Technische Universität München

Faculty of Civil, Geo and Environmental Engineering

Chair of Computational Modeling and Simulation
Prof. Dr.-Ing. André Borrmann

Spatial BIM Queries: A Comparison between CPU and GPU based Approaches

Robert Schweizer

Bachelor's Thesis
for the B.Sc. in Engineering Science

Abstract

From today's perspective, the future of computation lies in parallelization. This is a central design focus of modern GPUs, whose enormous calculation capabilities are now available for general purpose programming.

This thesis examines the portability of spatial queries to NVIDIA's popular GPGPU CUDA platform. The two algorithms analyzed are the query on the R-tree spatial indexing structure and an intersection test between triangle meshes. Both of them offer potential for the high-level parallelization necessary for efficient GPU programs.

The CUDA implementation showed no significant advantage over the single-threaded CPU one for the R-tree query, both for single as well as multiple parallel queries. The mesh intersection test on the other hand performed better using CUDA, at least when the tested objects were preselected by their bounding boxes using an R-tree.

The entire CPU code of the examined program was written in C#. A pure C implementation could yield different results. The analysis could also be repeated for a multi-threaded CPU program and on newer hardware.

Author:	Robert Schweizer Philipp-Foltz-Straße 31 D-81737 München
Student ID:	03630077
Supervisors:	Prof. Dr.-Ing. André Borrmann Dipl.-Ing. (FH) Simon Daum
Date of Issue:	October 15, 2014
Date of Presentation:	January 27, 2015

Contents

List of Abbreviations **7**

1 Introduction and Motivation **9**

2 Spatial BIM Queries **13**
- 2.1 History of BIM . . . 13
- 2.2 BIM Queries . . . 14
- 2.3 Spatial Operators in QL4BIM . . . 15
  - 2.3.1 Topological Predicates . . . 15
  - 2.3.2 Intersections between Triangle Meshes . . . 17
  - 2.3.3 Triangle Pair Meshes . . . 17
- 2.4 Queries Parallelized in this Thesis . . . 19
  - 2.4.1 Find All Meshes Intersecting One Search Mesh . . . 19
  - 2.4.2 Find All Intersection Pairs of the Model . . . 19

3 Parallel Programming **21**
- 3.1 History of GPU Programming . . . 21
  - 3.1.1 Development of Dedicated GPUs . . . 21
  - 3.1.2 Early Experiments with GPGPU . . . 23
- 3.2 Comparison of GPU and CPU Programming . . . 23
- 3.3 NVIDIA CUDA . . . 25
  - 3.3.1 Hardware Architecture . . . 26
  - 3.3.2 Thread Hierarchy . . . 27
  - 3.3.3 Memory Hierarchy . . . 28
  - 3.3.4 Application Structure . . . 29
    - 3.3.4.1 Host Functions . . . 29
    - 3.3.4.2 Device Functions . . . 30
  - 3.3.5 Maximization of Device Utilization . . . 31

4 Implementation **35**
- 4.1 R-Tree . . . 35
  - 4.1.1 Data Structure and Parameters . . . 35

4.1.2 R-Tree Creation 36
4.1.2.1 Implementation Used in QL4BIM System 36
4.1.2.2 Possible Optimization and Parallelization 37
4.1.3 Sequential R-Tree Query 38
4.1.4 Parallel R-Tree Query 38
4.1.4.1 Input Data Representation on the GPU 39
4.1.4.2 Parallelization Approaches 40
4.1.4.3 Overflow Treatment 40
4.2 Mesh-Intersection Test 41
4.2.1 Sequential Implementation 42
4.2.2 Parallel Implementation 42
4.2.2.1 Levels of Parallelism 42
4.2.2.2 Mesh Partitioning 43
4.2.2.3 Test on a Reduced Set of Meshes 44
4.2.2.4 Adjustments for Finding All Intersection Pairs . . . 45
4.2.2.5 Treatment of Kernel Timeouts on Windows 45

5 Performance Analysis **49**
5.1 Remarks on Platform Comparison 49
5.2 Test Description 49
5.2.1 Test Setup 49
5.2.2 Parameter Settings 50
5.2.3 Model Selection 50
5.3 Results 51
5.3.1 Model Properties 51
5.3.2 Data Structure Creation 52
5.3.3 Intersection With Search Mesh 54
5.3.4 All Mesh Intersections Query 57
5.3.5 Observed Trends 59

6 Conclusion and Outlook **63**

A Remarks on ManagedCUDA **65**

B Digital Files **67**

Bibliography **69**

List of Abbreviations

ALU	Arithmetic Logic Unit
BIM	Building Information Modeling
CAD	Computer-Aided Design
CIL	Common Intermediate Language
CLI	Common Language Infrastructure
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DP	Double Precision
DRAM	Dynamic Random-Access Memory
FLOPS	Floating-Point Operations Per Second
FPU	Floating-Point Unit
GPU	Graphics Processing Unit
IDE	Integrated Development Environment
IFC	Industry Foundation Classes
ILP	Instruction-Level Parallelism
JIT	Just-In-Time
KIT	Karlsruher Institut für Technologie
LINQ	Language Integrated Query
MEP	Mechanical, Electrical and Plumbing
NVCC	NVIDIA CUDA Compiler

OpenCL	Open Computing Language
PC	Personal Computer
PCIe	Peripheral Component Interconnect Express
PTX	Parallel Thread Execution
QL4BIM	Query Language for Building Information Models
RAM	Random-Access Memory
SDK	Software Development Kit
SFU	Special Function Unit
SIMD	Single-Instruction, Multiple-Data
SIMT	Single-Instruction, Multiple-Thread
SM	Streaming Multiprocessor
SP	Single Precision
SSE	Streaming SIMD Extensions
VES	Virtual Execution System

Chapter 1

Introduction and Motivation

The number of integrated components that microelectronic processors contain per area has been steadily increasing each year and with every development cycle that has passed since the 1950s. The exponential growth was first observed in Moore (1965) and is thus commonly referred to as "Moore's law", and has continued until today (see the green line in figure 1.1).

However, this does not directly translate into better performance, as other factors are also influential. Some of these have exponentially improved for many years in accordance with "Dennard scaling" (Dennard et al., 1974; McMenamin, 2013). The most important part of this theory is the processor's clock speed. When it increases, every calculation runs faster by roughly the same factor. However, this has not improved substantially since the mid 2000s (dark blue line in figure 1.1). This is because the operating voltage of the chips – an important factor in their power leakage per area – could not be reduced as significantly as in previous decades. Back then, the decrease in heat generation due to the voltage optimization had enabled the clock speed increases. Now, this is impossible without producing higher temperatures than the materials used can withstand.

However, as the minimum manufacturing scale continues to steadily decrease, ever more components can be built into the same chip area, even though their speed does not change. These can be used either to improve single threaded execution with complicated methods facilitating Instruction-Level Parallelism (ILP) – what Central Processing Unit (CPU) designers focus on – or to process a growing array of several threads simultaneously; this is the main target of Graphics Processing Unit (GPU) development (Wilt, 2013, p. 3 ff.). CPUs also now offer multiple cores for the concurrent execution of several threads, but their strength still lies in their optimized single thread performance.

According to Preshing (2012), this has been increasing by only approximately 21 % per year since the mid 2000s. GPU performance on the other hand has largely followed the exponential growth of electronic components per area. This can be seen

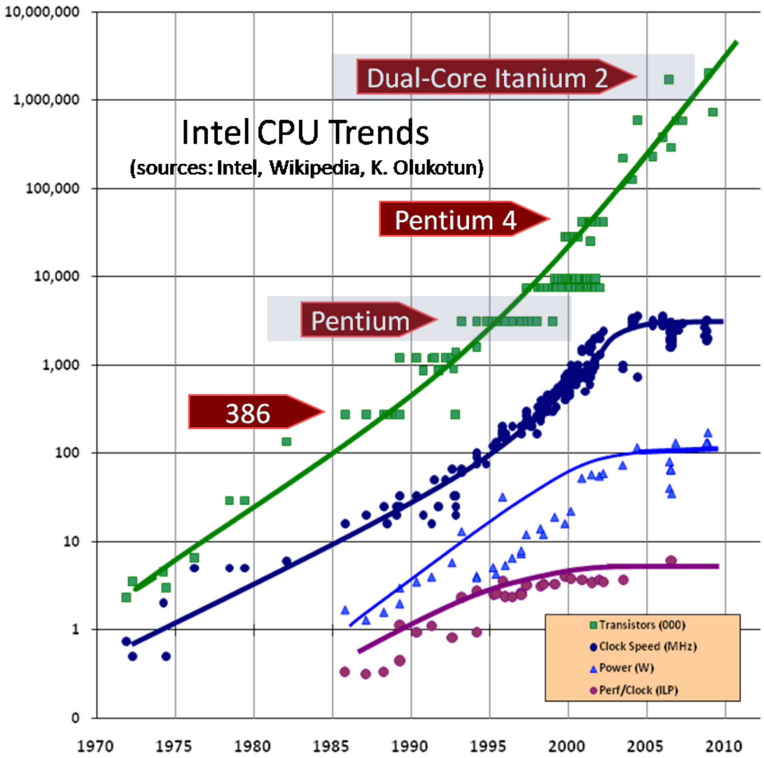


Figure 1.1: Trend overview of Intel processors regarding the component count ("Moore's law") as well as clock speed and heat production, which are connected by the so-called "Dennard scaling". The single-core performance per clock cycle is also specified by the available degree of ILP. From Sutter (2004).

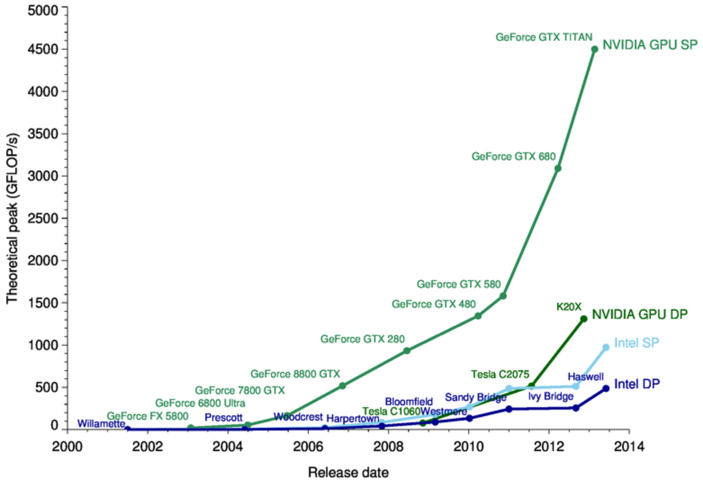


Figure 1.2: Performance comparison between Intel CPUs and NVIDIA GPUs with respect to Single Precision (SP) as well as Double Precision (DP) Floating-Point Operations Per Second (FLOPS) for each complete device. Most improvements in CPU throughput since the late 2000s can be attributed to the increasing number of cores which they feature. From Galloy (2013) under CC BY-NC-SA 2.5.

in figure 1.2, where the only visible jumps of CPU performance for the last years are owed to their increase in core numbers. But even with this, the raw computational power of GPUs is increasing much faster than that of CPUs.

However, the problem with GPUs is that their immense capabilities are difficult to harness for many important applications. Much research has been devoted to porting and optimizing a multitude of algorithms for GPU programming platforms such as NVIDIA's Compute Unified Device Architecture (CUDA).

In this thesis, this porting and optimizing was done for the topological analysis of 3D design files created by Building Information Modeling (BIM). To this end, algorithms were implemented for the querying of a spatial indexing structure called the R-tree, as well as intersection testing between triangle meshes. Both of these promise to be portable to CUDA, because their structure allows for the high-level parallelism required for an efficient GPU application.

The topological analysis of digital building models is a very important task, especially for larger models. However, an efficient implementation of the spatial algorithms with a favourable scaling is essential for larger models in particular. For the intersection query examined in this thesis – which finds all objects overlapping a search object – the R-tree is used as a preselector. This way, the number of objects for which the later mesh intersection test has to be done, can be decreased. This may facilitate a better than linear scaling of this query's runtime with increasing model sizes.

Chapter 2

Spatial BIM Queries

This chapter will give a short overview of BIM and describe the implementation of spatial queries thereof. A focus is put on topological queries and on how they can benefit from the algorithms examined in this thesis.

2.1 History of BIM

BIM is a widely used planning method in the construction industry of today (Autodesk, 2002). It interconnects models with databases supplying many kinds of additional information required for the construction process, e.g. lists of required parts or assembly specifications. Changes made to one of these databases are coordinated across all the others, so that the consistency of the model is always guaranteed.

The history of BIM starts with the adoption of PC-based Computer-Aided Design (CAD) in the building industry in the early 1980s. As soon as digital geometry models started to replace manual drafts on drawing boards, planners also began to specify additional data about objects surpassing their geometric properties. An example of this is information about the construction process or meaning of models, which can be derived from the different layers contained in the files.

About ten years later, in the beginning of the 1990s, object-oriented CAD was introduced. Here, the geometric model is only a part of the stored data object, which further includes non-graphical information like its type, ID, and logical relationships with other objects. This coincided with the wide adoption of 3D models in the building industry. Creating designs in three dimensions now offered strong advantages such as the possible automation of section view drawings or the generation of construction schedules.

However, object-oriented CAD is still just an extension of the graphics-based CAD, which was not initially designed to store further information about the building. It is thus called *building graphic modeling*, in contrast to *building information modeling*, which was introduced in the early 2000s as a response to these shortcomings. It uses object-based parametric modeling, which has been developed for mechanical engineering since the 1980s (Eastman et al., 2011, p. 40 f.). Here, geometric models can reflect dependencies between the dimensions of different components or other interconnected properties by calculating them from a list of parameters. In structural engineering, this additional information can include spacing rules between objects such as the minimum distance between windows or design thresholds such as the maximum slope of an excavation. Thus an even stronger focus is put on the inter-object relationships in order to capture the design intent of the engineer.

This vastly improves the interface between the mind of the designer and the stored model. Ideas about this kind of object relationships have always been crucial in every kind of planning process, but now they can be appropriately modeled in the computer. This also facilitates changes to the model, as the software can automatically determine and apply the additional changes required for related objects.

BIM uses databases for storage, and is therefore a perfect solution for projects relying on close collaboration, where the model has to be stored away from the designer's computer and possible parallel accesses have to be managed in order to keep the model consistent. This requires server structures, called *model management servers* or *product model servers* and are considered an IT infrastructure with growing importance (Borrmann et al., 2009c).

To facilitate the collaboration between different BIM softwares, the Industry Foundation Classes (IFC) have been developed since the 2000s as an open file format. They use entities which are organized in an inheritance hierarchy. Each of these describes either a part of the construction process, e.g. tasks, material or physical objects, called *IfcProducts*, or relationships and properties thereof. Thus all the dependencies and interconnections which appear in complex BIM models can be stored appropriately.

2.2 BIM Queries

With increasing project sizes, the extraction of partial models from the overall design file becomes more and more important for asynchronous collaboration, analysis and further processing. Not many of the designers working together require the complete model, and having a smaller subset of objects can facilitate their work considerably. The interface for retrieving such partial models is usually implemented in the form of a query language.

Various implementations exist of a query language specifically tailored to the needs of examining IFC models (Daum et al., 2014). Most of them provide methods commonly used for database accesses such as *Select*, *Update* and *Delete*. *Select* usually provides the functionality of filtering the model to obtain partial models. Furthermore, normally using the *Where* keyword, the data sets affected by these queries can be restricted. This means that only a subset fulfilling a user-defined condition is subject to the operation being executed.

Although these query languages provide capabilities for the semantic processing of IFC models, they have one restriction: they only take into account non-geometric information such as objects' IDs and types, and possibly interconnect these using nested conditions. They do not, however, use the geometry of the models to provide support for spatial conditions. This is only offered by the scientific QL4BIM proposed in Daum et al. (2014).

2.3 Spatial Operators in QL4BIM

The Query Language for Building Information Models (QL4BIM) is a BIM query language which stands out by supporting filter expressions with spatial operators. These consist of:

Metric Operators describing distance relationships between objects. Possible conditions are e.g. *Distance*, *CloserThan* and *FartherThan*. Specific implementations thereof are described by Borrmann et al. (2009a).

