KTH ROYAL INSTITUTE
OF TECHNOLOGY

Degree Project in Computer Science and Engineering

Second cycle, 30 credits

A domain-specific language for geospatial computations on the GPU

A study on the level of functionality and the performance gains

LOUISE TIDESTAV

Stockholm, Sweden, 2024

A domain-specific language for geospatial computations on the GPU

A study on the level of functionality and the performance gains

LOUISE TIDESTAV

Degree Programme in Computer Science and Engineering
Date: May 29, 2024

Supervisors: Ivy Bo Peng, Jakob Aasa
Examiner: Stefano Markidis
School of Electrical Engineering and Computer Science
Host company: Carmenta Geospatial Technology
Swedish title: Ett domänspecifikt språk för geospatiala beräkningar på GPU:n
Swedish subtitle: En studie på användbarhet och prestandaförbättringar

Abstract

This thesis explores how a domain-specific language (DSL) for simple geospatial operators on the GPU can be developed, and evaluates the level of functionality and performance of such a DSL. The purpose of such a DSL is to simplify implementation of geospatial operators on the GPU, in order to increase productivity and performance.

A DSL was designed and implemented according to the geospatial domain. The level of functionality of the DSL was evaluated based on what geospatial operators are supported and with the Framework for Qualitative Assessment of DSLs (FQAD). The level of functionality was evaluated to high for the DSL, as simple geospatial operators are supported and the FQAD evaluation determined that the DSL was effective. To evaluate the performance gains, three interpreters for the DSL were implemented, one sequential on the CPU, one parallel on the CPU and one on the GPU. The execution time for five geospatial operators and algorithms was measured for different problem sizes for the three interpreters. The GPU interpreter achieved large speedups for large problem sizes compared to the sequential CPU interpreter. The GPU interpreter achieved speedups for four out of five operators compared to the parallel CPU interpreter for large problem sizes. In conclusion, this thesis demonstrates the potential of a domain-specific language for geospatial computations on the GPU.

Keywords

Domain-specific language, Geospatial, Graphics processing unit

Sammanfattning

Denna uppsats undersöker hur ett domänspecifikt språk (DSL) för enkla geospatiala operatorer på GPU:n kan utvecklas, och utvärderar användbarheten och prestandan för ett sådant DSL. Syftet med ett sådant DSL är att förenkla implementationen av geospatiala operatorer på GPU:n, för att öka produktivitet och prestanda.

Ett DSL designades och implementerades utefter den geospatiala domänen. Användbarheten av DSL:en utvärderades baserat på vilka geospatiala operatorer som stöds och med Framework for Qualitative Assessment of DSLs (FQAD). Användbarheten utvärderades till hög för DSLen, eftersom enkla geospatiala operatorer stöds och FQAD utvärderingen kom fram till att DSLen var effektiv. För att utvärdera prestandaförbättringar implementerades tre interpreterare, en sekventiell på CPUn, en parallel på CPUn och en på GPU:n. Körtiden för fem geospatiala operatorer och algorithmer mättes för olika problemstorlekar för de tre interpreterarna. GPU implementationen hade bättre prestanda i jämförelse med den sekventiella CPU implementationen för stora problemstorlekar. GPU implementationen hade bättre prestanda för fyra av fem operatorer i jämförelse med den parallela CPU implementationen för stora problemstorlekar. Sammanfattningsvis visar detta projekt på potentialen för ett domänspecifikt språk för geospatiala beräkningar på GPU:n.

Nyckelord

Domänspecifikt språk, Geospatial, Grafikprocessor

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

The geospatial domain is concerned with objects, events, or features with a location relative to the surface of the Earth [1]. Geospatial computations are time demanding, due to how voluminous geospatial data is. An example of a problem of interest within the geospatial domain is to compute which of a large set of points lie within a specific geographic area. If the set of points is large, sequentially determining this for each point can be very time-consuming. Applying the same operators to large datasets is typical for geospatial computing. One approach to accelerate geospatial computation is to utilize the Graphics Processing Unit (GPU). GPUs were initially designed for computer graphics but have recently demonstrated their capabilities in scientific computing. GPUs are designed for parallel processing and can accelerate any parallelizable task. Many geospatial operators are trivially parallelizable, and therefore suitable for GPU acceleration. Utilizing the GPU for tasks traditionally executed on the CPU is known as General-purpose computing on graphics processing units (GPGPU).

However, GPGPU introduces several challenges related to parallel programming and GPU programming. Adapting geospatial algorithms to GPU execution can be both time-consuming and error-prone. One solution allowing developers to write parallel programs without prior knowledge of GPU programming is a Domain-Specific Language (DSL). A DSL is a small computer language specialized for a certain task. The purpose of a DSL is to reduce program complexity by providing domain-specific abstractions.

Currently, there exists several DSLs for GPU programming in different domains [2], [3]. However, there exists no DSL targeting geospatial computation on the GPU. Therefore, this project will explore how such a DSL can be developed. The project will also evaluate the level of functionality and the performance of the DSL.

1.1 Background

1.1.1 Domain-specific languages

A domain-specific language is a small programming language targeted at a specific problem or domain. A DSL provides domain-specific abstractions to simplify development and increase efficiency. There exist two main categories of DSLs: external and embedded (internal) DSLs [4]. An external DSL defines its own, tailored syntax while an embedded DSL resides within an already existing language. An embedded DSL instead extends a language by providing domain-specific abstractions. An external DSL is more time consuming to implement, but can be easier to use for domain experts once implemented. An embedded DSL is more suitable when DSL constructs should be used together with constructs of a General-Purpose Language (GPL). A DSL can be either compiled or interpreted. A compiled DSL generates code in a target language, while an interpreted DSL simply executes the domain constructs.

1.1.2 CUDA

CUDA is a parallel computing platform developed by NVIDIA to enable GPGPU on NVIDIA GPUs [5]. The CUDA toolkit contains libraries, a C/C++ compiler, a runtime library, and tools for debugging and optimization. CUDA C++ extends C++ by allowing developers to define *kernels*. Kernels are functions which are executed in parallel by different threads on the GPU. Programming in CUDA presents multiple challenges for developers. Challenges arising from parallel programming such as race conditions and deadlocks needs to be addressed, as well as challenges related to CUDAs thread and memory hierarchy.

1.1.3 The geospatial domain

The geospatial domain is concerned with objects, events or features with a location relative to the surface of the Earth. The geospatial domain differs from other domains in several respects, both related to geospatial data and geospatial analysis [1]. Geospatial data is interesting in at least two ways: it is multidimensional, e.g. two or three values are needed to specify a location, and it can be represented in different ways in the computer. Due to this, geospatial analysis requires many special methods. Another aspect that also impacts geospatial analysis is how voluminous geospatial data is.

1.2 Problem

Processing geospatial data is time-consuming. Mitigating this by writing parallel programs for the GPU is challenging and error-prone. DSLs have been used to simplify GPU programming in different domains [2], [3]. However, there exists no DSL targeted at geospatial computations on the GPU.

1.2.1 Research question

- What is the level of functionality of a domain-specific language for simple geospatial operators on a Nvidia GPU?
- What are the performance gains for a domain-specific language for simple geospatial operators on a Nvidia GPU?

1.3 Purpose

The goal and central message of this thesis is in the geospatial computation domain, more specifically, to understand how DSLs can be used to simplify GPU programming in the domain.

1.4 Goals

The goal of this project is to explore how a small DSL for geospatial computations can be developed, and what the level of functionality and performance gains for such a DSL are. This goal can be divided into the following sub goals:

1. Identify characteristics and understand the requirements arising from the geospatial domain.
2. Design and implement a DSL for a small set of simple geospatial operators.
3. Evaluate the level of functionality and performance of the DSL.

1.5 Research Methodology

The project was divided into three phases. First, a literature study was performed. DSLs for GPU programming in different domains as well as approaches for accelerating geospatial computation was studied. Then, a DSL was developed. The development of a DSL can be divided into five phases: decision, analysis, design, implementation and deployment [6]. The DSL was developed according to the analysis, design and implementation phases. Finally, the level of functionality and the performance of the DSL was evaluated.

1.6 Delimitations

The DSL will focus on abstractions of collections of vector data and rasters. The DSL will only support a small set of simple geospatial operators over collections of vector data and rasters. The DSL will be implemented with Thrust, a parallel algorithms library in CUDA, instead of hand-tuned CUDA.

The evaluation will evaluate the level of functionality of the DSL as well as the performance of the DSL. The performance of the DSL will only be measured for a small set of operators and algorithms. A user study will not be a part of the evaluation of the DSL.

1.7 Structure of the thesis

Chapter 2 presents relevant background information about DSLs, CUDA and the geospatial domain. The chapter also presents related work. Chapter 3 presents the methodology, including the different phases of developing a DSL. The chapter also presents the evaluation of the DSL. Chapter 4 presents the results obtained during the evaluation and the analysis of the results. Finally, Chapter 5 presents the conclusions and reflects on the project, its limitations and possible continuations.

