Acceleration of calculation of Third Party Risk around an airport using OpenCL

by

Ronald Erkamp

April, 2013
Vrije Universiteit Amsterdam

Master's Thesis

Supervisors

Dr. Rob van Nieuwpoort
Netherlands eScience Center &
Vrije Universiteit Amsterdam.
Amsterdam, The Netherlands.

Ir. Roalt Aalmoes
Dutch National Aerospace Laboratory (NLR),
Air Transport Environmental & Policy Support (ATEP),
Amsterdam, The Netherlands.

Abstract

During the past two decades, the Dutch National Aerospace Laboratory has developed a model to calculate the risk for third parties around airports. This Third Party Risk model is used in the decision making with respect to airport development and land use planning. Due to the increase of air traffic, the availability of improved individual flight track data, and the need for detailed calculations on a denser grid, the time required for a risk calculation has increased significantly. In this thesis, we present the research of using parallel programming hardware, in particular Graphical Processing Units, to accelerate a risk calculation. Because of different phases in the calculations, and the relation between adjacent grid cells, the translation of the Third Party Risk model into a parallel implementation is not straightforward. Results show that a calculation for Schiphol Airport, based on the traffic for a calendar year and a dense grid that took a week to complete in the original implementation, finishes well within fifteen minutes in the improved parallel implementation.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

ACM	Accident Consequence Model
ALM	Accident Location Model
API	Application Program Interface
APM	Accident Probability Model
AR	Accident Rate
CA	Crash Area
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DPU	Double Precision Unit
GPGPU	General-Purpose GPU
GPU	Graphical Processing Unit
IR	Individual Risk
MTOW	Maximum Take-Off Weight
OpenCL	Open Computing Language
PCIe	Peripheral Component Interconnect express
PDM	Probability Density Matrix
PLL	Potential Loss of Life
SIMT	Single Instruction Multiple Threads
SFU	Special Function Unit
SM	Streaming Multiprocessor
SP	Stream Processor
SR	Societal Risk
TRIPAC	Third-party RIsk analysis PACkage
VLIW	Very Long Instruction Word

Glossary

Device
When we speak about the device in this thesis, we mean the device on which parallel code is executed using OpenCL.

Host
The host is the machine controlling the execution of a parallel program running on an OpenCL-enabled device.

Parallel program
The parallel program is the version of TRIPAC in which parts of the code are executed concurrently on an OpenCL-enabled device.

Sequential program
The sequential program is the version of TRIPAC in which all code is executed consecutively. The original sequential version is the version provided by the NLR. The optimized sequential version is a version in which some inefficiencies of the original version have been eliminated.

Chapter 1

Introduction

During the past two decades, the Dutch National Aerospace Laboratory (NLR) has been performing research for third party risk around airports, and a calculation model and methodology for different types of airport are developed. The Third Party Risk Model is used to evaluate the risk for people living and working around an airport. The design of new or changed air routes and runway infrastructure at airports require that third party risk studies are conducted to determine the impact for the close surroundings. In the last decades, the amount of air traffic has increased significantly, and more detailed individual flight track data has become available. Also, a greater accuracy has been demanded, meaning that calculations have to be done on a denser grid. This all caused the required time to do a risk calculation to increase significantly, especially when multiple scenarios are involved and specific data output is requested.

In an attempt to accelerate risk calculations, we research the possibility to utilize many-core architectures, in particular focusing on Graphical Processing Units (GPUs), and using the standardized OpenCL technology. The many cores can be set to purpose by carrying out massively parallel operations. However, some algorithms are not as suitable for parallel execution as others, due to instruction dependencies, imbalanced workloads, and hardware limitations. It is a challenge to deal with the difficulties imposed by these factors, as it requires modifications to the original algorithm.

GPUs are often used to increase the performance of scientific computations. Scientific fields including computer graphics and the simulation of physical and chemical processes benefit especially, as these computations tend to be highly data-parallel. At the time of this writing, little research has been published on utilizing GPUs in the field of simulation, particularly regarding risk analysis. However, there are two of note. One publication focuses on the assessment of the risk in CO_2 geologic sequestration, which involves executing a simple and computationally efficient simulator on a GPU, resulting in a speedup of 64 over a CPU implementation [1]. Another publication is about a research to accelerate aggregate risk analysis used for supporting real-time pricing, in which a GPU implementation performs 9 times better than a CPU implementation [2]. The simulation in these two researches is based upon a Monte Carlo simulation, in which a computation is executed multiple times using randomized input to result in a distribution function. Since each simulation can be carried out independently, multiple simulations are run in parallel. So, not a simulation execution itself is run on multiple cores, but multiple runs are executed concurrently. Calculating third party risk involves only one run, hence the algorithm itself must be parallelised. Another difference is the programming framework used. The parallel applications of the two researches are implemented using CUDA, whereas our parallel program is implemented using OpenCL.

Our parallel implementation is tested on two NVIDIA GPUs, the Tesla C2075 and the recently released Tesla K20m. Risk calculations on the latter GPU achieve a speedup of over 3000 for a Schiphol Airport scenario over the original sequential implementation. Instead of six days, a risk calculation takes less than fifteen minutes. The significant increase in performance achieved does not only enable faster risk calculations, but also allows more accurate details to be included risk calculations. Besides the significant speedup, the process of translating the original sequential implementation into a parallel implementation, has also led to the finding of performance inefficiencies in the original sequential implementation. To make a fairer comparison between the sequential and parallel versions, we also implemented an optimized sequential version, in which found inefficiencies are eliminated.

The remainder of this thesis is structured as follows. In Chapter 2, we give a detailed explanation of the Third Party Risk model as developed by the NLR. In Chapter 3, we address Graphical Processing Units

(GPUs), mainly focusing on their architecture and how to program them. Chapter 4 is dedicated to a thorough explanation of the improved parallel program. All parts of the program are discussed, from conversion of the original implementation to improved parallel implementation, including the issues encountered. Also, the implementations and optimizations are thoroughly timed and evaluated. The overall conclusions are presented in Chapter 5.

Chapter 2

The Third Party Risk model

This chapter is dedicated to the Third Party Risk model. A good understanding of this model is essential for accelerating the Third Party Risk application (TRIPAC). In Section 2.1 we explain the Third Party Risk model, why it is built and what it computes. That section is followed by a description of the design of the Third Party Risk model, in Section 2.2.

2.1 What is the Third Party Risk model?

The Third Party Risk model is an analysis model for determining risk for third parties, such as homes, companies, and the population around airports. The model is developed by the Dutch National Aerospace Laboratory (NLR). The need for such a model arose in the last decades as the amount of air traffic increased and therewith the realisation of the population that they are exposed to risks because of air traffic. Since the airspace around airports is the most crowded, and the probability of an aircraft accident is the greatest in the take-off and landing phase (see Figure 2-1), the population in the vicinity of airports is exposed to the greatest risk. The increase in air traffic made the Ministry of Traffic decide to let the NLR map the risk of third parties around airports. The model is used for multiple purposes, e.g. determining risk zones and evaluating development plans of airports [3]. The Dutch government determined that the risk that a person, who is located at a fixed location for one year, dies as a consequence of an industrial or transport accident, must not be larger than 10^{-6}. This value includes the risks of other risk factors besides airports, such as the transport of hazardous substances [4].

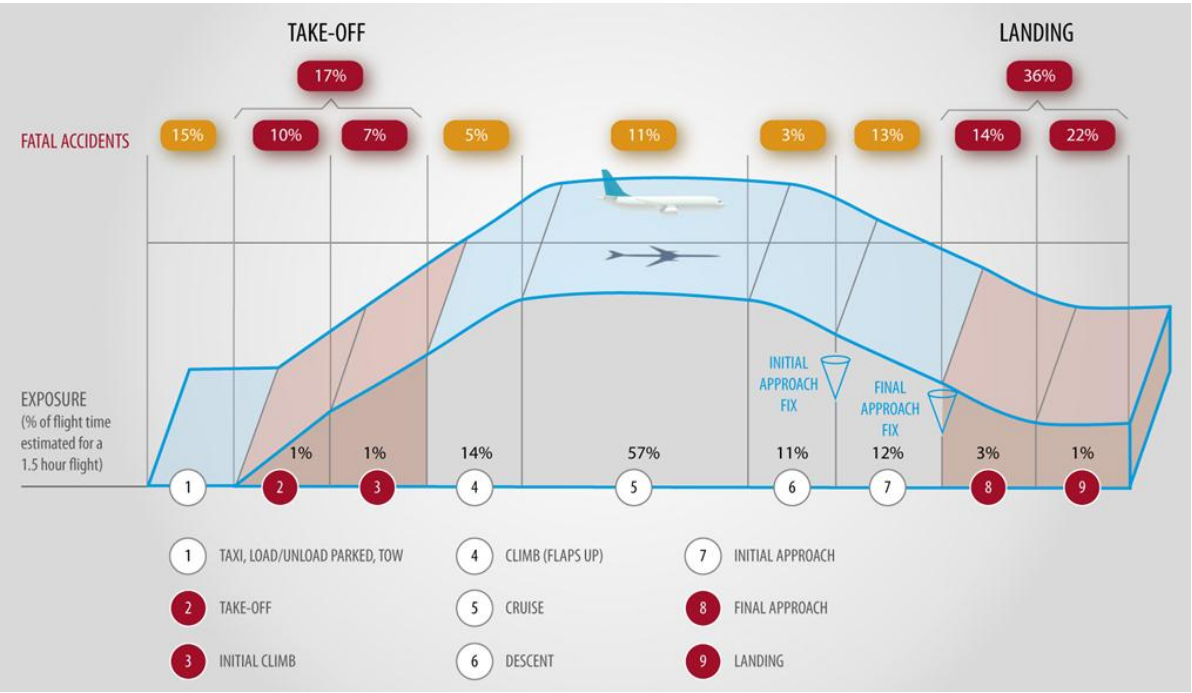

Figure 2-1: Percentage of fatal accidents by flight phase. Even though only 6% of a flight is spent in landing or take-off phase, most fatal accidents happen in these phases. (Source: Statistical Summary of Commercial Jet Airplane Accidents, 1959 – 2008, Boeing).

The NLR developed various calculation models: for large airports, for regional airports and for military airports. The difference between the models is based on a number of criteria like the amount of traffic and the type of airplanes, which may influence certain model parameters. For each run of the Third Party Risk application, one or more models may be requested [3].

The Third Party Risk model has a certain set of input parameters, on which internal processes operate to produce the requested output. The input parameters of the Third Party Risk model can be split into two groups: model parameters and input data of a scenario. The model parameters are characteristics of the model and are derived from past events. The input data of a scenario is the data which characterizes the scenario for which the risk analysis should take place, like which routes certain types of aircraft take and what the type of terrain is for certain locations. All this information will be used to calculate the risk for the scenario. The area for which calculations are requested is called the study area. The default study area for regional airports is 40 by 40 kilometres, with a cell width and cell height of 25 meters. For small airports, a smaller study area may be sufficient. For large airports, a larger study area may be needed. See Figure 2-2 for the study area of Schiphol Airport, which is 56 by 56 kilometres. The airport is situated within the orange circle.

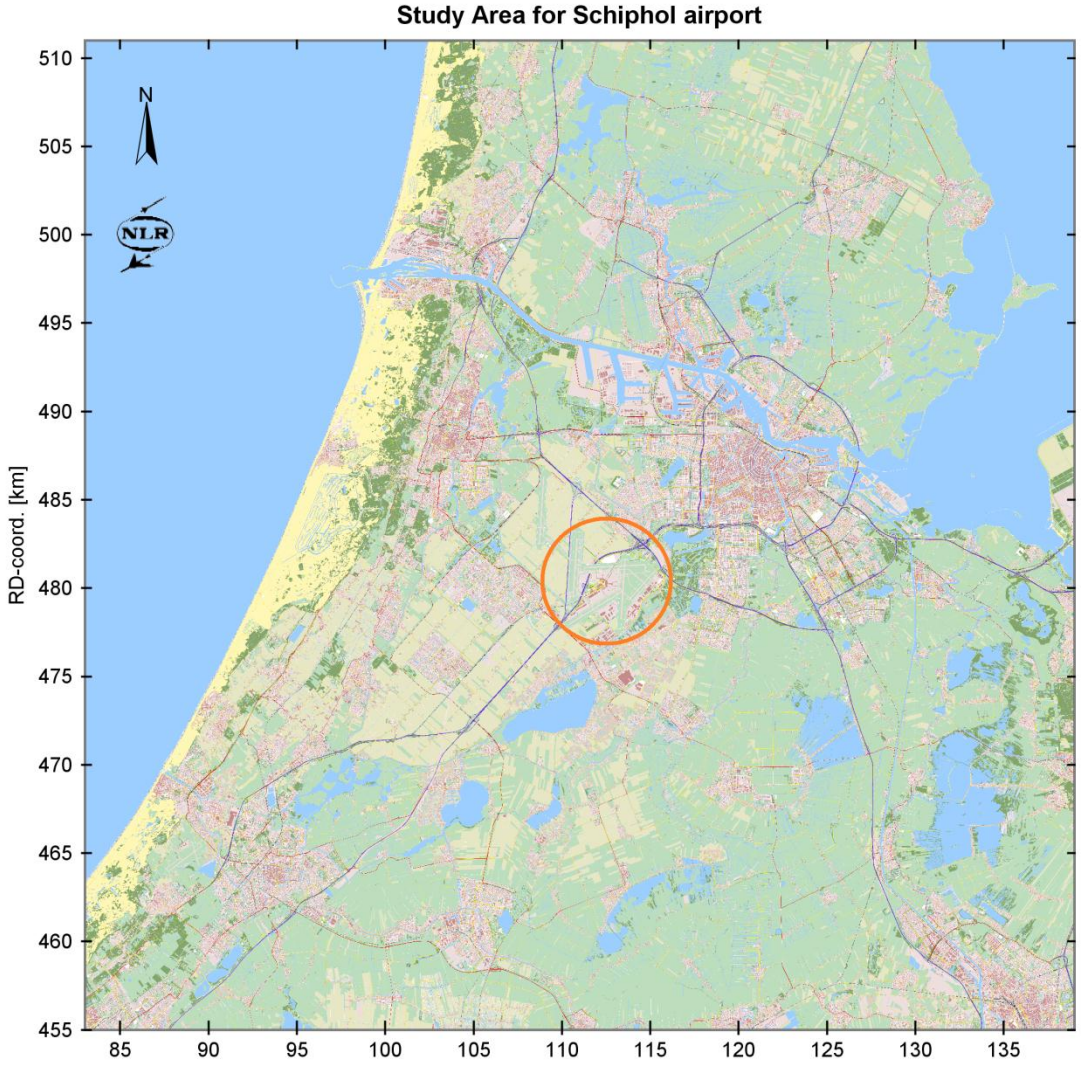

Figure 2-2: Study area for Schiphol Airport of 56 by 56 kilometres (Source: Topografische Dienst, Emmen).

The calculations that can be requested are Individual Risk (IR) and Societal Risk (SR). The Individual Risk is defined as the probability per year that an imaginary person permanently located on a fixed spot in the vicinity of an airport dies as a direct consequence of an aircraft accident [3]. The result of an IR calculation is a probability grid containing an IR value for each cell. The characteristics of this grid are the same as those of the study area as specified by the user. As an extra option, specific output can be requested. For example, one may request to compute additional grids containing the risk caused per accident type and per part of day (day and night). Requesting grids for specific subsets results in extra grids in addition to the overall grid.

The Societal Risk is defined as the probability per year that a group of greater than N persons, located in the study area, die as a direct consequence of a single aircraft accident [3]. To calculate the SR, population densities of the complete study area are required. The result of an SR calculation is a list of group sizes with for each group size the probability that there are more aircraft accident fatalities (see Table 2-1). Just as with IR calculations, specific results may be requested, resulting in extra output files. Also, output as a grid may be requested.

Additional calculations can be performed on the data resulting from IR and SR calculations. For example, from the IR grids contours can be computed to make the output more accessible, or houses and people exposed to a certain minimum risk level can be counted. The risk within a certain area can be calculated, resulting in an accumulated weighted risk. The SR results may be visualized by plotting the group sizes and the probabilities on a double-logarithmic scale. The Potential Loss of Life (PLL) can be derived from this plot by computing its integral [3].

Group size	Risk value
1	1.824315×10^{-3}
3	7.978614×10^{-4}
5	6.251645×10^{-4}
10	4.007661×10^{-4}
20	2.158520×10^{-4}
40	7.038592×10^{-5}
100	7.613916×10^{-6}
200	9.659953×10^{-7}
400	2.259261×10^{-12}

Table 2-1: Example output of a Societal Risk calculation.