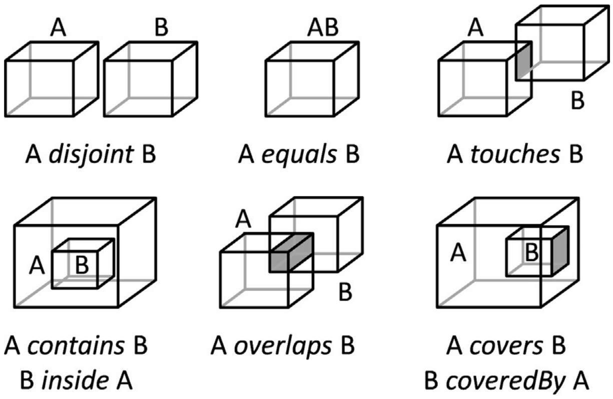
Directional Operators constraining the directional relationships between objects. These are usually expressed with respect to coordinate axes in conditions such as *Above*, *Below* or *NorthOf*. See Borrmann et al. (2009b) for details.

Topological Operators comprising primitive conditions like *Touch*, *Within* and *Contains*. This is what the algorithms developed in this thesis are targeted for.

Daum et al. (2014) provides a sequential reference implementation of the topological operators called QL4BIM System, which will be used as a point of reference here.

2.3.1 Topological Predicates

Using the classification proposed by Egenhofer et al. (1991), the entire range of possible topological relationships between two objects in 3D-space can be reduced to the eight predicates in figure 2.1. To determine the truth of these predicates for a pair of *IfcProducts*, their physical form has to be stored digitally. The IFC support many different types of such geometry representations. QL4BIM System uses a boundary representation called *IfcTriangulatedFaceSet*, which approximates the geometry of each *IfcProduct* with a closed mesh comprised only of triangles. To generate this for every *IfcProduct* when opening an IFC file, the xBIM Toolkit is employed.



(a) Visualization of the possible constellations for 3D objects. Figure from Daum et al. (2014) under CC BY-NC-ND 3.0.

$\partial \cap \partial$	$^\circ \cap ^\circ$	$\partial \cap ^\circ$	$^\circ \cap \partial$	
$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	A and B are disjoint
$\neg\varnothing$	$\neg\varnothing$	$\varnothing$	$\varnothing$	A equals B
$\neg\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	A and B touch
$\varnothing$	$\neg\varnothing$	$\varnothing$	$\neg\varnothing$	A contains B / B is inside A
$\varnothing$	$\neg\varnothing$	$\neg\varnothing$	$\varnothing$	A is inside B / B contains A
$\neg\varnothing$	$\neg\varnothing$	$\neg\varnothing$	$\neg\varnothing$	A and B overlap
$\neg\varnothing$	$\neg\varnothing$	$\varnothing$	$\neg\varnothing$	A covers B / B is covered by A
$\neg\varnothing$	$\neg\varnothing$	$\neg\varnothing$	$\varnothing$	A is covered by B / B covers A

(b) Combinations of boundary-boundary, inside-inside, boundary-inside and inside-boundary intersections between objects A and B corresponding with the predicates. The inside region here corresponds to the interior point set, and can thus not be modeled exactly using an inside mesh as described in section 2.3.3. Table from Egenhofer et al. (1991).

Figure 2.1: The eight possible topological predicates defining the relationships between 3D bodies using the classification from Egenhofer et al. (1991).

2.3.2 Intersections between Triangle Meshes

One of the most important tasks in the topological relationship analysis of two objects is the intersection test. It determines whether they intersect, touch or do not meet at all. Using the meshes provided by the *IfcTriangulatedFaceSets*, this breaks down to individual tests on the triangles of the two meshes: If any triangle of mesh A intersects any triangle from mesh B, the meshes have to be overlapping. The same holds for the touching relationship, although all possible triangle combinations still have to be checked in order to exclude a possible intersection. If no triangles from mesh A meet any triangles of mesh B, the meshes do not meet.

2.3.3 Triangle Pair Meshes

For distinguishing between intersecting and touching constellations of boundary representation models, different approaches exist. The one described above uses an intersection test which, for each triangle pair tested, distinguishes between the three possible results: intersecting, touching and disjoint. However, this requires the data to provide a high degree of floating-point precision, which normally can not be expected.

Many different kinds of imprecisions are introduced during the life span of a full fledged IFC model. These include truncation errors in floating-point numbers, e.g. when defining the position of the boundary relative to the *IfcProduct* and further positioning this with respect to the overall model. It also has to be taken into consideration here that in a general case, the *IfcTriangulatedFaceSet* created from the stored boundary definition can only be considered an approximation thereof. For example, when the meshes of two spheres meet in their exact model, e.g. as splines, their triangle representations at the meeting point are not defined clearly enough when using common tesselation techniques. Thus the result of the described elaborate triangle testing between their *IfcTriangulatedFaceSet* representations can not be relied upon either.

A possible solution to this shortcoming is an epsilon test in the triangle test implementation. This does not require the floating-point approximations of the triangles to equal each other exactly, but only within a certain user-defined tolerance. Thus the expected imprecisions within the model can be accounted for.

Apart from this, a user-defined parameter for what can still be counted as touching and what has to be considered disjoint might have further uses: it offers the possibility of setting a maximum distance from the searching mesh in which objects are still returned as intersecting. A possible query using this could be *FindObjectsWithinXMeters*.

However, for this work a different approach is used: instead of using only one triangle mesh, a *TrianglePairMesh* was generated from the *IfcTriangulatedFaceSet*.

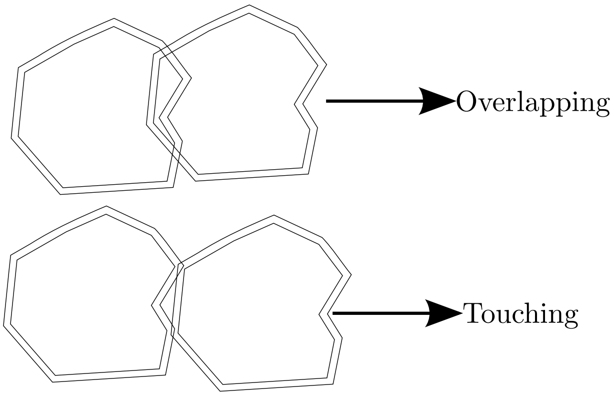


Figure 2.2: Example constellations of *TrianglePairMeshes* adapted to 2D which clearly define the overlapping and touching relationships from figure 2.1.

This basically corresponds to creating separate inside and outside triangle meshes of each *IfcProduct*'s surface, with a small offset in their respective directions from the single *IfcTriangulatedFaceSet* representation. The offset here is comparable to the epsilon constant used in the triangle test in that it is a user-defined measure for the acceptable imprecisions in the model.

The objects are defined as fully intersecting only if both meshes of object A intersect both meshes of object B. Although this way four times as many mesh intersection tests need to be run, there are several advantages:

No Mesh-Level Touching/Intersecting Determination

The complexity of the intersection test for the individual triangles decreases, because it does not have to discern between touching and intersecting relationships. Any meeting triangles are counted as intersecting, and testing can stop for that mesh combination as soon as the first one is found.

No Epsilon Test Necessary

The implementation of a correct epsilon test therein is not necessary. This is usually not trivial, which is also a problem with the one employed for this work: it uses coordinate transformations, where the different scaling of the epsilon "sphere" along the dimensions would have to be calculated appropriately.

Additional Information about Spatial Predicate

The combinations of intersections between the different meshes can provide more information than the single boundary test. For example, for a touching relationship between objects A and B they define whether the topological predicate A touches B, A covers B or B covers A is fulfilled (see the examples in figure 2.2).

This is possible because the inner meshes can often be considered equivalent to the inner region of the respective object. With that, table 2.1b can be used

> for the specification of the predicate. This simplification can be applied to all cases except the ones that do not yield an intersection of the meshes at all, i.e. the disjoint and contains/inside relationships. These have to be further inspected, e.g. using a ray test as proposed by Daum et al. (2013).

2.4 Queries Parallelized in this Thesis

This work covers only two topological queries, but many of the described concepts should be reusable for different spatial query implementations. The discussed algorithms for spatial look-up and intersection testing are crucial parts of many spatial analysis programs.

2.4.1 Find All Meshes Intersecting One Search Mesh

The first investigated query is the *IntersectsWith* operation. This basically retrieves every object in the IFC model which truly intersects with, i.e. does not merely touch, a search object. Potential uses of the retrieved subsets lie in error detection and compatibility checks for newly inserted objects. Unwanted or physically impossible intersections can easily be identified this way. This is often used when a complete building model has to accommodate changes for additional features, e.g when Mechanical, Electrical and Plumbing (MEP) services are implemented.

The brute-force approach to this problem is running a mesh intersection test (further described in section 4.2) between the search mesh and every single mesh stored in the model. However, to achieve better than linear runtime scaling with respect to the model size, preselection methods can also be considered. In this thesis, an R-tree is queried (see section 4.1) to obtain a reduced set of meshes on which the intersection test is run. It uses the axis aligned bounding box of each mesh to store it efficiently. Thus the intersection test only has to be run on the meshes whose bounding boxes intersect that of the search mesh.

2.4.2 Find All Intersection Pairs of the Model

Even though it is only an extension of the single *IntersectsWith* query, finding all mesh intersection pairs can be important in model consistency and error checking. Implementing it separately promises huge performance benefits, especially when considering the kind of natural parallelism it offers; sequentially running the *IntersectsWith* query for every single mesh in the model would be a waste of optimization potential here.

This query is a good example of a computationally complex, but also parallelizable procedure. As such, it is even better suited for the GPU than the single mesh intersection query, and can thus be used as a measure of how the GPU performs under optimal circumstances.

Chapter 3

Parallel Programming

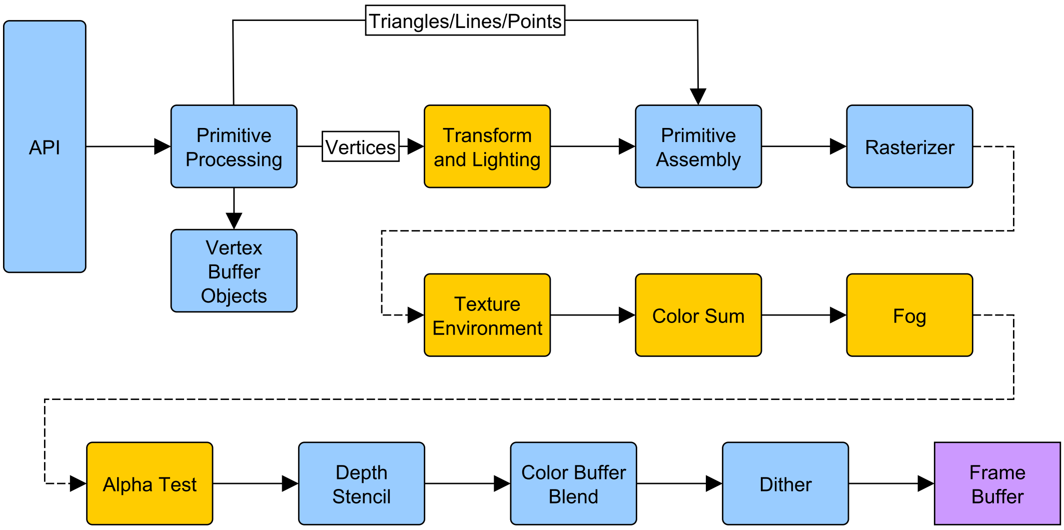
The core of this thesis was the implementation of spatial queries on the massively parallel computation platform offered by recent GPUs using the NVIDIA CUDA Software Development Kit (SDK). The following section will briefly introduce GPUs and their programming in general, and then give a more specific description of the CUDA programming model in particular.

3.1 History of GPU Programming

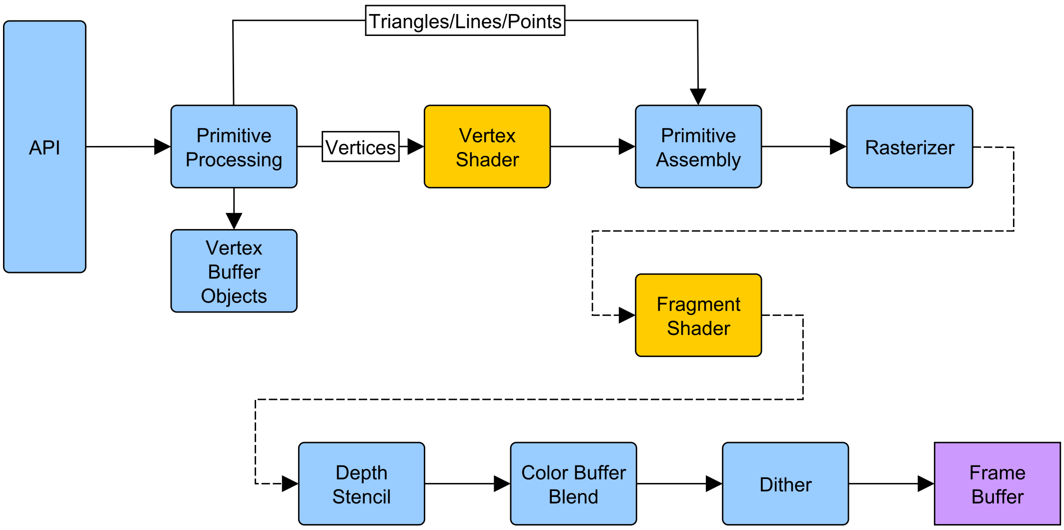
3.1.1 Development of Dedicated GPUs

The development of the modern GPU began in the 1970s, when the first display controllers were introduced with the sole purpose of preparing the data stream produced by the main processor for being displayed on the screen. The main function of these devices was creating a video output stream containing luminance and color information for the respective display, which also included ensuring horizontal and vertical sync (Singer, 2013).

Until the late 1990s, GPUs were programmed using the so-called *fixed-function-pipeline* (see figure 3.1a). This used predefined functions for each step in the rendering process and only allowed the programmer to change some of their parameters (Behrmann, 2007). However, the clearly defined structure of the computations also facilitated extremely efficient hardware design, as the different computational units could be built to fit the needs of their specific tasks very tightly (Owens et al., 2008). This simple design permits two kinds of parallelism: First, task parallelism, which means that the different stages in the rendering pipeline can be working on different chunks of data simultaneously if that data has already passed through the previous steps. Secondly, data parallelism, i.e. the division of data on several processors of the same pipeline stage. This is possible because the different vertices/fragments of



(a) The fixed-function-pipeline with the separated shading stages marked in yellow. The pipeline structure is completely predetermined here and can only be influenced by parameters.



(b) The new programmable pipeline with the flexible vertex and fragment shading stages marked in yellow. These offer the developer a wider range of options by allowing for freely customizable shader programs.

Figure 3.1: Comparison of the two graphics pipeline architectures. Based on Khronos Group (*OpenGL ES 2.X - The Standard for Embedded Accelerated 3D Graphics*).

a mesh being rendered can pass the calculation independently and thus at the same time.

Around the turn of the millennium, the graphics pipeline started to become more flexible: the programmer was now able to write his own customized programs, called *vertex shaders* and *fragment shaders*, which were then run as their respective stages in the pipeline (see figure 3.1b, the replaced stages are marked yellow in figure 3.1a). In the beginning, the functionality that the hardware offered for these programs was very limited, but the amount of available memory, the quality of the instruction sets and the control-flow flexibility increased steadily. At this point, fragment and vertex shaders were still run on different hardware units, as their instruction sets were quite different. However, this carries the disadvantage of load imbalances, which appear when one of the shader programs is more computationally complex than the other. As a result, one of the specialized processors can not be used at full capacity, because the throughput of the pipeline is limited by its slowest component.

The solution to this problem was the *unified shader* architecture, which introduced the concept of *stream processors*. These are able to execute any kind of shader program, so that the load balancing between the different rendering stages is no longer a problem. They still employ both kinds of parallelism mentioned above, and can thus offer the high amount of throughput required for graphics processing. The GPU was now sufficiently programmable for a wide range of developers to use it for general computations aside from pure rendering tasks. This gave rise to new GPGPU abstractions like NVIDIA CUDA and the Open Computing Language (OpenCL) in the following years, which made it possible to program the GPU relatively easily using high-level languages such as C.

3.1.2 Early Experiments with GPGPU

Facilitated by the advent of programmable shaders, one of the first experiments with using the computational power of GPUs for non-graphical computations was published in Larsen et al. (2001). However, this early implementation of matrix multiplication on the GPU did not yield performance superior to that of CPUs. One of the problems at the time was that native support for single-precision floating-point operations on GPUs was not introduced until 2003. This enabled the first GPGPU programs, one of which was LU factorization in Galoppo et al. (2005), to outperform comparable CPU implementations (Du et al., 2012).

3.2 Comparison of GPU and CPU Programming

Implementing an algorithm for the GPU is vastly different from doing the same for a CPU. The core reason for this distinction are the two opposing optimization maxims under which the hardware is designed:

Throughput Optimization

GPUs are typically designed to handle a large amount of data, all of which has to be processed as fast as possible. The work is distributed on an array of processors, each of which handles its assigned tasks sequentially. However, the processors work in parallel and thus increase throughput; alone they are not very fast compared to CPUs, but combined they can usually handle more data and perform more arithmetic operations in the same time.