Chapter 2

Background

This chapter presents the background. In sections 2.1 to 2.3, compilers, DSLs, and the development of DSLs are presented. Section 2.4 introduces GPU programming and CUDA. In Section 2.5, the geospatial domain and geospatial computations are discussed. Finally, Section 2.6 presents related work by discussing approaches for simplifying GPU programming and accelerating geospatial computations.

2.1 Compilers

Compilers for GPLs typically consist of a front-end, an optimization phase, and a back-end. The front-end consists of lexical analysis (lexing) and syntactic analysis (parsing). Lexing is the process of transforming text into a sequence of predefined lexical tokens, with known meaning. Lexers can be manually constructed by consuming one character at a time until a token is identified. Lexers can also be generated by lexer generators, by specifying regular expressions (regex) for the lexical tokens. Parsing is the process of creating an Abstract Syntax Tree (AST), representing the program, from sequences of tokens. Parsing can be performed manually, by consuming tokens until a specific token sequence is identified. Parsers can also be generated by parser generators from a specification defining the meaning of different token sequences.

From the AST, an intermediate representation (IR) is created. Different optimizations can be applied to the IR, to be able to later generate more efficient code. From the optimized IR, the back-end generates code in a target, low-level language.

2.2 Domain-specific languages

In contrast to GPLs, a DSL is a language targeted at a specific domain. The purpose of a DSL is to reduce program complexity by providing domain-specific abstractions. DSLs are easy to learn and use for domain experts, resulting in increased productivity.

2.3 Development of a DSL

The development of a DSL can be divided into five phases: decision, analysis, design, implementation and deployment [6]. Creating a DSL can be both hard and time consuming. Therefore, the decision phase is about identifying if a DSL is suitable for the problem domain. The final phase, the deployment phase is about deploying and using the DSL.

2.3.1 Domain analysis

The purpose of the domain analysis is to understand the requirements of the domain and to identify key concepts in the domain. This is usually done by talking to experts in the domain, reading technical documentation and analysing existing implementations for the problems in GPLs [6]. Two formal methods for domain analysis are Family-Oriented Abstractions, Specifications, and Translation (FAST)[7], and Feature Oriented Domain Analysis (FODA)[8]. An informal approach for domain analysis is presented by Wasowski and Berger [9], based on the following five questions:

- What is the purpose of the language? What are the use cases?
- Who are the key stakeholders and the intended users of the language?
- What are the key domain concepts that users care about?
- How are domain concepts related, and what are their relevant properties?
- What examples of language instances are available or can be prototyped?

The result of a domain analysis consists of domain terminology, domain concepts and identified characteristics of the domain [6]. This is used in the language design and implementation phases.

2.3.2 Language design

During the language design phase, it should be determined whether the DSL should be external or embedded. An external DSL defines its own syntax, allowing a syntax closer to the domain. An external DSL is more time-consuming to implement, but can be easier to use once implemented. An embedded language resides within an host language and is limited to the syntax of that language. Instead, an embedded language define domain-specific constructs and abstractions. An embedded DSL is easier to implement, but can be harder to use for domain experts. An embedded DSL can be suitable if the users of the DSL are already familiar with the host language.

What domain concepts and constructs the language should support and provide is also determined during the language design phase. The design should be described either formally or informally [6]. An informal design is composed of a description of the DSL in natural language and a set of example programs in the DSL. A formal design is often defined by a concrete syntax for the language. A concrete syntax defines the syntax of a language, and states how valid programs are structured.

2.3.3 Implementation

The implementation phase differs depending on whether the DSL is external or embedded. For external DSLs, the first phase is similar to implementing the front-end of a compiler for a GPL, e.g. lexing and parsing. The output from the parsing is an AST, later transformed into a *semantic model*, a model representing the domain constructs. A semantic model can also be created directly during parsing.

For embedded DSLs, the implementation phase consists of defining domain-specific data types and operators in the host language. These represent the semantic model. Data types and operators must be carefully defined, to ensure that they can be used as a language rather than as an API.

The result of the first phase of both an external and embedded DSL is a semantic model. The semantic model can then be compiled or interpreted. In the case of an interpreted DSL, the model is simply executed as it is traversed. For a compiled DSL, code is generated in a target language as the model is traversed. This might be suitable if the DSL should be possible to execute on different platforms. An interpreter is easier to implement and more common.

2.3.4 Evaluation

The Framework for Qualitative Assessment of DSLs (FQAD) presents ten basic characteristics that can be used for evaluation of a DSL [10]. Examples of characteristics are usability and productivity. Each characteristic contains sub-characteristics. Initially, the characteristics to evaluate are chosen and assigned an importance degree. The importance degree determines the minimum support level that is required for each sub-characteristic belonging to the characteristic. Then, the support level of each sub-characteristic is evaluated. The support-level of the sub-characteristics are then compared to their minimum level, determining the success level of the DSL, either incomplete, satisfactory, or effective.

2.4 Graphics processing units

Graphics processing units (GPUs) are specialized parallel hardware, initially designed to be used for accelerating computer graphics. The last twenty years, GPUs have been found to be highly useful for accelerating scientific computations.

2.4.1 GPGPU

GPGPU is the use of a GPU for tasks traditionally performed on the CPU. GPGPU utilizes the parallel execution enabled by the GPU to improve performance. GPGPU computing is especially useful for problems involving large amounts of data parallelism, e.g. when the same task is applied to large amounts of data in parallel. Figure 2.1 shows how data parallelism works. Examples of problems well-suited for GPU execution are large-scale matrix and vector operations.

2.4.2 CUDA

Compute Unified Device Architecture (CUDA) is a parallel computing platform developed by NVIDIA, enabling GPGPU on GPUs [5]. The CUDA toolkit includes GPU-accelerated libraries, a compiler (nvcc), debugging and optimization tools, and a runtime library. CUDA supports languages such as C and C++.

CUDA C++ extends C++ by allowing developers to extend programs written in C++ with kernels. Kernels are functions that are executed multiple

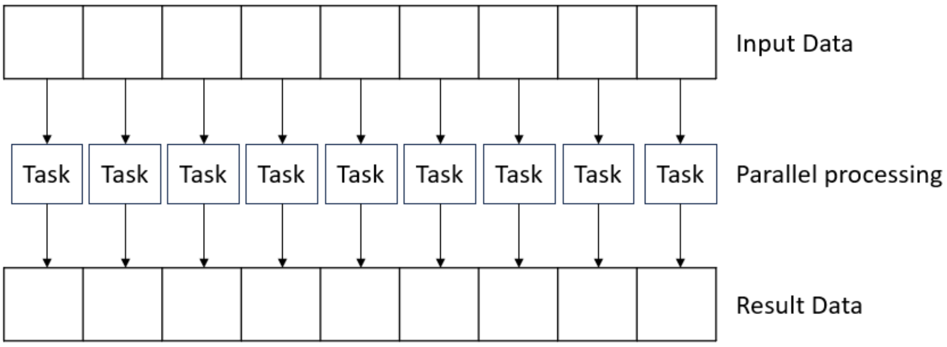


Figure 2.1: Data parallelism visualized.

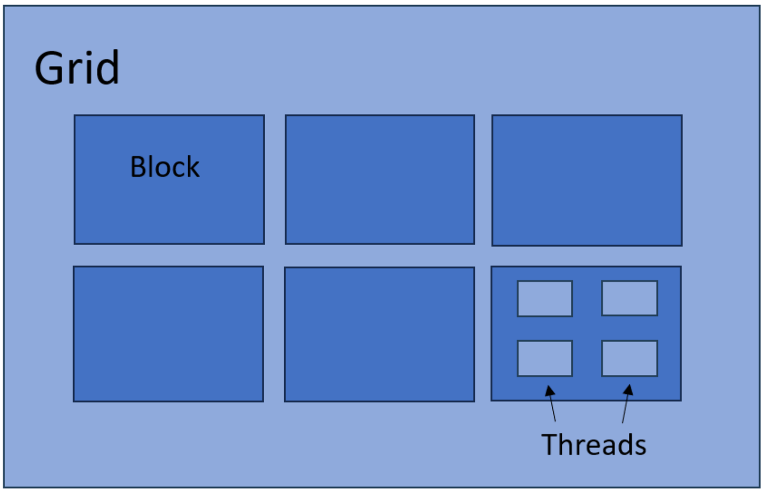


Figure 2.2: CUDAs thread hierarchy.

times in parallel by different *CUDA threads*. CUDA threads are organized into *thread blocks*. All threads within a thread block reside on the same streaming multiprocessor (SM) core, limiting the number of threads in one block. All threads within a thread block are assigned a unique thread ID, and can share and synchronize data with threads in the same thread block. Thread blocks are organized into *grids*. Grids and blocks can be either one, two, or three-dimensional. Figure 2.2 visualizes the thread hierarchy of CUDA.