2.2 Design

The Third Party Risk application can be split into three main parts: the Individual Risk calculation part, the Societal Risk calculation part, and the additional calculations part. The IR and SR calculations are independent of each other. However, both calculations partially rely on the same data that needs to be computed in advance: the probability densities and the sizes of the crash areas. This data is calculated by a number of sub models: the Accident Probability Model, the Accident Location Model, and the Accident Consequence Model. See Figure 2-3 for the flow chart of these sub models.

Since the data produced by these models is used by the IR and SR calculations, these three models are described first. The additional calculations part relies on the output of the IR and SR calculations, as explained in Section 2.1. The Third Party Risk model design is shown in Figure 2-4. Data is represented by rectangles with squared corners; processes by rectangles with rounded corners.

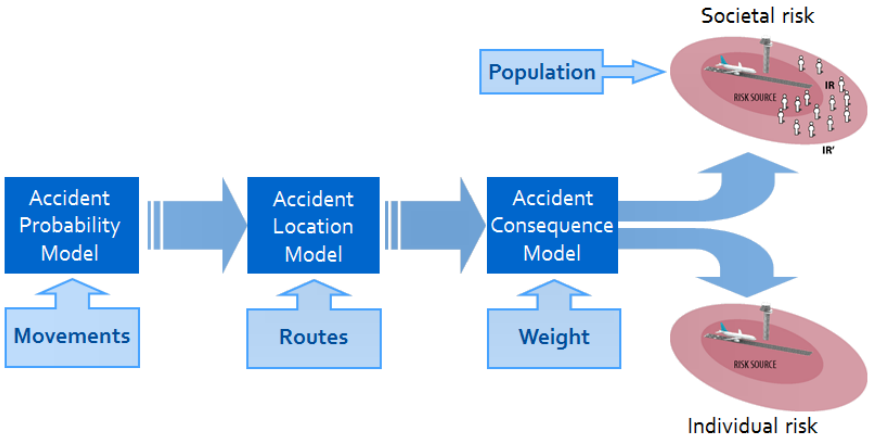

Figure 2-3: Flow chart of the sub models (Source: NLR).

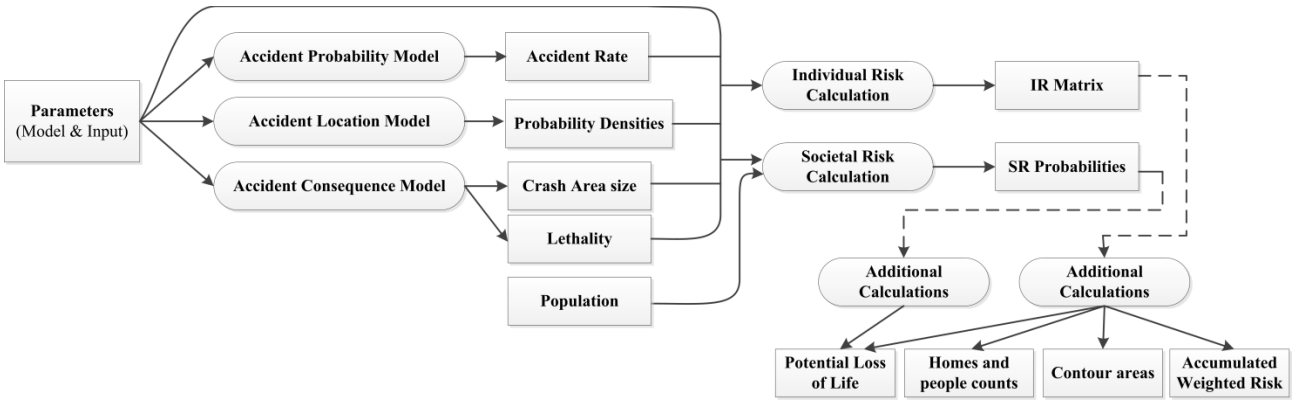

Figure 2-4: Design of the Third Party Risk model.

2.2.1 The sub models

The Accident Probability Model selects the accident rate (AR) of a certain accident type based on a couple of parameters: the aircraft's generation and maximum take-off weight, whether the flight is a cargo flight, passenger flight or a business jet, and whether the accident happens during take-off or landing. The accident rate is the probability than a certain accident occurs [3]. The accident types that are supported by the Third Party Risk model are overrun, undershoot and overshoot. An overrun is a situation in which an aircraft runs off the end of the runway during take-off or landing. A landing undershoot is a situation in which a landing aircraft contacts the ground before the runway threshold. An overshoot is a situation in which an aircraft takes off and contacts the ground beyond the runway end. These accident types are illustrated in Figure 2-5. Veer-offs are not supported by the standard Third Party Risk model. Note that the AR values selected are model parameters, so there are no computations involved in this model.

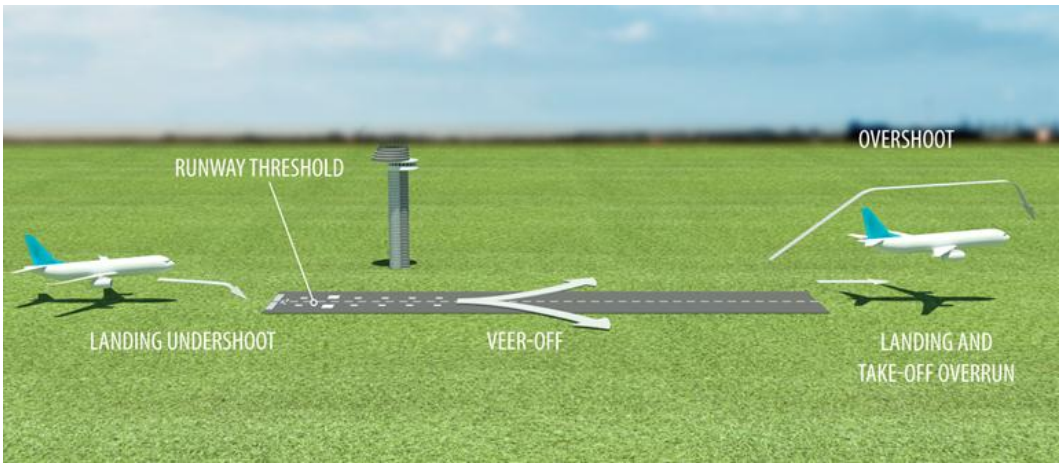

Figure 2-5: Aircraft accident types (Source: NLR).

In the Accident Location Model, a probability density matrix is calculated for each accident rate selected in the Accident Probability Model by using one or more distribution functions. The probability density matrix is a grid with a probability density value for each cell. The probability density in a cell gives the probability that if an accident happens, the crash will occur in the cell. Each cell has its own probability density since its distance to the flight path or the runway threshold most likely differ. Selection of the right distribution functions to use is based on the aircraft's maximum take-off weight, whether it is in take-off or landing phase, and the type of accident. Furthermore, the distribution function choice depends on whether the probability density matrix should be calculated relative to the route or relative to the runway [3]. The distribution functions are mathematically complex and are listed in Appendix A.

The Accident Consequence Model selects two parameters, i.e. crash area and lethality. The size of the crash area is based on the aircraft's maximum take-off weight, and depending on the models and the terrain type where the aircraft crashed. The crash areas are modelled as circles with its centre point located in the centre of a grid cell. Lethality is a constant and is dependent of the chosen model [3]. The selection schemes for all three sub models are listed in Appendix A.

2.2.2 Individual Risk calculation

As mentioned before, the Individual Risk is the probability that a person who stays at exactly the same location for a year dies as a consequence of an aircraft crash. This probability is an accumulation of the risk incurred by all aircraft movements that take place in that year. Each movement contributes to the risk depending on the path the aircraft takes and the type of aircraft. Also, there are several types of accidents, each having their own probability of occurrence [3]. The calculation of the Individual Risk can be split up in two phases. The first one is computing the total risk caused by an aircraft accident taking place at a certain location. The second phase is distributing this risk over the cells present in the accident's crash area. These two phases are performed for every cell for every movement and for each type of accident that may occur.

Computing the total risk caused by an aircraft accident in a certain cell depends on several factors. First, there is the probability that an accident actually occurs, also called the accident rate. Then, there is the probability that if the accident happens, the crash will occur in the cell. Note that this is the value in the probability density matrix. Two other factors are the total area of the crash and the lethality in the cell. The lethality is the probability of a person dying as a consequence of an aircraft accident when being in the crash area. Multiplying these four factors gives the total risk for a certain cell for a given accident type and movement [3].

The risk calculated in the first phase applies for the complete crash area. Only when the crash area is entirely within the cell, the calculated risk is added to the cell's risk. If this is not the case, it is distributed over the cells within the crash area accordingly. Thus, for each cell (partially) covered by the crash area, the percentage of the crash area in that cell should be computed. Since the crash area is always modelled as a circle and the centre of that circle is always the centre of a cell, the same distribution can be applied to multiple cells. Therefore, for each crash area size, a template is computed containing the indices of the cells in the crash area, relative to the centre cell of the crash area, and the corresponding percentages [3]. See Figure 2-6 for an example crash area, its derived template, and the application of that template to the crash area in a grid. Note that for cells which lie close to the edge of the study area the same template can be used, only those indices mapping to cells out of range are ignored. All computed templates are stored in memory for later use. The total amount of templates that needs to be computed and stored is equal to the number of different crash area sizes. Computing a template only concerns the first quadrant of the crash area, because the values in the other quadrants are equal to the first only mirrored vertically, horizontally or diagonally. When these two phases are performed for every cell, for each accident type, and for each movement, the Individual Risk calculation is finished.

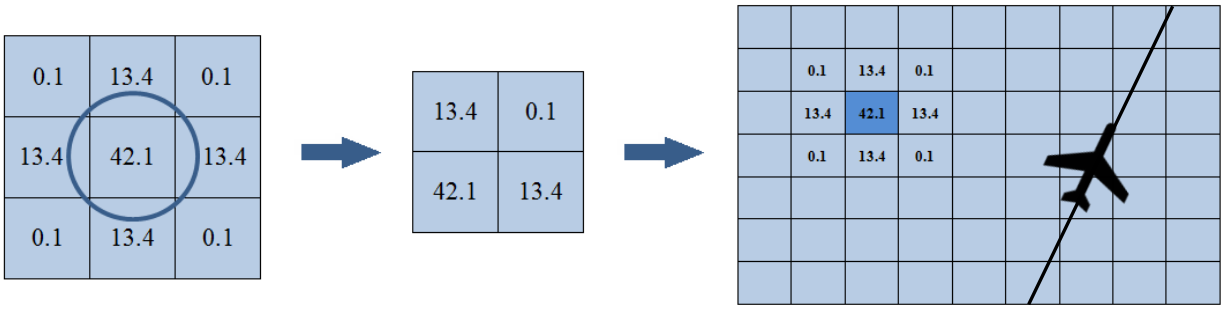

Figure 2-6: Derivation of a template from a crash area (blue circle), and application of that template to a grid.

2.2.3 Societal Risk calculation

The calculation of the Societal Risk results in the probability that more persons than in a given group size die as a direct consequence of an aircraft accident for every group size requested. Just as with the Individual Risk calculation, the probability is the accumulation of the risk incurred by each movement in a year [3]. The calculation involves a number of steps performed for each cell in the study area.

First, the probability that a certain type of accident occurs and the cell is part of the crash area is calculated by multiplying the accident rate with the cell's probability density. This probability is called the accident location probability. Then, the maximum number of victims is determined by a similar approach as the risk distribution in the Individual Risk calculation. Instead of neighbouring cells contributing to the cell's risk, in the SR calculation the neighbouring cells contribute to the maximum number of victims. How much they contribute depends on the population density in the concerning cell, and how much of its area is covered by the crash area size. For this calculation, templates are used in the same way as explained for IR calculations. When the maximum number of casualties is computed, the probability for each requested group size smaller than this maximum is calculated. This probability is the accident location probability calculated earlier multiplied by the probability of having more victims than the group size. Computing the latter probability involves the accumulation of the amount of permutations defined by the binomial coefficient [3]. When these calculations are done for every cell, for every accident type, and every movement, the SR calculation is finished.

Chapter 3

GPU computing

Usually, when an application is run, the Central Processing Unit (CPU), also known as the processor, executes the program's instructions and performs the required calculations. With GPU computing, one or more Graphical Processing Units (GPUs) are used to accelerate the application. Originally, GPUs were designed to be used for graphical programs only. Nowadays, applications do not necessarily have to be graphical applications to benefit from the computing power of GPUs. We describe how the first graphical cards evolved to the modern GPUs in the first section of this chapter. In order to optimally benefit from GPUs, it is crucial to understand the architecture of the underlying hardware. Therefore, Section 3.2 is dedicated to several GPU architectures and their benefits and drawbacks. That section is followed by a description of OpenCL, a programming model used for programming on GPUs, in Section 3.3. Important factors influencing the suitability of an application for parallelisation are discussed in Section 3.4.

3.1 The evolution of GPU computing

The first GPUs were only responsible for the graphical output. The operations they performed can be explained by the graphics pipeline. The graphics pipeline, also called the rendering pipeline, is a series of operations that are typically performed on a 3D representation and results in the pixels to be displayed on a 2D monitor. The operations can be divided in two parts: the conversion of 3D object coordinates to 2D screen coordinates, and the rendering, which consists of operations to obtain the final pixels. The first graphical cards were designed to take care of the rendering part only. The conversion part was implemented in software and run on the CPU. Driven by the demand for faster and higher-definition graphics, the graphical cards evolved. More operations of the graphics pipeline got implemented in hardware, therewith releasing the CPU from performing compute-intensive calculations [5].

In 1999, NVIDIA started a new generation by launching the first graphical card that implemented the complete graphics pipeline: the GeForce 256. At this point, the term Graphical Processing Unit (GPU) was introduced and can be defined as a dedicated hardware device which is responsible for transforming 3D data into a 2D raster. The benefit of having all the pipeline stages performed by the GPU is not only that each stage is finished more quickly, it also eliminates the need to transfer lots of graphics data between the CPU and the GPU [6, 7].

The downside of having the complete graphics pipeline implemented in hardware is that developers have minimal influence on the graphics data as it runs through the pipeline. This kind of pipeline is called a fixed function pipeline. In order to provide developers a greater flexibility, the GPUs evolved towards the use of a programmable pipeline. Again it was NVIDIA who introduced a new generation. Their GeForce 3, launched in 2001, gave programmers the ability to program a part of the pipeline using shader programs, which operate on dedicated processors and are written in a low-level shader language. Fortunately, two major Application Program Interfaces (APIs) have been created to help writing shader programs: SGI's OpenGL and Microsoft's Direct3D, part of DirectX. To further aid the programmers, two much alike high-level programming languages were developed: Cg and HLSL (High-Level Shader Language) [6].

The first generation of fully-programmable GPUs was introduced with the launch of NVIDIA's GeForce FX and ATI's Radeon 9700 graphical cards in 2002. The use of shader programs caught the attention of the scientific community, who started to use the powerful GPUs for general purpose applications [8]. In the years to follow, extra processors were added, allowing multiple pixels to be processed simultaneously. There were, however, downsides to the GPU architecture at this point. One was that each piece of dedicated hardware had its own instruction set, which increased the complexity of

using the GPU for general purpose applications. Also, computer games varied wildly in the portion of specific shader usage. Having a fixed amount of dedicated processors led to inflexibility and inefficient processor utilization [5].

Primarily to ease load balancing, a unified shader model was introduced in 2006. Instead of having dedicated hardware for specific tasks, a single type of processor was used for all kinds of tasks. Because the processors had a unified instruction set, it became more suitable for non-graphic tasks [9]. The unified shader model was first implemented in NVIDIA's G80 architecture for the GeForce 8 series GPU. New programming models were developed to use these GPUs: CUDA by NVIDIA and OpenCL by the Khronos Group, a collaboration initiated by Apple including AMD, IBM, and NVIDIA. Around the year 2010, the first GPU designed especially for GPGPU programming was released by NVIDIA based on the Fermi architecture, which is explained in the next section [5].

In the evolution of the GPU two trends can be distinguished. First, the trend was to add more programmability to GPUs by moving towards a fully-programmable pipeline in contrast to the fixed function pipeline. Next, the focus was more on making the GPUs more parallel and general purpose by adding processing units and 'merging' the separate dedicated processors to a unified one. The next section focuses on the architectures of the most popular modern GPUs.

