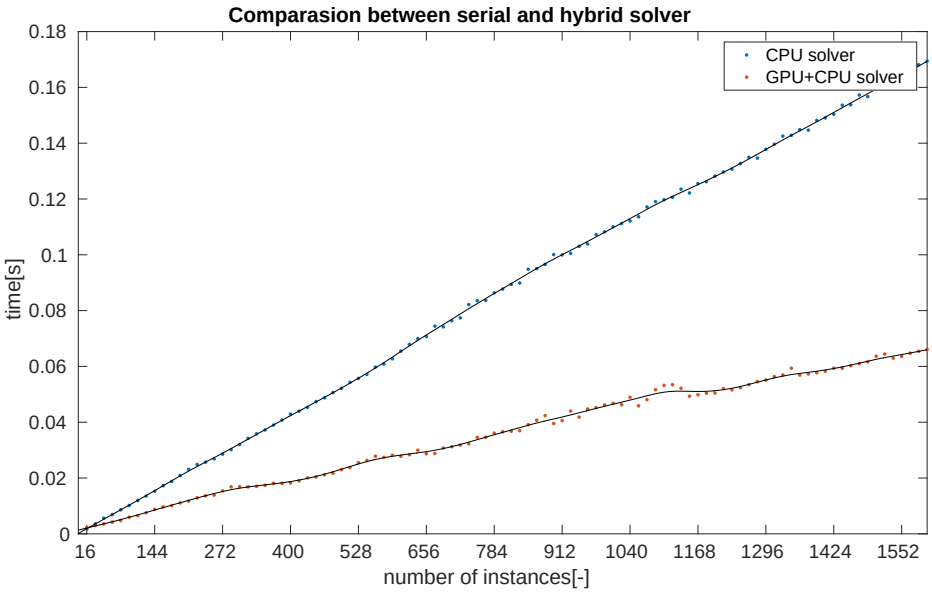


Figure 7.4:

7.5.2 The triangulation kernel

The number of tested hypotheses is 160, the number of points is 2000. Selected results are in Figure 7.11 and Figure 7.12.

7.5.3 The sampson kernel

The number of tested hypotheses is 160, the number of points is 2000. Selected results are in Figure 7.13.

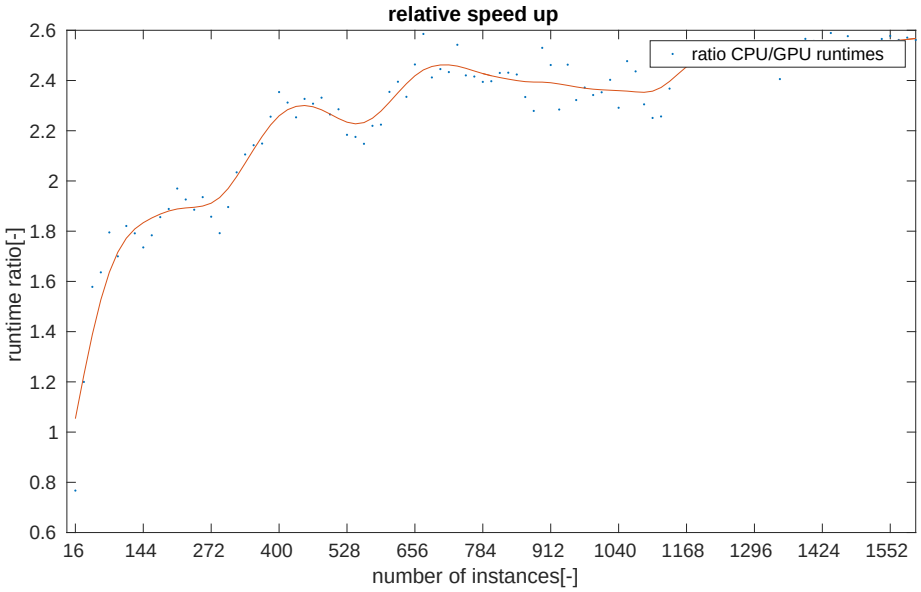


Figure 7.5:

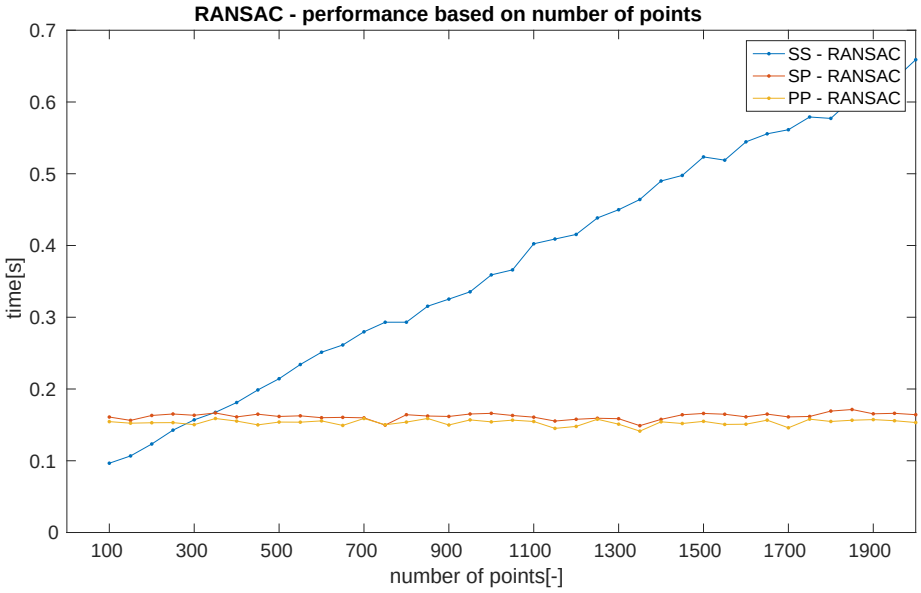


Figure 7.6: SS - RANSAC: serial implementation; SP - RANSAC: parallel verification; PP - RANSAC: parallel verification and solver

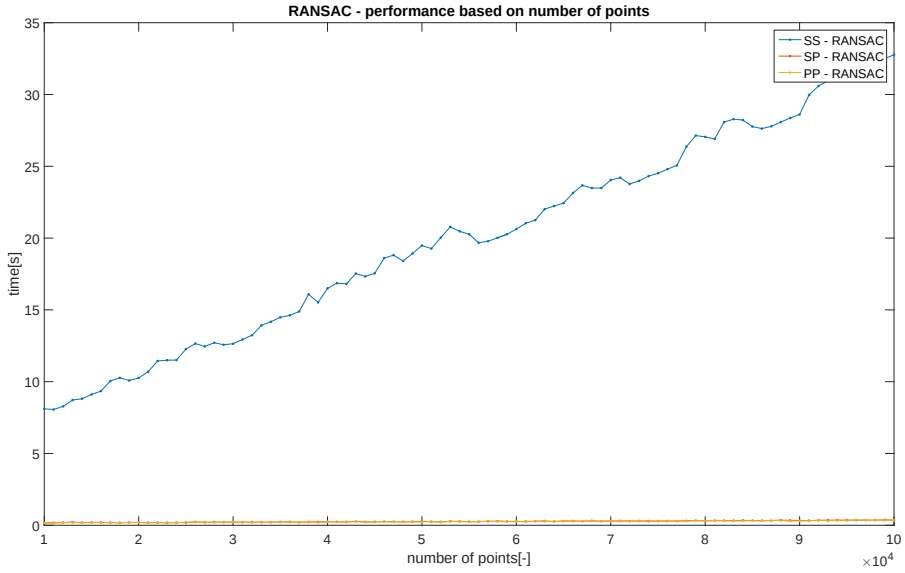


Figure 7.7: SS - RANSAC: serial implementation; SP - RANSAC: parallel verification; PP - RANSAC: parallel verification and solver