Latency Optimization

CPUs on the other hand are naturally built to minimize the reaction time as much as possible. A classic example for this optimization process is the calculation of the mouse cursor's position. Despite the involvement of a tiny amount of data, it has to be calculated in as few clock cycles as possible to maximize the responsiveness of the system. Due to the nature of the required work, its distribution on multiple processors is neither possible nor sensible. Accelerating the sequential execution of the routine is the only feasible optimization approach here.

Although the raw computing power of CPUs is typically lower than that of GPUs, they will still be an integral part of most computers in the future. This is because harnessing the GPU's superior capabilities requires a completely different approach to software design. Furthermore, the development effort is usually higher, while the benefit to be gained depends strongly on the problem type. Many kinds of algorithm can not be parallelized at all, e.g. if they are intrinsically iterative procedures like series of numbers (e.g. the Fibonacci sequence) and common root finding techniques (e.g. Newton's method).

Additionally, due to the long-lasting difficulties in improving single threaded computing performance apart from clock frequency increases, and especially because of the factual standstill of the achievable maximum frequency due to currently insurmountable physical barriers since the early 2000s, CPU vendors have been introducing two levels of parallelism since the late 1990s (Wilt, 2013, p. 439):

SIMD Instructions

The first of these were introduced in the late 1990s on Intel's x86 architecture under the name Streaming SIMD Extensions (SSE). They offer instructions for simultaneously performing vector operations like additions on 32-bit float arrays.

Multithreading

Widely available since 2006, this has been the main source of computing power increase on CPUs from then on. Nowadays, twelve or more cores can be found in high-end CPUs, each of which can work separately from the others.

The main problem of these improved CPU capabilities is that many programs do not make use of them. It requires additional time and effort – and thus development cost – to exploit any of these different levels of parallelism. Minimizing the execution time of simple single-threaded programs will therefore remain one of the most important efforts of CPU vendors. A currently popular way of doing this is increasing the ILP that a processor offers, which is either exploited automatically at runtime, or through compiler optimizations.

The critical requirement for an algorithm to be implemented on GPUs is that it involves numerous parallel operations. If it does, performance improvements can be expected from employing a GPU implementation instead of a CPU one. Classic examples of this are of course the routines of the GPU's original application field of graphics processing and rendering; here each pixel or polygon can be considered separately. However, with the increasing popularity of GPGPU, many different applications where GPUs offer superior performance have become known, e.g. linear algebra.

3.3 NVIDIA CUDA

The CUDA SDK by NVIDIA is one of the two major GPGPU implementations used today (NVIDIA Corporation, 2014b). It was announced in 2006 as a way of making the numerous possibilities opened up by the newly introduced stream processors available to a larger community of general purpose programmers. Since then, a research community has evolved which tries to optimize every kind of algorithm that offers some degree of parallelism for CUDA. For common applications, NVIDIA offers highly optimized libraries, e.g. for random number generation (cuRAND) or fast Fourier transform (cuFFT).

The CUDA SDK comes with an IDE called Nsight, which allows developers to use C and C++ as programming languages for their CUDA programs. It also includes a debugger for host and device code, as well as a profiler, which indicates potential performance bottlenecks in the program and helps to determine the optimum configuration parameters. However, many other languages like Fortran, MATLAB, Java or C# offer interfaces and wrappers for CUDA. In this thesis, managedCUDA (Kunz, 2014) was used as a way of calling CUDA programs written in C from a managed C# environment.

C# uses the Common Language Infrastructure (CLI), which translates the code to Common Intermediate Language (CIL) at compile time (ECMA, 2012). The resulting code, comparable to assembly for CPUs, is then executed in the Virtual Execution System (VES) on the machine running the code. The VES further processes the CIL code using a Just-In-Time (JIT) compiler and runs it with a high level of abstraction from the hardware. This includes "managing" the data, meaning that memory is automatically allocated and later deallocated through garbage collection.

As it was developed by NVIDIA, CUDA is only available on high-end NVIDIA GPUs. A more widely available standard is OpenCL, which is implemented by all major GPU vendors, and thus available on most current PCs. However, several studies (Karimi et al., 2010; Fang et al., 2011; Du et al., 2012) imply that CUDA tends to perform slightly better. The NSight IDE is also superior to the IDEs offered for OpenCL in terms of analyzing and profiling capabilities.

CUDA offers a high degree of downward compatibility. Although new features are introduced with every generation of devices and every version of the SDK, code written and optimized for older hardware can usually run on newer machines as well. The standard output of the NVIDIA CUDA Compiler (NVCC) is a Parallel Thread Execution (PTX) file, which is comparable to assembly code. Only once the kernel is executed, it is translated to actual binary code by the running graphics driver, similar to the JIT compilation done within the CLI for C#. With this procedure, the code finally run on the device can be optimized specifically for that device's capabilities.

To denote the supported CUDA functionality of different generations of their hardware, NVIDIA introduced the *compute capability* versioning scheme. The first digit marks the hardware generation, with versions 1.x for Tesla, 2.x for Fermi, 3.x for Kepler and 5.x for Maxwell microarchitecture devices. The second digit indicates minor differences within the same architecture.

3.3.1 Hardware Architecture

NVIDIA GPUs which support CUDA use the *unified shader* architecture described above. This means that their stream processors have general-purpose computing capabilities, which can be used for all possible tasks in the rendering pipeline. CUDA offers a way of programming these processing units, then called CUDA cores, in higher-level languages like C. Each CUDA core has an Arithmetic Logic Unit (ALU) for integer operations and a Floating-Point Unit (FPU) for floating-point operations (NVIDIA Corporation, 2009). These are grouped together in an array of Streaming Multiprocessors (SMs), each of which contains, in addition to its CUDA cores (Wilt, 2013, p. 46):

- Special Function Units (SFUs) for the fast and precise (22 bits) computation of the transcendental functions sine, cosine, logarithm, exponential, reciprocal and reciprocal square root. This is better capability than that offered by CPUs, which yield a native precision of only twelve bits and implement fewer functions in the hardware (Wilt, 2013, p. 244, p. 251 f.).

- Warp schedulers with one instruction dispatch unit each, which select the warps to be executed, and issue their instructions to the corresponding logic units in parallel.
- Shared memory accessible to multiple threads.
- L1 and L2 cache.

The Quadro 600 GPU used for this work is a 2nd-generation CUDA device based on the Fermi architecture. It features 2 SMs, each of which has 48 CUDA cores, four SFUs, two warp schedulers, 64 KB of on-chip memory (usable as L1 cache and shared memory) and 768 KB of separate L2 cache. The on-chip memory can be configured at runtime to offer either 48 KB of shared memory with 16 KB of L1 cache or 16 KB of shared memory with 48 KB of L1 cache.

3.3.2 Thread Hierarchy

This hardware architecture is abstracted for the programmer as a three level hierarchy: The top level is the grid, made up of one or multiple thread-blocks. Each of these consists of one or several warps, each of which is again made up of exactly 32 threads. This structure is shown in figure 3.2; the warps are not shown there because they are usually not directly exposed to the programmer. Each thread corresponds to one execution of the so-called kernels, which are little programs written by the developer and are very similar in nature to the shader programs the GPU was originally designed for.

These kernels can be executed from the CPU (host) in a similar way to normal functions (see section 3.3.4.1). However, the size of each block as well as the number of blocks always have to be passed as arguments to the kernel call. Each block is then assigned to one of the SMs on the GPU (device) and further split into warps of 32 threads. The distribution of the blocks onto SMs is done automatically, so that a CUDA application developed for one device can also run on different devices with different numbers of SMs, and also benefit from the increasing number of SMs found in newer generations of GPUs. There are no guarantees made concerning the execution order of the threads: They can only be forced to wait for all other threads of the same block at certain points in the kernel code using the *__syncthreads()* command.

Following NVIDIA's Single-Instruction, Multiple-Thread (SIMT) hardware design, which is a realization of the Single-Instruction, Multiple-Data (SIMD) architecture commonly used by GPUs, every thread of a warp executes the same instruction at the same time. This also means that if threads inside a warp take different paths through conditional code branches, only one of those branches can be executed at the same time. Because of that, the threads taking one branch have to wait while

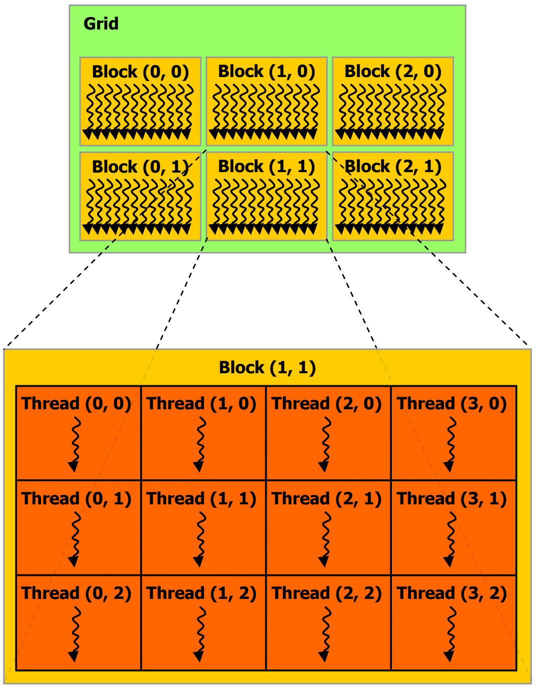


Figure 3.2: Sketch of the hierarchy between threads, blocks and the grid in CUDA. From Wikimedia Commons (2010a) under CC BY 3.0.

the other threads execute the code of their branch, and vice versa. For performance optimizations, this is a significant detail, making the minimization of code branching inside of warps an important point of interest.

3.3.3 Memory Hierarchy

The memory available to CUDA kernels is also arranged in a hierarchy as follows (see also figure 3.3):

Per-Thread Local Memory

Local memory is private to each thread, and typically stored in the thread's registers. Registers are tiny memory units (32 bits each on CUDA devices) primarily used to store the instructions of a program on a processor. As this type of memory is built on the respective processor, it is instantly available when read, and the fastest kind of memory on the GPU. Every variable declared inside a CUDA kernel without special qualifiers is stored in local memory.

However, if the number of registers available to each thread (63 on the Quadro 600) is insufficient to store the instructions as well as the local variables, the spillover is stored in global memory (register spilling). Furthermore, arrays whose lengths can not be determined at compile time are also stored in global memory. As the access time for data in global memory is more than two orders of magnitude higher, this can become a significant performance bottleneck.

Per-Block Shared Memory

Shared memory is available to every thread of a block, and can thus be used for inter-thread communication. It is stored in the SM's on-chip memory, which results in an access latency of two clock cycles (Luo et al., 2010).

Global Memory

Global memory can be accessed by every thread running on the GPU as well as by the host CPU via the connecting PCIe bus. Due to its storage on the device's off-chip DRAM, the access latency lies between 400 and 800 clock cycles for compute capabilities 1.x and 2.x and between 200 and 400 for newer hardware. This makes it the slowest kind of memory available on the GPU. As described above in Per-Thread Local Memory, parts of that can also stored here, which can lead to a massive drop in performance.

An important detail to keep in mind when accessing global memory is that all transactions here are performed for either 32-, 64- or 128 byte words. This means that if all 32 threads of a warp access one bit of memory with a 128 byte offset from each other, 32 separate costly global memory transactions are initiated. However, if the offset is only 4 bytes, one 128-byte memory access is sufficient. This increases memory throughput to 32 times that achieved with the spaced memory access. The strategic issuing of accesses such that consecutive threads access closely spaced consecutive memory regions is called *memory coalescing.*

3.3.4 Application Structure

Programs developed using CUDA's C++ environment are always split into two parts: On the one side the host code is present. This works the same way as normal C++, apart from some small syntax extensions introduced by CUDA. It contains the starting point of program execution, either in the main function if there is one, or in a function called by another C++ or CUDA file. On the other side are the device functions, which are kernels running exclusively on the GPU. These are invoked from CUDA host functions, using a special syntax.

3.3.4.1 Host Functions

This is the default function type in CUDA source files. However, the *__host__* keyword can be put before the method's name to clarify its status. It can only be called from other host functions and is executed on the CPU. CUDA offers libraries and syntax extensions for GPU memory management and kernel execution, which include:

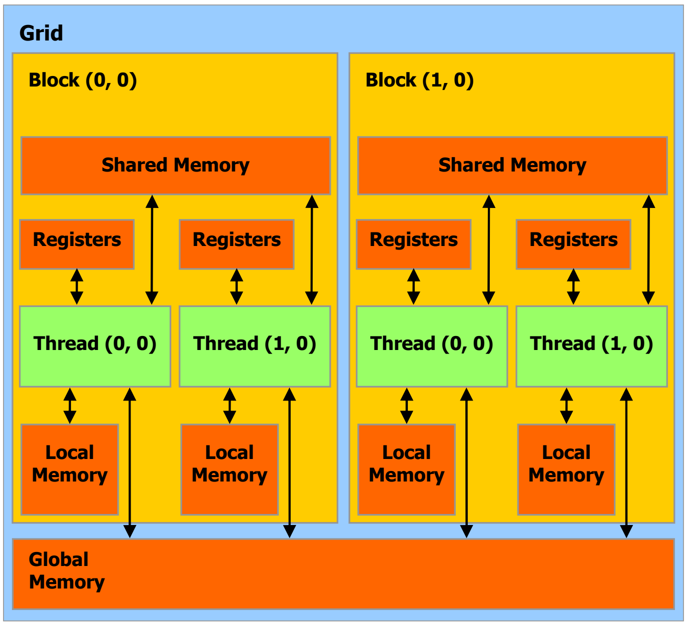


Figure 3.3: Different types of memory which CUDA kernels commonly access. Adapted from Wikimedia Commons (2010b) under CC BY 3.0.

cudaMalloc() allocates memory in the device's global memory in a similar manner as *malloc()* does on the host Random-Access Memory (RAM).

cudaMemcpy() copies data between host and device memories, comparable to *memcpy()* for host-only copies. The copy direction has to be passed as an argument.

kernelName<<<gridSize, blockSize>>>(par1, par2, ...) is the normal way of calling a CUDA device function. It is invoked in the same way as any other C method, using the name and passing the necessary arguments. However, the number of blocks has to be specified in *gridSize*, and the number of threads per block in *blockSize*.

In a typical CUDA program which uses these functions, memory for input and output data is first allocated on the device. Then, the input data is copied there and the kernel is run. In order to make the input and output memory available to the kernel, pointers to it are passed as arguments. Finally the results are copied back to the host for further use.

3.3.4.2 Device Functions

Functions written as kernels to be exclusively run on the device are marked either with the *__device__* or the *__global__* qualifier. They support specific CUDA methods like *__syncthreads()* and can access custom runtime constants, the most important of which are *blockIdx* and *threadIdx*. These uniquely identify each thread, by specifying the block's position in the grid as well as the thread's position in its block.

A *__global__* method is only callable by host functions. This is where the code execution starts on the device, so e.g. static shared memory allocation is usually handled here. The return type always has to be *void*, so the only way of returning results is to store them in an array in device memory and later copy them from the host side using *cudaMemcpy()*.

__device__ on the other hand marks a method that can only be called from other device functions. This works in the same way as in standard C and values can be returned to the calling function. Starting with compute capability 2.0, recursive calls of *__device__* functions are supported. However, the code is still run in a kernel, so the complexity of the code should be kept to a minimum. Code branching can cause massive performance drawbacks due to each warp's threads sharing their instructions, and if the kernel uses too many registers, register spilling could become a problem.

An example CUDA program with one host and one *__global__* device function is shown in listing 3.1. It is written in CUDA's C++ environment, and makes use of the host functions described above in section 3.3.4.1.

3.3.5 Maximization of Device Utilization

An important part of the optimization of CUDA programs is minimizing the amount of processors idling during the execution of a program. Using all the available hardware resources will increase the data throughput.

There are three levels on which to optimize device utilization in CUDA (NVIDIA Corporation, 2014b, pp. 72-76):

Application Level

The most important way of constantly keeping the GPU occupied here is the reduction of the latency created by kernel calls. When the host program calls a CUDA kernel, the passed arguments are transferred to the GPU and the blocks assigned to their SMs. This can be a reason for bad performance if many separate calls are issued to a kernel executing comparatively quickly.

A possible solution to this are asynchronous kernel calls running in parallel, if the program does not need them to be sequential. CUDA provides a stream architecture exactly for this purpose. This allows every method involving a CUDA device to be assigned to a stream. Functions attached to the same stream are then executed one after another.