3.2 GPU architectures

The two biggest competitors in the field of high-end graphics cards are NVIDIA and AMD. In this section, we describe the architectures of the modern GPUs of these two vendors. By modern GPUs we mean GPUs implementing the unified shader model. We mainly focus on the details which are of interest for general purpose computations, rather than graphical computations. The general design of NVIDIA's and AMD's architectures has many similarities, but also a couple of essential differences. First, the general design is introduced, followed by a description of each element, including the essential design choices made by both vendors.

3.2.1 General design

One of the characteristics of graphics computations is that there is a large set of data on which the same instructions are performed. This characteristic fits nicely with the Single Instruction Multiple Threads (SIMT) execution model, in which multiple threads simultaneously execute the same instruction [10]. The general design complies with this execution model by providing lots of processing elements able to

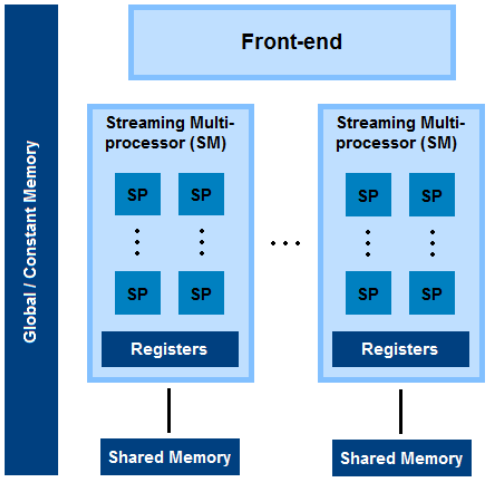

Figure 3-1: General architecture of modern GPUs.

process instructions concurrently. Since the GPU is a separate device, data cannot be read from host memory. Instead, GPUs have their own memory. Therefore, data needs to be copied from host to device memory, which is much slower than reading from memory directly. The default interface of the memory bus, via which the data is transferred, is PCIe (Peripheral Component Interconnect Express) [11]. More information about the memory model is given later. A simplified graphical representation of this design is depicted in Figure 3-1. Note that this picture only shows the general design; not the physical architecture.

Relevant specifications of NVIDIA and AMD GPUs that are described in this section are listed in Table 3-1. The specifications in this table are of the release models using the considered architecture. That is, for NVIDIA's G80, GT200, Fermi and Kepler the GeForce 8800 GTX, the GeForce GTX 280, the GeForce GTX 480, and the GeForce GTX 680, respectively. For AMD/ATI's R600, Cayman, and Tahiti, specifications are listed of the Radeon HD 2900 XT, the Radeon HD 6970, and the Radeon HD 7970, respectively.

	NVIDIA				**AMD / ATI**		
	G80	**GT200**	**Fermi**	**Kepler**	**R600**	**Cayman**	**Tahiti**
Released	Nov. '06	Jun. '08	Mar. '10	Mar. '12	May. '07	Sep. '09	Dec. '11
Processing elements	128	240	480	1536	320	1536	2048
Graphics core clock (MHz)	575	602	700	1006	742	880	925
Shader core clock (MHz)	1350	1296	1401	-	-	-	-
Memory clock (MHz)	900	1107	1848	6008	825	1375	1375
Memory bandwidth (GB/s)	86.4	141.7	177.4	192.2	128	176	264
Memory bus interface	PCIe	PCIe 2.0	PCIe 2.0	PCIe 3.0	PCIe	PCIe 2.1	PCIe 3.0
Max. global memory (MB)	768	1024	1536	2048	512	2048	3072
Single-precision GFLOPS	518	933	1345	3090.4	475	2703	3788.8
Double-precision GFLOPS	-	78	168.5	129	-	675	947.2
Constant memory (KB)	64	64	64	64	64	64	256
Shared memory (KB per SM)	16	16	16/48	16/32/48	16	32	64
Private memory (KB per SM)	32	64	128	256	32	256	64

Table 3-1: Specifications of several NVIDIA and AMD/ATI GPUs [12, 13, 14].

3.2.2 Front-end

The main task of the front-end is to assemble the data and the instructions to be performed on this data. Also, threads need to be set up and dispatched to the processing elements. The front-end does not issue the threads directly to the processing elements, but to a Streaming Multiprocessor (SM). How the processing elements are grouped together within an SM is vendor-dependent. Also, the number of threads that are scheduled together depends on the vendor. NVIDIA schedules threads in groups of 32, which is called a warp; AMD schedules groups of 64 threads, called a wavefront [5, 15].

3.2.3 Processing elements

In NVIDIA architectures, starting with the G80, processing elements are called Stream Processors (SPs), and are able to perform single-precision floating point and integer operations. A single instruction is carried out by multiple SPs concurrently. Each SM has its own instruction handler in order to store instructions and dispatch them to its processing elements. Further, an SM also contains two special function units (SFUs) to carry out transcendental and mathematical functions, like square root and cosine [15, 16].

In the GT200 architecture, G80's successor, a Double Precision Unit (DPU) was added to each SM. As the name suggests, the DPU performs double-precision floating point operations, which are

essential for many scientific applications [16]. In the Fermi architecture, double-precision floating point capability is improved by letting the double-precision calculations be performed by the SPs themselves, instead of using a special unit [17, 18]. Since the Third Party Risk application uses double-precision floating point numbers, GPUs from this generation on are very interesting for Third Party Risk calculations. Another big improvement is that each SM contains two warp schedulers and instruction dispatch units to allow issuing and executing two warps simultaneously, with possibly different instructions. Also, two kernels can be executed concurrently for better GPU utilization. In the Kepler architecture the number of warp schedulers was doubled to four, each able to dispatch two instructions per warp every clock cycle [13].

The way processing elements are grouped in AMD architectures has changed from the R600 to the Southern Islands family. In the R600 architecture, processing elements are grouped per 5 in a shader cluster based upon the Very Long Instruction Word (VLIW) architecture. Hence, within one shader cluster five different instructions can be executed in parallel. All five processing elements are capable of performing single-precision floating point and integer operations. The fifth processing element is also able to do transcendental and mathematical functions. Each SM contains multiple shader clusters. The reason for AMD to cluster five processing elements is that most common graphics operations consist of a 4 component dot product and a scalar operation. For non-graphics computations, however, keeping all five cores busy is quite hard, due to instruction dependencies. Also, providing instruction-level parallelism requires instruction ordering and packing, which reduces the overall efficiency compared to NVIDIA's architecture. Another issue of the VLIW architecture is that optimizing a program is hard, because debugging and predicting the performance of pieces of code is difficult [19].

In the Cayman chip, the VLIW5 architecture was replaced by a VLIW4 architecture, meaning that only four processing cores are contained within the shader cluster, and therewith only four different instructions can be executed simultaneously. All processing elements have equal capabilities. Special functions are executed by combining three to four units [20]. For non-graphics computations this architecture is more efficient relative to VLIW5, but the VLIW issues still apply. In AMD's latest GPU family, Southern Islands, an architecture more like NVIDIA's is used, in which all processing elements of an SM execute the same instruction [19].

When comparing the number of processing elements between NVIDIA's and AMD's first architectures, it is striking that the R600 and Cayman have many more processing elements than the G80, GT200, and even Fermi. This does not mean, however, that the performance is higher. This is because of the VLIW architecture used in the R600 and Cayman. As explained above, keeping all cores busy is difficult to achieve. If, for example, only one instruction can be handled concurrently, the actual working processing elements of the R600 and Cayman should be divided by 5 and 4, respectively. Also, the AMD cores run at a lower clock frequency compared to their competitors.

A GPU's performance is given by the number of floating point operations it can perform in a second, noted as GFLOPS. A striking observation is that NVIDIA's double-precision GFLOPS peaks are way below AMD's. NVIDIA's latest GPU, the Tesla K20, makes up for that gap. It has 2496 processing cores and has a single- and double-precision floating point peak performance of 3.52 and 1.17 Tflops, respectively [21].

3.2.4 Memory

Obviously, the processing elements are of great importance as they perform the actual calculations, but without memory to store data, the processing elements are incapable of doing their jobs. There are several kinds of memory in a GPU, each with their own characteristics. In Figure 3-1, the rectangles with white text represent memory spaces.

The largest memory is the global memory. It can be accessed by all threads and by the host. Therefore, the global memory is where initial input data and final output data is stored, as well as other

data that needs to be accessible to all threads. The constant memory is read-only and is part of the global memory. Data is transferred from host to device and vice versa through a memory bus, of which the default interface is PCIe, as mentioned earlier. The amount of data that can be transferred in a second (i.e., the memory bandwidth) depends on the width of the memory bus, the type of memory, and the memory clock [22].

Shared memory is located closer to the processing elements and, therefore, can be accessed more quickly than global memory. The amount of shared memory, however, is limited and it can only be accessed by a certain group of threads. Therefore, shared memory is typically used to store data that will be used by multiple threads of the same group [22].

The fastest and smallest type of memory is the private memory. As the name suggests, this memory belongs to an individual thread exclusively and cannot be accessed by others. Private memory is implemented by a fixed size on-chip register file. This way, the size of the private memory available to a processing core can vary. This hierarchy of memory spaces is implemented on all modern GPU architectures of both NVIDIA and AMD.

From Table 3-1 can be concluded that not only the sizes of the global, shared and private memory increased with the release of new architectures, also the memory bandwidth improves. The latter is mainly caused by the increase in speed of the memory clock and by improvements made on the memory bus interface. The transfer rate of the most recent PCIe interface, PCIe 3.0 as used in Kepler and Tahiti architectures, has almost quadrupled with respect to PCIe 1.0.

3.2.5 Clocks

The speed at which the elements discussed above operate is determined by several clocks. For memory accesses a memory clock is used. In AMD architectures, all non-memory operations are performed based on a main clock, which typically runs at a lower frequency than the memory clock. In NVIDIA architectures, there is another clock, called the shader core clock, used by the processing elements. All clocks have their own frequencies, but it is important to maintain a good balance between them. For example, having a main or shader core clock with a too high frequency compared to the memory clock, results in idle processor cores waiting for memory to be loaded. Fortunately, memory access latency can be hidden by the GPU, by scheduling another warp or wavefront when all threads of a warp or wavefront are waiting [13, 15].

In the Kepler architecture there is no shader core clock anymore. This is because the development of the Kepler architecture was focused on reducing power consumption. Since a higher clock frequency means higher power consumption, processing elements in the Kepler architecture use the main clock instead of the higher-frequency shader core clock. This approach led to significantly less power consumption with respect to its predecessor, Fermi [13].

3.3 Programming the GPU with OpenCL

Up till this point, we have explained how graphical cards have evolved over time and what their architecture looks like, but not how to program them. Standard C and C++ libraries can only be used for traditional CPUs. For GPUs, however, special functions are needed in order to copy data from and to the GPU device, and execute code on the processing elements. NVIDIA introduced CUDA (Compute Unified Device Architecture) to program their GPUs. The use of CUDA is restricted to NVIDIA GPUs only. The Khronos group developed OpenCL (Open Computing Language), which can be used for all supporting GPUs and CPUs from major manufacturers like AMD, NVIDIA, and Intel. Since CUDA is focused on NVIDIA GPUs, it can outperform OpenCL on these devices. A comprehensive performance comparison shows that CUDA performs at most 30% better than OpenCL, but similar performance is achieved under a fair comparison [23]. Because of its portability, we chose to use OpenCL to accelerate the Third Party Risk program. In the remainder of this chapter a typical OpenCL program is described.

3.3.1 Initialisation

A typical OpenCL program starts by selecting a platform (e.g. AMD or NVIDIA). Then, one or more devices from the selected platform are chosen. These devices may be GPUs, but also CPUs or accelerators. The chosen devices are contained in a `Context`. This abstraction allows multiple groups of devices, possibly containing devices of different platforms, to be used in a single program.

The core of a parallel program is the code that needs to be run in parallel. In CUDA and OpenCL this piece of code, which is actually a single method, is called a kernel. A parallel application may contain multiple kernels, which need to be read in and compiled during runtime. This results in a `Program` object and multiple `Kernel` objects.

In order to let the device perform an operation, such as transferring data, the host sends a message to the device. Such a message is called a command and must be sent using a `Command Queue`. For each device selected, a `Command Queue` needs to be created. If a device supports simultaneous data transfer and kernel execution, multiple `Command Queues` for the device can be created. By default, commands are issued in order, but out of order issuing of commands can be enabled [24, 25].

3.3.2 Preparing for kernel execution

When the initialisation is done, the actual computation can be started. Before a kernel can be executed, device memory must be allocated for both input and output. This is done in global (or constant) memory since that is the only memory accessible by the host. To store data in the allocated memory, `Buffers` are used, which are basically just one-dimensional arrays. Then, required data, if any, should be transferred to these `Buffers`. This is done by issuing a data transfer command using the `Command Queue`. The device must be told which `Buffers` correspond to which kernel arguments. This process is called setting kernel arguments. The last action is to provide a configuration of work-groups and work-items. Determining a good configuration is difficult, and knowledge of the way the data is distributed and how OpenCL groups the processing elements is required [24, 25].

Usually, a sequential program iterates over a certain set of data, performing the same operations on each data element. In parallel programs, the operations performed on the data elements are contained in the kernel. In OpenCL, an individual execution of the kernel is called a work-item. Work-items are grouped together in a work-group, which is executed on a Streaming Multiprocessor. The work-items belonging to a certain work-group can access the same block of high-speed shared memory and can be synchronized. Work-items of different work-groups cannot be synchronized within the kernel execution and only share the ‘slow’ global memory. Each work-item has a global and a local index, which is used by the work-item to select the correct piece of data to operate on. The global index is a unique index for

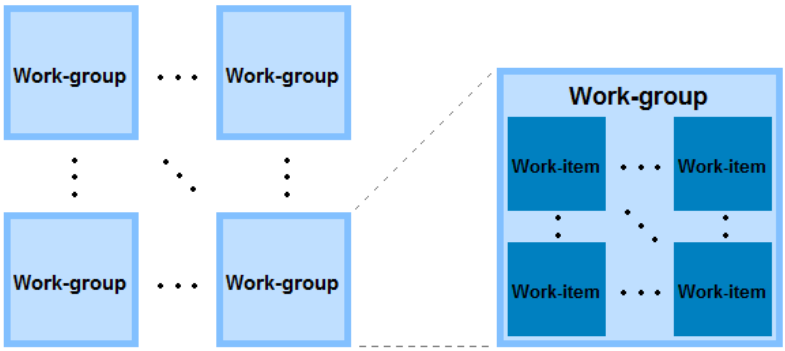

Figure 3-2: Two-dimensional work-group and work-item space.

all work-items. The local index is the work-item's index within the work-group. Work-groups and work-items can be ordered multi-dimensionally, which makes it easier to determine the correct indices when the dataset is multi-dimensional. In case of a two-dimensional ordering, as in Figure 3-2, global and local indices both consist of an x and y value; in case of a three-dimensional ordering, both indices consist of an x, y, and z value [25].

Determining the total amount of work-groups and the number of work-items in those work-groups depends on several factors. The total number of work-items should be chosen in such a way that the computations are performed for the complete dataset. The maximum work-group size is determined by the device, and depends on the private memory required by each work-item, the number of registers used by the kernel, and the total amount of private memory available to a work-group. The number of work-groups depends, for performance reasons, on the amount of SMs present on the device. Since a work-group is executed by a single SM, multiple work-groups should be scheduled to an SM in order to hide memory latency as mentioned in Section 3.2. Further, the number of work-groups and work-items depends on the number of dimensions in which they should be ordered. Fortunately, OpenCL programmers have the opportunity to let OpenCL decide which configuration is best suitable [25, 26].

3.3.3 Kernel execution and program finalisation

When a configuration is determined, the kernel execution can be started by issuing a command to the `Command Queue`. Some devices support multiple kernels to be executed concurrently if the kernels have no dependencies. Executing several kernels consecutively is always supported. Because global memory is persistent, data required by consecutive kernels does not have to be transferred to the device again. Data in shared and private memory is not persistent, meaning that required data needs to be reloaded for every new kernel execution. When all kernels have been executed, the results should be placed in global memory and copied back to the host.

3.4 Suitability of applications for parallelisation

Using GPUs may result in a big performance improvement, but not all applications are fit for parallelisation. There are several factors that limit the profitability of parallelisation, such as instruction dependencies, execution path branching, and whether floating points and atomic operations are used. To which extent these factors are limiting parallelisation is described next.