A CUDA kernel is defined using the `__global__` declaration. The syntax for invoking a kernel is $<<< N, M >>>$, where N is the number of blocks and M is the number of threads per block. The variables `threadIdx`, `blockIdx` and `threadIdx` are built into CUDA. Listing 2.1, shows a modified example from [5] of how a kernel can be used for vector addition of two vectors of length N. Each thread in a thread block is executed in parallel on the streaming multiprocessor. Each thread block is required to execute independently, allowing thread blocks to be scheduled in any order (parallel or sequential), depending on the number of SMs of the GPU.

```
// Kernel definition
__global__ void VecAdd(float A*, float B*, float C*){
    // Compute thread ID
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    C[i] = A[i] + B[i];
}

int main(){
    ...
    // Kernel invokationg
    dim3 threadsPerBlock = ..;
    dim3 numBlocks = N / threadsPerBlock;
    VecAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

Listing 2.1: Vector addition using CUDA kernels.

The kernel function is executed on the GPU, called device, while the main program is run on the CPU, referred to as the host. The host and device have their own separate memory spaces, and memory must therefore be transferred between the host and the device. Each thread has a *private memory*. Each thread block has a *shared memory*, accessible for all threads in the block. All threads also have access to a *global memory*. Two additional read-only memories are also accessible for all threads: *constant memory* and *texture memory*.

2.4.3 Thrust

Thrust is a parallel algorithms library for CUDA C++ [11]. Thrust provides a high-level interface to CUDA with algorithms such as sort and reduce. Thrust also provides memory management by providing two vector containers, `host_vector` and `device_vector`, living in host memory and device memory respectively. All algorithms can be executed either on the GPU or in parallel on the CPU. This is possible by either specifying the execution policy or by using device or host vectors. Functions on device vectors are executed on the GPU, while functions on host vectors are executed in parallel on the CPU. Listing 2.2 shows the usage of host and device vectors and Thrust algorithms.

```
int main(void){
    // Create a host vector with 50 elements
    thrust::host_vector<int> h_vec(50);
    // Populate the vector with random numbers
    thrust::generate(h_vec.begin(), h_vec.end(), rand);
    // Copy the result to a device vector
    thrust::device_vector<int> d_vec = h_vec;
    // Sort the random numbers on the device
    thrust::sort(d_vec.begin(), d_vec.end());
    // Copy the result to a host vector
    return 0;
}
```

Listing 2.2: Thrust vectors and algorithms.

User-defined operators are also possible in Thrust. One example of when a user-defined operator is needed is the vector operation `y = a * x + y`, where `a` is a scalar, and `y` and `x` are vectors. For this operator, a solution would be to use two calls to the transform algorithm. The first call would compute `a * x` and temporary store the result. The second call could then add `y` to the temporarily stored result. Another, more efficient, solution is a user-defined functor. A functor is a class overloading the () operator, holding a state. Listing 2.3, a modified example from [12], shows how a functor can be used for the vector operation:

```
struct user_defined_functor{
    const float a; // State

    user_defined_functor(float _a) : a(_a) {} // Constructor

    __host__ __device__
        float operator()(const float& x, const float& y)
   const {
        // Overloading of the () operator
            return a * x + y;
        }
};