Device Level

On the level of a single CUDA device, it is important to keep all its SMs busy

```
// Device code:
__global__ void ArrayAdd(float* A, float* B, float* C)
{
    // Determine position of thread in grid
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    // If the number of entries in the arrays (N) is not divisible
    // by the block size, there are more than N threads running.
    if (i < N)
        C[i] = A[i] + B[i];
}

// Host code:
int main()
{
    ...
    size_t size = N * sizeof(float);

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

    // Copy vectors from host memory to device memory
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Calculate the block and grid dimensions
    int threadsPerBlock = 32;
    // Make sure that at least N threads are run
    int numBlocks = (N + threadsPerBlock - 1) / threadsPerBlock;

    // Kernel invocation
    ArrayAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);

    // Copy results back to host
    cudaMemcpy(h_A, d_C, size, cudaMemcpyDeviceToHost);
    ...
}
```

Listing 3.1: Simple CUDA program for parallel addition.

at all times. This can be achieved by issuing enough blocks (either in one or several parallel kernel calls) with a sufficient block size. Each SM can handle multiple blocks simultaneously, the exact number of which is dependent on the factors described below. For that reason, a great number of small blocks can use the same number of SMs as only a few larger blocks.

SM Level

This third level of optimization is the most important one, as it directly affects the programming of the kernels. There also have to be taken into account numerous device properties and runtime parameters varying strongly between the different core generations (compute capabilities).

The main concern here is keeping the arithmetic units of the SM busy. The percentage of time that they actually spend doing calculations during a kernel call is referred to as *occupancy*.

Usually there are multiple warps resident on an SM during the kernel run. These may belong to one or several thread blocks, depending on the number of blocks and warps which the SM can hold for this specific kernel. The number of blocks is limited by the amount of shared memory that the kernel requires, or by the maximum amount of warps which the SM can handle. This again is limited by the register count that each thread of the kernel uses: The maximum number of stored registers per SM, and thus per block, is 32768 on the Quadro 600. The maximum block size for the triangle intersection test kernel, which uses 53 registers, is thus 618 threads. Using a block size of 32 threads, a theoretical amount of 19 blocks can run simultaneously on the same SM. However, the maximum number of resident blocks per SM is limited to eight on this device. A detailed table of the hardware limits of different compute capability versions can be found in NVIDIA Corporation (2014b, pp. 182-184).

If the computing units of the SM are kept busy by the resident warps depends on the arithmetic intensity of the kernel being executed. This is the ratio of calculations done per memory access. When a warp issues a global memory access, this takes between 200 and 800 clock cycles to be processed. In the meantime, the threads of this warp can not perform any more computations. To still keep the arithmetic units busy, it is stored and later reactivated when the access query returns. Then the SM's warp schedulers select a different warp whose instructions are then issued to the processors. However, if the arithmetic intensity of the running kernel is too low, sometimes all warps residing on this SM might be waiting for memory transactions to finish. This, although sometimes unavoidable, is a waste of throughput theoretically processable by the device.

Chapter 4

Implementation

The following chapter first specifies the two algorithms which were implemented on the GPU to be used for the discussed spatial BIM queries: the querying of an R-tree as well as intersection testing between triangle meshes. Both their sequential and parallel implementations are described, together with their possible application and combination.

The sequential implementations were largely reused from the available QL4BIM System. They are, together with the host (CPU) code of their parallel counterparts, programmed in C# atop the .NET Framework. This provides Language Integrated Queries (LINQs) (Meijer et al., 2006), which are methods for data querying available for all classes which implement the *IEnumerable* interface (most importantly *Array*, *List*, *HashSet* and *Dictionary*). They are heavily used in all the following algorithms, both in those taken from QL4BIM System (see Daum et al., 2014) as well as in the host code of the newly developed parallel ones.

4.1 R-Tree

4.1.1 Data Structure and Parameters

The R-tree is a multidimensional indexing structure for managing data objects with regard to their positions in space proposed by Guttman (1984). It uses a multi-level tree-like structure with each node from one level containing pointers to several nodes on the next level (child nodes). Each node is only referenced by exactly one parent node, with the exception of the root node. Those on the bottommost level are called "leaves" and only reference the single data object stored in them. Every node has an axis-aligned bounding box enclosing all of its children's bounding boxes, or its stored data object.

The tree is height balanced, which means that all leaf nodes are on the same level. Its defining variables are its fanout M, i.e. the maximum number of children

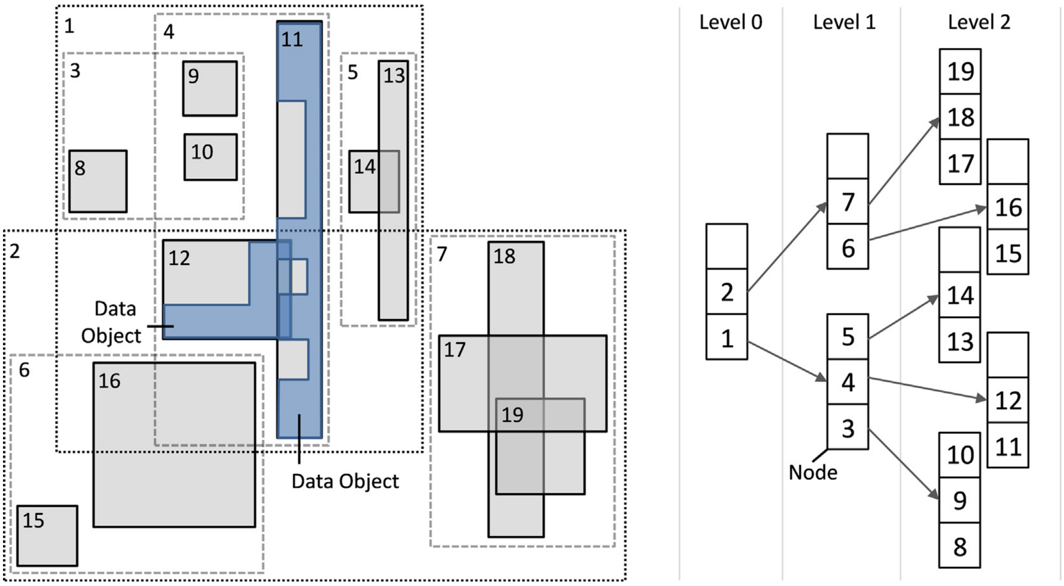


Figure 4.1: Possible structure of an R-tree storing objects in 2D-space with $M = 3$ and $m = 1$ over three levels. From Daum et al. (2014) under CC BY-NC-ND 3.0.

a node can have, and m, i.e. the minimum number of children a node (except for the root node) must have. Through appropriate restructuring during *Add* and *Remove* operations, the tree can always fulfill these requirements as long as $m \leq \frac{M}{2}$.

Figure 4.1 shows an example R-tree illustrating these properties.

4.1.2 R-Tree Creation

4.1.2.1 Implementation Used in QL4BIM System

The creation of an R-tree is a very complex and time-consuming task; it usually takes much longer than a single query on the complete tree. Because of that, the runtime reduction of the algorithm employed for this, as well as the reuse of once created R-trees, are important aspects in the optimization of the overall application performance.

QL4BIM System's implementation, which is also used for this work, supports two methods for tree construction: One uses the original algorithms proposed by Guttman (1984). The other one increases performance by incorporating optimizations to construct the R*-tree proposed by Beckmann et al. (1990). Both of these are based on a sequential approach, where all the elements of the final tree are added to an initially empty tree one after another. The most important procedures for this are the following:

ChooseLeaf decides to which node a newly inserted leaf should be added. In Guttman (1984), the best node of each level is chosen recursively based on which node needs the least enlargement of its bounding box area/volume. The thus determined node's children are compared as candidate nodes for the next level in the next recursion step.

SplitNode is called every time the node decided upon by *ChooseLeaf* is full. The tree then has to be adjusted by dividing the children of the overflowing node in two newly created nodes as intelligently as possible. The *Linear Split* algorithm of Guttman (1984) first decides on two seed leaves to start the new nodes, and then divides on them the remaining leaves with the same criterion as *ChooseLeaf*. The initial leaves are selected by calculating which of the bounding boxes have the highest distance from each other in relation to the extent of the R-tree along each respective dimension. The pair with the highest relative distance along any of the considered dimensions is then chosen as the seed pair.

4.1.2.2 Possible Optimization and Parallelization

For possible optimizations of R-tree creation, one has to consider two different aspects: On the one hand, a focus can be put on decreasing the computational cost of the creation process itself, by e.g. choosing cheap implementations of *ChooseLeaf* and *SplitNode*. On the other hand, e.g. for massively reused R-trees or when only the search time is important, it might be more sensible to focus on lowering the cost of querying the tree. A possible approach to this could be to reduce the average amount of overlap between different nodes in the same level, and thus the mean number of candidate nodes that need to be checked.

An alternative to the original algorithms for *ChooseLeaf* and *SplitNode* is proposed by Beckmann et al. (1990). This R*-tree promises superior performance in query as well as construction time.

Furthermore, there have been efforts to improve construction performance by parallelizing this algorithm and running it on the GPU. Apart from the possible speed-up of the pure R-tree creation, this method offers an additional benefit in combination with a GPU-based query algorithm: The copy overhead between host and device is reduced. As this is comparatively high and sometimes responsible for significant portions of the overall runtime, running both operations without moving the R-tree structure between GPU and CPU can improve their combined performance substantially.

One of the first studies on parallel R-tree construction, also called bulk loading, was published in Kamel et al. (1992). Further research on porting this technique to the GPU was done in the last few years: Luo et al. (2012) found that GPU-based bulk loading offers speeds of between 26% and 600% of those of comparable CPU implementations. Which of the two implementations needs less time depends mostly on the number of objects to be stored here; as expected, the GPU tends to perform better for larger data sets. However, You et al. (2013) detected possible inferior query performance when searching trees created in this way.

4.1.3 Sequential R-Tree Query

Using this index structure, it is possible to retrieve all the stored objects whose bounding boxes intersect with an arbitrary search box. This can easily be done by first checking for intersections between the search box and the bounding boxes of the topmost level of nodes. Then, the same test is done for all child nodes of the nodes whose bounding boxes intersect with the search box. Repeating this procedure will eventually yield the stored objects of the leaf nodes whose bounding boxes intersect with the search box.

The sequential code for the R-tree query was implemented for QL4BIM System in the scope of Daum et al. (2014) and reused identically for this work. It largely follows the query algorithm presented in Guttman (1984). However, as recursive functions in combination with C#'s *yield return*-statement lead to complicated code, a loop over a queue was employed; see algorithm 1 for the specific implementation.

Algorithm 1 Sequential C# algorithm for the R-tree query. Returns every object stored in *RootNode*'s R-tree whose *BoundingBox* intersects with *SearchMesh.BoundingBox*.

```
if not BoxesIntersect(RootNode.BoundingBox, SearchMesh.BoundingBox) then
  yield break
end if
queue.Enqueue(RootNode)
while queue.Count > 0 do
  parent ⇐ queue.Dequeue()
  for each node in parent.Nodes do
    if BoxesIntersect(node.BoundingBox, SearchMesh.BoundingBox) then
      if parent.IsLeaf then
        yield return node.Item
      else
        queue.enqueue(node)
      end if
    end if
  end for
end while
```

4.1.4 Parallel R-Tree Query

The R-tree query potentially offers parallelism, as the box intersection tests can be performed simultaneously on the different nodes of each level. This implies possible performance benefits from a GPU implementation. Because of that, there have been various studies of accelerating this procedure using CUDA, e.g. Kunjir et al. (2009), Luo et al. (2012), Yampaka et al. (2012), and You et al. (2013). The algorithm used here is largely based on that presented in Luo et al. (2012).

4.1.4.1 Input Data Representation on the GPU

The first problem which has to be solved is the representation of the R-tree data in the GPU's global memory. This has three main requirements:

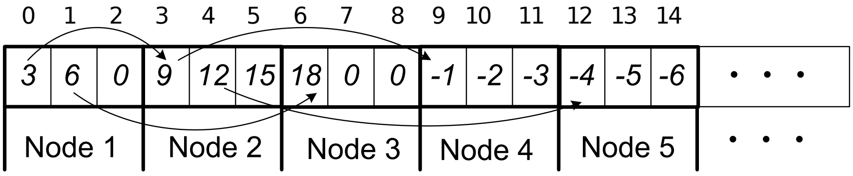
1. The size of the data has to be minimized, by reducing the amount of redundant or unused information. The benefit of this is that the individual *cudaMemcpy()*-operations take less time.
2. The data should be stored in as few different places as possible, so that the number of *cudaMemcpy()*-commands which have to be issued is reduced. This saves the overhead produced by each *cudaMemcpy()* call.
3. It is best if nodes belonging to the same parent are located in adjacent regions in device memory. This makes coalescing memory accesses possible, which can speed up program execution significantly.

It is thus evident that the original C# representation is not suitable for the GPU's needs. The referenced C# objects are distributed randomly across the application's memory heap, with no continuous chunk of memory reserved exclusively for them. Therefore a different memory model proposed in Luo et al. (2012) is employed.

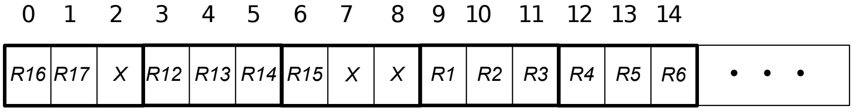
As the R-tree search only takes into account the bounding boxes, but not the spatial objects stored in the leaf nodes, it is sufficient to copy the tree structure and the box coordinates to the GPU. They are represented by two arrays:

Index Array Every group of M entries corresponds to the child nodes of one parent here, with the first M entries belonging to the root. Each positive entry is a non-leaf node and contains the array index of its first child node. If a parent has less than M child nodes, the remaining entries are filled up with zeros. Negative entries are leaf nodes; their value uniquely identifies their respective spatial object stored in the R-tree, similar to an ID (figure 4.2(a)).

Box Array It has exactly the same length as the index array, because it stores each bounding box at the same offset that the corresponding node has in the index array. Where the corresponding entry in index array is zero, the value is a don't-care (figure 4.2(b)).



(a) Index-- Array of node structures



(b) *Rect*-- Array of rectangle coordinates

Figure 4.2: Array structure employed for storing R-trees on the GPU. From Luo et al. (2012), © 2012 IEEE.

4.1.4.2 Parallelization Approaches

As mentioned above, a single R-tree query has the parallelism offered by simultaneously checking all the candidate nodes of the same level for intersections. However, even higher performance improvements can be achieved when multiple queries are run simultaneously. This can be used e.g. when searching for all mesh intersections in an IFC model.

These two levels of parallelism can directly be applied to the CUDA architecture, so that each block handles a separate query and each thread checks the box intersection for one node (see figure 4.3). This fits the purpose of each level of parallelism in CUDA perfectly: The different blocks have no way of communicating with each other during a kernel call, which is not necessary as each individual query has its own independent search box. Inside a block, however, threads can work together and quickly exchange data using shared memory and the *__syncthreads()* command. This is necessary, because the threads of each block have to have a common pool of nodes which have to be checked, somewhat similar to the queue employed for the sequential implementation.

4.1.4.3 Overflow Treatment

However, unlike the sequential implementation, the queue can overflow here, i.e. there are too many candidates and too few threads to perform all the intersection tests in parallel. In each level, the threads check all the children of the nodes stored in the queue for intersections. Even for zero-nodes in the index array, a dummy thread which is not doing meaningful work is running. This means that there cannot be more than $blockSize/M$ nodes in the queue at once.

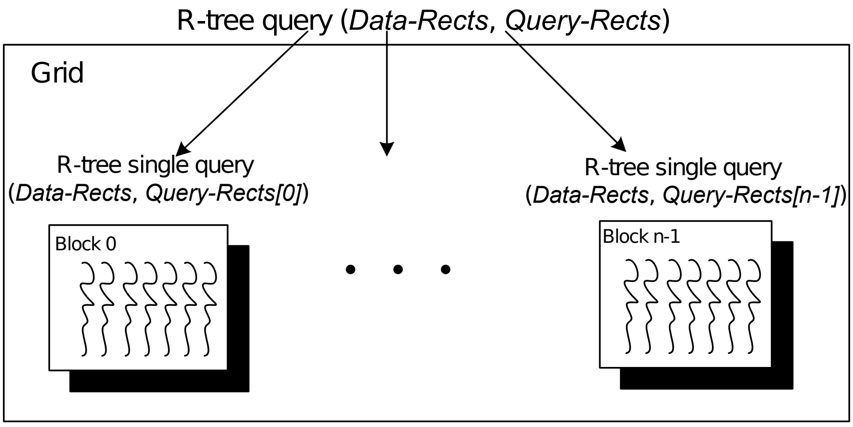

R-tree query (Data-Rects, Query-Rects)
Grid
R-tree single query (Data-Rects, Query-Rects[0])
R-tree single query (Data-Rects, Query-Rects[n-1])
Block 0
Block n-1

Figure 4.3: Structure of a kernel call employing two levels of parallelism. From Luo et al. (2012), © 2012 IEEE.