3.4.1 Instruction dependencies and branching

Some parts of a program cannot be executed in parallel, because the instructions rely on the result of other instructions. If the part of the program, in terms of execution time, that can be parallelised is small, overall performance gain, if any, will be small as well. This also depends on the size of the dataset on which the parallelised part operates. It would be ideal if this data set is large, in order to keep as many processing elements as possible busy.

Another important factor in parallelisation is whether there is a lot of branching in the execution path, which is incurred by conditional statements. This factor is important because a GPU executes one branch of the execution path at the time. Imagine an if-statement in the kernel code for which just a couple of work-items satisfy the condition. A couple of processing elements have to execute the instructions for these work-items while the rest of the processing elements are idle. Therefore, programs with execution path branching are less suitable for parallelisation than program without branching.

3.4.2 Floating point and atomic operation support

Many scientific applications use single- or double-precision floating points instead of integers. Floating points are data types able to represent any real number, whereas integers only represent natural numbers (i.e. numbers without a fractional or decimal component). The difference between single- and double-precision floating points is that double-precision floating points support a wider range of values because 8 bytes of memory are used to store a value instead of 4 bytes. Not all GPUs support floating point operations, and the ones that do, at the time of this writing, do not support atomic functions.

Atomic functions are functions used to eliminate race conditions, which cause the output of the program to be dependent on the timing and sequence of uncontrollable events. Assume two threads, A and B, which both want to update a global value X. Updating a value consists of three separate operations. First, the value to be updated is read and stored locally. Then, the local value is updated. Finally, the global value is overwritten by the updated value. A possible scenario is illustrated in Figure 3-3. A reads the global value, which happens to be 10, and wants to add 2 to this value, so it increments its local copy of the value to 12. Before A can overwrite the global value, B reads the global value, which is still 10. Now, A overwrites the global value to 12, while B increments its local copy by 4 and overwrites the global value to 14. However, incrementing 10 by 2 and by 4 should result in 16. Such race condition scenarios can be eliminated by the use of atomic functions. For example, an atomic addition function ensures that the three separate operations (read, update and write) are performed as a single operation. Atomic operations are only supported in OpenCL for integer values.

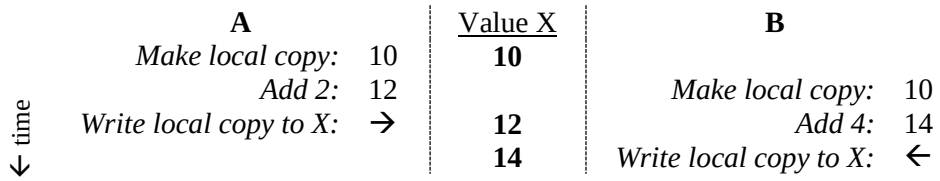

Figure 3-3: Sample scenario of race conditions.

Chapter 4

Evaluation of the accelerated program

In this chapter, the parallel program is described and evaluated. Some parts of the sequential code are easily parallelised; other parts require to be restructured. For each part is described how it is parallelised, which optimizations we have done, and what the results of the parallelisation are. The test environment in which the parallel program is run is described first, in Section 4.1, followed by the main structure of the program in Section 4.2. Then, the parallelisation and evaluation of the sub models, distribution template calculations, Individual Risk calculations and Societal Risk calculations are addressed in Sections 4.3, 4.4, 4.5 and 4.6, respectively. Section 4.7 is dedicated to the implementation of subset output support, followed by a discussion of further optimizations we applied, in Section 4.8. Finally, the complete accelerated program is analysed with respect to performance, memory requirements, and scalability, in Section 4.9.

4.1 Testing the code

This section contains a description of how we tested the code, including the hardware used, the input data, and how the output is validated.

4.1.1 Hardware

The parallel program is tested on two different machines. Machine A contains an Intel Core 2 E8400 CPU, which has two processing cores running at 3 GHz. The GPU in this machine is the NVIDIA Tesla C2075, which contains 6 GB of global memory and 448 processing cores running at 1.15 GHz.

The CPU of machine B is an Intel Xeon E5-2670, in which 8 processing cores run at 2.6 GHz. The GPU residing in this machine is NVIDIA's Tesla K20m, containing 5 GB of global memory and 2496 processing cores running at 705.5 MHz.

To which extent the GPUs should be able to perform better than the CPU can best be determined by their theoretical peak performance, which is given in floating point operations per second (FLOPS). Since the Third Party Risk program contains many double-precision floating point operations, we look at the double-precision FLOPS. The CPU used for the sequential version (Intel Core 2 E8400) has a

	Tesla C2075	**Tesla K20m**
Released	July '11	November '12
Processing elements	448	2496
Streaming Multiprocessors	14	13
Processor clock (MHz)	1150	705.5
Memory clock (MHz)	3000	5200
Memory bandwidth (GB/s)	144.0	208.0
Memory bus interface	PCIe 2.0 x16	PCIe 2.0 x16
Max. global memory (MB)	6144	5120
Shared memory (per SM)	48 KB	48 KB
Registers (per SM)	128 KB	256 KB
Double-precision GFLOPS	515	1173
OpenCL version	1.1	1.1

Table 4-1: Specifications of used NVIDIA GPUs: Tesla C2075 and Tesla K20m [27].

theoretical peak performance of 22.4 GFLOPS. The Tesla C2075 and Tesla K20m have a peak performance of 515 and 1173 GFLOPS, respectively, which is 23 and 53 times higher than the CPU. More specifications of the GPUs used in machine A and B are listed in Table 4-1.

4.1.2 Dataset

The dataset used as input to evaluate the program is a real dataset used for calculating third party risk around Schiphol (further referred to as the EHAM dataset). The amount of traffic lines in the traffic file is 4738, and the size of the study area is 56 × 56 km. The program is run with three different cell sizes: 100, 50, and 25 meter. Hence, the amount of cells to be computed for these cell sizes is 313600, 1254400, and 5017600, respectively. The group sizes for which the Societal Risk must be calculated are 1, 3, 5, 10, 20, 40, 100, 200, 400, and 1000.

4.1.3 Timing and validation

The timings, as presented later in this chapter, are the average of three runs, except for the time of the sequential program with a cell size of 25 meter, which is just run once, since there is not enough time to run it multiple times. The sequential parts of the code, which are executed on the CPU, are timed using the *timeval* structure as defined in the <*sys/time.h*> header. The transfer time and kernel execution time is timed using OpenCL's built-in profiling functionality.

The output of all runs is validated by comparing it with the output of the sequential runs. For this, we created a small program, which is also able to compare probability density matrices. Since the accuracy of floating point arithmetic operations on the CPU and GPU may differ, results of the parallel version may deviate from the sequential version's results. The maximum relative error depends on the functions used [28]. A run of Individual Risk calculation with 25m cells shows a maximum absolute difference of 7.2×10^{-15} and a maximum relative difference of 1.2×10^{-7} when probability densities are computed in the parallel version. When identical pre-computed probability densities are used for both sequential and parallel execution, the results are exactly the same. Hence, the difference is caused by the probability density calculation. The reason for this is that many trigonometric functions are used for calculating probability densities. This conclusion was also made in a study to the difference in results when running TRIPAC on a Linux and a Windows machine [29].

The maximum absolute and relative difference is small enough to pass the validation tests, as there are other TRIPAC versions which passed the tests with larger differences with their predecessors [30]. A complete validation test has not yet been executed, however.

4.2 Structure of the program

4.2.1 What is parallelised?

The structure of the parallel program mainly adheres to the original program's structure. This means that at the highest level the program loops over all requested models. In other words, requested models are performed one by one. For each model, the program performs calculations movement by movement. Depending on the type of movement, multiple accident types may apply. For each accident type, the sequential program loops over all cells in the study area, performing either Individual Risk calculations or Societal Risk calculations for the concerning cell. This is the part of the program where most of the calculations happen and most time is spent. Therefore, these calculations are carried out concurrently in the parallel version. However, the IR and SR calculations are not the only calculations done in parallel. Part of the operations performed by the sub models are parallelised as well. The structure of the sequential program is given in pseudo code in Listing 4-1. The bold lines are executed concurrently in the parallel version.

We choose to do the calculations of multiple cells in parallel, rather than multiple movements. One reason is that there are way less movements than cells, which could lead to processor under-utilization. Another reason is that multiple work-items would need to update the same values, leading to race conditions.

```
FOR all models {
    FOR all movements {
        FOR all accident types applying to movement {
            FOR all cells in study area {
                Calculate risk value;
            }
        }
    }
}
```

Listing 4-1: Main structure of the sequential program. The bold lines are executed in parallel in the parallel version.

4.2.2 The data

The grids used to store intermediate and final results are stored in device memory in a single one-dimensional Buffer. The main reason for this is that the number of grids to be stored is variable. A drawback of using a single Buffer is that a more complex calculation must be done to find the right index to store grid results. However, index calculation is more efficient than having to check conditions to select the correct Buffers.

The risk values, which are stored in these grids, are very small, even smaller than 3.4×10^{-38}. Therefore, the risk values cannot be stored as single-precision floating point numbers, but must be represented as double-precision floating point numbers. This has several drawbacks over integers and single-precision floating points. First, a double-precision floating point number requires 8 bytes of memory, which is twice as much as an integer or a single-precision floating point number. Second, not all devices support double-precision floating point, which limits the amount of devices able to run the parallel program. Finally, the use of floating point numbers also limits the program itself, because atomic operations are only supported for integer values, as explained in Section 3.4.2.

4.2.3 OpenCL initialisation

Before OpenCL can be used, it must be initialised. First, there is checked which platforms and devices are installed on the machine. Only devices supporting double-precision floating point numbers are listed as available devices. The user can choose which device to use, when multiple are available. If only one device is available, it is automatically selected.

Then, a `Context` is created for the selected device, containing two `CommandQueue`s. We chose to use two `CommandQueue`s to enable concurrent data transfer and kernel execution, as is explained in Section 4.8. Also, two `Program`s are created within the `Context`. One `Program` contains the code of the kernels needed for probability density calculations. The other `Program` contains all code required for the parallel Individual and Societal Risk calculations. From the code of these `Program`s, `Kernel`s are created. The reason two `Program`s are used, is that both `Program`s need to be compiled with different compiler options set. A bug in NVIDIA's OpenCL compiler forces us to compile the probability densities `Program` to be compiled with the `-cl-opt-disable` option [31]. Without setting this option, output of several runs mutually differs. However, for the other `Program`, this problem does not occur. Compiling it with the `-cl-opt-disable` option would not produce wrong results, but execution time would increase substantially. Therefore, the `Program` with the code for IR and SR calculations is compiled without any options set.

When a `Kernel` is invoked, OpenCL needs to know the number of work-items (global range), the work-group size (local range) and the dimensions, as explained in Section 3.3.2. The local range can be determined by OpenCL based on the number of work-items requested. However, for some kernels, this information is needed by the host to allocate shared memory, as is described in Section 4.8.1. Therefore, the size and dimensions of the work-group should be determined beforehand. This determination takes a couple of rules into account. One is that the size in each dimension should be able to divide the sizes of the corresponding dimensions of the global range. Another rule is that the size in each dimension can best be a power of 2 [26].

4.3 Sub model parallelisation

As stated in Chapter 2, there are three sub models: the Accident Probability Model, the Accident Consequence Model, and the Accident Location Model. Since the Accident Probability Model only selects a single value used for all cells for each accident type, parallelisation is not necessary. The Accident Consequence Model calculates a crash area size for each cell, so this calculation is suitable for concurrent execution. The computation of the probability densities, performed by the Accident Location Model, can also be parallelised, but is more complex. The parallelisation of the crash area sizes and probability density calculations is described in detail in this section.

4.3.1 Parallelising crash area size calculations

The crash area size at a given location for a certain movement and accident type is calculated by the following formula: $CA = a \times MTOW + b$, where CA is the crash area size and *MTOW* is the maximum take-off weight of the aircraft. Parameters *a* and *b* are selected based on the aircraft's weight category, the accident type and the terrain type of the cell. In total, there are five possible weight categories, three accident types, and four terrain types. An example mapping of these factors to an *a* and *b* pair is given in Table 4-2. In the sequential version, for each movement and accident type, the formula is applied to each cell in the study area.

Weight category	Accident type	Terrain type	*a*	*b*
Light	Undefined	Open I	69	80
Light	Undefined	Obstacle I	78	28
Heavy	Run	Obstacle II	83	0
Heavy	Shoot	Open II	83	10

Table 4-2: Example mapping scheme.

Parallelising the formula is trivial: the formula only needs to be placed in a kernel. Obtaining a performance gain is less trivial. This is because of the mapping that needs to be done to provide correct *a* and *b* values. Performing the mapping for all cells on the host and transferring the obtained values to the device, gives just a small performance improvement over the sequential version. A significant percentage of the time is spent on parameter selection and data transfer.

A better approach would be to parallelise the parameter selection as well, therewith reducing mapping time and data transfer. In order to parallelise the parameter selection, each work-item should be able to do a mapping from three values to an *a* and *b* value. This cannot be done in the same way as in the sequential version, since the map functionality of the C/C++ library is not supported in the kernel. Fortunately, there are only four different terrain types possible, and the weight category and accident type is the same for each cell in a kernel. Therefore, only four pairs of *a* and *b* values are possible for each

kernel invocation. Transferring only these eight values to the device costs little time compared to transferring one pair per cell. The work-items can easily select the correct *a* and *b* values based on their terrain type. Obviously, this requires the terrain types for the complete study area to be present on the device. Since the terrain types are constant during the entire program, these values need to be copied to the device only once. This way, each work-item calculates its crash area size itself, and the data that needs to be copied for each movement and accident type is only four pairs of *a* and *b* values.

One might think that data transfer can be eliminated completely by maintaining the complete mapping in device memory and letting each work-item do the complete mapping themselves. Unfortunately, this will not make a significant difference in performance, since the host should send the correct weight category and accident type instead. Also, extra mapping operations need to be performed by the work-items.

Table 4-3 shows the performance gain over the sequential version when using the approach described above. The big difference in execution times between the sequential run and the parallel runs is not only due to the fact that multiple cores are used, but also because of inefficient code in the sequential version. For each cell the complete mapping to the correct *a* and *b* values is done, while only one of the three factors, the cell's terrain type, is variable for a given movement and accident type. In the parallel version, only the mapping from a cell's terrain type to the correct *a* and *b* values are done for each cell, where the first part of the mapping is done on the host only once for each movement and applying accident type.

<table>
<tr><th></th><th>Sequential</th><th>Tesla C2075</th><th>Tesla K20m</th></tr>
<tr><td>Transfer terrain types</td><td>-</td><td>0.014</td><td>0.004</td></tr>
<tr><td>Transfer a and b values</td><td>-</td><td>0.029</td><td>0.025</td></tr>
<tr><td>Mapping to correct a and b values (part 1)</td><td rowspan="2">3998.0</td><td>0.023</td><td>0.002</td></tr>
<tr><td>Mapping to correct a and b values (part 2)</td><td rowspan="2">1.306</td><td rowspan="2">0.632</td></tr>
<tr><td>Calculate crash area sizes</td><td>1587.0</td></tr>
<tr><td>Total time</td><td>5585.0</td><td>1.372</td><td>0.661</td></tr>
</table>

Table 4-3: Timings of crash area sizes calculations. Cell size is 25m. Time in seconds.

4.3.2 Parallelising probability density calculations

The calculation of probability density matrices accounts for a large portion of the total execution time. Therefore, the sequential program stores the calculated probability density matrices in a file, to be read when needed in future runs. However, when movement or study area properties are changed, a new probability density matrix must be computed. Therefore, probability density calculations are parallelised as well.

The probability density at a certain location is computed by applying a distribution function, which is a combination of Weibull, Generalized Laplace, Gauss, Log Normal and/or Dirac functions [3]. Which distribution function to use is determined by a selection, based on the aircraft's maximum take-off weight, whether the aircraft is in take-off or landing phase, the type of accident, and whether calculation is requested with respect to the runway or the route. See Appendix A for the complete selection scheme and the formulas of the distribution functions.

The value computed by the distribution functions depends on the distance of the concerning cell to the runway or route. Since the probability near the runway or route is the highest and may differ significantly between locations within a cell, the probability density of cells located close to the runway or route are computed by refinement. These cells are divided in a number of sub cells (default is 100), and for each of these sub cells a probability density value is calculated in the same way as the cells of the

study area. The resulting probability density of the refined cell is the average of the probability densities of its sub cells. This way, a more accurate value is obtained and outliers are avoided. Determining whether a cell should be refined is done by checking whether a part of the cell is located in a specified area, known as the refinement area. For this determination, the coordinates of the cell's centre point with respect to the runway or route is computed. In case of a runway-dependent calculation this results in a single coordinate pair. Since a route may consist of multiple route segments, multiple coordinate pairs may be found in case of a route-dependent calculation. Besides determining refinement, the computed coordinates are also used as input for the distribution function.