void user_defined(float A, thrust::device_vector<float>& X,
   thrust::device_vector<float& Y>){
    // Y <- A * X + Y
    thrust::transform(X.begin(), X.end(), Y.begin(), Y.begin
    (), user_defined_functor(A));
}
```

Listing 2.3: User-defined functor in Thrust

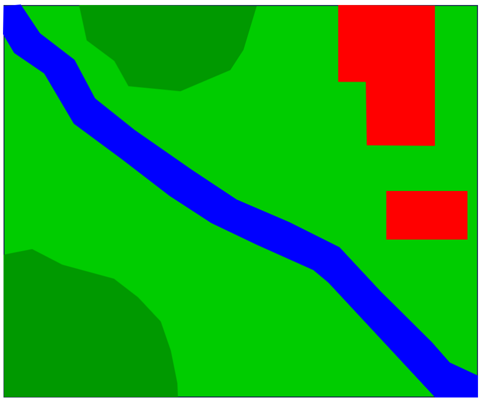


Figure 2.3: A landscape represented with vector data.

2.5 Geospatial data and analysis

Geospatial data is information that describes objects, events, or other features with a location relative to the Earth's surface. Such objects, events, or features can be described as entities or fields. Entities describe distinct objects, for example, buildings and roads. Fields are used to describe continuous surfaces, for example, elevation or rainfall. Many objects or features can be described as either entities or fields, depending on the purpose. A landscape can either be described by its entities (roads, rivers, buildings) or as a field where different colors represent different land cover.

2.5.1 Vector data

An entity can be represented as *vector data*. Vector data is represented by one or more (interconnected) coordinates, typically a point, line or polygon (representing an area). Figure 2.3 shows how a landscape can be represented with vector data. An entity is also described by its *attributes*. Attributes are key-value pairs holding additional information about an entity. Examples of attributes for an entity representing a city are name and population. An entity is also described by its relation to other entities, via its topology. Topology describes relationships that are preserved during distortion of space. For example, a shared border between countries is preserved when space is distorted.

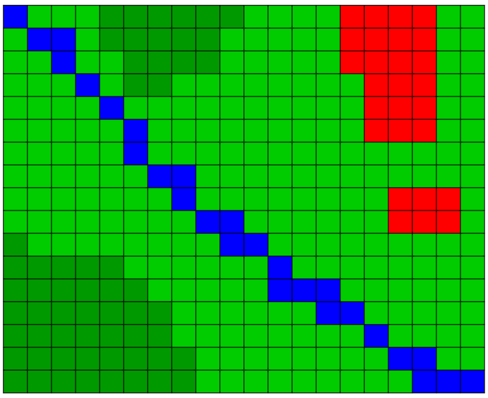


Figure 2.4: A landscape represented with a raster.

2.5.2 Raster data

Fields (continuous surfaces) can be represented as *raster data*. Raster data is represented as a grid of cells, where each cell represents an area. The value of the cell corresponds to the value of an attribute in that area. For an elevation raster, the value in each cell represents the elevation for the corresponding area. The spatial resolution of a raster determines how large area each cell corresponds to. Figure 2.4 shows how a landscape can be represented as a raster.

2.5.3 Spatial analysis on entities

Spatial analysis of entities is often related to the shape of the entity and the attributes of the entity. One of the most basic spatial analyses is calculating length of lines, perimeter of polygons, and areas of polygons [13]. These operations are used regularly, as they are needed in other more complex analyses. The length of lines is, for example, used when calculating the distance of different routes in a road network. These analyses are referred to as *measurements* [1]. A related concept is buffers. A buffer adds an offset to a point, a line or a polygon. When planning cities, buffers could be added around lakes to ensure that buildings and roads are not at risk of flooding. Buffers and similar operations are referred to as *transformations* [1].

Another common spatial analysis is *queries* [1]. Querying is about filtering entities based on attributes. A query could be used when a user wants to filter cities with a population greater than a certain value, or find restaurants within a specific distance.

How entities can be combined and how entities relate to each other is another field of interest within spatial analysis. Overlay operators combine information from two or more sets of spatial data to create a new set [13]. Polygon overlay is one overlay operator, where one polygon layer overlays another polygon layer, creating a new polygon layer. One type of polygon overlay is intersection, where the result is the areas where the two input polygons overlap. Union is another type of polygon overlay, where the result is the areas that at least one of the polygons covers. Subtract (erase) and clip are two other types of polygon overlays. For intersection and union, attributes are often combined and transferred to the resulting polygon. For erase and clip, only the attributes for one of the input layers are transferred to the resulting polygon. Line-in-polygon overlay and point-in-polygon overlay are two other overlay operators. Line-in-polygon and point-in-polygon overlay is about determining which polygon a specific line or point lies within. This is useful for example when assigning postal codes to buildings.

Network analysis is another area in spatial analysis. A network is composed of nodes and edges representing locations and paths between them [13]. Each path is typically assigned a value representing the cost of that path, often relating to the distance of the path. A common problem within network analysis is finding the shortest path between two nodes in the network. One algorithm for finding the shortest path is Dijkstra's algorithm. Another problem of high interest related to network analysis is the travelling salesman problem. The travelling salesman problem is about finding the shortest route which visits a specified set of nodes and then returns to the starting node.

2.5.4 Spatial analysis on fields

Spatial analysis can also be performed on fields. One of the most commonly used operators is map algebra. In map algebra, operations are applied to one or multiple rasters to create a new raster [14]. The operations are typically mathematical or boolean operations. Map algebra operations can be divided into three types: local-, focal-, and zonal operations [15]. Local operations are applied to corresponding cells in multiple layers. A local operation can be used to add together two rasters, by adding together the cells at the same position in the two rasters. Focal operations operate on a cell and its neighborhood. A focal operation can be used to compute the slope from a grid of elevation values. The values of the neighboring cells are used to calculate the new value of each cell. Zonal operations operate on regions of identical values. A zonal operator could be used to compute the mean of the rainfall for different land covers. The input to such an operator would be one raster representing the

Table 2.1: Overview of spatial operators.

Operators on entities
Measurements
Transformations
Spatial queries
Overlay operators
Network analysis

Operators on fields
Map algebra
Transformations
Measurements
Spatial interpolation

rainfall, and one raster representing land cover. All cells corresponding to the rainfall for one land cover will be used as input to one computation, and the result of that computation is written to all corresponding cells in the output raster.

One of the most common uses of rasters is to represent a *digital elevation model* (DEM) [1]. In a DEM, each cell represents the elevation of the Earth's surface. Common operations on a DEM are to compute the *slope* and *aspect*. The slope and aspect is computed based on the neighbouring cells, which are given different weights. The slope represents the rate of change of elevation, and the aspect is defined as the direction of the slope. These operators can also be classified as measurement operators.

Another area of interest related to spatial analysis on fields is spatial interpolation. Spatial interpolation predicts values where no sample is available [13]. The basis for interpolation is that values that are close together tend to be similar. One example of spatial interpolation is to create a DEM from a set of data points. Nearby known points will be used to predict the elevation for cells where the value is unknown. One common method for spatial interpolation is inverse distance weighting (IDW). In the IDW method, sample points are given different weight depending on the distance to the point that are being predicted. Table 2.1 presents an overview of the landscape of geospatial operators and analysis.

2.6 Related work

2.6.1 DSLs for image processing

There exist DSLs targeted at simplifying GPU programming for specific domains. One such DSL is HIPA^{CC}, targeting image processing [2].

The HIPA^{CC} framework consists of two main components: A DSL for image processing, and a source-to-source compiler (transpiler). The DSL is embedded in C++ and provides domain specific abstractions for image processing. The source-to-source compiler generates code in different languages, including CUDA. The source-to-source compiler is implemented as a Clang plugin, rewriting DSL constructs into the target language based on the AST. The generated code outperforms libraries and other DSLs for image processing. The DSL also results in increased productivity, compared to other approaches.

Another DSL for image processing is Halide [16]. In Halide, the algorithm (what is computed) is separated from the scheduling (how to run on a particular machine). This allows programmers to specify the algorithm, and then run the algorithm with different scheduling to achieve higher performance. In Halide, both the algorithm and the schedule are expressed using functional programming. The resulting code achieves state-of-the-art performance. Ragan-Kelley *et al.* [17] present a compiler optimizer for the Halide language, achieving large speedups compared to hand-tuned code in fewer lines of code.

2.6.2 hiCUDA

There also exists a number of source-to-source compilers for compiling high-level sequential code to high-level parallel code. One such source-to-source compiler is hiCUDA [18]. hiCUDA is directive-based, e.g. the programmer annotates how different sections of the code should be translated. hiCUDA is built with the Open64 compiler, by extending the front-end, adding a compiler pass using Open64 modules, and extending the code generator with a CUDA code generator. hiCUDA requires excessive use of directives, which forces the programmer to explicitly state which parts of the program should be translated, and how.

2.6.3 Mint

A similar source-to-source compiler is Mint, translating annotated C to CUDA C [19]. Mint is targeting stencil computations, e.g. updating array elements after some fixed pattern (a stencil). Mint makes use of five pragmas to denote the C program, stating how different parts should be translated. Mint is built using the ROSE framework. The ROSE framework consists of multiple front-ends, a midend enabling changes to the AST, and back-ends generating source code from the AST. Mint also includes a Mint optimizer, applying optimizations for stencil computations in CUDA. The generated programs reach performance similar to hand-tuned CUDA.

2.6.4 Algorithmic skeletons

Bones is another source-to-source compiler, transforming C to code for NVIDIA GPUs, AMD GPUs and OpenML [20]. Bones is based on algorithmic skeletons, e.g. commonly used parallel algorithms, for example map and reduce. Bones also requires annotations, although a very limited amount. The generated code achieves high performance and high readability, facilitating manual fine-tuning. SkelCL is also based on algorithmic skeletons, but implemented as a library based on OpenCL [21]. SkelCL is based on OpenCL and provides a vector data type and four algorithmic skeletons: reduce, map, zip and scan. Programming effort is greatly reduced by using SkelCL, while only suffering from a minor performance overhead. Another skeleton library is the Muesli library, presented by Ernsting and Kuchen [22]. Muesli allows a single application to be executed on a wide range of parallel machines, including multi-GPU systems and GPU clusters.

2.6.5 OpenMP and OpenACC

OpenMP and OpenACC enable directive-based high-level parallel programming in C, C++ and Fortran [23] [24]. OpenMP and OpenACC provide compiler directives that can be used to denote parts of a program that should be run in parallel. Compiler directives for copying and synchronizing data are also provided. OpenMP focuses on shared-memory systems while OpenACC targets heterogeneous systems.

2.6.6 Geospatial algorithms on the GPU

Implementing specific geospatial algorithms on the GPU is one approach for accelerating performance for geospatial computation. Xia *et al.* [25] present a methodological framework for mapping geospatial algorithms to the GPU, consisting of domain decomposition and thread allocation. They also present implementations of two common geospatial algorithms using this framework, inverse distance weighting (IDW) interpolation and viewshed. The result shows large speedups for the two algorithms and indicates that geospatial algorithms are suitable for GPU parallelization. Stojanovic and Stojanovic [26] present how an algorithm for slope computing efficiently can be implemented on a GPU using CUDA. Compared to a sequential implementation, the GPU implementation achieved large speed-ups. Zhang [27] presents a framework for accelerating rasterization of large scale polygons with GPGPU. The results achieve almost 20x speedup. You *et al.* [28] present an implementation of R-Trees on GPUs for accelerating spatial queries. R-Trees provides a way of storing and organizing data making it suitable for spatial queries. The implementation achieves about 10x speedups compared to a parallel CPU implementation.

2.6.7 Libraries and GIS

Zhang and You [29] present CudaGIS, a geospatial information system (GIS) implemented on the GPU. The implementations of different operators are presented and described. The authors also discuss how geospatial algorithms in general can be adapted for parallel hardware, using decomposition and general parallel algorithms.

Another approach for accelerating geospatial analysis using GPUs is libraries. cuSpatial is a C++ and Python high-performance library for geospatial analysis implemented on CUDA [30]. cuSpatial provides access to a set of common geospatial algorithms, for example, spatial indexing functions and spatial join functions.

Chapter 3

Method

This chapter presents the methodology of the project. Section 3.1 presents an overview of the research process. In Section 3.2, the domain analysis and language design phases of developing a DSL are discussed. Section 3.3 desribes the implementation of the DSL. Sections 3.4 and 3.5 present the templated interface of the parallel algorithms for vector data and rasters. Finally, Section 3.6 presents the evaluation of the project.

3.1 Overview of method

A DSL for geospatial computations on the GPU was developed. The development of the DSL was supported by three phases: domain analysis, language design, and implementation. The domain analysis explored the requirements, possibilities, and limitations of a DSL for geospatial computations on the GPU. During the language design, the structure of the language and what abstractions it should provide was determined. The DSL and the GPU interpreter were then implemented according to the design. To evaluate the performance, a sequential interpreter on the CPU and a parallel interpreter on the CPU was implemented. Finally, the level of functionality and the performance of the DSL were evaluated.

The research process can be summarized in the following steps:

1. Perform a domain analysis.
2. Design and implement the DSL and the GPU interpreter.
3. Implement two additional interpreters on the CPU.
4. Evaluate the level of functionality and performance of the DSL.

3.2 Development of DSL

3.2.1 Domain analysis

The domain analysis was divided into two parts. The first part included reading about geospatial operators and algorithms. The second part of the domain analysis was an interview with a domain expert at Carmenta. The questions were based on the five questions presented in [9]. The answers are summarized below:

- What is the purpose of the language? What are the use cases?
  - To simplify the implementation of geospatial computations on the GPU.
  - To reduce the time needed to implement geospatial operators on the GPU.
- Who are the key stakeholders and the intended users of the language?
  - The language will be used by experienced developers with knowledge about geospatial computations but without prior knowledge of GPU programming.
- What are the key domain concepts that users care about?
  - Vector data (entities) with attributes and rasters (fields).
  - Large datasets.
  - Simple operators are often applied to large datasets.
- How are domain concepts related, and what are their relevant properties?
  - Vector data and rasters both represent real-world objects.
  - Entities are in a 2D or 3D space.

3.2.2 Language design

During the language design, it was determined that the DSL should be embedded into C++. Then, appropriate abstractions were identified based on the domain analysis. It was decided that the DSL should facilitate abstractions for rasters and collections of vector data. The DSL should focus on parallel algorithms over rasters and collections of vector data, and allow user-defined lambdas (anonymous functions) within the parallel algorithms. This was also deemed suitable for GPU execution since it enables exploiting data parallelism.

3.2.2.1 Language constructs

The DSL should provide the following language constructs:

- Collections of vector data
- Rasters
- Parallel algorithms with user-defined lambdas over collections of vector data, more specifically:
  - Reduce, on all coordinates, per line or polygon.
  - Transform, on all coordinates.
  - TransformReduce, per line or polygon
  - Filter, on all coordinates
- Parallel algorithms with user-defined lambdas over rasters, more specifically:
  - Reduce
  - Transform
  - Local operators (combining two rasters)

3.2.2.2 How the language is used

A raster or a collection of vector data needs to be created and values added to the raster or collection. Values can be added manually, or generated. There exist three classes for collections of vector data and three classes for rasters, which all have the same interface but are interpreted differntly. Listing 3.3 shows how the three interpreters can be used to manually create two lines with two coordinates in each and display the collections.