If a thread finds an intersecting node, which does not fit into the queue anymore, it sets the block-wide shared Boolean *overflowing* to true. After the next *__syncthreads()*, the other threads check this variable and start the overflow treatment: The indices of the nodes whose children were checked in this level, i.e. the results from the previous level, are written to the output array. Finally the output overflow flag for this block is set to true.

The host code handles blocks with a positive overflow flag as follows: All the node indices returned by overflowing blocks are gathered into one list. At the same time, another list is made of their respective search boxes. These can be different if multiple queries were issued in parallel. Using these lists, the kernel is called again, but this time an entire block is assigned to each index. Due to the storage model of the R-tree on the device, all these blocks can be issued together and search the same tree. They simply have to start at the previously returned indices, which are transferred to the device before the kernel is called. For initial (standard top-level) queries, this is set to zero (the root node). If these blocks again return positive overflow flags, the procedure is repeated recursively until all the searches have finished.

4.2 Mesh-Intersection Test

This test uses the concept of intersection testing of triangle pair meshes described in section 2.3.3. The basic idea here is to discern between touching and intersecting relationships between objects by testing both the inner and the outer mesh of object A against both meshes of object B. A real intersection is only proven if all these tests return true, in accordance with the definition of the *touching* relationship in figure 2.1.

As described in section 2.3.2, the individual tests between two meshes are performed by checking each triangle of mesh A for an intersection with any triangle of mesh B. Because the meshes form closed surfaces, this directly proves that they meet. For this project, the triangle intersection algorithm presented in Möller (1997) was used. This is publicly available at Möller (1999) in highly optimized standard C. As such, it can be used by any CUDA kernel without further changes besides adding the *__device__* qualifier in front of the function names.

4.2.1 Sequential Implementation

For it to run in C# however, some expressions had to be adapted, while the algorithm itself remains unchanged. The overall mesh intersection test for *TrianglePairMeshes*, which uses this triangle intersection test, is already present in QL4BIM System. It expects the R-trees containing the individual triangle pairs of the two *TrianglePairMeshes* as input arguments. The two trees are then checked against each other level by level using a queue similar to the one in the sequential R-tree query described above. However, each queue entry is now a pair of R-tree nodes, with the initial entry being the two root nodes. The pairs of their child nodes whose bounding boxes intersect are then added to the queue and processed in the same way again. Using this procedure, eventually the queue only contains pairs of leaf nodes with intersecting bounding boxes. Then the individual triangle tests are run on them one after the other. However, if a combination of mesh intersections (e.g. inner-inner) has already been proven, the individual triangle pairs are no longer tested for that.

4.2.2 Parallel Implementation

Unlike the sequential implementation, the parallel one separately checks the four mesh combinations which have to be considered for each intersection test. Additionally, it is designed as a search, where all the meshes of the current model are already stored on the device beforehand. While this increases the initialization time of the program, the latency of the individual intersection queries is vastly reduced. The search mesh is then tested against all the stored triangle meshes by looking for intersections between all possible triangle combinations.

4.2.2.1 Levels of Parallelism

This algorithm again exposes two levels of parallelism. As such it fits the CUDA programming model perfectly: One thread can be assigned to each triangle of the stored meshes, and then sequentially check the triangles of the search mesh for intersections. However, if another thread of the same mesh has already found an intersection, further testing would be a waste of computational resources. For that

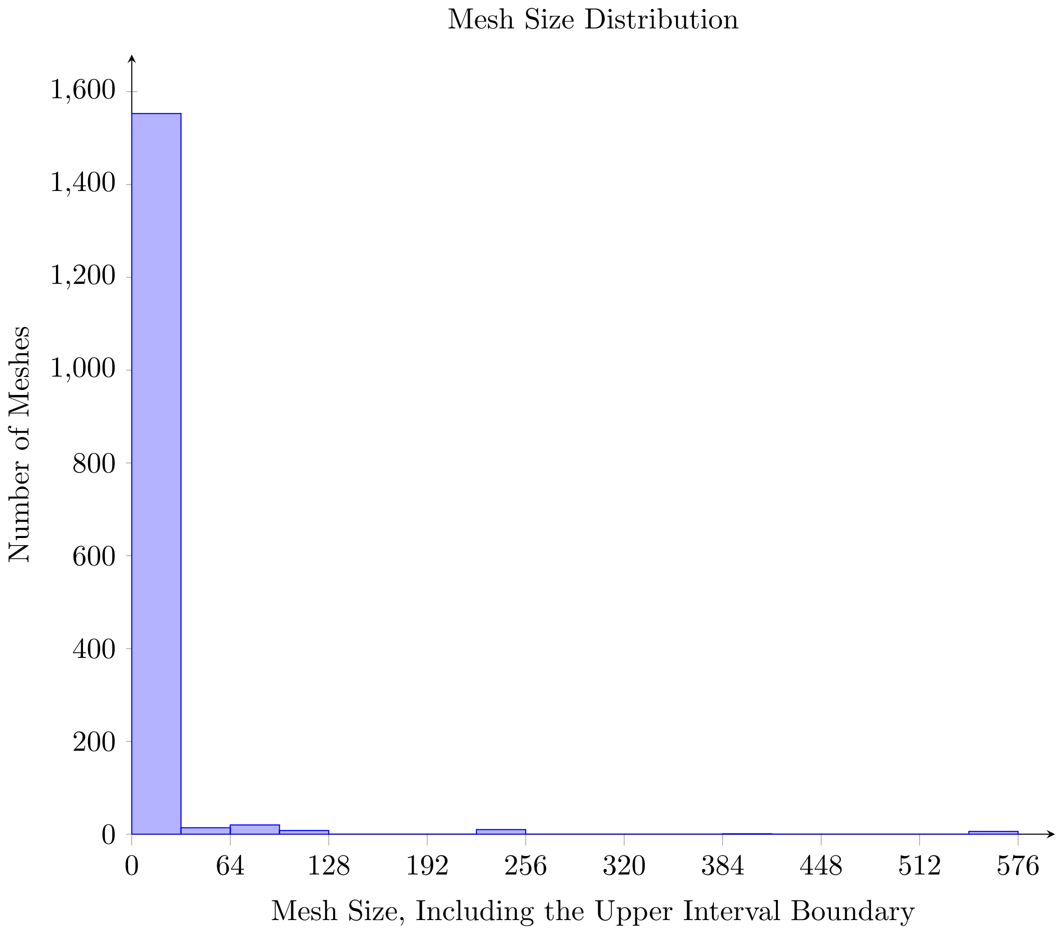


Figure 4.4: Histogram of the mesh sizes found in the *AC11-Institute* IFC model. The bins include their upper boundaries and have a size of 32. Thus every successive bin needs one more warp per mesh.

reason, the threads assigned to the same mesh communicate to each other whether they have already found a positive result. To that end, each mesh is checked by one block, with a shared Boolean result variable.

4.2.2.2 Mesh Partitioning

A problem that had to be overcome here are the vast differences in the number of triangles per mesh. This ranges from 12 to 576 in the examined *AC11-Institute* IFC model. Therefore, a block size of 576 is necessary to fit the testing of even the largest mesh in only one block. This wastes a great amount of potential performance, because over 96% of the meshes have 32 or less triangles, and can thus be tested by only one warp (see figure 4.4).

As the control flow is only fixed within each warp, the surplus warps in blocks which are checking a small mesh can immediately return. Although this would free

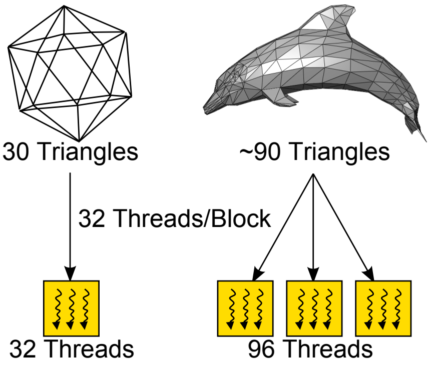


Figure 4.5: Example of the distribution of a simple as well as a complex mesh on blocks of 32 threads each. Made of figures from Wikimedia Commons (2007) and Wikimedia Commons (2008, under CC BY-SA 3.0).

the computational units on their SMs, their register memory remains allocated. This prevents other blocks from being run on the same SM. Its processors might thus not have enough work to keep them occupied during memory accesses of the remaining active warps.

To resolve this issue, the large objects are distributed on several blocks as shown by example in figure 4.5. Every block then only tests a set of triangles, which do not form a closed mesh. However, the intersection is still proven for the entire mesh if at least one of these mesh fragments intersects with the search mesh. The disadvantage of this is that when one of the blocks finds an intersection, the others have to continue their checks anyway because communication is impossible. This results in extra computational effort being done, which can only be neglected because merely a minuscule portion of the meshes consists of more than a few dozen triangles.

4.2.2.3 Test on a Reduced Set of Meshes

As from a certain model size upwards the numerous blocks can not be handled by the device's SMs all at once, they would eventually have to wait for execution. This ultimately leads to a close to linear scaling of the runtime with respect to the number of checked meshes. As this is impractical for large BIM models, preselection can be done using an R-tree which indexes all the meshes in the IFC file. Then only those whose bounding boxes intersect with that of the search mesh have to be tested triangle by triangle.

To minimize data transfer between host and device, the complete set of meshes for the entire model is still first copied to the GPU. Although this causes a large initial overhead, subsequent queries can then be carried out much faster. This is because only the pointers to the candidate meshes or their mesh fragments need to be copied to the device prior to each function call. Every issued block then tests the triangles which its pointer references.

4.2.2.4 Adjustments for Finding All Intersection Pairs

The search for all mesh-mesh intersection pairs in the entire model was implemented slightly differently. The single search mesh kernel would require one separate kernel call for every mesh in the model. For typical model sizes, thousands of them would have to be issued. This is impractical in CUDA, as they would either have to be run completely sequentially, or partly parallelized using concurrent kernel execution. This can be done using CUDA's asynchronous function call framework, which works with several streams running in parallel. However, the maximum number of kernels running simultaneously is limited to 16 on compute capability 2. Moreover, each of these separate calls still causes a lot of overhead due to SM allocation and parameter copying.

Therefore a single kernel handling this massive intersection query was devised. As for the single query, each block handles one mesh, with large ones being split into several block sized fragments. However, instead of checking its triangles against one single search mesh as before, each block now performs intersection tests against several other meshes.

These are again preselected individually for every mesh. This is done by first querying an R-tree and then removing from its candidate list all the meshes which have an index in the global mesh list of lower or equal the current mesh's index (see figure 4.6). This way, each combination of meshes is ensured to only be checked once. Additionally, unnecessary tests of meshes against themselves are prevented. The resulting lists of candidate indices for each mesh are finally concatenated into one array. The interval in this array which each block should process is stored in a second array.

4.2.2.5 Treatment of Kernel Timeouts on Windows

One problem that was encountered here lies in the long runtime of this extremely computation intensive kernel. Upwards of a few thousand meshes, the execution takes more than a few seconds to finish. This leads to an error, because Windows automatically stops all shader programs running for more than a certain time threshold (NVIDIA Corporation, 2014a). This crashes the application, and has hung the entire machine several times during the development of this program. To still be able to carry out benchmarks of larger IFC models on Windows systems, the maximum execution time has to be reduced.

To that end, all blocks which need to be run are split up into several grids, each with a low enough runtime, and executed sequentially. The offset that each thread has to add to its runtime *blockIdx* constant to achieve identical behaviour as with one single grid is passed as an argument in each kernel call.

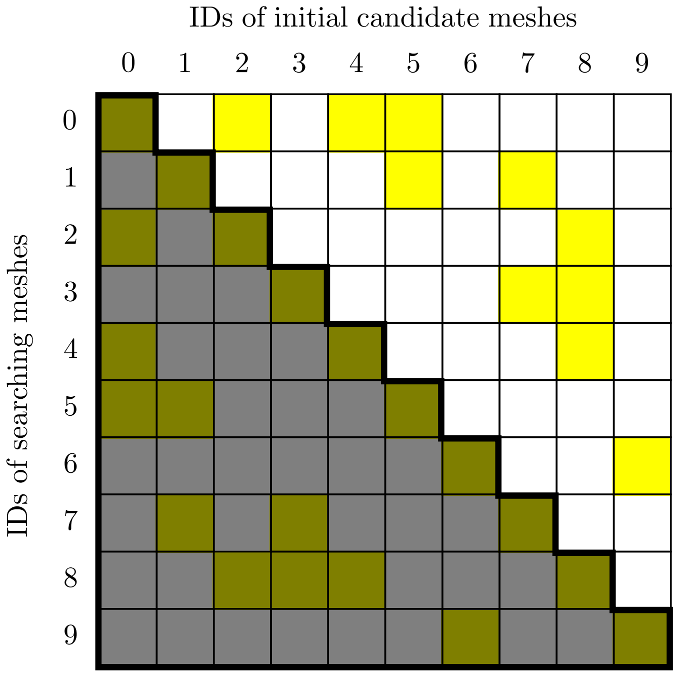


Figure 4.6: Sketch of the selection process of candidates for the self intersection test:

1. The candidates (🟨) for each searching mesh (corresponding to rows) are selected using a multiple parallel R-tree queries. As every bounding box intersection is found for both meshes involved, the R-tree produces a symmetric pattern in the matrix.
2. The candidates on the lower left side of the matrix, i.e. with smaller indices than the search meshes, as well as those on the diagonals, with equal indices, are removed (thick black line). This prevents the same mesh combination from being checked twice as well as intersection tests of meshes against themselves.
3. The indices of the remaining candidates in the top right half of the matrix are stored in an array.

To determine the number of blocks that each grid should run, a measure of the expected computational complexity is needed for each searching mesh. The number of its candidate blocks multiplied with the number of blocks into which the mesh is divided itself is a very good approximation of this. Basically, the number of run blocks is multiplied with the number of blocks which they have to check – which yields the number of block-block intersection tests. However, this can not predict how quickly the blockwise tests find intersections, and can thus stop their triangle testing. Additionally, many blocks are not full, which also reduces the number of necessary tests.

These variations depend on the examined IFC file, the block's rough position therein as well as the type of object stored in it. Their effects can thus be approximated by the runtime of previously run blocks. This is especially promising because in most IFC files, the *IfcProducts* are numbered by either spatial position or type. To benefit from this, the expected runtime of each mesh is calculated by multiplying its expected complexity with the runtime per expected complexity of the previously run grid.

However, the runtime of each grid still shows variations with respect to the predicted one. It thus needs a corresponding safety margin, especially because a timeout immediately causes a crash of the program. However, the chosen grid size has to minimize the overhead caused by individual kernel calls at the same time. Assuming a constant runtime per candidate block checked, the theoretical number of candidate blocks which can be checked in 500 ms can easily be calculated from the previously run grid. Together with a minimum of 100 blockwise intersection tests per grid, this proved to be a good solution: The overall runtime of this test was not significantly increased, and none of the examined models caused a timeout.

Chapter 5

Performance Analysis

To judge the strengths and weaknesses of the different implementations of the examined spatial BIM queries, a benchmark testing the runtimes of several combinations thereof was performed on multiple real-life models with different sizes.

5.1 Remarks on Platform Comparison

When analyzing the CPU and GPU implementations of an algorithm, a clear statement about which one performs better is often difficult (Lee et al., 2010). Apart from the almost impossible performance comparison of the different hardware, the programming on each platform is also difficult to standardize. For both architectures, there exist virtually unlimited possibilities for optimization and fine-tuning.

The most important factor in these performance tests is that the host code is written in C#. As this is run in a hardware abstraction called the Virtual Execution System (VES), the developed application might be running slower than it would written in C. This affects both pure CPU operations as well as CUDA functions, but to a different degree. This makes a direct comparison of the runtimes extremely difficult.

For those reasons, only the rough trends visible within the numbers were compared, which contain information about the scaling behaviour of the different platforms for different model sizes.