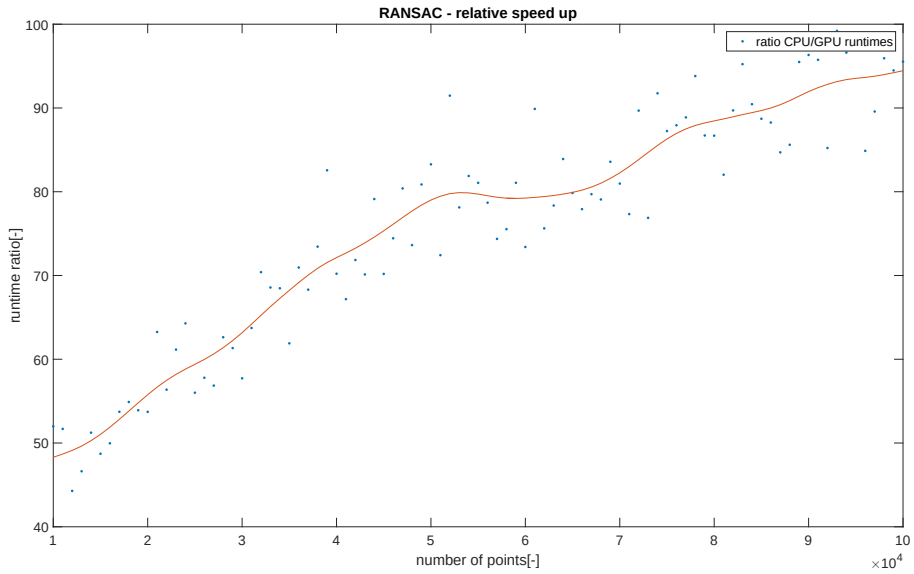


Figure 7.8:

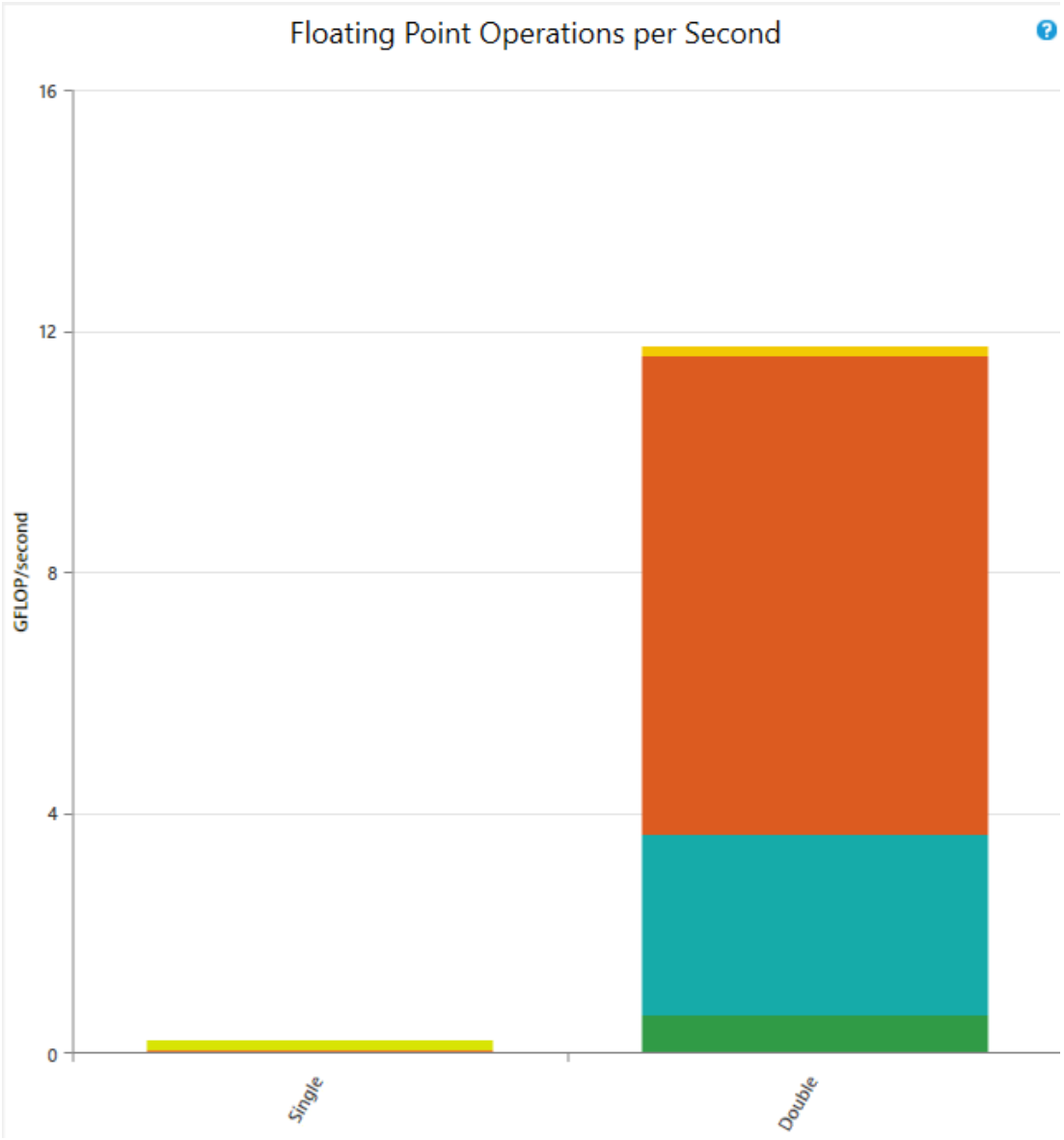

Floating Point Operations per Second
16
12
8
4
0
GFLOP/second
Single
Double

Figure 7.9:

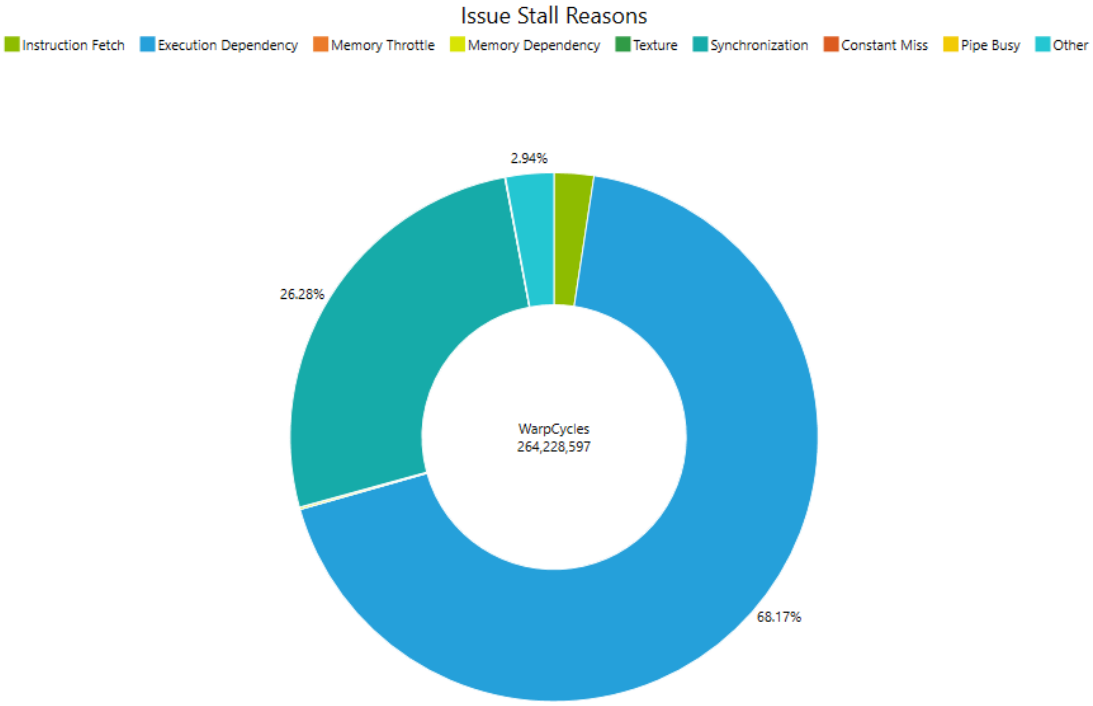

Issue Stall Reasons
Instruction Fetch
Execution Dependency
Memory Throttle
Memory Dependency
Texture
Synchronization
Constant Miss
Pipe Busy
Other
2.94%
26.28%
68.17%
WarpCycles
264,228,597

Figure 7.10:

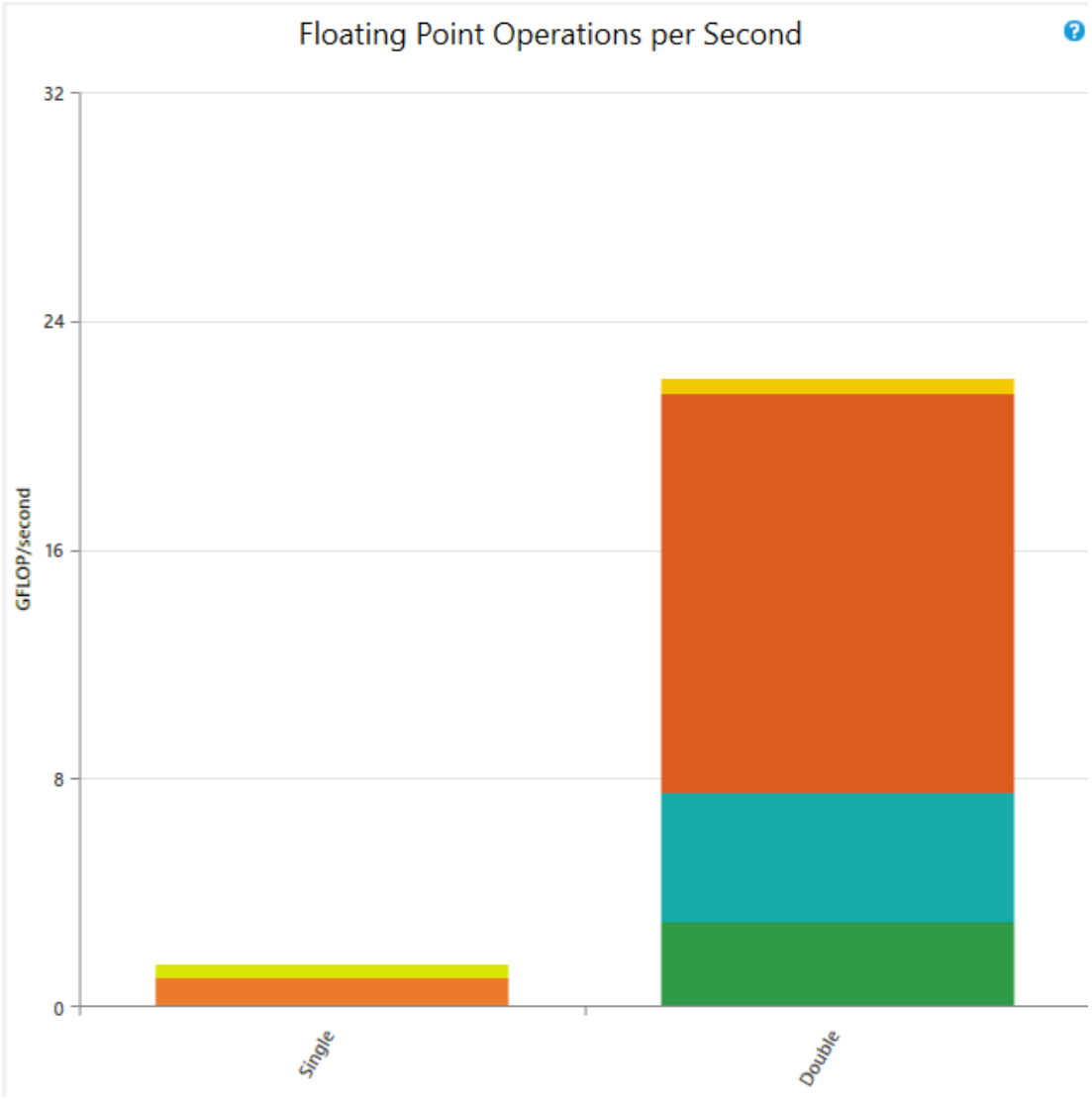

Floating Point Operations per Second
32
24
16
8
0
GFLOP/second
Single
Double

Figure 7.11:

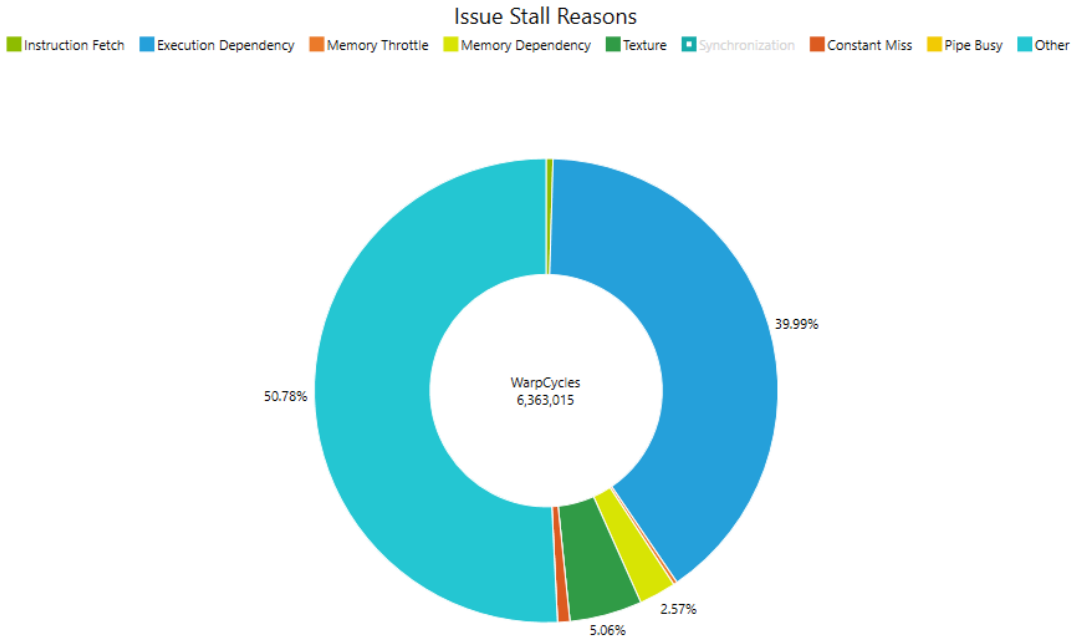


Figure 7.12:

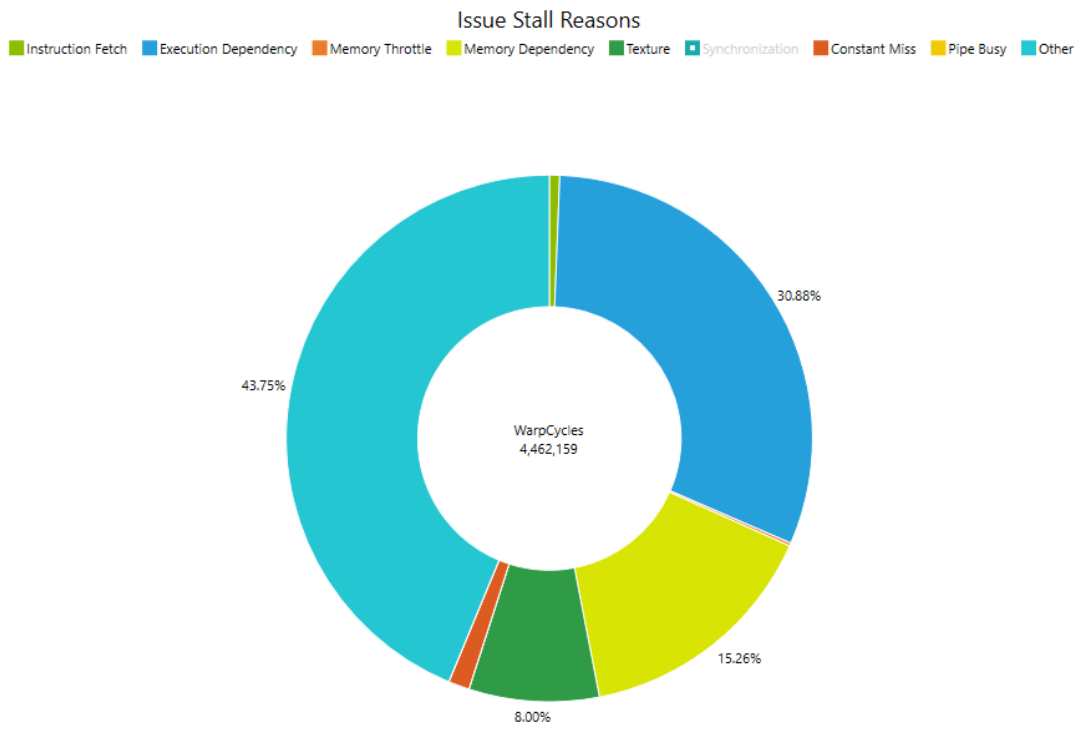


Figure 7.13:

Chapter 8

Discussion

In this chapter, we will point at interesting observations and discuss the merits of proposed methods.

8.1 The QR algorithm

The direct comparison between the GPU and CPU implementation requires caution. The library function for solving the eigenvalue problem is computing redundant complex eigenvectors. Moreover, there is an associated overhead with repeatedly calling the serial function inside the loop. The reader should bear this in mind when reading Chapter 7.