```
auto coordinateVector = {{Coordinate(2, 3), Coordinate(4,5)},
    {Coordinate(0, 0), Coordinate(2, 2)}};
CPUCollection collCPU;
collCPU
    .lines(coordinateVector)
    .show();

CPUCollectionThrust collCPUThr;
collCPUThr
    .lines(coordinateVector)
    .show();

GPUCollection collGPU;
collGPU
    .lines(coordinateVector)
    .show();
```

Listing 3.1: Example program with vector data in the DSL.

Additionally, vector data can be created by inputting the coordinates as a list and the offset vector as another list, instead of grouping the coordinates by which entity it belongs to. Also, constructors for random data are accessible. Listing 3.2 shows how a CPUCollection with random lines can be created.

```
auto randomLines = utils::randomLines(nrOfLines,
   verticesInEachLine, maxValueCoordinates, seed1, seed2);
CPUCollection collCPU(randomLines);
collCPU.show();
```

Listing 3.2: Generating random vector data in the DSL.

Listing 3.3 shows an example program in the DSL with vector data. The example demonstrates how one CPU interpreter and one GPU interpreter can be used, and how the lambda must be modified to enable GPU execution (`__device__` needs to be added). The example also shows the fluent interface of the DSL. The lambda cannot capture values by reference in the capture clause when it is executed on the GPU.

```
auto coordinateVector = {{Coordinate(2, 3), Coordinate(4,5)},
    {Coordinate(0, 0), Coordinate(2, 2)}};
auto lambdaGPU = [] __device__(Coordinate c) { return c + 5;
   };
auto lambdaCPU = [] (Coordinate c) { return c + 5; };
CPUCollection collCPU;
collCPU
    .lines(coordinateVector)
    .transformCoordinates(lambdaCPU)
    .show();

GPUCollection collGPU;
collGPU
    .lines(coordinateVector)
    .transformCoordinates(lambdaGPU)
    .show();
```

Listing 3.3: Example program with vector data in the DSL.

Listing 3.4 shows how the different interpreters for rasters can be used. In this example, the rasters stores integers. Other datatypes can be stored by inputting another type when the raster is created.

```
int i = 5;
auto lambdaGPU = [i] __device__(int val) { return val + i; };
auot lambdaCPU = [i] (int val) { return valu + i; };
RasterCPU<int> rasterCPU;
rasterCPU
    .raster(rasterValues, nrOfColumns, nrOfRows)
    .transformRaster(lambdaCPU)
    .show();

RasterCPUThrust<int> rasterCPUThr;
rasterCPUThr
    .raster(rasterValues, nrOfColumns, nrOfRows)
    .transformRaster(lambdaCPU)
    .show();

RasterGPU<int> raster;
rasterGPU
    .raster(rasterValues, nrOfColumns, nrOfRows)
    .transformRaster(lambdaGPU)
    .show();
```

Listing 3.4: Example program with rasters in the DSL.

3.3 Implementation

3.3.1 Overview of implementation

The DSL was implemented as an embedded, interpreted DSL in CUDA C++. CUDA C++ was chosen since the target GPU is NVIDIA GPUs and the intended users are familiar with C++. Three interpreters of the DSL were implemented: one sequential on the CPU, one parallel on the CPU and one on the GPU. The GPU interpreter and the parallel CPU interpreter were implemented using Thrust, a parallel algorithms library in CUDA. This reduced the complexity of the GPU programming, allowing the interface of the DSL to be the focus of the project. Thrust was also suitable since the DSL is based on parallel algorithms. For the sequential CPU interpreter, all parallel algorithms were implemented manually.

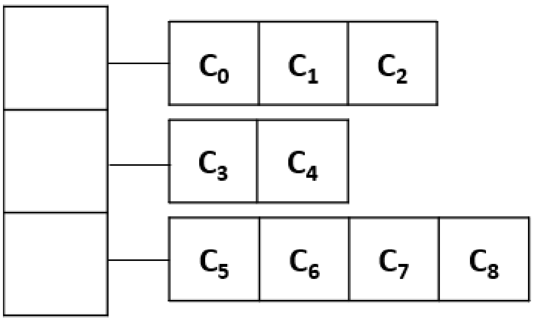


Figure 3.1: Semantic model of the sequential CPU interpreter for vector data, where C_i denotes the ith coordinate.

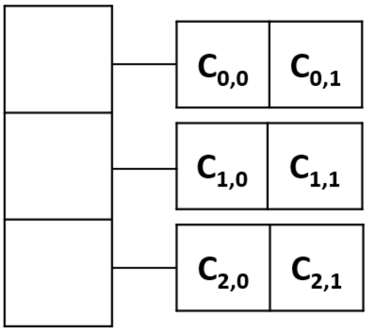
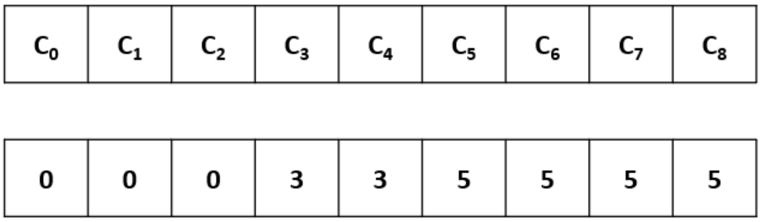


Figure 3.2: Semantic model of the sequential CPU interpreter for rasters, where C_i, j denotes the value on row i and column j.

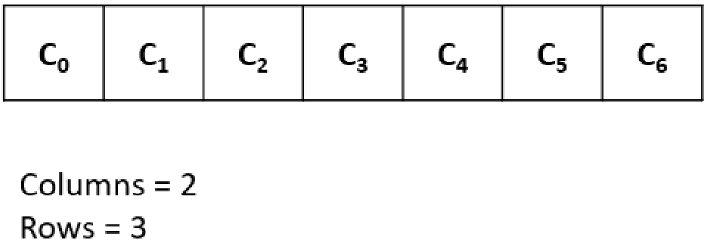
3.3.2 Sequential CPU interpreter

A sequential CPU interpreter was implemented for the DSL. The semantic model of the sequential CPU interpreter for vector data is based on vector of vectors, where each vector represents an entity and stores all coordinates belonging to that entity. Figure 3.1 shows the semantic model for the sequential CPU interpreter. The semantic model of the sequential CPU implementation for rasters is also based on vector of vectors, where each inner vector represents one row in the raster. Figure 3.2 shows the semantic model for the raster.



C_0	C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8
0	0	0	3	3	5	5	5	5

Figure 3.3: The semantic model of the CPU implementation, where C_i denotes the ith coordinate.



C_0	C_1	C_2	C_3	C_4	C_5	C_6

Columns = 2
Rows = 3

Figure 3.4: The semantic model of the parallel CPU implementation for rasters, where C_i denotes the ith cell.

3.3.3 Parallel CPU interpreter

A parallel CPU interpreter was implemented for the DSL. The semantic model of the parallel CPU interpreter for a collection of vector data consists of two vectors. The first vector, referred to as the coordinate vector, contains the coordinates for all entities. The second vector, the offset vector, contains the offset of each coordinate. This approach allows a two-dimensional vector to be represented by two one-dimensional vectors. Both vectors are stored as Thrust host vectors, which reside on the host (CPU). Figure 3.3 shows the semantic model. The figure displays three entities (lines), with the first entity containing three coordinates, the second containing two coordinates and the third containing four coordinates.

The semantic model of the parallel CPU interpreter for rasters consists of one vector, and the dimensions of the raster. The vector stores the values for all cells in the raster. The dimensions are stored as two integers, one for the number of rows and one for the number of columns. The vector is stored as a host vector. Figure 3.4 shows the semantic model of the parallel CPU interpreter. This example represents a raster with two columns and three rows.

3.3.4 GPU interpreter

A GPU interpreter for the DSL was also implemented. The semantic model is similar to the one for the parallel CPU implementation, except that all vectors are Thrust device vectors. When Thrust functions are called on device vectors, the computations will be executed on the GPU.

3.4 Templated interface for vector data

This section presents the templated interface for the parallel algorithms over vector data. The parallel functions are to be used as described in Subsection 3.2.2. The examples only demonstrate the parallel algorithms with the GPUCollection, however, both CPUCollection and CPUCollectionThrust can be used in the same way. Some parallel algorithms mutate the collection, while some store the result in a variable passed as a parameter to the functions. The reason for not returning results is to allow for a fluent interface (method chaining) in the DSL.

3.4.1 Reduce

The reduce function takes in a lambda which takes in two coordinate arguments and return a coordinate. The lambda is first applied to the first and second value in the collection. The lambda is then applied to the result of the previous application and the next coordinate in the entity until all values in the collection or entity are processed. Depending on which reduce function is applied (reduce on coordinates, lines, polygons), the result is one coordinate for the whole collection, one for each line, or one for each polygon. The reduce does not mutate the collection, but rather stores the result in a vector that is provided as input to the function. Listing 3.5 demonstrates how lines can be reduced on the GPU to calculate the coordinate with the largest x value for each line.