5.2 Test Description

5.2.1 Test Setup

The performance analysis was performed on a workstation with a quad-core Intel Xeon E31225 CPU at 3.10 GHz of clock speed and 8 GB of DDR3-1333 RAM. The installed dedicated graphics device was an NVIDIA Quadro 600 workstation GPU with two SMs. These have 48 CUDA cores each, resulting in a total of 96 CUDA cores, which are clocked at 1.28 GHz. It further offers 1 GB of global memory, as well as 64 KB of on-chip memory per SM. The latter was left at its default settings, yielding 16 KB of L1-cache and 48 KB of shared memory (see section 3.3.1 for the configuration of on-chip memory). Each SM can also store up to 32768 registers.

The machine was running Windows 7 Enterprise Service Pack 1 in its 64-bit version. The installed version of the CUDA driver and runtime was 6.5, on top of the NVIDIA graphics driver version 340.62. The C# code was built and executed using the .NET Framework 4.5. All compilations were done targeting the x64 platform in release mode, and the program was finally run without any debuggers attached. All floating-point calculations were performed with double precision.

As a measure of the accuracy of the results, the standard deviation is a good indicator. For a machine not otherwise occupied, according to Haveraaen et al. (2001), it decreases fastest when taking the minimum of the obtained runtimes as the correct one and measuring it from there. Therefore, the described benchmark was run three times for each model, with the lowest collected times assumed as the measurement results.

5.2.2 Parameter Settings

In the source code of the developed programs, numerous constant parameters had to be defined. Almost all of these have impacts on the overall performance, but the scrupulous analysis and optimization of each one would go beyond the scope of this thesis. Educated guesses were made for their optimal values, which are listed in table 5.1.

5.2.3 Model Selection

The goal of this performance test is the comparison of the scaling behaviours of the different query implementations. This is evaluated with respect to model size, i.e. the number of objects in an IFC file, as well as other query-specific properties, especially the number of found results. The perfect test subjects would thus only change in these characteristics, but none else. This can only be achieved using artificial models, such as the one used in Daum et al. (2014).

However, they can only be compared with representations of real-world buildings to a limited degree. As the tests show, their distinct properties might severely impact the performance of spatial operations executed on them. One of the most important peculiarity of actual models is the varying complexity which each mesh has. A very large meshes, i.e. consisting of many triangles, requires more computations per tested mesh intersection.

Parameter	Description (see section)	File	Value
NumBenchmarkRuns	Number of times that a benchmark is repeated to obtain meaningful results (5.2.1)	CUDACaller.cs	3
MinNumNodes	The minimum number m of children which an R-tree non-root node must have (4.1.1)	CUDARTree.cs	1
MaxNumNodes	The maximum number M of children which an R-tree node can have (4.1.1)	CUDARTree.cs, rTreeSearch.h	8
RTreeQueryBlockSize	Block size used for the R-tree query kernel (4.1.4)	CUDARTree.cs, rTreeSearch.h	512
CUDAMeshIntersectorBlockSize	Block size used for the two mesh intersector kernels (4.2.2.2)	CUDAMeshIntersector.cs	32
StartingMaxNumRunBlockTests	Block tests performed in the first partial run of the all mesh intersections kernel (4.2.2.5)	CUDAMeshIntersector.cs	2000
MinNumRunBlockTests	Minimum number of block pairs to test per kernel run (4.2.2.5)	CUDAMeshIntersector.cs	100
TargetMillisecondsPerKernelRun	Target runtime of each partial all mesh intersections kernel run in milliseconds (4.2.2.5)	CUDAMeshIntersector.cs	500

Table 5.1: Descriptions and chosen values of all the constants in the project.

Furthermore, unless constructed very elaborately, the objects in artificial models are regularly distributed across space. However, most real-world models are spatially inhomogeneous. This corresponds to a building having big, open rooms with a small amount of furniture in one section, and smaller rooms with a high density of room setup and MEP equipment in the other. Although it did not become apparent in the tests done in this thesis, this can have an impact on the R-tree performance: It might lead to an unfavorable tree structure, which can influence its construction as well as query time.

For those reasons, an extensive test using real-world models with different object counts was performed. Ten models were supplied by the Chair of Computational Modeling and Simulation of TUM. However, only four of these could be tested, because the others either could not be exported in the IFC format from Autodesk Revit, or required too much memory to be evaluated on the test system with 8 GB of RAM. Additional models were downloaded from the Open IFC Model Repository (University of Auckland, 2012), which were originally supplied by researchers from Karlsruher Institut für Technologie (KIT).

5.3 Results

5.3.1 Model Properties

First, the average number of *TrianglePairs* per object was analyzed for each IFC file (see figure 5.1). This shows that the average mesh complexity indeed varies

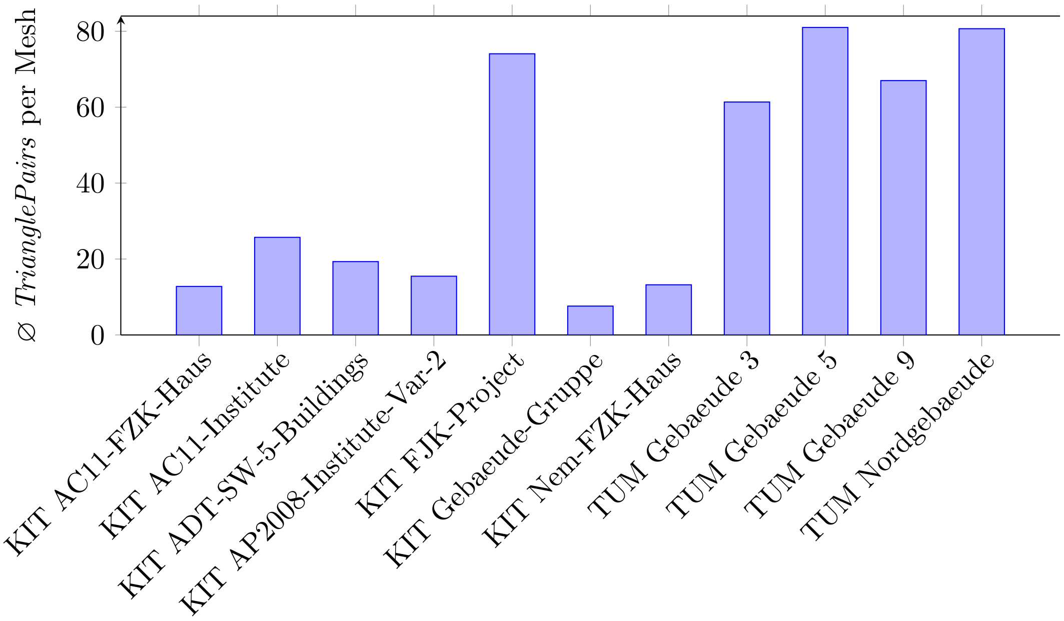


Figure 5.1: Plot of the average number of *TrianglePairs* per *TrianglePairMesh* for each examined IFC file.

strongly between different building models. Thus the construction time of the *TrianglePairMeshes* as well as the average mesh intersection time should be compared with respect to the number of *TrianglePairs*, not meshes, in a model.

5.3.2 Data Structure Creation

Next, the construction times of the different data structures, including their copy time to the CUDA device if necessary, were examined. They have to be considered separately from the individual query processing times, because, once created, they can be used for multiple queries.

The first created object was the *CUDAMeshIntersector*. Although each mesh causes a small minimum overhead here, the deciding factor for the instantiation time is the number of *TrianglePairs* (see figure 5.2). This has a much clearer connection with the measured time than the number of meshes, showing an almost perfectly linear correlation except for one outlier.

The computationally intensive tasks in this constructor function are the calculation of the *TrianglePairMeshes* from the original single *TriangleMesh* representation and their storage in separate triangle arrays for the CUDA device as described in section 4.2.2. This is all done in sequential C# code making heavy use of LINQ, which naturally scales linearly with increasing amounts of processable data. The copying of the resulting arrays takes only a small portion of time in the order of milliseconds, which only has a minor impact here.

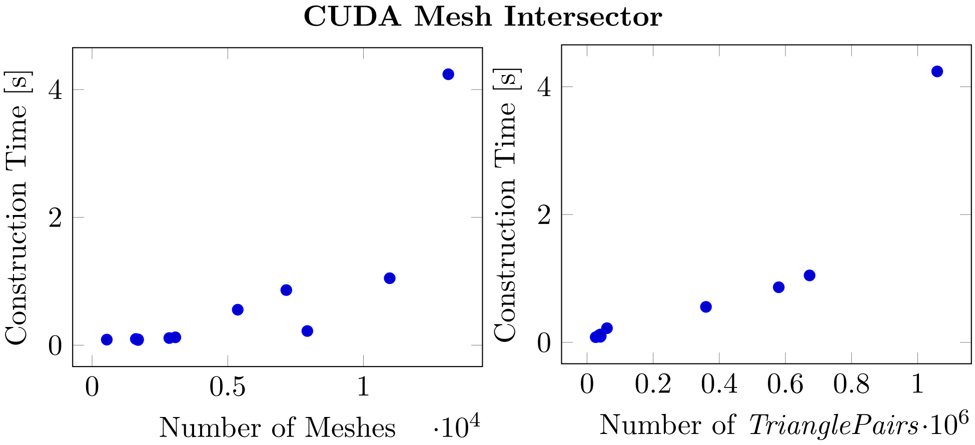


Figure 5.2: Scatter plots of the *CUDAMeshIntersector* construction times. While they only show a limited connection with the number of meshes, they depend on the number of *TrianglePairs* almost perfectly linearly when considering the data point with over four seconds of runtime an insignificant outlier.

The IFC file with over four seconds of runtime, which does not fit in the linear model for this test, is TUM Nordgebaeude. The exceptionally bad performance here can perhaps be attributed to a memory bottleneck. When loading this IFC file in QL4BIM System, almost all of the test machine's 8 GB of memory were occupied. This possibly slowed down all further analysis, because memory allocation therein might have either been fragmented or required moving part of the main memory to the hard drive first.

The next called constructor is that of the *CUDARTree* which indexes all the meshes – one per *IfcProduct* – of the IFC file. From this, the candidates for individual mesh intersection tests can be obtained. The created instance references both the original C# R-tree as well as its array representation (described in section 4.1.4), which is stored on the CUDA device. The object can thus be used for both parallel CUDA queries as well as sequential ones in C#.

The main constructor tasks are the creation of the R-tree in C#, the derivation of the corresponding array structure and finally the copying of this to the CUDA device. The tree construction dominates the runtime here. It only takes the individual meshes' bounding boxes into consideration, not their complexity. Thus its runtime can be expected to only depend on their number. This relation is plotted in figure 5.3, and proves to be linear. This corresponds with the almost constant insertion time for sufficiently large R-trees described in Guttman (1984).

The last constructed object is a list of the C# R-trees containing all the *TrianglePairs* of each object. These are used by the C# mesh intersector, which compares the individual *TrianglePairs* against each other using these R-trees. Constructing all of them facilitates the reuse of the data structures for multiple queries. How-

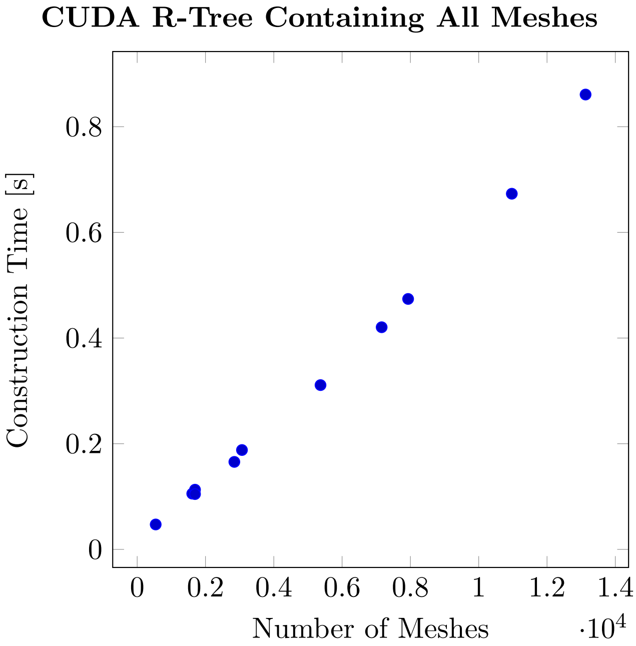


Figure 5.3: Scatter plot of the *CUDARTree* construction times. Their dependence on the number of meshes is almost perfectly linear for all examined models.

ever, it has to be noted that for a single intersection query with a search mesh, only the objects which are selected by the global meshes R-tree have to be indexed and searched. This can reduce the overall runtime significantly, which is why constructing all R-trees beforehand is only beneficial if they are used for a large number of queries.

The timing results plotted in figure 5.4 show an overall linear relation between the construction time and the number of *TrianglePairs*. As with the model-wide R-tree before, this can be attributed to the practically constant R-tree insertion times of each *TrianglePair*.

5.3.3 Intersection With Search Mesh

After all the necessary data structures were created, a simple query looking for all meshes intersecting with one search mesh was executed. The intersection criterion used for this was that the two inner as well as the two outer *TriangleMeshes* intersect. To achieve this, these two mesh combinations were checked separately in the parallel CUDA implementation, while the C# mesh intersection tester was adjusted accordingly. The search mesh which this benchmark uses is the octant with the smallest x-, y- and z-coordinates of the largest mesh's bounding box.

The first test, whose results are depicted in figure 5.5, used both implementations to check the search mesh against every single object in the model. As the parallel CUDA program uses a brute-force method which tests every triangle in the entire

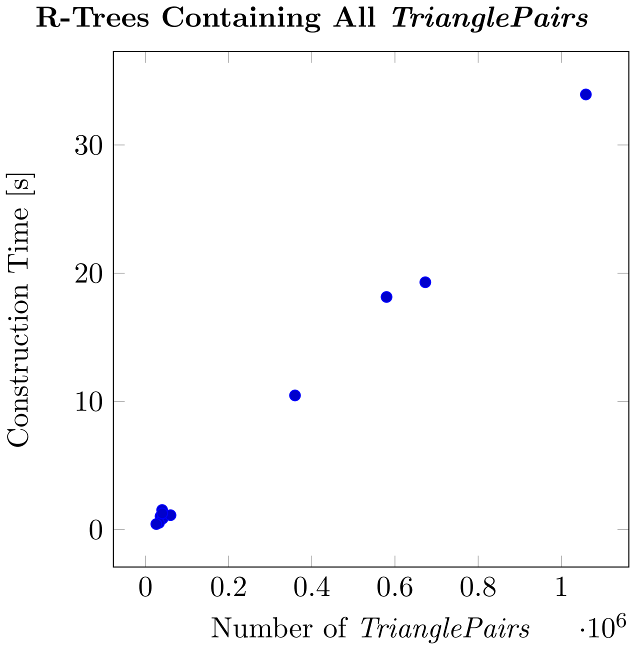


Figure 5.4: Scatter plot of the construction times of all the R-trees containing the individual *TrianglePairs*, showing a clearly linear correlation.

scenery against all triangles of the search mesh, its linear scaling with the number of *TrianglePairs* in the scenery makes sense. In the more intelligent C# approach, non-intersecting meshes are quickly excluded from testing in the first level of the R-tree test. Thus the number of found results has a higher impact on the overall runtime here.

The next conducted test did the same single mesh intersection query, but only using a selection of candidate meshes. These are extracted from the total set of objects using an R-tree query (see figure 5.6).

The runtime of this does not scale linearly, but increases by trend with the number of intersections for both methods. However, both methods need roughly the same relative amount of time for every examined model. This could be caused by the R-tree structure and search box setting particular to each test subject.

The plots also show that the CUDA implementation has a minimum latency of more than 2.5 ms. This can largely be attributed to the relatively high overhead of each data copy and kernel launch operation. Additionally, the interfacing by managedCUDA slows down execution, with a pure C++ implementation promising better results.