When the centre point of a cell coincides with the rotation point of a circle segment, the centre point is marked as invalid as the computation will lead to an invalid value. When the probability density of all valid cells is calculated, the probability density for the invalid cells is computed by taking the average of their valid neighbours. This process is shown in Figure 4-1. Note that invalid values may only occur in route-dependent probability density calculations as there are no circle segments in runway-dependent calculations.

1.1	1.3	1.3	1.4	1.4
1.2	1.2	1.3	1.4	1.5
1.2	1.3	1.4	X	1.6
1.3	1.4	1.5	1.5	1.6
1.4	1.5	1.5	1.6	1.7

$\times 10^{-18}$

Sum of valid neighbours:
1.3 + 1.4 + 1.5 + 1.4 + 1.6 + 1.5 + 1.5 + 1.6 = **11.8**

Average:
11.8 / 8 neighbours = 1.475

Probability density at X:
$\mathbf{1.475 \times 10^{-18}}$

Figure 4-1: Computation of probability density of invalid cell X.

In the parallel version, two separate main kernels are used, one for runway-dependent calculations and one for route-dependent calculations, in which the probability density of a cell is computed and refinement is determined. The initial probability densities are computed concurrently using one of these kernels. For calculating the probability densities of the sub cells, in case refinement is necessary, the same operations have to be performed as in the main kernel, except for the determination of refinement. Therefore, two special kernels are created for refinement, performing only the probability calculation.

Calling a refinement kernel for just one refinement cell at a time may lead to underutilization of the device, since there are only 100 cells to be computed. Calling a refinement kernel for all refinement cells may require too much device memory, especially when many of the study area's cells need to be refined. Therefore, to avoid memory problems and improve processor utilization, a refinement kernel is called for number of refinement cells equal to the size of the study area. So if the size of the study area is 160,000 cells, the maximum number of cells to be refined in one kernel invocation is 160,000 / 100 = 1,600. This way, the memory allocated for the initial probability densities can be reused. So, since there are 100 refinement cells per study area cell, the amount of study area cells that can be refined in one kernel invocation is at most the total number of study grid cells divided by 100.

Using separate kernels for initial probability density calculation and refinement calculation requires the complete matrix, as computed by the first kernel, to be transferred from device to host. The matrix is needed to determine which cells have to be refined. To avoid this transfer, work-items could do the refinement, if necessary, themselves. This would lead to a significant load imbalance, because not all cells require refinement and not all refinements are of equal load. Also, an extra kernel would be necessary for computing the probability density of invalid results. This cannot be done in the main kernel, because all the cell's values must be computed before taking care of the invalid values. As there will only

be few invalid values per matrix or maybe even none, unnecessary work would be done. Therefore, the calculation of the probability density of the invalid cells is done sequentially.

In case of the EHAM dataset, only approximately 33,000 of the 5,017,600 cells are refined, which is less than 1%. However, doing the refinement of these cells on the device is over 6 times faster than the sequential version.

The probability density of all sub cells of a refinement cell should be accumulated. Due to the lack of atomic functions for double-precision floating point numbers, this cannot be done in the refinement kernel itself. Therefore, a special accumulation kernel is created in which each work-item accumulates the probability densities of all sub cells of a refinement cell. The resulting values are the probability densities of the corresponding refined cells. This makes the total amount of kernels used for the calculation of probability density matrices five: a main and refinement kernel for both runway- and route-dependent calculations and one accumulation kernel for the refinement cells. Listing 4-2 shows in pseudo code the structure of a route-dependent probability density matrix calculation in the parallel version.

```
Start route-dependent kernel;

FOR all cells {
    IF cell's value is -1 or not-a-number {
        Mark cell as invalid;
    } ELSE {
        IF refinement is necessary {
            Mark cell as to be refined;
            IF enough cells to be refined OR cell is last cell {
                Start route-dependent refinement kernel;
                Start refinement accumulation kernel;
            }
        }
    }
}

FOR all invalid cells {
    Calculate average of valid neighbours;
}
```

Listing 4-2: Pseudo code of the calculation of a route-dependent probability density matrix.

The computation of all probability density matrices is done, if necessary, before Individual or Societal Risk calculations are started. Another possibility is to compute the required probability densities during the IR and SR calculations when needed. This would avoid copying the probability density matrix from host to device, but has a downside concerning device memory. First of all, an extra `Buffer` must be used, as the probability densities calculated initially should not be overwritten by the values of the refinement cells. Also, less memory would be available for the IR and SR calculations, because a part of the memory stays in use for probability density calculations, where this memory can be reused when all probability density matrices are computed beforehand.

Execution times of the calculation of probability densities are listed in Table 4-4. The parallel versions are about 12 and 24 times faster using the Tesla C2075 and K20m, respectively, which is about half of the theoretical peak performance gain. This difference is partially caused by the part that cannot be executed in parallel. This part includes checking initial results for refinement and invalid cells, as well as the calculation of a probability density for the invalid cells. About 25% of the time is spent on these parts of code. Most of the time, however, is spent by the kernels.

		Sequential	Tesla C2075	Tesla K20m
Sequential part		10666.276	116.446	111.069
Transfer	- static data	-	< 0.001	< 0.001
	- dynamic data	-	0.001	0.001
	- initial results	-	2.823	1.641
	- refined cells	-	1.526	0.895
Kernels	- Initial KDH calculation	-	195.989	90.981
	- Refinement	(1483.193)	550.496	232.854
	- Accumulation	-	1.102	0.836
Total time		**10666.276**	**868.383**	**438.277**

Table 4-4: Timings of probability densities calculation. Cell size is 25m. Time in seconds.

In case of the EHAM dataset, the refinement kernels are called as often as the regular kernels. Nevertheless, refinement kernels require about 2.5 times as much time as the regular kernels. This has to do with the difference in dimensions of work-items between regular and refinement kernels. The work-items of the regular kernels are ordered one-dimensionally. For the refinement kernels, the work-items are ordered two-dimensionally, so that work-items can (easily) calculate their coordinates. Because of the use of two dimensions instead of one, a less efficient work-item configuration is made. This is one of the points of the probability density calculation of which we think further improvements can be made.

The data that needs to be allocated and transferred for the probability density calculations mainly depends on the size of the study area and the number of segments of the flight paths. In case of the EHAM dataset, the amount of data allocated is approximately 70 MB. The total amount of data transferred is 8.3 GB. The main part of this data is the probability densities, which are copied form device to host after each (refinement) kernel invocation.

4.4 Distribution template calculation parallelisation

Calculating distribution templates from a crash area size is a compute-intensive task. Based on the crash area we compute which cells are covered by the crash area, and for how much these cells are part of the crash area. In the sequential version, distribution templates are stored in main memory to be re-used by cells with the same crash area size. For the parallel version, re-using templates from main memory requires the host to transfer the necessary templates for each kernel invocation. Fortunately, the number of different crash area sizes is limited, so storing all templates in device memory is possible. The calculation of the templates can be done in parallel, so no templates have to be transferred. In the initialization phase of the parallel program all templates are calculated once and concurrently.

All distribution templates are stored consecutively in a single one-dimensional `Buffer`. As mentioned before, only the percentages of the upper right quadrant of the crash area is stored, as shown in Figure 4-2. For the work-items to easily find a template, the templates should be stored at fixed intervals. Since not all templates have the same size, the interval is determined by the size of the largest template.

With the parallelisation of the distribution template calculation, a new problem arises: the work-items should know which template to use. The template to use depends on the work-item's crash area size. If the crash area sizes would have been integers, a direct mapping to the correct index could have been made. Unfortunately, crash area sizes are floating point values, so a direct mapping is not possible as multiple floating point values may map to the same integer. One solution is to let the host make the mapping, but it will lead to a large overhead. This is because the crash area size of each cell needs to be copied from device to the host, and after mapping these values to template identifiers on the host these identifiers must be transferred to the device.

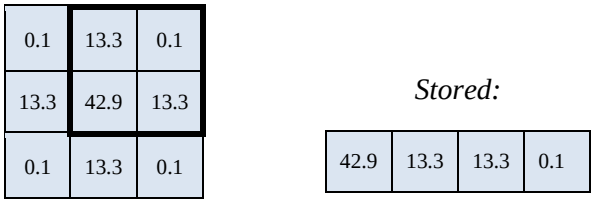

Figure 4-2: Complete distribution template. Only the part surrounded by the thick border is actually stored.

To eliminate the data transfer overhead the mapping should be done on the device. One approach is to use a mapping array, containing all possible crash area sizes. The index of each crash area size in the mapping array corresponds to the identifier of the corresponding distribution template. Looking up the correct template identifier requires a work-item to do a linear search in the mapping array. A linear search operation is very expensive in kernels, because it will lead to load imbalance since some work-items will need more time to search than others, especially when a value to be found is located near the end of the array.

To reduce the lookup time another mapping structure must be used. In the current parallel version, a mapping is made as illustrated in Figure 4-3. This mapping also uses an array containing all possible crash area sizes. However, each crash area size is located at the index equal to its crash area size, converted to an integer, or higher. When performing a lookup, a work-item converts its crash area size to an integer value and uses it as a start index in the mapping array, from which it starts searching. This way, work-items only have to search a small part of the array and load imbalance is reduced significantly. The index where the crash area size is found is the identifier of the distribution template. Note that this requires the templates to be placed at correct indices in the templates array as well. The starting index of a template in the templates array is its identifier multiplied by the maximum template size. In Figure 4-3 this approach is clarified by means of three sample crash areas.

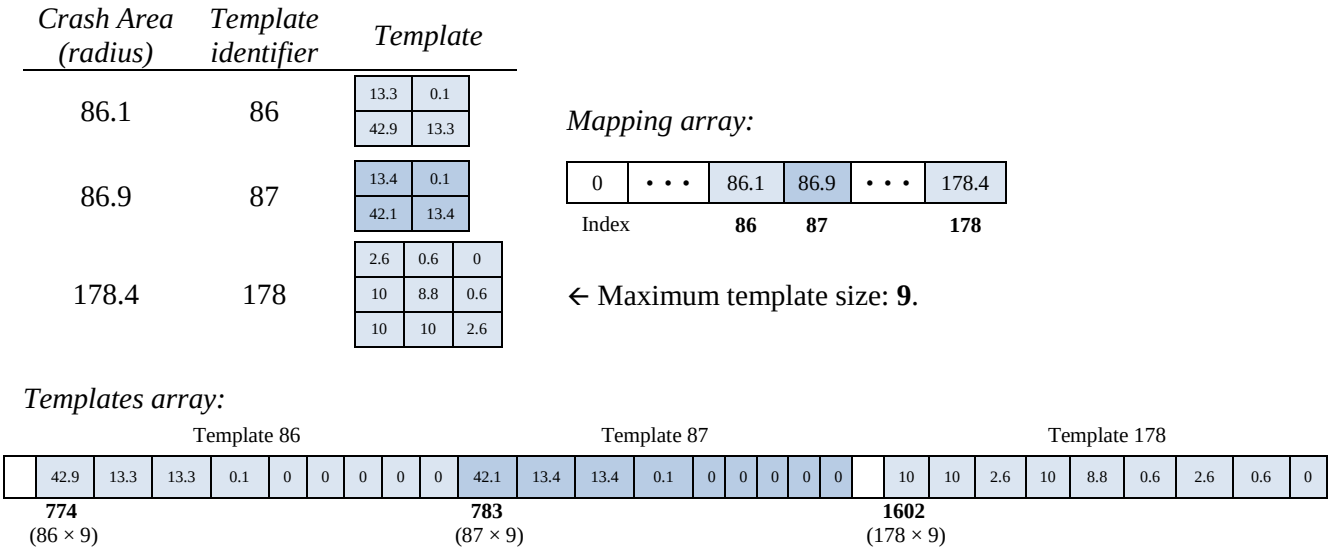

Figure 4-3: Example of mapping of crash area sizes to the corresponding templates. Note: the crash area radii are chosen to demonstrate a mapping conflict when directly mapping from floating point to integer. They are not representative.

The approach described above is not optimal for memory usage, because most likely not all the indices in the mapping array, and therefore also not in the distribution template array, are used. The amount of required memory can be reduced by mapping to a template identifier based on a crash area's radius instead of size. However, more crash areas will map to the same start index, leading to a higher mapping time. Thus, a trade-off has to be made between memory usage and performance. In case of the EHAM dataset with 25m cells, the mapping array and templates array require approximately 258 kB and 806 kB, respectively, instead of 1.3 kB and 4.2 kB when all crash area sizes are stored consecutively. The space wasted is small in comparison to the total memory space required by the application and available on the GPU, so we chose performance over memory.

In Figure 4-3 the mapping of three sample crash areas to the corresponding templates is given. The crash area radii 86.1 and 86.9 show why a direct mapping from crash area radius to template identifier cannot be made: they both map to 86. Assume a work-item has a crash area radius of 86.9. It should check in the mapping array which template corresponds to its crash area. The start index from where to search is 86. Since the value at index 86 is not the work-item's crash area radius, it looks to the value at the next index. At index 87 it finds its crash area radius, meaning that the identifier of the corresponding distribution template is 87. The found template identifier is not the index at which the template can be found in the templates array, because the size of a template is probably larger than 1. The index of the template in the templates array is calculated by multiplying the template identifier by the size of the largest distribution template. Since the size of the largest template in this example is 9, the start index of template 87 is 783 (87×9).

The found identifiers of the cells are kept in a separate array in device memory for use in the Individual Risk distribution phase and the Societal Risk calculation phase. These identifiers do not have to be stored for each cell specifically. Because the crash area size depends on a cell's terrain type, only the identifiers for each terrain type have to be stored. Note that this leads to multiple cells modifying the same value. Atomic operations are not necessary, however, because all these cells write the same value.

	Sequential	**Tesla C2075**	**Tesla K20m**
Data preparation	-	0.009	0.006
Transfer required data	-	< 0.001	< 0.001
Search/map template	1434.982	5.568	3.521
Template calculation	1152.883	0.001	< 0.001
Store template	15644.500		
Total time	**18232.365**	**5.578**	**3.527**

Table 4-5: Timings of distribution template calculations. Cell size is 25m. Time in seconds.

Table 4-5 lists the timings of the calculation of the distribution templates. Data preparation only applies to the parallel version, and consists of the creation of the mapping array and the calculation of the maximum template size. In the sequential version, searching a template concerns the lookup of the distribution template for a certain crash area size, which is done for each cell, each iteration. In the parallel version, the only lookup that takes place is efficiently done by multiple work-items in parallel, based on their converted crash area size as explained above. The big difference in execution times for calculating and storing the distribution templates between the sequential and parallel executions is partially caused by inefficient sequential code.

4.5 Individual Risk parallelisation

We now describe how the Individual Risk calculation is parallelised. In Chapter 2, we mentioned that the Individual Risk calculation can be split in two phases. In the parallel version, each of those phases is implemented in a separate kernel. The reason for this is that we need to be sure that all work-items are finished calculating the risk value in the first phase, before distributing this risk value in the second phase. Since work-items can be synchronized only within their own work-group, global synchronization must be achieved by using separate kernels.

4.5.1 Parallelising IR calculation

In the first phase, the risk caused by a movement, at a certain location, for a given accident type is calculated by the following formula: $IR_{(x,y)} = AR_{acc} \times PD_{(x,y)} \times CA_{(x,y)} \times lethality$, where AR_{acc} is the accident rate of the given accident type, $PD_{(x,y)}$ is the probability density of the given accident type at location *(x,y)*, and $CA_{(x,y)}$ is the crash area size of the accident type at location *(x,y)*.

The parallelisation of this phase is straightforward. Each work-item performs the multiplications itself. The accident rate, probability density, and lethality are provided by the host for each movement and accident type. Hence, at least one complete probability density matrix must be transferred for each movement, depending on the amount of applicable accident types. Unfortunately, the probability density matrices require a lot of memory to keep them all in device memory. For a realistic dataset which generates 125 probability densities of approximately 5 million cells, the required memory is about 4.7 GB. This fits in the device memory of the latest GPUs, but little memory would be available for the remainder of the calculation. Calculating the necessary probability densities when needed has its drawbacks as explained in Section 4.3.2. The crash area size is calculated by the work-items themselves as described in Section 4.3.1.