```
thrust::device_vector<Coordinate> maxGPU(nrOfLines);
auto lambdaMaxGPU = [] __device__(Coordinate c1, Coordinate
   c2) { return (c1.getX () < c2.getX()) ? c2 : c1; };
GPUCollection collGPU;
collGPU
    .lines(coordinateVector, offsetVector)
    .reduceLines(lambdaMaxGPU, maxGPU)
    .show();
```

Listing 3.5: Example of how the largest x value in each line can be found with reduce lines.

3.4.2 Transform

The transform function takes in a lambda and applies it to each element in the coordinate vector. The lambda should take a coordinate and return a coordinate. This function mutates the collection. Listing 3.6 demonstrates how transform can be used to offset each coordinate with the value five.

```
auto lambdaGPU = [] __device__(Coordinate c) { return c + 5; };
GPUCollection collGPU;
collGPU
    .lines(coordinateVector, offsetVector)
    .transformCoordinates(lambdaGPU)
    .show();
```

Listing 3.6: Example of how coordinates can be offseted with transform coordinates.

3.4.3 Transform-reduce

The transform-reduce function takes in either one or two functions. The first function transforms a sequence of coordinates to a sequence of any type. The second function reduces the sequence, if no function is passed, the values will be added together. The transform-reduce function also takes in an initial value, which is used as initial value in the reduce (and between entities). The result is one value for each entity, of any type. This function does not mutate the collection, rather stores its result in a vector which is passed to the function. Listing 3.7 shows how the length of lines can be computed with a transform-reduce function.

```
double init = 0.0;
thrust::device_vector<double> lengthGPU(nrOfLines);
auto lambdaGPU = [] __device__(Coordinate c1, Coordinate c2) { return length(c1, c2); };
GPUCollection collGPU;
collGPU
    .lines(coordinateVector, offsetVector)
    .transformReduceLines(lambdaGPU, lengthGPU, init)
    .show();
```

Listing 3.7: Example of how the length of lines can be computed with transform-reduce.

3.4.4 Filter

The filter function takes in a predicate based on a coordinate, and then filters out coordinates which does not fulfill the predicate. This mutates the collection. Listing 3.8 shows how all coordinates with an x value smaller than three can be filtered out.

```
auto lambdaFilterGPU = [] __device__(Coordinate c) { return c
   .getX() > 3; };
GPUCollection collGPU;
collGPU
    .lines(coordinateVector, offsetVector)
    .filterCoordinates(lambdaFilterGPU)
    .show();
```

Listing 3.8: Example with filter coordinates

3.5 Templated interface for rasters

This section presents the templated interface for the parallel algorithms over rasters. The parallel functions are to be used as described in section 3.2.2. The example only demonstrates the parallel algorithms with the RasterGPU, however, both RasterCPU and RasterCPUThrust can be used in the same way. Some parallel algorithms mutate the raster, while some store the result in a variable passed as a parameter to the functions. The reason for not returning results is to allow for a fluent interface (method chaining) in the DSL.

3.5.1 Local operator

The local operator function takes in a second raster and a lambda and applies the lambda cell vise to the two rasters. The result is stored in the raster which the operation is applied to. Listing 3.9 shows how this can be used to sum up two rasters of integers.

```
auto lambda = [] __device__ (int first, int sec) { return
    first + sec; };
RasterGPU<int> raster1;
RasterGPU<int> raster2;
raster1
    .raster(init);
raster2
    .raster(init)
    .localOperator(raster1, lambda)
    .show();
```

Listing 3.9: Example of how two rasters can be summed up with a local operator.

3.5.2 Transform

The transform function takes in a lambda that takes in one value and returns a transformed value of the same type. This operation mutates the raster. Listing 3.10 shows how the value 5 can be added to all cells in a raster using the transform function.

```
int i = 5;
auto lambda = [i] __device__(int val) { return val + i; };
RasterGPU<int> raster;
rasterGPU
    .raster(rasterValues, nrOfColumns, nrOfRows)
    .transformRaster(lambda)
    .show();
```

Listing 3.10: Example of how a value can be added to all cells in a raster using the transform function.

3.5.3 Reduce

The reduce function takes in a lambda. The lambda is first applied to the initial value passed and the first value in the raster. The lambda is then applied to the result from the previous application and the next value in the raster, until all values in the rasters have been processed. This function does not mutate the collection, but rather stores the result in an inputted variable. Listing 3.11 displays how the sum of all cells in a raster can be computed with reduce.

```
int result;
int init = 0;
auto lambda = [] __host__ __device__(int sum, int val) {
    return sum + val; };
```