The GPU has the overhead connected with memory allocation(which can be removed) and memory transactions(which cannot be removed). Both overheads are shown in Figure 7.2.

Figure 7.1 shows that the speed up is achieved in groups. The group size is determined by device limits (i.e., if k instances can run simultaneously on the device the groups have size k).

The results show that a single-core CPU is usually outperformed by the GPU. However, the difference is not big (usually twice as fast) and the GPU implementation would be slower than the implementation using multiple CPU threads. It is because the GPU limits are relatively soon exhausted due to low occupancy (about 20 %).

8.2 The hybrid solver

The solver speed up in Figure 7.4, even for smaller instances, originate in the latency hiding. We are not comparing CPU with GPU, but rather a combination of CPU+GPU with CPU. One advantage is that we can keep the data on GPU for the following step(the verification).

We have seen in Figure 7.5 that the achieved speed up is, therefore, actually higher than that for the unsymmetric eigenvalue problem alone.

8.3 The RANSAC

The RANSAC mainly profits from the parallelized verification. The excellent scaling properties in Figure 7.6 make this method truly interesting. Note that the speed up of the solver is insignificant for selected batch sizes and the number of points seemingly does not matter.

We have seen the extent of the speed up in Figure 7.8.

8.4 Future work

The verification is currently running on default stream (overlapping with CPU is disabled). The results in Figure 7.4 and Figure 7.7 suggest that overlapping the generation of hypotheses with verification might be even more efficient.

An open question remains on whether the QR algorithm is the best option. Perhaps some simpler algorithm (power iteration, inverse iteration) would achieve better results. Moreover, the testing of other problems and various RANSAC modifications should be the next priority.

Chapter 9

Conclusion

In this work, we have examined how to speed up the solver provided by the automatic generator [35].

First, we have shown that since the solver is usually used inside RANSAC [20] a more pressing matter is the effective verification of the hypotheses. We suggested and successfully parallelized the verification of the 5-point relative pose problem hypotheses. The implementation is branchless and considerably faster (including overhead) for larger data sets. We noticed that the size of the test set is much less significant, and the runtime is often dictated by overhead and number of generated hypotheses. We conclude that the resulting method has excellent scalability.

Next, we concentrate on the automatically generated solvers. We have seen [20] that the eigenvalue problem is the most time-consuming part. Hence, we designed a specialized solver of the eigenvalue problem. We used the special properties of the matrices and modified the known algorithm [12]. We parallelized the algorithm [12], and we have shown its speed and correctness on random data. The scalability properties were satisfactory, with open possibilities of improvement.

The predetermined goals were fulfilled to our expectations. The suggested methods can significantly speed up hybrid systems.

Bibliography

[1] Mario Ahues and Francoise Tisseur. A new deflation criterion for the QR algorithm. Technical Report 122, LAPACK Working Note, March 1997.

[2] Derek Wilson Anand Lal Shimpi. Nvidia's geforce 8800 (g80): Gpus re-architected for directx 10, 2006. Accessed: 2016-05-24.

[3] Peter Arbenz. The QR algorithm (lecture notes), March 2016.

[4] Zhaojun Bai and James W. Demmel. On a block implementation of hessenberg multishift QR iteration. Technical Report 8, LAPACK Working Note, January 1989.

[5] Wieb Bosma, John Cannon, and Catherine Playoust. The Magma algebra system. I. The user language. *J. Symbolic Comput.*, 24(3-4):235–265, 1997. Computational algebra and number theory (London, 1993).

[6] Karen Braman, Ralph Byers, and Roy Mathias. The multishift qr algorithm. part ii: Aggressive early deflation, 2002.

[7] Alain Cosnuau. Computation on GPU of eigenvalues and eigenvectors of a large number of small hermitian matrices. In *Proceedings of the International Conference on Computational Science, ICCS 2014, Cairns, Queensland, Australia, 10-12 June, 2014*, pages 800–810, 2014.

[8] John Little David Cox and Donal O'Shea. *Ideals, Varieties, and Algorithms*. Undergraduate Texts in Mathematics. Wiley, 2nd edition, 1997.

[9] John Little David Cox and Donal O'Shea. *Using Algebraic Geometry*. Graduate Texts in Mathematics. Springer New York, 2005.

[10] Martin A. Fischler and Robert C. Bolles. Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography. *Commun. ACM*, 24(6):381–395, June 1981.