Besides the calculation of the risk value, the computed crash area is mapped to a distribution template identifier as discussed in Section 4.4. The calculated risk values and mapped template identifiers are stored in a `Buffer` in global memory. Note that it is necessary to store the value in global memory, because this memory is the only persistent memory region on the device, and the calculated risk values are needed in the next kernel.

4.5.2 Parallelising IR distribution

The second phase is the distribution of the risks, calculated in the first phase, over all cells being part of a cell's crash area. The distribution is based on the percentage of the crash area that lies within the relevant cell, and is only dependent on the crash area, for reasons explained in Section 2.2.2. The same section showed an example of how a distribution template is produced from a crash area, and how this template is used. How the templates are computed concurrently and how work-items map their crash area size to the correct template is described in Section 4.4. At the point where the risk values are distributed, the template identifiers applying to the crash areas of each cell are available in global memory.

In the sequential version, each cell's calculated risk is immediately distributed over the relevant cells. This is not possible in the parallel version, because it would lead to race conditions, and atomic operations are not supported for double-precision floating points, as explained in Section 3.4.2.

The solution to this problem is that each work-item should be allowed to update the risk value of its own cell only. Therefore, each cell should check its neighbouring cells for how much they contribute to its own risk value. This is done by multiplying a neighbour's calculated risk value by the correct percentage from that neighbour's distribution template. The result of this multiplication is added the cell's own risk value. Of course, it would be very inefficient if every cell would need to check each other cell in the entire study area for contribution. Therefore, an offset is calculated based on the maximum template

size, indicating the maximum distance in vertical and horizontal direction of the cell that may contribute to a cell's risk. To avoid unnecessary checking of neighbours, the offset is recomputed every movement rather than once at the beginning when all templates are computed. If an offset were to be computed based on the largest distribution template of the complete calculation, for movements for which the offset is smaller, too much neighbours would be checked. Since there are only four distinguishing terrain types, there are only four different distribution templates per movement. Note that to avoid cells taking a percentage of an already updated risk value, the risk values calculated in phase one are stored in a different `Buffer` than the final risks.

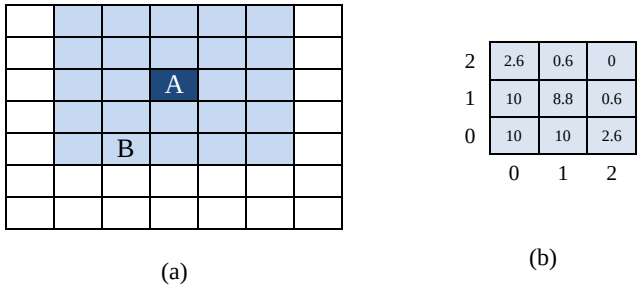

Figure 4-4: (a) Sample grid showing the neighbours which may contribute to cell A's risk. (b) Sample distribution template of cell B.

Figure 4-4 (a) shows for a sample cell which neighbours it needs to check for contribution. Assume cell A is calculating its risk value. If the maximum distribution template size is 3 × 3, the maximum offset is 2. This means that cell A needs to iterate over all cells with an *x* and *y* value of at most 2 less or 2 more than cell A, and take the correct percentage of that cell's calculated risk value. Assume cell A needs to calculate cell B's contribution, and the distribution template applying to cell B is as given in Figure 4-4 (b). Note that this is the upper-right part of the complete distribution, so it must be mirrored vertically and horizontally, if required. The position of cell A relative to cell B is (1,2). Thus, cell A gets the percentage of cell B's calculated risk value which is located at (1,2) in cell B's distribution template, which is 0.6%.

During the distribution phase, multiple work-items in a work-group (partially) require the same data located in global memory, including initial risk values of other cells and distribution templates. To reduce accesses to global memory, this data is read into shared memory. See Section 4.8.1 for a more detailed explanation and timing results.

	Sequential	**Tesla C2075**		**Tesla K20m**	
	Time	Time	*Speedup*	Time	*Speedup*
Read probability densities	6567.210	208.827	*31.4*	66.591	*98.6*
Obtain risk calculation data	2715.904	0.045	*60353.4*	0.033	*82300.1*
Obtain risk distribution data	18342.202	0.045	*407604.5*	0.038	*482689.5*
Transfer: - Probability densities	-	143.930	-	63.244	-
- Risk calculation data	-	0.046	-	0.029	-
- Risk distribution data	-	0.031	-	0.021	-
- Results	-	0.019	-	0.011	-
Calculate risk values	4963.124	10.395	*477.5*	7.034	*705.6*
Distribute risk values	603089.837	92.278	*6535.6*	37.646	*16020.0*
Store results	-	0.067	-	0.051	-
Accumulated time	**635678.277**	**455.682**	***1395.0***	**174.682**	***3639.1***
IR time with concurrency	**-**	**348.375**	-	**132.820**	-

Table 4-6: Timings of Individual Risk calculation. Cell size is 25m. Time in seconds.

Table 4-6 shows the execution times of an Individual Risk calculation run sequentially on a CPU (Intel Core 2 E8400), and run in parallel on the Tesla C2075 and Tesla K20m. Note that the time spent to store the results by the sequential version is included in the calculation times. The accumulated total time is computed by simply accumulating the separate operations. The second total time is the average of 3 runs of the application in which data transfer and computation on both host and device are done concurrently. See Section 4.8.2 for a more detailed explanation and more timing results.

In the sequential version, the main part of the execution time is spent on distributing the risk values. This makes sense since most of the computations take place in the distribution phase, but compared to the fastest parallel time it is over 16,000 times slower. This difference cannot be attributed to the use of 2496 cores only. The main difference is caused by some very inefficient storing of the data in the sequential version. Remember that in the sequential version the risk calculated in a cell is immediately distributed to its neighbouring cells. For each update of a cell's value, the correct grid is searched. This searching is very expensive, especially since it is done for each update operation. In the parallel version, the index of the value to update is computed instead of searched.

The difference in time required to read the probability densities by the sequential version and the parallel version on the Tesla C2075, cannot be caused by the hardware since the same hardware is used for this operation. The probability densities are read one by one in the sequential version, requiring a search for the correct probability density matrix for each cell. In the parallel version, a probability density matrix is read entirely, requiring only one search per matrix.

Another observation that can be made from the timings in Table 4-6 is that, in the parallel version, a significant part of the time is spent on reading and transferring probability densities. With the EHAM dataset, reading probability densities include reading 125 probability densities of over 5,000,000 double-precision floating point numbers, and searching for requested probability density matrices. Transferring the requested probability densities involves transferring over 350 GB of data. The difference between both parallel timings is due to difference in hardware used.

4.6 Societal Risk parallelisation

The way the Societal Risk calculation is parallelised is quite similar to the Individual Risk parallelisation. This applies to the structure of the parallel versions, as well as to the operations. However, there are some differences, mainly regarding the organization of the kernels.

4.6.1 Parallelising SR calculation

In the Societal Risk calculation, the risks for requested group sizes are computed for each cell in the study area. To do so, each cell must know how many people are possibly present in the crash area. The population is given per cell, so each cell should compute the population within a crash area by taking the percentage of the population equal to the percentage of that cell covered by the crash area. Note that this calculation is much like the IR distribution phase. In fact, the distribution templates are calculated and applied in the same way as for the IR calculation.

Just as for the IR calculation, work-items compute the crash area sizes of their cell and map these to the right distribution templates themselves. In the IR calculation, these two operations are done in the same kernel as the actual risk calculation. In the SR calculation, a separate kernel is used to compute the crash area sizes and do the mapping. The reason for this is that the application of the distribution templates is done within the same kernel as the risk calculation, instead of in two separate kernels as for the Individual Risk. Remember that two kernels are used for IR calculation because of the global synchronization that needs to take place between the computation and distribution phase. For SR calculations, this global synchronization is not necessary, because the distribution templates are applied to the population, which is constant throughout the program. Global synchronization is still necessary,

however, but now between the kernel computing and mapping the crash areas and the kernel doing the risk calculations, because the mapping must be available when the second kernel is started.

4.6.2 Parallelising SR accumulation

When these two kernels have been executed for all movements, a risk value for each group size for each cell has been computed. These values should be accumulated to a total risk value for each group size. Instead of looping over all cells for each group size, an accumulation kernel is used, in which each work-item iterates the study grid and adds all risk values of a certain group size. Using a kernel for the accumulation of the risk values does not only ensure that the risk values of the group sizes are accumulated concurrently, also the transfer of these risk values from device to host is avoided.

It may seem logical to use a number of work-items equal to the number of cells, since there are substantially more cells than group sizes, and have them add their risk values to the correct group size. However, this would lead to race conditions, similar to the case explained in Section 4.5.2, as multiple work-items need to update the same variable.

	Sequential	**Tesla C2075**		**Tesla K20m**	
	Time	Time	*Speedup*	Time	*Speedup*
Read probability densities	511.684	20.061	*25.5*	6.379	*80.2*
Obtain required data	-	0.011	-	0.007	-
Transfer: - probability densities	-	15.185	-	6.664	-
- other data for kernels	-	0.006	-	0.005	-
- results	-	< 0.001	-	< 0.001	-
CA calculation/mapping	180.474	1.544	*116.9*	1.125	*160.4*
Calculate max. number of victims	1108.303	9.115	*237.0*	6.280	*344.0*
Risk distribution	1052.001				
Store results	5046.630	< 0.001	-	< 0.001	-
Accumulated total	**7899.092**	**45.922**	***172.0***	**20.460**	***386.1***
SR time (with concurrency)	-	**38.763**	-	**16.464**	-

Table 4-7: Timings of Societal Risk calculation. Cell size is 25m. Time in seconds. 500 movements.

In Table 4-7 execution times of a Societal Risk calculation of the first 500 movements of the EHAM dataset are listed. Due to time constraints we could not do a sequential run of the complete dataset. Timings of parallel runs of the complete dataset are shown in Section 4.9.

The same observations as for the Individual Risk calculation can be made. The parallel version is faster on the Tesla K20m than on the Tesla C2075 due to hardware differences, and reading the probability densities is slower in the sequential version due to inefficient code. Note that the CA calculation and mapping operation is performed in the first kernel, and the calculation of the maximum number of victims and distribution of the risk is done in the second kernel. The timings for the latter also include the time of the accumulation kernel, as accumulation is done implicitly in the sequential version's risk distribution step.

4.7 Implementing subset output support

The standard output of an Individual Risk calculation is a grid containing the total risk of the provided study area and traffic. Additional output grids may be requested, which contain, for example, the risk incurred by aircrafts of a certain generation only, or the risk incurred by aircrafts using a certain runway. The complete set of possible subsets, that may be requested as additional output, consists of output per

part of day (day/night), per runway head, per aircraft weight category, per aircraft generation, per direction (take-off/landing), per accident type, per operation type (passenger, cargo, business jets), per route, and per movement. Also, two combinations are possible, per runway head and direction, and per direction and accident type.

For all these options the dedicated grids are updated for each movement, if applicable, except for output requested per movement. In that case, the output contains only the risk values incurred by the movement itself. To save device memory, the movement grid is copied to the host immediately after each movement calculation. Another special case is the part-of-day subset option. In contrast to the other subset options, this option is applied in combination with other options. Thus, for each subset output grid (containing risk values for day and night movements combined), two extra output grids are created: one for day movements and one for night movements.

Besides the grids that need to be stored for subset output, extra grids need to be stored when multiple models (i.e. traffic files) are given. In that case, all requested grids are stored for each model and for all models accumulated. Since the amount of device memory is limited, the number of models and subset output grids that can be requested is limited as well. Note that this limit depends on the size of the study area. More on this limit is given in Section 4.9, where the parallel program is evaluated in terms of memory usage, among others.

In the sequential version, whenever a risk value is calculated, it is immediately added to the correct grids, which are selected using a mapping based on movement-specific data. To support subset output in the parallel version, device memory must be allocated for the additional grids. The amount of additional grids depends on the number of grids required by each requested subset. The grids are allocated in device memory as a one-dimensional array, for reasons explained in Section 4.2.2.

When a work-item has computed a risk value, it should know which grid(s) it must update. Selecting the right grid(s) to update involves the mapping of a string to a grid index. This mapping is done on the host for each movement and accident type, and results in a small array containing the indices of the grids that need to be updated. Note that these indices apply for all work-items, so no specific mapping for each cell has to be done. Each work-item loops over the indices array and updates the corresponding grids.

To conclude, supporting subset output in the parallel version requires extra device memory to store the requested output grids, and a small mapping has to be done on the host for each movement. Also, more complex index calculation takes place. However, this leads to more efficient storing of the data and therewith to a performance gain over the sequential version.

4.8 Further optimizations

Besides by parallelising operations of the sequential program, performance can be improved by other optimizations, such as using shared memory and registers in kernels instead of global memory. Other optimizations include concurrent data transfer and calculation on host and device, and improvements to the sequential part.

4.8.1 Using registers, pinned memory, and shared memory

Since the global memory is the largest piece of device memory and the only device memory which is persistent and directly accessible by the host, data required by the kernels are stored in this memory. Accessing global memory, however, is much slower than accessing shared memory or registers. Therefore, reads from and writes to the global memory should be minimized. In the parallel version, work-items that use a data value multiple times store it in a register. This way, the data value has to be read from global memory only once. The same applies to intermediate results. Instead of updating a value in global memory multiple times within a kernel, intermediate results are kept in registers and only the

final result is written to global memory. The use of registers should be limited, though, as there is only a fixed amount of registers available to a multiprocessor. When processors require too much registers, not all processors can be used simultaneously.

The highest bandwidth between host and device can be achieved by using pinned memory. When a memory region is pinned, the operating system will not swap data in this region to secondary storage, therewith ensuring the data is always present at the same location. The use of pinned memory should be limited, because excessive use can reduce overall system performance [26]. Therefore we decided to only pin the memory regions accessed most frequently, which are the memory regions of data copied for (almost) every movement and accident type. Without the use of pinned memory, the time spent on data transfer in an IR calculation is 200.610 seconds. When pinned memory is used, the same data transfer takes only 63.114 seconds, which is a speedup of 3.18. In case of the EHAM dataset with 25m cells, the amount of pinned memory is less than 40 MB.

If data is used by multiple work-items of a work-group, storing the data in shared memory may improve performance. Using shared memory, the amount of accesses to global memory can be reduced substantially, resulting in less traffic on the memory bus and faster data access. A kernel in which shared memory is used is the Individual Risk distribution kernel. In this kernel, each cell needs the risk value of a certain number of neighbouring cells, which are also neighbours of each other. To reduce the number of accesses to global memory, the risk values needed by the work-items in a work-group are copied to shared memory. Also, the required distribution templates can be stored in shared memory. Since there are only four different terrain types, there are only four different crash area sizes possible, hence only four distribution templates are used each kernel invocation. To be able to determine which of the four distribution templates applies to a certain cell, a work-item needs to know the cell's terrain type. These can best be stored in shared memory as well, since it is requested multiple times within a work-group.

The terrain, the risk values, and the distribution templates are copied to shared memory in the beginning of the kernel. Obviously, this must be done as efficiently as possible to avoid wasting the time gained by using the shared memory. Each work-item is responsible for copying one or more values from global to shared memory. After this, work-items need to synchronize to ensure all the required data is present in shared memory, before continuing.

	Individual Risk
No shared memory	70.768
Shared risk values	50.220
Shared templates and terrain	55.484
Shared risk values, templates and terrain	44.680

(a)

	Societal Risk
No shared memory	134.035
Shared population	137.867
Shared templates and terrain	140.030
Shared population, templates and terrain	139.050

(b)

Table 4-8: Timings of Individual and Societal risk calculations with and without shared memory usage. Run on NVIDIA Tesla K20m. Cell size is 25m. Time in seconds.

Individual Risk calculations are almost 37% faster when shared memory is used, as can be seen in Table 4-8 (a). Note that this only concerns the actual calculations (i.e., the IR calculation and IR distribution kernel), rather than the complete Individual Risk calculation, which includes data acquisition and transfer.

Using the same strategy for the Societal Risk calculation does not give a performance improvement, as can be concluded from Table 4-8 (b). In fact, the calculation is slowed down. This is caused by a conditional statement in the SR calculation kernel. If the risk value in a cell is smaller than 10^{-99}, the remainder of the calculation is skipped. This check is done after the transfer of data from global to shared memory. Thus, even if all work-items in a work-group skip a part of the calculation, time is spent on shared memory. Therefore, using shared memory is not beneficial for Societal Risk

calculations for the EHAM dataset. A logical action would be to do the check before work-items copy data to shared memory. However, it is possible that a part of the work-items in a work-group need to do the computation, and the other part not. In that case, still all work-items are needed to help preparing shared memory. Hence, work-items may only skip shared memory preparation when all group members have a risk value smaller than 10^{-99}. Checking the risk value for each other group member would induce too much overhead to be profitable.