The mesh intersection tests run using the thus acquired candidate list had the runtimes shown in figure 5.7. They scale almost identically for all three examined parameters. This can probably be attributed to the interdependence between the parameters themselves: The number of candidates is of course higher for larger result

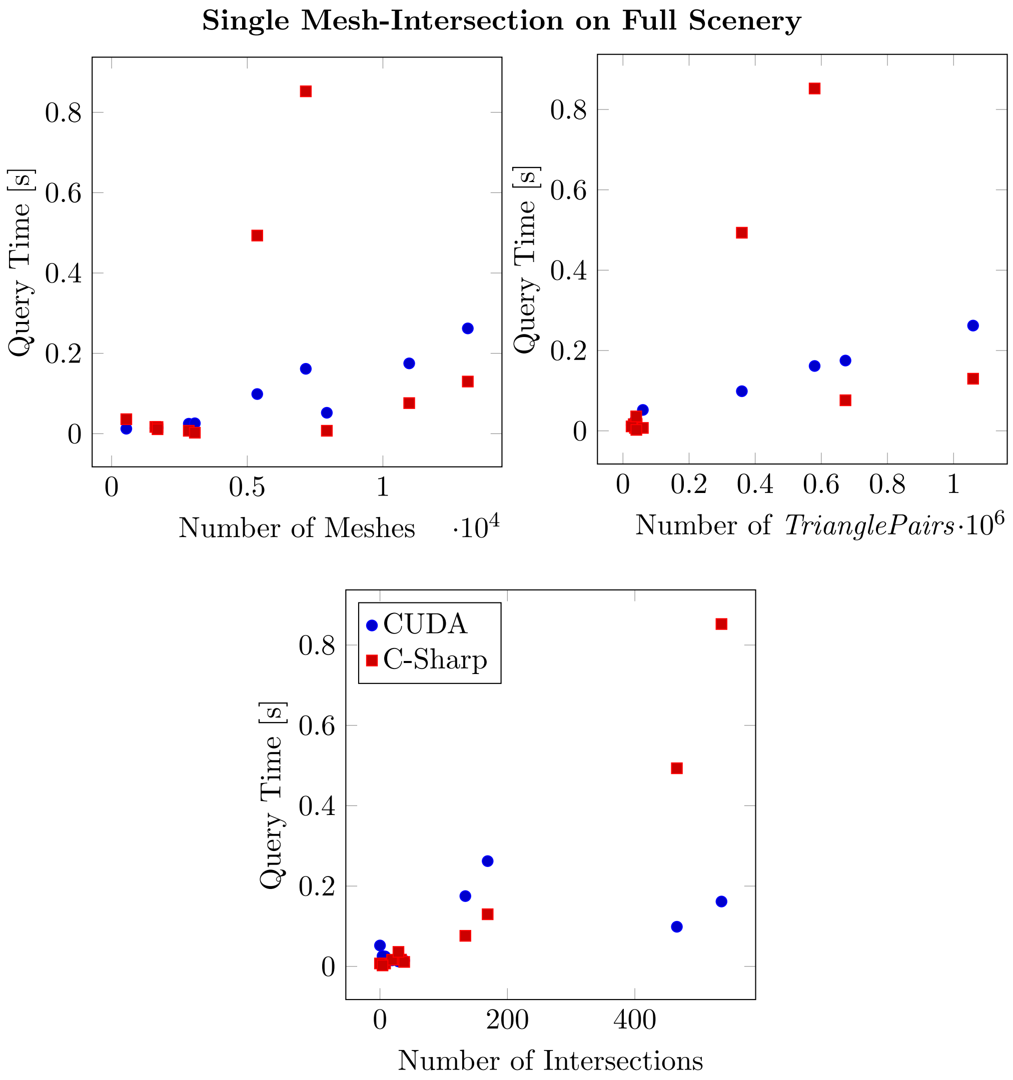


Figure 5.5: Scatter plots of the single mesh intersection query times without using the global meshes R-tree to reduce the number of checked meshes. The CUDA intersection test scales linearly with respect to the number of *TrianglePairs* in the model, while the C# one correlates most strongly with the number of found intersecting meshes.

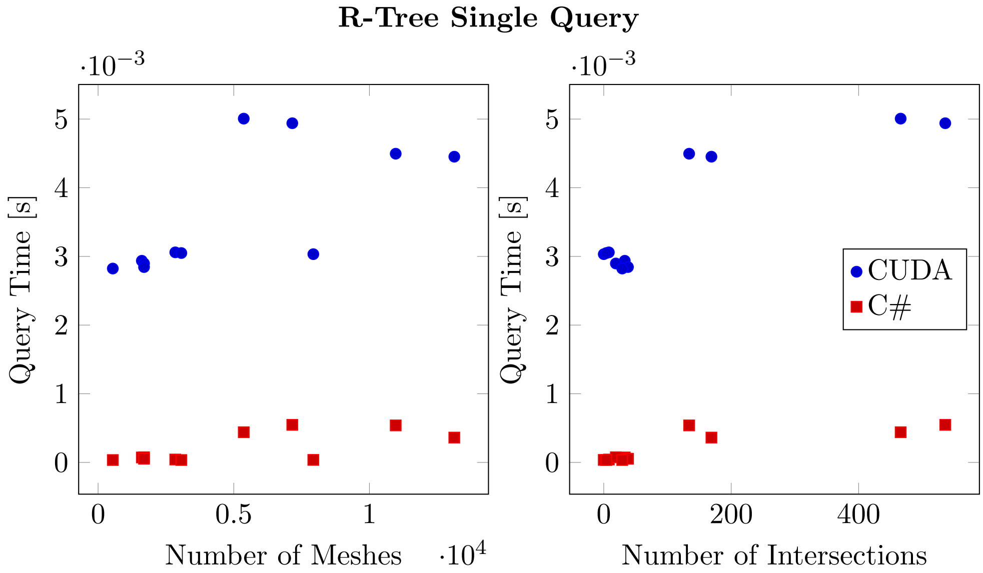


Figure 5.6: Scatter plots of the single query times on the global meshes R-tree. For both methods, there is a slightly positive correlation between query time and the number of intersections.

mesh sets. The number of candidate *TrianglePairs* again depends on the number of candidate meshes.

The C# test shows almost identical performance to that without prior candidate selection in figure 5.5. This corresponds with its intelligent approach that rapidly disqualifies all distant meshes completely, as described above.

5.3.4 All Mesh Intersections Query

Finally, all intersections between any meshes were determined. This is largely a presentation of CUDA's scaling behaviour when executing several queries in parallel. The intersection criterion used here is the same as for the single mesh search.

This test was only performed using lists of candidates created with an R-tree. As shown above, the CUDA results would be much worse in a full mesh search, while the C# ones would remain largely unaffected.

Thus the first analyzed runtimes here (see figure 5.8) were those of the multiple R-tree queries selecting a candidate list for every single object in the entire examined scenery. The quadratic increase of query time for both methods with respect to the number of meshes can be explained with the double impact of this value: It describes the R-tree size (including the statistical number of results) as well as the number of individual searches in it.

However, the results still vary strongly from a perfectly quadratic line. Each model seems to have individual properties, which affect the query time of each

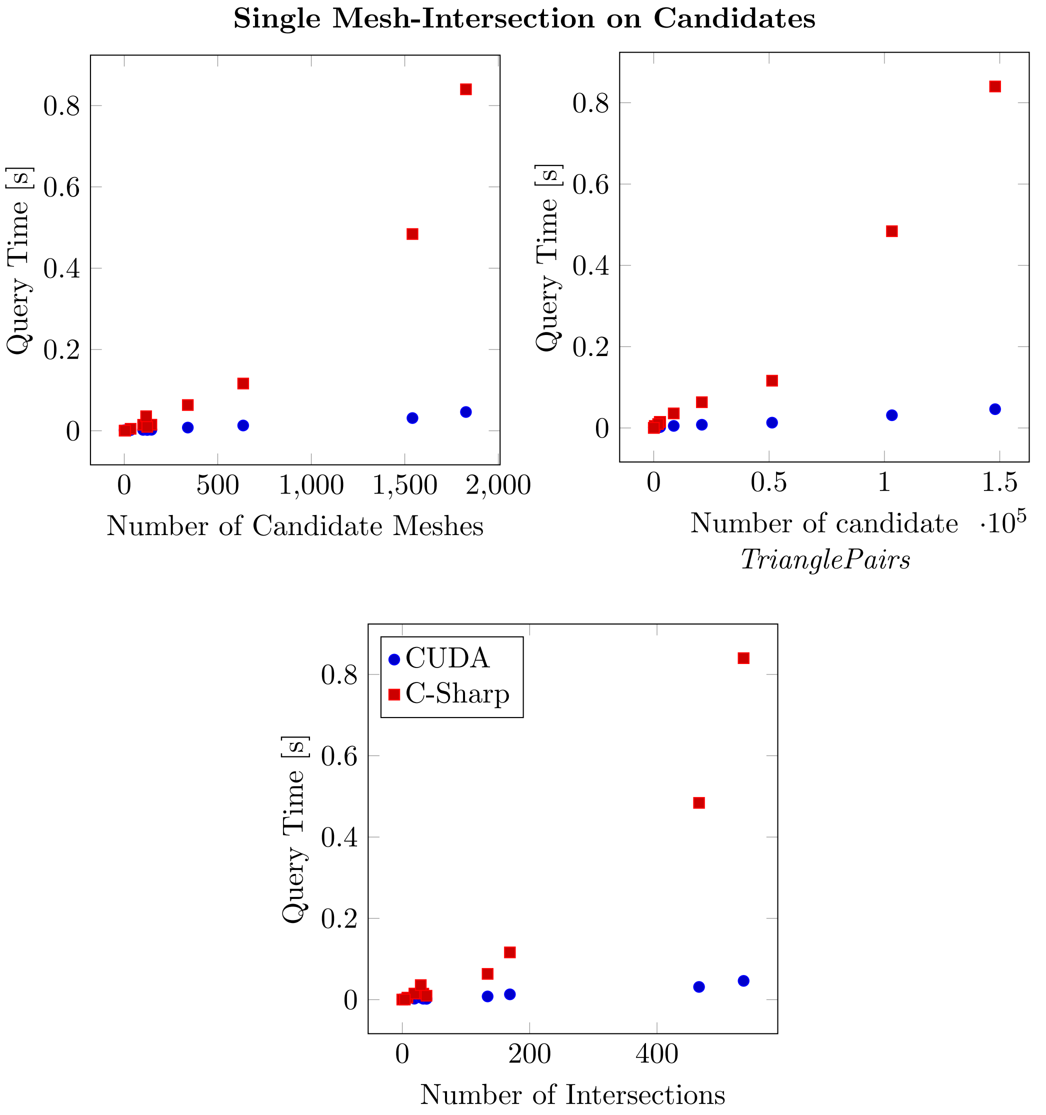


Figure 5.7: Scatter plots of the single mesh intersection query times when previously using the global meshes R-tree to reduce the number of checked meshes. The number of candidate *TrianglePairs* is calculated from using the number of candidate meshes and the average number of *TrianglePairs* per mesh for each model. The query times scale similarly for all three examined parameters.

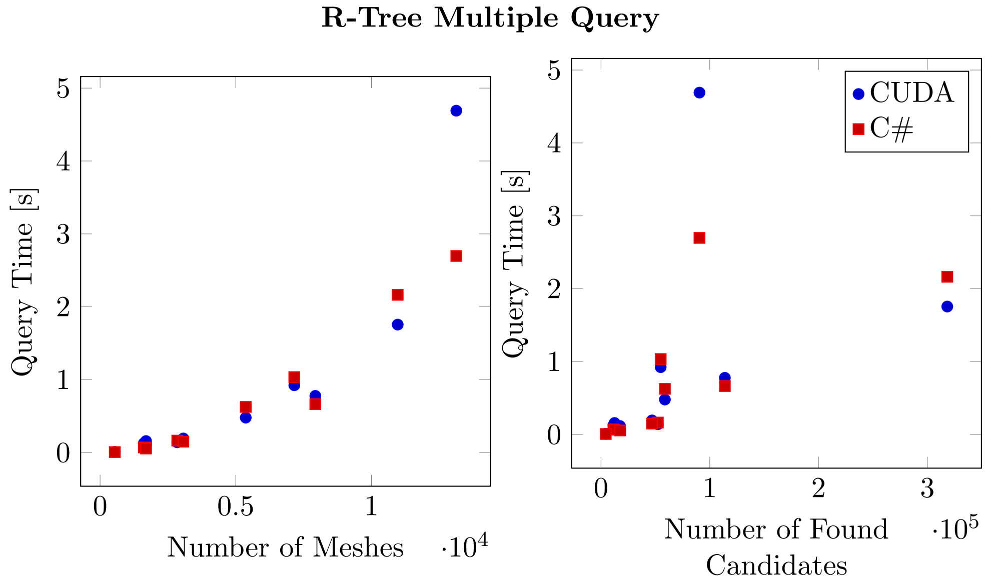


Figure 5.8: Scatter plots of the query times for the intersection candidates of every mesh on the global meshes R-tree. The dependence of both methods' runtimes on the number of meshes appears to be almost quadratic, while that on the number of intersections does not show a very clear trend.

method significantly. Even the number of found candidates only has a minor impact here.

The inconsistent scaling of the CUDA queries might be caused by the overflow treatment: When a thread block finds more nodes than it can handle in one level, another search is started with a separate block assigned to each of the previous nodes. Every time this is done, the corresponding parameters have to be shuffled in C#. Additionally, the overhead caused by their copying to the device and the kernel run has to be accepted. If this process has to be done once for one model file, but several times for another, it can impact the resulting query time in CUDA significantly.

Finally, the acquired candidate lists were optimized as described in figure 4.6. Then the mesh intersection tests were run, yielding the results shown in figure 5.9. The number of candidates alone is evidently no reliable indicator of the required computational effort. The number of candidate *TrianglePairs*, which is extrapolated from each model's average number of *TrianglePairs* per mesh, is a better indicator here. However, it shows no clearly trending curve for any of the two examined methods either.

5.3.5 Observed Trends

When comparing the two implemented algorithms, it can be observed that the R-tree query performance showed no or only minor improvements on the GPU over the

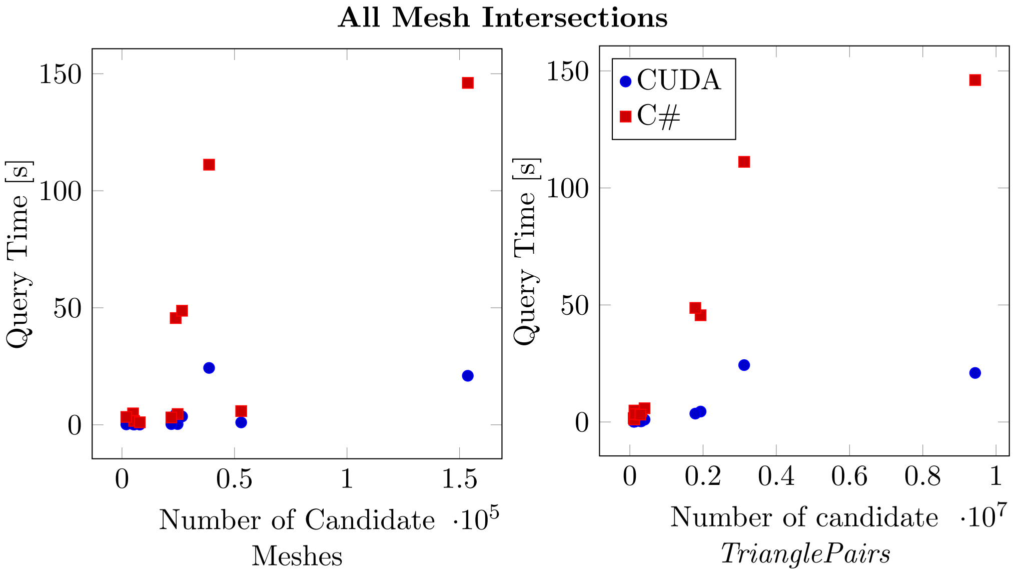


Figure 5.9: Scatter plots of the all mesh intersections query times. The query times increase with the number of candidate *TrianglePairs*, but no consistent correlation model can be deduced from the data.

CPU. However, the mesh intersection test generally ran faster on the GPU, at least when candidate preselection was done. This showed that an R-tree of all objects in a model should be employed in combination with the CUDA mesh intersection tester to achieve optimal results. The runtime sum from this scheme is shown in figure 5.10.

It can be observed that the combined query consistently performed better on the GPU. However, this direct comparison does not carry much significance because the test can not really be considered fair (see section 5.1). The much more important information is the trend of the computing time for each platform: Both clearly have a direct dependence on the number of found intersections. The CUDA results form a linear function with a rather flat slope. The C# ones seem to increase almost quadratically, which appears unlikely from the way the sequential implementations are programmed. In any case, the slope of the sequential calculations is much steeper than that of the parallel ones. For larger models – which is the only application where a different implementation would be sufficiently rewarding – the runtimes can thus be expected to be shorter by orders of magnitude on the GPU.

It has to be noted that the above tests were all done under the assumption that the initial creation of the described data structures would be necessary. As such, they were not added to the respective query times. However, not all of the C# R-trees of *TrianglePairs* would have to be created for a single mesh intersection test.