```
RasterGPU<int> raster;
raster
    .raster(rasterValues, nrOfColumns, nrOfRows)
    .reduce(result, lambda, init)
    .show();
```

Listing 3.11: Example of how the sum of all cells in a raster can be computed with reduce.

3.6 Evaluation

3.6.1 Level of functionality

The level of functionality of the DSL was evaluated based on two parameters: what geospatial operators were supported and the FQAD evaluation. What geospatial operators were supported was evaluated by going through Table 2.1 and evaluating the support level of each category of operators. The support level for each category of operators was evaluated as either no, some, or strong support. The DSL was also evaluated with the FQAD framework [10]. The evaluation was done by a developer at Carmenta, a possible user of the DSL. Before the evaluation, the overall idea of the DSL and the abstractions it provides was presented. A number of example programs were also shown. The evaluation began by choosing which characteristics to evaluate, and the importance degree of each characteristic. For each characteristic, the support level for each sub-characteristic was evaluated. The support level for each sub-characteristic was then compared to the importance degree of the characteristic, deciding the success level of the DSL, either incomplete, satisfactory or effective.

3.6.2 Performance

The performance of the DSL was evaluated by comparing the execution time for the three interpreters for a set of algorithms and operators written in the DSL. The following five algorithms and operators were created with the DSL for evaluating performance:

- Offset operator
    - Offsets all coordinates by a specific value.
- Line-length operator

  - Computes the length of all line segments.
- Raster operator
  - Offsets all values in a raster and sums up two rasters.
- Naive point-in-polygon algorithm
  - Filters out points that are clearly outside a polygon.
- Ramer-Douglas-Peucker algorithm
  - Removes points from a line segment while preserving the overall shape of the line segment.

The execution time for the operators was measured for the three interpreters. The execution time was measured for different input sizes, ranging from 10^2 to 10^8. The same input sizes were not used for all operators, since the execution time for some operators was too long.

Chapter 4

Results and Analysis

This chapter presents the results of the project. Section 4.1 presents the results from the evaluation of the level of functionality of the DSL. In Section 4.2, the results from the evaluation are presented.

4.1 Level of functionality

4.1.1 Operators supported

Tables 4.1 and 4.2 present the support level for the categories of geospatial operators. The DSL has strong support for two out of five categories of operators on entities and some support for one out of five categories. The DSL has strong support for one of four categories of operators on rasters, and some support for one of four categories.

Overall, the DSL assists with multiple geospatial analysis and operators. The initial scope of the project was simple geospatial operators. Measurements and transformations of entities and transformations of rasters are to be considered simple geospatial operators, and the support is evaluated as strong for those categories.

4.1.2 FQAD evaluation

The following four characteristics were chosen to be evaluated:

- Functional suitability
  - *Functional suitability of a DSL refers to the degree to which a DSL supports developing solutions to meet stated needs of the*

Table 4.1: Support of spatial operators on entities.

Operators on entities	Support level
Measurements	Strong support
Transformations	Strong support
Spatial queries	No support
Overlay operators	Some support
Network analysis	No support

Table 4.2: Support of spatial operators on fields.

Operators on fields	Support level
Map algebra	Some support
Transformations	Strong support
Measurements	No support
Spatial interpolation	No support

application domain.

- Usability
  - *Usability of a DSL is the degree to which a DSL can be used by specified users to achieve specified goals.*
- Productivity
  - *Productivity of a DSL refers to the degree to which a language promotes programming productivity. Productivity is a characteristics related to the amount of resources expended by the users to achieve specified goals.*
- Expressiveness
  - *The degree to which a problem-solving strategy can be mapped into a program naturally.*

Table 4.3 shows the chosen importance degree for each characteristic. Table 4.4 shows the support level for each sub-characteristic. One of the sub-characteristics belonging to the productivity characteristic was deemed not

Table 4.3: The importance degree for each characteristic.

Characteristic	Importance degree
Functional suitability	Mandatory
Usability	Desirable
Productivity	Mandatory
Expressiveness	Desirable

possible to evaluate without a user-study. Overall, the support level of 11 of 17 sub-characteristics was evaluated as strong. For the remaining five that was possible to evaluate, the support level was evaluated as some. The sub-characteristics with some support are mostly related to how the symbols and attributes of the DSL.

By comparing the support level of the sub-characteristics to the importance degree for each characteristics, the DSL is deemed to be effective. All characteristics with importance degree mandatory have strong support in all sub-characteristics and all characteristics with importance degree desirable has at least some support in all sub-characteristics. The sub-characteristics belonging to productivity that could not be evaluated is not considered.

Overall, the DSL has strong or some support for the chosen characteristics. To improve the DSL, it needs to be easier to map a problem to the DSL and the attributes and symbols of the DSL need to be clearer.

4.2 Performance

4.2.1 Setup

The programs were executed on an Intel i7 CPU with 6 cores and a NVIDIA Quadro P2000. The programs were compiled with nvcc in Visual studio, with the flag –extended-lambda set in order to allow for lambdas on the device. Random points, lines, polygons and rasters were generated. A program involves both transferring data to the GPU and executing the specified operators. Each program was run twice as warmup. Then, the time for running each program two additional times was measured, and the average was calculated.

Table 4.4: The support level for each sub-characteristics.

FQAD		Support level
Functional suitability		
1.	All concepts and scenarios of the domain can be expressed in the DSL.	Strong support
2.	DSL is appropriate for the specific applications of the domain.	Strong support
Usability		
3.	The required amount of effort for understanding the language is small.	Strong support
4.	The concepts and symbols of the language are easy to learn and remember.	Strong support
5.	Language has capability to help users achieve their tasks in a minimum number of steps.	Strong support
6.	Users can recognize whether the DSL is appropriate for their needs.	Some support
7.	DSL has attributes that makes it easy to operate and control the language.	Some support
8.	DSL has symbols that are good-looking	Some support
9.	The language provides mechanism for compactness of the representation of the program	Strong support
Productivity		
10.	The development time of a program to meet the needs is improved	Strong support
11.	The amount of human resource used to develop the program is improved	N/A
Expressiveness		
12.	A problem solving strategy can be mapped into a program easily.	Some support
13.	The DSL provides one and only one good way to express every concept of interest	Some support
14.	Each DSL construct is used to represent exactly one distinct concept in the domain.	Strong support
15.	The language constructs corresponds to important domain concepts. DSL does not include domain concepts that are not important.	Strong support
16.	DSL does not contain conflicting elements.	Strong support
17.	DSL is at the right abstraction level such that it is not more complex or detailed than necessary.	Strong support

4.2.2 Offset operator

The offset operator takes in a set of entities and offsets each coordinate with a specific value. The offset operator was implemented in the DSL by one call to the transform functions, with a lambda which performs the offset.