4.8.2 Concurrent host calculation, data transfer and kernel execution

The order in which operations on the host are performed, and data transfers and kernel executions are invoked, may influence the performance. This mainly depends on whether the device supports concurrent data transfer and kernel execution. Figure 4-5 (a) shows the operations that take place for each movement for an Individual Risk calculation without paying attention to concurrency. Obtaining the probability density matrix (PDM) is modelled separately from the calculation of data for IR calculation, because it takes substantially more time. Also, copying the PDM takes more time than copying the rest of the data for the IR calculation kernel.

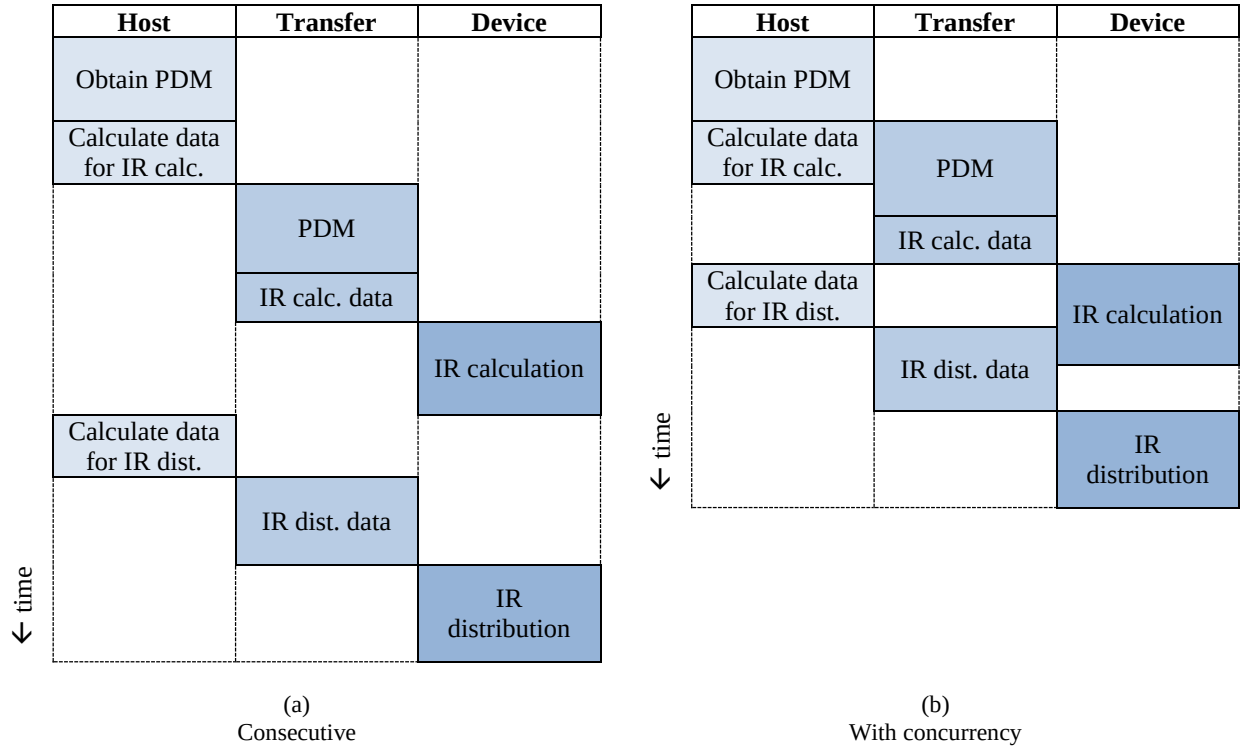

Figure 4-5: Operation flow of a part of the Individual Risk calculation.

Since data transfer can be done non-blocking, part of the data preparation which takes place on the host can overlap with the data transfer. The operations can be parallelised even further if the device supports concurrent data transfer and kernel execution. This concurrency can be achieved by using two separate Command Queues. One queue should be used for data transfer commands, the other for kernel execution commands. To make sure all necessary data has been transferred before a kernel execution starts, the kernel invocation command is added to the queue only when all commands in the data transfer queue are finished. To prevent the host from getting ahead of the device, the host waits for the IR calculation kernel

to finish before continuing to the next movement or accident type. See Figure 4-5 (b) for the operation flow with concurrency. In most cases, operations of the next movement can already start while the device is still performing the IR distribution kernel, leading to an even higher level of parallelism. Note that when concurrent data transfer and kernel execution is not supported, executing operations on the host and transferring data can still be done in parallel. The Individual Risk calculation part is approximately 24% faster with concurrency than without, for a cell size of 25 by 25 metres.

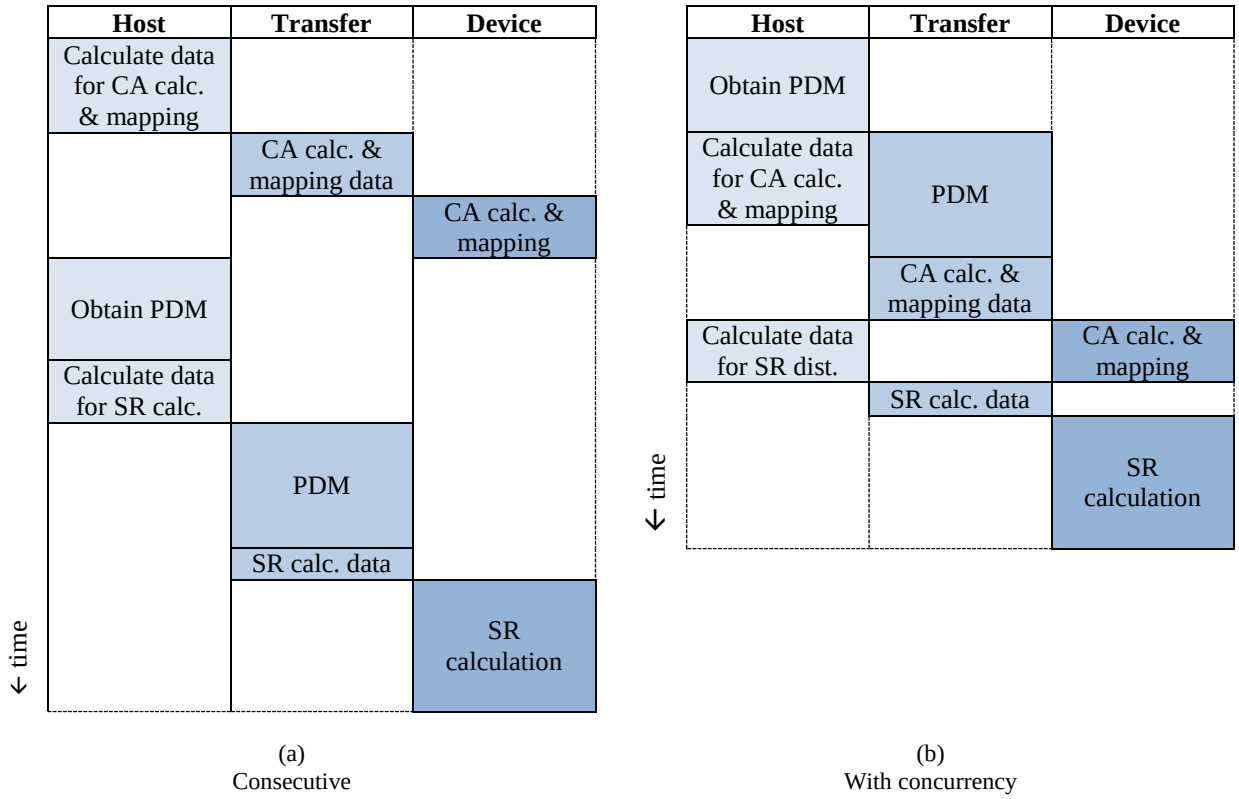

Figure 4-6: Operation flow of a part of the Societal Risk calculation.

For Societal Risk calculations, the same strategy can be applied. See Figure 4-6 (a) for the operation flow of the Societal Risk calculation. The main difference between the consecutive operation flows of Individual and Societal Risk calculations is the moment at which the probability densities are obtained. In the IR calculation this is done before the first kernel. In the SR calculation this is done after the first kernel, because the probability densities are only used in the second kernel. This implies that transferring probability densities and executing the SR calculation kernel cannot be done concurrently. However, since these operations require most time, it would be best to do them in parallel. Therefore, we modified the kernels to use the probability densities in the first kernel. This way, the operation flow as depicted in Figure 4-6 (b) is made possible, in which obtaining and transferring probability densities can be done in parallel with the SR calculation kernel. The performance gained by this approach, for 25 × 25 metre cells, is approximately 14 to 19%.

4.8.3 Improving the sequential code

Some parts of the parallel program need to be done sequentially. Optimizing these parts of the program is as important as optimizing the parallel parts. Thanks to the extensive study to convert the algorithm to a

parallel version, inefficiencies in the sequential version have been found. One sequential operation that proved to be time-consuming is obtaining the probability density values. The main reason for this is that in the original sequential version the probability densities are obtained cell by cell, and for each cell a linear search in a mapping takes place. We improved this operation by only doing a search when a new probability density matrix is required, and obtaining the complete matrix in one call.

The same inefficiency applies for other operations in the original sequential version where data is read from or stored in matrices. Especially inefficient is the way a calculated Individual or Societal Risk value is stored. First, a string is constructed from movement-specific information. Then, using the string as a key, the corresponding grid is searched. When found, the risk value is stored. Also, the program checks whether subset output grids need to be updated as well and performs the same operations for each requested subset. Note that this inefficiency is eliminated in the parallel version by the way the results are stored in the kernels.

We eliminated the inefficiencies described above from the original sequential version as well, by implementing an optimized sequential version. Using the timings of the optimized sequential version, rather than the original sequential version, gives a fairer comparison regarding the speedup due to the use of GPUs. Note that the comparison still is not completely fair, as some of the optimizations used in the parallel program have not been ported back to the sequential version, such as the crash area size mapping, and the way distribution templates are used. These optimizations have not been implemented in the optimized sequential version due to time constraints, as it would require a modification of the program's structure. Nevertheless, the optimizations that are implemented attain a speedup of almost 10, where the original sequential version takes about 7 days, the optimized sequential version takes about 17.5 hours. Note that the sequential timings documented in this thesis are timings of the original sequential version.

4.9 Overall evaluation

Up till now, the timing results shown are for the argued parts of the calculation only. In this section, the complete program is timed, including reading the input from the given files. Also, the parallel version is run for various cell sizes and a various amount of movements to determine its scalability.

4.9.1 Complete program timings

In Table 4-9 the execution times of the complete program are listed for Individual and Societal Risk calculation, either with or without calculating probability densities. From these timings can be concluded that probability density calculations take more time than the risk calculations themselves. This observation is not very surprising, because the calculation of probability densities is very compute intensive. However, we think probability density calculation performance can be improved, as is discussed in Chapter 5.

	Probability density calculation?	**Tesla C2075**	**Tesla K20m**
Individual Risk	No	384.705	163.292
	Yes	1224.063	637.724
Societal Risk	No	528.941	227.007
	Yes	1369.921	888.708

Table 4-9: Execution times of complete program. Cell size 25m. Time in seconds.

A sequential run of the Individual Risk calculation on an Intel Core 2 Duo E8400 CPU takes 515,208 seconds, which is approximately 6 days. On the CPU of machine A, the calculation even takes about 7 days. In Table 4-10, the speedups for the Individual Risk timings are shown. A similar table for Societal Risk timings cannot be made, because we could not do a sequential run in time. We expect the speedups to be at least as high as for Individual Risk calculations, as more calculations are performed and inefficient code in the sequential version is executed more often.

The most striking observation is that the speedup exceeds the number of cores for the calculations with pre-computed probability densities. This is caused by the inefficient code in the sequential program, in particular regarding reading and writing data. Further, there can be concluded that running the parallel version on the Tesla K20m is about 2 times faster than running it on the Tesla C2075.

Probability density calculation?	**Sequential**	**Tesla C2075** (448 cores)		**Tesla K20m** (2496 cores)	
	Time	Time	Speedup	Time	Speedup
No	515208.0	384.705	**1339.2**	163.292	**3155.1**
Yes	524199.0	1224.063	**428.2**	637.724	**822.0**

Table 4-10: Individual Risk timings and speedups. Cell size 25m. Time in seconds.

The maximum speedup achievable due to parallelisation only can be computed with Amdahl's Law, which states that the maximum speedup depends on the portion of a program that can be made parallel. The formula is: $max\ speedup = 1/(P_S + (P_P/cores))$, where P_S and P_P are the percentages of the total execution time spent on inherently sequential parts and parts which can be run in parallel, respectively. For Individual Risk calculations, run with the optimized sequential version with pre-computed probability densities, these percentages are found to be 0.02% and 99.98% (see Table 4-11). The sequential part consists of hard disk and main memory accesses and determining which grids needs to be updated for each movement. The parallel part consists of all risk computation parts, but also the mapping and computation of crash area sizes and distribution templates. The update operation is a combination of calculation and storing. We counted this operation as parallel, because storing to device memory is also done in parallel, and risk values are stored in the host's main memory only once in the entire parallel calculation.

Using the number of GPU cores for the *cores* variable in Amdahl's Law would be unfair, because in general, GPU cores are slower than CPU cores. Therefore, it would be fairer to use the increase in theoretical maximum performance of the GPUs, which is 23 for the Tesla C2075 and 53 for the Tesla K20m. Applying Amdahl's Law with these values give that the maximum speedup using a Tesla C2075 is 22.9 and using a Tesla K20m is 52.5, which is close to their theoretical maximum performance gain, since the inherently sequential part is very small. If the number of cores would be infinite, the upper bound of maximum speedup can be calculated by $1/P_S$. In case of Individual Risk calculations the maximum speedup, disregarding the amount of cores used, is 5000.

The speedups listed in Table 4-10 exceed the maximum speedups, because some parts of the algorithm are done in a more efficient way. For reasons given in Section 4.8.3, these optimizations are not back-ported to the optimized sequential version.

	Time	Percentage
Sequential	70.095	0.0002
Parallel	454238.873	0.9998

Table 4-11: Proportion of inherently sequential operations and operations suitable for parallelisation. Time in seconds.

4.9.2 Scalability

The timings shown up till now all concerned using a cell size of 25 × 25 metres. To determine the scalability of the program, several cell sizes are used. See Table 4-12 for timings of Individual Risk calculations. The movements used for these timings are only the first 500, so in case of 1500 movements, the same set of 500 movements is calculated three times. We chose this approach because some movements take longer than others. Note that with a cell size of 50 × 50 metres, the number of cells has quadruples with respect to a 100 × 100 metres cell size, and a cell size of 25 × 25 metres quadruples the number of cells with respect to 50 × 50 metre cells.

Cell size Movements	**100 × 100 m** (313,600 cells)	**50 × 50 m** (1,254,400 cells)	**25 × 25 m** (5,017,600 cells)
500	0.998	3.574	14.287
1500	2.731	10.159	41.191
2500	4.488	16.706	67.889
3500	6.184	23.284	94.916
4500	7.924	29.908	121.583

Table 4-12: Individual Risk calculation time. Cell size is 25m. Time in seconds.

As expected, the execution time increases linearly when the number of movements to be calculated increase. The execution time also tends to increase linearly when the number of cells increase. Actually, execution times increase even less than the number of cells. This is because the extra work some operations have to do hardly influence the total execution time, such as calculating distribution templates, and transferring and storing results. For the sequential version, these operations have a bigger influence. Therefore, we expect the speedup of the parallel version to increase even further as the number of cells increase. A similar trend can be seen for probability density calculations. See Table 4-13 for probability density calculation timings.

100 × 100 m (313,600 cells)	**50 × 50 m** (1,254,400 cells)	**25 × 25 m** (5,017,600 cells)
31.034	119.310	443.068

Table 4-13: Probability densities calculation time. Cell size is 25m. Time in seconds.

Of course, there is a limit to the maximum amount of cells of the study area. As the number of cells increase, by either using a larger study area or a smaller cell size, the amount of memory required on the device increases as well. However, the number of cells is not the only factor influencing the required memory. See Table 4-14 for the complete formulas of the required memory. Note that the amount of movements (i.e., traffic lines) has no influence on the required memory.