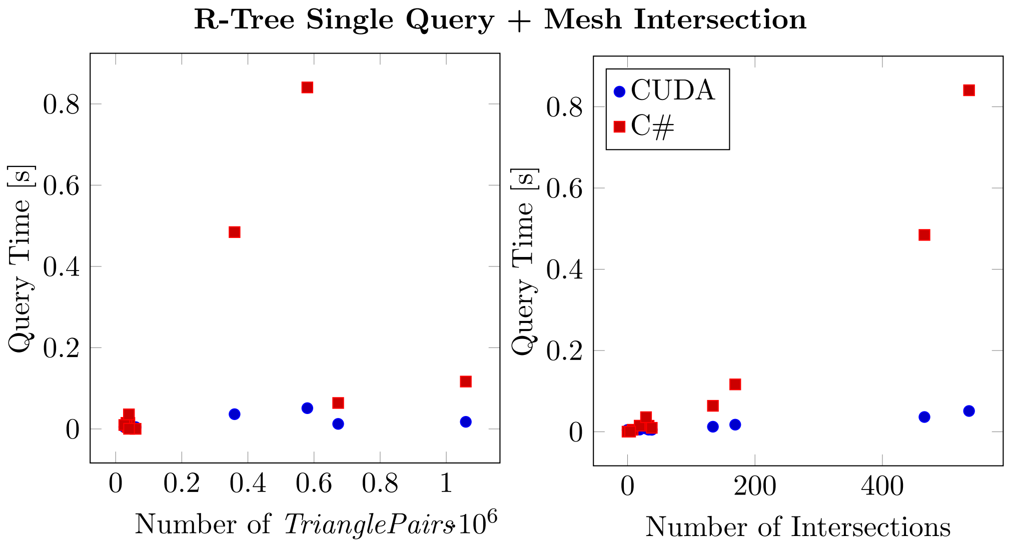


Figure 5.10: Scatter plots of the R-tree single query in combination with the mesh intersection on the thus obtained candidates. The plot shows the sums of the runtimes. CUDA consistently performs better in this scheme than C#, while both implementations show a clear dependence on the number of found intersecting meshes.

Chapter 6

Conclusion and Outlook

In this thesis, the performance of queries on the R-tree spatial indexing structure, as well as the intersection checking of triangle meshes was compared for two architectures: A single CPU core programmed in C# and a CUDA-capable GPU. The conducted tests demonstrated that GPU programming is beneficial for spatial BIM queries, but only for selected queries and especially for larger models.

For the R-tree search, the sequential implementation was generally faster. However, multiple R-tree queries could be executed reasonably fast with the CUDA implementation, because it takes advantage of this additional high level of parallelism. The mesh intersection test, which in fact performs an enormous number of triangle overlap analyses, also runs faster on CUDA. As it exhaustively checks every input mesh combination, these have to be preselected by their bounding boxes to obtain reasonable scaling. This is a promising application of the R-tree.

The performance analysis was performed on a workstation PC with an NVIDIA GPU released in late 2010 and a CPU from early 2011. Repeating it with current hardware would probably improve the results of the CUDA implementation. This is because current GPUs feature multiple times as many SMs as the one used for the analysis. All CUDA programs which run a large grid of many blocks can thus be expected to run faster on newer devices in proportion to the increased number of SMs. The latest NVIDIA workstation GPU is the Tesla K40 and features 2880 CUDA cores, which is 30 times as many as were available for this benchmark. The CPU single core performance has increased only by a much smaller factor since 2011.

Recent CUDA devices have also introduced a number of new features, which might be useful for more efficient implementations of the described algorithms. Among the most promising ones is *Dynamic Parallelism* (NVIDIA Corporation, 2014b, p. 134 ff.). This lets CUDA kernels dynamically launch new grids on the device, without returning control to the host. It might be especially useful for the R-tree query, where the amount of computations that need to be done can not be sufficiently estimated beforehand.

Another interesting idea would be to do all the computations only with single floating-point precision. As shown in figure 1.2, the theoretical peak performance is much higher this way, especially for GPUs. These are highly optimized for a large number of low-precision operations, because these are commonly necessary for graphics output. Thus the CUDA implementations might benefit more from performing only single precision calculations than the CPU ones.

To provide a fairer comparison, an analysis of how well the runtime scales when distributing the program on several parallel CPU threads should be conducted. This is a promising option because today's high end processors already have up to a dozen and more cores. This number can be expected to increase further in the next years, as this is the easiest way for CPU vendors to improve performance.

Appendix A

Remarks on ManagedCUDA

There is a tutorial on the basic workings of and how to set up a managedCUDA project within Visual Studio at Kunz (2014). Most classes and methods exposed by this library have documentation comments within their code, which can easily be accessed using Visual Studio's IntelliSense.

Here are some further remarks in addition to that:

- ManagedCUDA 6.5 has a bug treating device variables of Boolean type. In the early development of the programs for this thesis, large Boolean arrays were used to store the results of the R-tree queries and mesh intersection tests on the device. When these were copied back to the host immediately after a kernel run, an unspecified error occurred sporadically, which crashed the program.

  The risk of this happening could be reduced by producing a time gap between kernel run and the copy operation, e.g. by performing a dummy copy operation on one of the input arrays. However, the problem was only solved completely when using full integer variables to store this Boolean information. These do not cause any errors.

  This is probably due to the fact that managedCUDA extends Boolean variables to a size of four bytes per element. This facilitates coalescing memory accesses, which need that as a minimum word size (NVIDIA Corporation, 2014b, p. 187).

- When a CUDA kernel is run through managedCUDA, the Nsight debugger cannot be used on it. Thus there is no benefit from compiling it in debug mode. However, the performance is multiple times faster without debug symbols included. This is why the kernels for managedCUDA should always be compiled in release mode.

Appendix B

Digital Files

The CD that is handed in with this thesis contains the following files:

- This thesis as a PDF file, with internal and external hyperlinks.
- The developed Visual Studio 2012 solution. This is located in the `ParallelBimQueries` folder and includes the IFC files on which the benchmarks were run in `Data\Ifc\`. Its three projects are the following:

  CudaKernels This is the CUDA project with the two kernel files and the local headers which they need. They are compiled to two PTX files, which can be loaded by managedCUDA.

  QL4BIM2015 This contains the code of the spatial queries developed for QL4BIM System.

  - The `CUDA` folder holds the additional classes developed for this thesis, where CUDACaller is the implementation of some test queries, including the benchmark described above. The CUDA kernel files are loaded in the CUDAMeshIntersector and the CUDARTree class constructors.
  - `Spatial\Topology\MeshIntersector.cs` is where the intersection criterion had to be set, e.g. that inner with inner and outer with outer are both required.
  - In `Spatial\Topology\TriangleIntersector.cs`, small changes to the C#-implementation of the triangle intersection test from Möller (1999) were made.
  - `Spatial\IndexedFaceSet.cs` was modified to prevent the inclusion of degenerate triangles in the created *TrianglePairMeshes*.
  - In `Spatial\Indexing\RTree\RTreeQuery.cs`, the intersection check between the items of found leaves and the search box was removed,

because this requires an intersector for each new item type and is not part of the standard R-tree query.

QL4BIMClient This is the graphical interface, from where the functions of the CUDACaller class are run. For this, buttons were added in `MainWindow.xaml` and functionality assigned to them in `MainWindow.xaml.cs`.

- The benchmark results as a table in `Benchmark.csv`.

Bibliography

Autodesk, Inc. (2002). *White Paper: Building Information Modeling.* URL: http://www.laiserin.com/features/bim/autodesk_bim.pdf.

Beckmann, Norbert et al. (1990). “The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles”. In: *SIGMOD Rec.* 19.2, pp. 322–331.

Behrmann, Jan-Hendrik (2007). *OpenGL und die Fixed-Function-Pipeline.* Tech. rep. URL: https://wwwcg.in.tum.de/fileadmin/user_upload/Lehrstuehle/Lehrstuhl_XV/Teaching/WS07_08/Seminar/talks/Jan-Hendrik_Behrmann.pdf.

Borrmann, André, Stefanie Schraufstetter, and Ernst Rank (2009a). “Implementing Metric Operators of a Spatial Query Language for 3D Building Models: Octree and B-Rep Approaches”. In: *Journal of Computing in Civil Engineering* 23.1, pp. 34–46.

Borrmann, André and Ernst Rank (2009b). “Specification and Implementation of Directional Operators in a 3D Spatial Query Language for Building Information Models”. In: *Advanced Engineering Informatics* 23.1, pp. 32–44.

— (2009c). “Topological Analysis of 3D Building Models using a Spatial Query Language”. In: *Advanced Engineering Informatics* 23.4, pp. 370–385.

Daum, Simon and André Borrmann (2013). “Boundary Representation-Based Implementation of Spatial BIM Queries”. In: *Proc. of the EG-ICE Workshop on Intelligent Computing in Engineering.* Vienna, Austria.

— (2014). “Processing of Topological BIM Queries using Boundary Representation Based Methods”. In: *Advanced Engineering Informatics.*

Dennard, Robert H. et al. (1974). “Design Of Ion-implanted MOSFETs with Very Small Physical Dimensions”. In: *IEEE Journal of Solid State Circuits* 9.5, p. 256.

Du, Peng et al. (2012). “From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming”. In: *Parallel Computing* 38.8, pp. 391–407.

Eastman, Chuck et al. (2011). *BIM Handbook: A Guide to Building Information Modeling for Owners, Managers, Designers, Engineers and Contractors.* Wiley. ISBN: 9780470541371.

ECMA (2012). *Common Language Infrastructure (CLI) (Standard ECMA-335).* Ecma International. URL: http://www.ecma-international.org/publications/standards/Ecma-335.htm.

Egenhofer, Max J. and Robert D. Franzosa (1991). "Point-set topological spatial relations". In: *International journal of geographical information systems* 5.2, pp. 161–174. URL: http://www.tandfonline.com/doi/abs/10.1080/02693799108927841.

Fang, Jianbin, Ana Lucia Varbanescu, and Henk Sips (2011). "A Comprehensive Performance Comparison of CUDA and OpenCL". In: *2011 International Conference on Parallel Processing.* Taipei City: IEEE, pp. 216–225.

Galloy, Michael (2013). *CPU vs GPU performance.* URL: http://michaelgalloy.com/2013/06/11/cpu-vs-gpu-performance.html (visited on 12/20/2014).

Galoppo, Nico et al. (2005). "LU-GPU: Efficient Algorithms for Solving Dense Linear Systems on Graphics Hardware". In: *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing.* IEEE Computer Society.

Guttman, Antonin (1984). "R-trees: A Dynamic Index Structure for Spatial Searching". In: *ACM SIGMOD Record* 14.2, pp. 47–57.

Haveraaen, Magne and Hogne Hundvebakke (2001). "Some Statistical Performance Estimation Techniques for Dynamic Machines". In: *Norsk Informatikkkonferanse 2001.* Trondheim, pp. 176–185.

Kamel, Ibrahim and Christos Faloutsos (1992). "Parallel R-Trees". In: *Proceedings of the 1992 ACM SIGMOD international Conference on Management of Data.* New York, New York, USA: ACM Press, pp. 195–204.

Karimi, Kamran, Neil G. Dickson, and Firas Hamze (2010). "A Performance Comparison of CUDA and OpenCL". In: *arXiv* abs/1005.2. URL: http://arxiv.org/abs/1005.2581.

Khronos Group. *OpenGL ES 2.X - The Standard for Embedded Accelerated 3D Graphics.* URL: https://www.khronos.org/opengles/2_X/ (visited on 11/25/2014).

Kunjir, Mayuresh and Aditya Manthramurthy (2009). *Using Graphics Processing in Spatial Indexing Algorithms.* Tech. rep. Indian Institute of Science.

Kunz, Michael (2014). *ManagedCUDA Project Page.* URL: http://managedcuda.codeplex.com/ (visited on 12/14/2014).

Larsen, E. Scott and David McAllister (2001). "Fast Matrix Multiplies Using Graphics Hardware". In: *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*. New York, New York, USA: ACM, p. 55.

Lee, Victor W. et al. (2010). "Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU". In: *SIGARCH Computer Architecture News* 38.3, pp. 451–460.

Luo, Lijuan, Martin D. F. Wong, and Wen-mei Hwu (2010). "An Effective GPU Implementation of Breadth-First Search". In: *Proceedings of the 47th Design Automation Conference*. New York, New York, USA: ACM, pp. 52–55.

Luo, Lijuan, Martin D. F. Wong, and Lance Leong (2012). "Parallel Implementation of R-Trees on the GPU". In: *Design Automation Conference (ASP-DAC), 2012 17th Asia and South Pacific*. IEEE. IEEE Computer Society, pp. 353–358.

McMenamin, Adrian (2013). *The end of Dennard scaling*. URL: https://cartesianproduct.wordpress.com/2013/04/15/the-end-of-dennard-scaling/ (visited on 12/20/2014).

Meijer, Erik, Brian Beckman, and Gavin Bierman (2006). "LINQ: Reconciling Objects, Relations and XML in the .NET Framework". In: *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*. Chicago, p. 706.

Moore, Gordon E. (1965). "Cramming more components onto integrated circuits". In: *Electronics* 38.8, pp. 114–117.

Möller, Tomas (1997). "A Fast Triangle-Triangle Intersection Test". In: *Journal of Graphics Tools* 2.2, pp. 25–30.

— (1999). *Triangle/Triangle Intersection Test Routine*. URL: http://fileadmin.cs.lth.se/cs/Personal/Tomas_Akenine-Moller/code/opttritri.txt.

NVIDIA Corporation (2009). *Whitepaper - NVIDIA's Next Generation CUDA Compute Architecture: Fermi*. URL: http://www.nvidia.com/content/pdf/fermi_white_papers/nvidia_fermi_compute_architecture_whitepaper.pdf.

— (2014a). *CUDA FAQ*. URL: https://developer.nvidia.com/cuda-faq (visited on 12/07/2014).

— (2014b). *NVIDIA CUDA C Programming Guide*. Version 6.5. URL: http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.

Owens, John D. et al. (2008). "GPU Computing". In: *Proceedings of the IEEE* 96.5, pp. 879–899.

Preshing, Jeff (2012). *A Look Back at Single-Threaded CPU Performance*. URL: http://preshing.com/20120208/a-look-back-at-single-threaded-cpu-performance/ (visited on 12/20/2014).

Singer, Graham (2013). *The History of the Modern Graphics Processor*. URL: http://www.techspot.com/article/650-history-of-the-gpu/ (visited on 09/23/2014).

Sutter, Herb (2004). *The Free Lunch Is Over - A Fundamental Turn Toward Concurrency in Software*. Updated in August 2009. URL: http://www.gotw.ca/publications/concurrency-ddj.htm (visited on 12/20/2014).

University of Auckland (2012). *Open IFC Model Repository*. URL: http://openifcmodel.cs.auckland.ac.nz/ (visited on 12/07/2014).

Wikimedia Commons (2007). *Dolphin triangle mesh.png*. URL: http://commons.wikimedia.org/wiki/File:Dolphin_triangle_mesh.png (visited on 01/23/2015).

— (2008). *Wire frame.svg*. URL: http://commons.wikimedia.org/wiki/File:Wire_frame.svg (visited on 01/23/2015).

— (2010a). *Block-thread.svg*. URL: http://commons.wikimedia.org/wiki/File:Block-thread.svg (visited on 01/23/2015).

— (2010b). *Memory.svg*. URL: http://commons.wikimedia.org/wiki/File:Memory.svg (visited on 01/23/2015).

Wilt, Nicholas (2013). *The CUDA Handbook: A Comprehensive Guide to GPU Programming*. Pearson Education. ISBN: 9780321809469.

Yampaka, Tongjai and Prabhas Chongstitvatana (2012). "Spatial Join with R-Tree on Graphics Processing Units". In: *IC2IT*. URL: http://www.cp.eng.chula.ac.th/%7Epiak/paper/2012/final_SpatialJoin_IC2IT2012.pdf.

You, Simin and Jianting Zhang (2013). "GPU-based Spatial Indexing and Query Processing Using R-Trees". In: *Proceedings of the 2nd ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data*, pp. 23–31.

Some figures contained in this thesis were published under the following licenses, which require the links to their descriptions to be provided:

CC BY-NC-SA 2.5:	http://creativecommons.org/licenses/by-nc-sa/2.5/legalcode
CC BY-NC-ND 3.0:	http://creativecommons.org/licenses/by-nc-nd/3.0/legalcode
CC BY 3.0:	http://creativecommons.org/licenses/by/3.0/legalcode
CC BY-SA 3.0:	http://creativecommons.org/licenses/by-sa/3.0/legalcode

Declaration of Originality

I hereby declare that this thesis is my own unaided work. All direct or indirect sources used are acknowledged as references.

I am aware that my work may be examined for the unmarked use of someone else's intellectual property by means of plagiarism recognition software.

This paper was not previously presented to another examination board and has not been published.

I confirm that my thesis in electronic form is identical to the printed version.

Place, Date	Signature