[11] J. G. F. Francis. The QR Transformation A Unitary Analogue to the LR Transformation (Part 1 and 2). *The Computer Journal*, 4:265–271, 332–45, 1961.

[12] Gene Howard Golub and Charles F. Van Loan. *Matrix computations*. Johns Hopkins studies in the mathematical sciences. The Johns Hopkins University Press, Baltimore, London, 1996.

[13] Robert Granat, Bo Kagstrom, and Daniel Kressner. A novel parallel qr algorithm for hybrid distributed memory hpc systems. Technical Report 216, LAPACK Working Note, April 2009.

[14] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[15] Gaël Guennebaud, Benoît Jacob, et al. Eigen benchmark. http://eigen.tuxfamily.org/index.php?title=Benchmark, 2011.

[16] Richard Hartley and Andrew Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, New York, NY, USA, 2 edition, 2003.

[17] Alejandro Hidalgo-Paniagua, Miguel A. Vega-Rodríguez, Nieves Pavón, and Joaquín Ferruz. A comparative study of parallel ransac implementations in 3d space. *Int. J. Parallel Program.*, 43(5):703–720, October 2015.

[18] K. E. Spagnoli A. L. Paolini E. J. Kelmelis J. R. Humphrey, D. K. Price. Cula: Hybrid gpu accelerated linear algebra routines,. *SPIE Defense and Security Symposium (DSS)*, April 2010.

[19] Lars Karlsson and Daniel Kressner. Optimally packed chains of bulges in multishift qr algorithms. Technical Report 271, LAPACK Working Note, August 2012.

[20] Zuzana Kúkelová. *Algebraic Methods in Computer Vision*. PhD thesis, Department of Cybernetics, Faculty of Electrical Engineering, Czech Technical University in Prague, February 2013.

[21] Charles F. Van Loan. *Introduction to Scientific Computing*. The MATLAB Curriculum Series. Prentice-Hall, 2nd edition, 2000. A Matrix-Vector Approach Using MATLAB.

[22] Francesco Mezzadri. How to generate random matrices from the classical compact groups. *Notices of the American Mathematical Society*, 54(5):592 – 604, 5 2007.

[23] NVIDIA. *CUBLAS Library User Guide*. NVIDIA, v7.5 edition, September 2015.

[24] NVIDIA. *cuSOLVER Library*. nVidia, v7.5 edition, September 2015.

[25] NVIDIA Corporation. CUDA Samples, 2015. Version 7.5.

[26] NVIDIA Corporation. CUDA toolkit documentation, 2015. Version 7.5.

[27] NVIDIA Corporation. NVIDIA CUDA C best practices guide, 2015. Version 7.5.

[28] NVIDIA Corporation. NVIDIA CUDA C programming guide, 2015. Version 7.5.

[29] NVIDIA Corporation. NVIDIA CUDA Compiler Driver NVCC, 2015. Version 7.5.

[30] A. Quarteroni, R. Sacco, and F. Saleri. *Numerical Mathematics*. Texts in applied mathematics. Springer, 2000.

[31] Philip E. Ross. Why cpu frequency stalled, 2008. Accessed: 2016-05-24.

[32] FooManchu sfackler, jcmacdon and Arnie. How cuda's abstractions map to a gpu implementation, 2013. Accessed: 2016-05-24.

[33] E.J.D. Tebbens and Univerzita Karlova. Matematicko fyzikální fakulta. *Analýza metod pro maticové výpočty: základní metody*. Matfyzpress, 2012.

[34] P. Trivedi, T. Agarwal, and K. Muthunagai. Mc-ransac: A pre-processing model for ransac using monte carlo method implemented on a gpu. pages 1380–1383, Aug 2013.

[35] Pavel Trutman. Minimal Problem Solver Generator, Bachelor thesis, 2015.

[36] van Oosten J. Cuda memory model, 2014. Accessed: 2016-05-24.

[37] Radim Šára. 3d computer vision, October 2015.

[38] D.S. Watkins. *Fundamentals of Matrix Computations*. Pure and Applied Mathematics: A Wiley Series of Texts, Monographs and Tracts. Wiley, 2004.

[39] J.H. Wilkinson. *The Algebraic Eigenvalue Problem*. Monographs on numerical analysis. Oxford: Clarendon Press, 1965.

[40] Ruigang Yang. Cuda memory architecture, 2011. Accessed: 2016-05-24.