Calculation	Formula
Probability densities	$4 \times (13 + nrOfCells) + 52 + 104 +$ $8 \times (2 + nrOfCells + 3 \times (nrOfCells/refinementFactor))$ Extra memory for per-runway calculation: $8 \times (nrOfRadii + 2 \times nrOfSectors)$ Extra memory for per-route calculation: $4 \times nrOfSegments + 8 \times 11 \times nrOfSegments$
Individual Risk	$4 \times (nrOfCells + nrOfSubsets + 3 \times nrOfTerrainTypes + nrOfDayparts) + 68 +$ $8 \times (3 \times nrOfCells + nrOfUniqueCAs \times templateSize + nrOfUniqueCAs + nrOfGrids \times nrOfCells) +$ Extra memory if output per movement is requested: $8 \times nrOfCells \times (nrOfDayparts\text{-}1)$
Societal Risk	$4 \times (2 \times nrOfCells + nrOfSubsets + 3 \times nrOfTerrainTypes + nrOfDayparts + nrOfGroupSizes) + 68 +$ $8 \times (3 \times nrOfCells + nrOfUniqueCAs \times templateSize + nrOfUniqueCAs + nrOfCells \times nrOfDayparts) +$ $8 \times (nrOfDayparts+1) \times nrOfModels \times (nrOfSubsetOptions+1) \times nrOfGroupSizes \times (nrOfCells+1) +$ Extra memory if output per movement is requested: $8 \times nrOfAccidentTypes \times (nrOfGroupSizes+1) \times (nrOfDayparts+1) \times (nrOfCells+1)$

Table 4-14: Formulas to calculate required memory (in bytes).

The minimum amount of memory required for either an Individual or Societal Risk calculation of the EHAM dataset is listed in Table 4-15. There is no subset output requested. Since the size of global memory of recent GPUs is 5 or 6 GB, calculations with 10 × 10 metre cells is possible.

The extra memory required when a subset output is requested, is a complete matrix for each possible value within a subset and a few extra bytes. In case of 25 × 25 metre cells, approximately 38 MB extra memory is required per subset possibility. For example, when output is requested per operation type, and the possible operations types are business, cargo, and passenger, about 114 MB (3 × 38 MB) extra memory is required. The limit of 5 or 6 GB may easily be exceeded, especially when per-day-part output is requested in combination with other subsets, tripling the amount of extra memory needed. For Societal Risk calculations, the required extra memory is also multiplied by the amount of group sizes requested. Note that for 10 × 10 metre cells the limit will be reached sooner than for 25 × 25 metre cells as one matrix requires approximately 239 MB.

	100 × 100 m (313,600 cells)	**50 × 50 m** (1,254,400 cells)	**25 × 25 m** (5,017,600 cells)	**10 × 10 m** (31,360,000 cells)
Probability densities	4.3	17.2	68.9	430.7
Individual Risk	10.8	43.1	172.3	1067.7
Societal Risk	45.3	165.0	658.0	4055.6

Table 4-15: Required memory (in MB) for EHAM dataset with various cell sizes. No subset output requested.

Chapter 5

Conclusions

The main research question was whether it is possible to accelerate Third Party Risk around an airport calculations using a GPU. From the results we have shown can be concluded that it is possible. Due to the use of thousands of processing cores, a complete third party risk calculation for Schiphol Airport, which takes approximately 6 days to finish on a single core, can be executed within 15 minutes. This performance gain enables to compute risks on a dense grid (25 × 25 metre cells), as preferred by the Dutch government, and to compute risks with individual aircraft movements instead of aggregated movements. Also, it enables computations to be done within a reasonable time, such as Societal Risk calculations on a dense grid. It is even feasible to do computations for a grid with 10 × 10 metre cells. There is, however, a limit to which extent the program scales, imposed by the available memory of a GPU. The more number of cells that need to be computed, the more memory required. This limit is expected to be eliminated in the next couple of years because of the new architectures announced by NVIDIA and AMD. NVIDIA's unified virtual memory [32] and AMD's heterogeneous system architecture [33] will make the main system's memory and device memory accessible for both CPU and GPU.

Achieving the results presented in this thesis is not the result of simply converting the sequential implementation to a parallel one. There are many factors that needed to be taken into account when deciding how to parallelise an algorithm. One is that instructions may be depending on other instructions, i.e. an instruction may need to wait for the result of another instruction. Synchronization may be necessary between the processing cores to avoid incorrect results. Such a situation occurs in the Individual Risk calculation, for example, where the risk distribution can only be started after a risk value is calculated for each cell. So, some calculations must be divided into several sub calculations to enable synchronization.

Some parts of the algorithm may have to be rewritten to be (more) suitable for parallel execution. In the parallel program, instead of distributing a cell's risk to other cells, each cell should gather parts of risks calculated by others. Without this modification to the algorithm, the current level of parallelism would not have been possible. Also, a part of the Societal Risk calculation is done at an earlier moment than in the original algorithm to enable a higher level of concurrency.

Another point is that, ideally, a piece of code executed in parallel induces the same workload on all processing cores. This implies that branching, as a consequence of conditional statements, should be avoided. However, sometimes branching is unavoidable. In the probability density calculation and distribution template calculation kernels, branching occurs.

The results presented in this thesis show a significant performance gain, but we consider further improvements may be possible. The timings and performance breakdowns show that most time is spent on calculating, reading and transferring probability density matrices. In the current parallel program, at least one probability density matrix is read and transferred per movement. However, a probability density matrix is used by multiple movements. Changing the order in which movements are processed can drastically reduce the number of times a probability density matrix must be read and transferred. Ideally, each probability density matrix is read and copied only once. Time spent on reading probability density matrices from memory may be reduced by eliminating the use of the custom-designed matrix. Storing the probability densities in a one-dimensional vector when read from hard disk avoids converting the two-dimensional matrix to a one-dimensional vector. Note that this requires all methods applying these matrices to be modified, and that the software will be more complex, as index computation is required. Whether the calculation of the probability densities itself can be improved is difficult to say. For this, further study is recommended.

Bibliography

[1] Y. Zhang, P.Vouzis and N.V. Sahinidis. *GPU simulations for risk assessment in CO_2 geologic sequestration.* Computers & Chemical Engineering. Volume 35, Issue 8, pages 1631-1644. August, 2011.

[2] A.K. Bahl, O. Baltzer, A. Rau-Chaplin, B. Varghese and A. Whiteway. *Multi-GPU Computing for Achieving Speedup in Real-time Aggregate Risk Analysis*. High Performance Computing on Graphics Processing Units (hgpu.org). February, 2013.

[3] J.Weijts, R.W.A. Vercammen, Y.A.J.R. van de Vijver and J.W. Smeltink. *Voorschrift en procedure voor de berekening van Externe Veiligheid rondom luchthavens.* NLR-CR-2004-083. Dutch National Aerospace Laboratory. February, 2004.

[4] CBS, PBL, Wageningen UR (2007). *Externe veiligheidsrisico's: de kans op een ongeluk (inleiding)*. Indicator 0300, Ver. 05. September, 2007.

[5] C. McClanahan. *History and Evolution of GPU Architecture: A Paper Survey*. Georgia Tech, College of Computing. 2010.

[6] T. S. Crow. *Evolution of the Graphical Processing Unit.* Professional Paper, University of Nevada, Reno. December, 2004.

[7] J. Nickolls and William J. Dally. *The GPU Computing Era.* IEEE Micro. Vol. 30, Issue 2, pages 56-69. March/April, 2010.

[8] C.J. Thompson, S. Hahn, M. Oskin. *Using Modern Graphics Architectures for General-Purpose Computing: A Framework and Analysis*. Micro, pp. 306, 35th annual IEEE/ACM International symposium on microarchitecture (MICRO'02), 2002.

[9] J. Palacios and J. Triska. *A Comparison of Modern GPU and CPU Architectures: And the Common Convergence of Both*. March, 2011.

[10] Y. Kreinin. *SIMD < SIMT < SMT: parallelism in NVIDIA GPUs.* http://www.yosefk.com/blog/simd-simt-smt-parallelism-in-nvidia-gpus.html. November, 2011.

[11] *PCI Express specifications.* http://www.pcisig.com/specifications/pciexpress/

[12] *NVIDIA GeForce 8800 GPU Architecture Overview*. Technical Brief, NVIDIA. November, 2006.

[13] *NVIDIA's Next Generation CUDA Compute Architecture: Fermi*. Whitepaper, NVIDIA. 2009.

[14] *NVIDIA GeForce GTX 680. The fastest, most efficient GPU ever built*. Whitepaper, NVIDIA. 2012.

[15] *NVIDIA GT200 GPU and Architecture Analysis*. Beyond3D. http://www.beyond3d.com/content/reviews/51/1. June, 2008.

[16] M. Papadopoulou, M. Sadooghi-Alvandi and H. Wong. *Micro-benchmarking the GT200 GPU*. Technical Report, Computer Group, ECE, University of Toronto. 2009.

[17] C. M. Wittenbrink, E. Kilgariff and A. Prabhu. *Fermi GF100 GPU Architecture*. IEEE Micro. Vol. 31, Issue 2, pages 50-59. March, 2011.

[18] D. Patterson. *The Top 10 Innovations in the New NVIDIA Fermi Architecture, and the Top 3 Next Challenges*. Parallel Computing Research Laboratory (Par Lab), University of California, Berkeley. September, 2009.

[19] A.S. Nielsen, A.P. Engsig-Karup and B. Dammann. *Parallel Programming using OpenCL on Modern Architectures*. IMM-Technical Report-2012. Technical University of Denmark. 2012.

[20] D. Kanter. *AMD's Cayman GPU Architecture*. Real World Technologies. December, 2010. http://realworldtech.com/cayman/

[21] *Tesla GPU Accelerators for Servers*. NVIDIA. http://www.nvidia.com/object/tesla-servers.html

[22] *OpenCL Programming Guide for the CUDA Architecture*. Version 4.2, NVIDIA. September, 2012.

[23] J. Fang, A.L. Varbanescu and H. Sips. *A Comprehensive Performance Comparison of CUDA and OpenCL*. 2011 International Conference on Parallel Processing (ICPP). Pages 216-225. September, 2011.

[24] B.R. Gaster, L. Howes, D. R. Kaeli, P. Mistry and D. Schaa. *Heterogeneous Computing with OpenCL*. Morgan Kaufmann, Elsevier. ISBN 978-0-12-387766-6. First edition, 2012.
[25] M. Scarpino. *OpenCL in Action: how to accelerate graphics and computation*. Manning, 2012.
[26] *OpenCL Best Practices Guide*. NVIDIA. February, 2011.
[27] *NVIDIA Tesla C2075 Companion Processor*. NVIDIA. September, 2011. http://www.nvidia.com/docs/IO/43395/NV-DS-Tesla-C2075.pdf
[28] *OpenCL 1.2 specification.* Khronos Group. Revision 19, November, 2012.
[29] Y.A.J.R. van de Vijver . *Ondersteuning NLR bij validatieproces GEVERS.* MN-GEV-16. Dutch National Aerospace Laboratory. October, 2007.
[30] Y.A.J.R. van de Vijver and M.A. Guravage. *Validatie van TRIPAC, de nieuwe software voor analyse van externe veiligheid rond luchthavens.* NLR-TR-2003-651. Dutch National Aerospace Laboratory. November, 2003.
[31] P. Collingbourne, C. Cadar and P.H.J. Kelly. *Symbolic Testing of OpenCL code*. Haifa Verification Conference (HVC). LNCS. Vol. 7261, pages 203-218. 2011.
[32] A. Cunningham. *Nvidia's next Tegra chips will get a big boost from new GeForce GPUs*. March, 2013. http://arstechnica.com/gadgets/2013/03/nvidias-next-tegra-chips-will-get-a-big-boost-from-new-geforce-gpus/
[33] *What is Heterogeneous System Architecture (HSA)?* AMD. http://developer.amd.com/resources/heterogeneous-computing/what-is-heterogeneous-system-architecture-hsa/

Appendices

Appendix A Selection schemes and formulas

A1 Selection scheme of the Accident Probability Model [3]

Maximum take-off weight (MTOW)	Operation-type	Generation	Flight phase		Accident Rate
Light (MTOW < 5.700 kg)	*n.d.*	*n.d.*	Start	→	AR
			Landing	→	AR
Heavy (MTOW > 5.700 kg)	Business Jet	*n.d.*	Start	→	$AR^{overrun}$ $AR^{overshoot}$
			Landing	→	$AR^{overrun}$ $AR^{overshoot}$
	Cargo	*n.d.*	Start	→	$AR^{overrun}$ $AR^{overshoot}$
			Landing	→	$AR^{overrun}$ $AR^{overshoot}$
	Passenger	1	Start	→	$AR^{overrun}$ $AR^{overshoot}$
			Landing	→	$AR^{overrun}$ $AR^{overshoot}$
		2	Start	→	$AR^{overrun}$ $AR^{overshoot}$
			Landing	→	$AR^{overrun}$ $AR^{overshoot}$
		3	Start	→	$AR^{overrun}$ $AR^{overshoot}$
			Landing	→	$AR^{overrun}$ $AR^{overshoot}$

A2 Selection scheme of the Accident Location Model [3]

Maximum take-off weight (MTOW)	Flight phase	Accident type	Route dependent	Runway dependent
Light (MTOW < 5.700 kg)	Start	-	$f_{route}^{take-off\ shoot}(s,t)$	-
	Landing	-	$f_{route}^{landing\ shoot}(s,t)$	$f_{runway}^{landing\ run}(u,v)$
Heavy (MTOW > 5.700 kg)	Start	(over)shoot	$f_{route}^{take-off\ shoot}(s,t)$	$f_{runway}^{take-off\ shoot}(u,v)$
		(over)run	-	$f_{runway}^{take-off\ run}(u,v)$
	Landing	(over)shoot	$f_{route}^{landing\ shoot}(s,t)$	$f_{runway}^{landing\ shoot}(u,v)$
		(over)run	-	$f_{runway}^{landing\ run}(u,v)$

A3 Distribution functions used for the Accident Location Model [3]

Weibull: $f_W(x;\beta,\eta) = \frac{\beta}{\eta}\left(\frac{x}{\eta}\right)^{\beta-1} e^{-\left(\frac{x}{\eta}\right)^{\beta}}$

Generalised Laplace: $f_{GL}(x;a,b) = \frac{1}{2ab\Gamma(b)} e^{-\left|\frac{x}{a}\right|^{1/b}}$

Gauss: $f_{Gauss}(x,\sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x}{\sigma}\right)^2}$

LogNormal: $f_{LN}(x;\mu,\sigma) = \frac{1}{\sigma x\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{log(x)-\mu}{\sigma}\right)^2}$

Dirac: $\delta(x)$

For light movements:

$f_{route}^{landing\ shoot}(s,t) = f_W(s;\beta,\eta)\cdot\{p\cdot\delta(t) + (1-p)\cdot f_{GL}(t;a_0+a_1\cdot s,b)\}$

$f_{runway}^{landing\ run}(u,v) = f_{LN}(u;\mu,\sigma)\cdot\{p\cdot\delta(v) + (1-p)\cdot f_{GL}(v;a_0+a_1\cdot u,b)\}$

$f_{route}^{take-off\ shoot}(s,t) = f_W(s;\beta,\eta)\cdot\{p\cdot\delta(t) + (1-p)\cdot f_{GL}(t;a_0+a_1\cdot s,b)\}$

For heavy movements:

$f_{route}^{landing\ shoot}(s,t) = f_W(s;\beta,\eta)\cdot f_{Gauss}(t;\sigma_0+\sigma_1\cdot s)$

$f_{runway}^{landing\ shoot}(u,v) = f_W(u;\beta,\eta)\cdot f_{GL}(v;a_0+a_1\cdot u,b)$

$f_{runway}^{landing\ run}(u,v) = f_W(u;\beta,\eta)\cdot\{p\cdot f_{Gauss}(v;\sigma_0) + (1-p)\cdot f_{GL}(v;a_0+a_1\cdot u,b)\}$

$f_{route}^{take-off\ shoot}(s,t) = f_W(s;\beta,\eta)\cdot f_{Gauss}(t;\sigma_0+\sigma_1\cdot s)$

$f_{runway}^{take-off\ shoot}(u,v) = f_W(u;\beta,\eta)\cdot f_{GL}(v;a_0+a_1\cdot u,b)$

$f_{runway}^{take-off\ run}(u,v) = f_W(u;\beta,\eta)\cdot\{p\cdot f_{Gauss}(v;\sigma_0) + (1-p)\cdot f_{GL}(v;a_0+a_1\cdot u,b)\}$