The execution time for the offset operator was measured for input sizes from 10^2 to 10^7 points. Figure 4.1 shows the execution time as a function of the number of points. The parallel CPU implementation has shorter execution time than the sequential CPU implementation for any problem size. The GPU implementation has shorter execution time than the sequential CPU implementation when the number of points exceeds 10^3. The parallel CPU implementation and the GPU implementation perform roughly the same when the input size exceeds 10^4.

The GPU interpreter outperforms the sequential CPU interpreter when the input size grows. This is expected, since the GPU will benefit from the parallel processing when the input size grows. The parallel CPU interpreter and the GPU interpreter perform roughly the same. One reason for this is that the offset operator is only composed of one Thrust call. This means that the overhead for transferring data to the device gets large compared to the computations performed on the GPU. This could explain why they perform similarly.

4.2.3 Line-length operator

The line-length operator calculates the length of all lines. It is implemented in the DSL by a call to the transform-reduce function with a lambda which computes the length between two coordinates. The transform operator transforms the values into the length of the different line segments, which is then summed up by the reduce.

The execution time is measured for different number of lines and coordinates in each line. Figure 4.2 shows the execution time as a function of the total number of coordinates in all lines. The execution time is first measured for 100 lines with 10 coordinates in each, and the number of lines and coordinates is alternately increased until there are 10^4 lines with 10^3 coordinates in each. The result shows that the sequential and parallel CPU implementations performs roughly the same for any problem size. When the total number of coordinates exceeds 10^4, the GPU has shorter execution time than both CPU implementations.

The line-length operator, even if it only uses one parallel algorithm in the DSL, uses multiple Thrust functions. The overhead for transferring to the device gets less significant when multiple computations are done on the data.

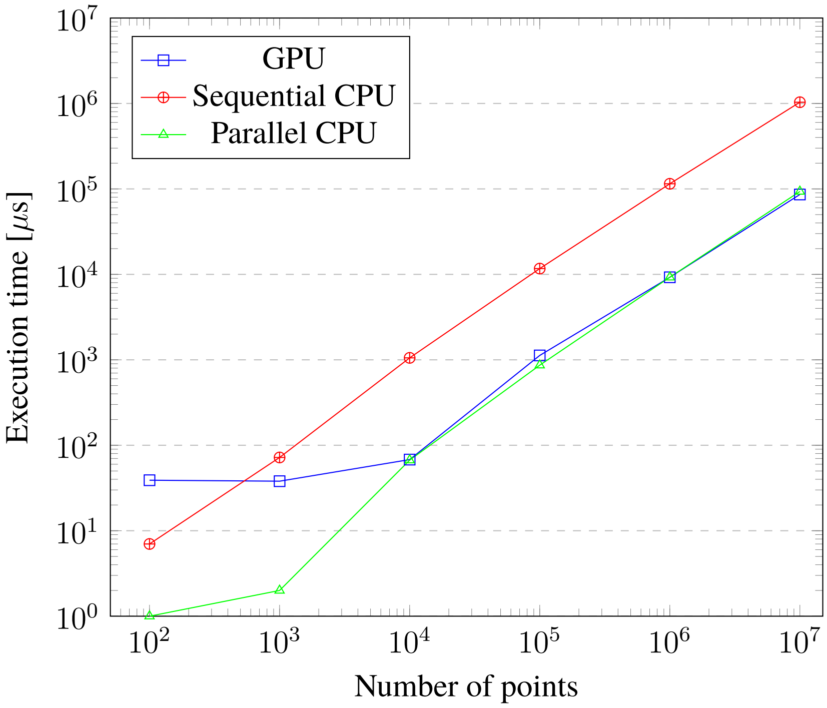


Figure 4.1: Execution time for offset operator

It is also reasonable to assume that the sequential CPU interpreter performs well for this operator, since computing something for each line get very natural with the semantic model of the sequential CPU interpreter.

4.2.4 Raster operator

The raster operation consists of first transforming a raster, and then cell wise addition with a second raster. It is implemented in the DSL by two function calls. First, a call to the transform-raster function with a lambda which adds the value two to each cell. Then, a call to the local operator function with the second raster as input and a lambda which computes the sum of two cells.

The execution time is measured for different number of rows and columns. Figure 4.3 displays the execution time as a function of the total number of cells in the raster. The execution time is measured for rasters with sizes ranging from 10^2x10^2 to 10^4x10^4.

The result shows that the parallel CPU implementation performs better than the sequential CPU implementation for all problem sizes. The GPU implementation performs better than the sequential CPU implementation

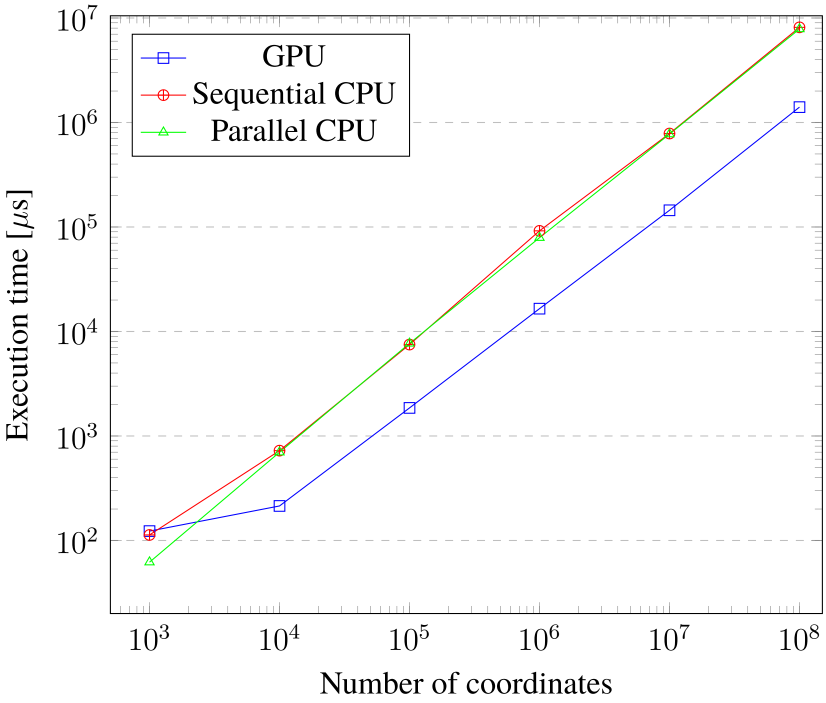


Figure 4.2: Execution time for line-length operator

when the total number of cells exceeds 10^5. The GPU implementation performs better than the parallel CPU implementation when the total number of cells exceeds 10^6.

The GPU interpreter only performs slightly better than the parallel CPU interpreter. One reason for this could be that some Thrust algorithms better utilize the GPU than others. The transform function in Thrust might be one which utilizes the GPU worse, which should also explain why the performance gains is small for the offset operator as discussed previously. Another reason could be that the actual work done in the transform, e.g. adding together two values, is too small to be able to utilize the GPU fully.

4.2.5 Naive point-in-polygon

A naive point-in-polygon algorithm was implemented using the DSL. The point-in-polygon algorithm takes in one polygon and a set of points. The algorithm uses the reduce-functionality from the DSL on the polygon to find the largest and smallest x and y values, called $xMax$, $xMin$, $yMax$ and $yMin$. The algorithm then uses filter to filter out all points with $x < xMin$, $x > xMax$, $y < yMin$ or $y > yMax$.

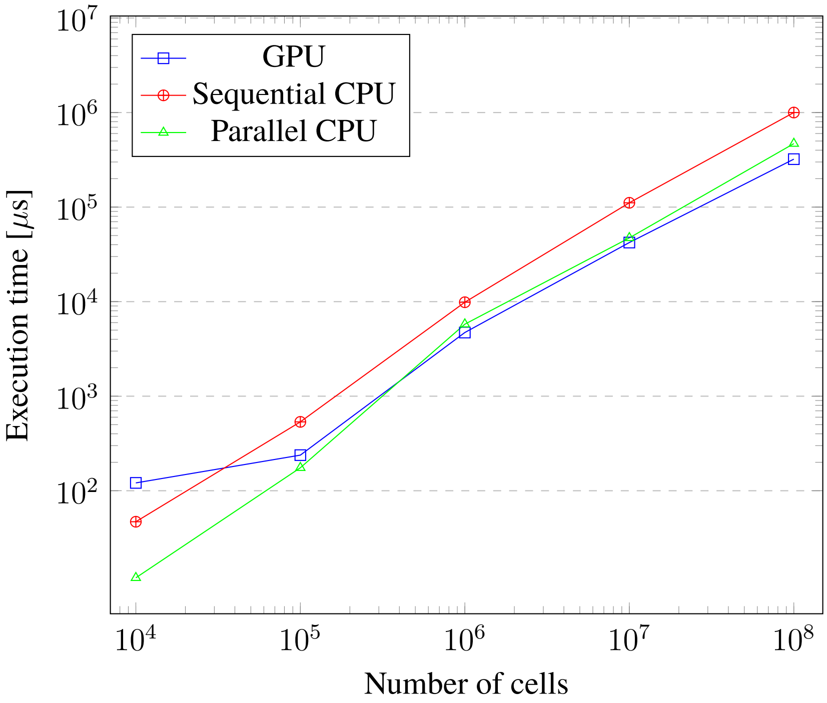


Figure 4.3: Execution time for raster operation

The execution time of the algorithm was measured for different number of points for two sizes of polygons. The polygon had 10^3 respective 10^4 coordinates. The number of coordinates ranged from 10^2 to 10^7. Figure 4.4 shows the execution time for point-in-polygon as a function of the number of points when the polygon has 10^3 coordinates (denoted by v). The parallel CPU interpreter performs better than the sequential CPU interpreter for all input sizes. The GPU interpreter performs better than the sequential CPU implementation when the input size exceeds 10^4. The GPU interpreter performs better than the parallel CPU implementation When the input size exceeds 10^5. Figure 4.5 shows the execution time for point-in-polygon as a function of the number of points, with $v = 10^4$. The trends are the same as Figure 4.4.

The overall trend is the same as for previous operators. When the input size is large, the GPU interpreter performs better than the other two interpreters. This operator consists of multiple parallel algorithms in the DSL, and therefore multiple Thrust parallel algorithms, resulting in speedups compared to the parallel CPU interpreter. In both 4.4 and 4.5, the polygons are relatively small. The GPU interpreter could benefit from if the polygon inputted was

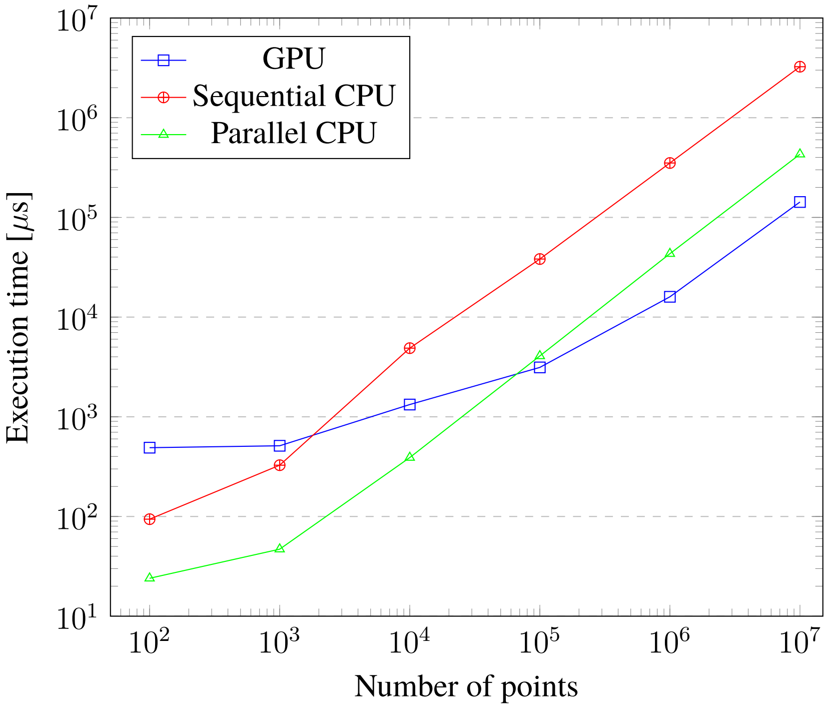


Figure 4.4: Execution time for point in polygon, with v = 1000

even larger, as the GPU performs relatively better compared to the parallel CPU interpreter when the input size grows.

4.2.6 Ramer-Douglas-Peucker algorithm

The Ramer-Douglas-Peucker algorithm was implemented using the DSL. The Ramer-Douglas-Peucker algorithm removes points from a line segment while preserving the overall shape of the line segment. The algorithm marks the first and last coordinate in the line segment and creates a line between the two points. The point which is furthest away from the line is marked as kept. The algorithm is then called recursively, once with the line segment between the first and the marked coordinate, and once with the line segment between the marked coordinate and the end coordinate. This process is repeated until the coordinate which is furthest from the line is within a specified limit. The DSL is used to compute the coordinate furthest away with the reduce functionality. It also uses the find functionality in the DSL to find the index of the marked coordinate.

The execution time of the algorithm was measured for input sizes from 10^2

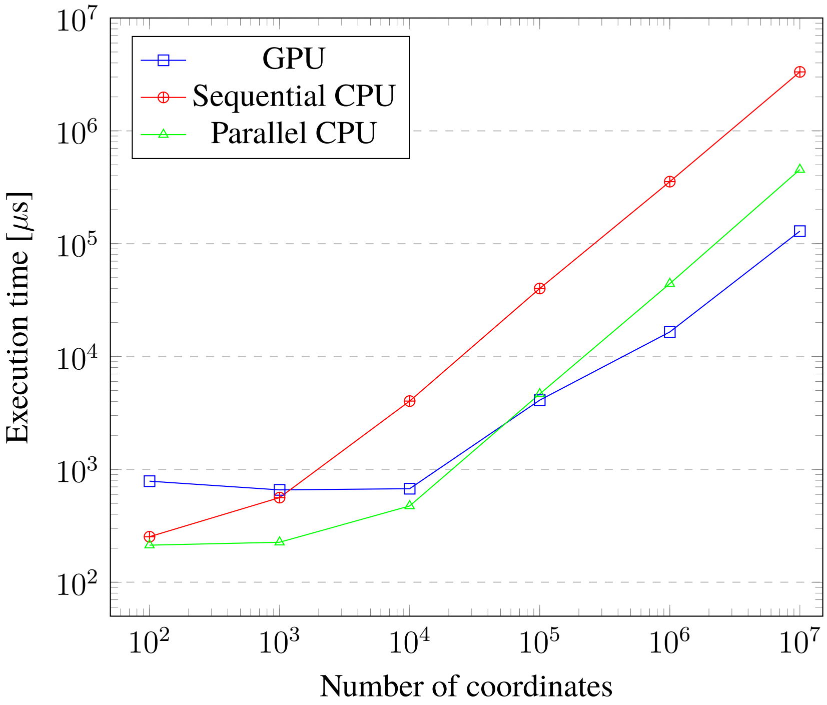


Figure 4.5: Execution time for point in polygon, v = 10 000

to 10^5 points. Figure 4.6 shows the execution time for the Ramer-Douglas-Peucker algorithm for different number of points in the line segment. The parallel CPU interpreter has shorter execution time than both the sequential CPU interpreter and the GPU interpreter for any number of points. The GPU interpreter has shorter execution time than the sequential CPU interpreter when the number of coordinates exceeds 10^4.

For the Ramer-Douglas-Peucker algorithm, the parallel CPU interpreter performs better than the GPU interpreter. One reason for this could be the relatively small input size, which was chosen due to the time the took to execute the algorithm. Another reason is that the algorithm recursively operates on a smaller line segment, and applying the parallel GPU algorithms on the small line segments could possibly introduce more overhead than the performance gained from it.

4.2.7 Speedup

Figure 4.7 shows the speedup for the GPU implementation compared to the sequential CPU implementation for the different operators and algorithms. R-D-P refers to the Ramer-Douglas-Peucker algorithm. Speedups smaller than

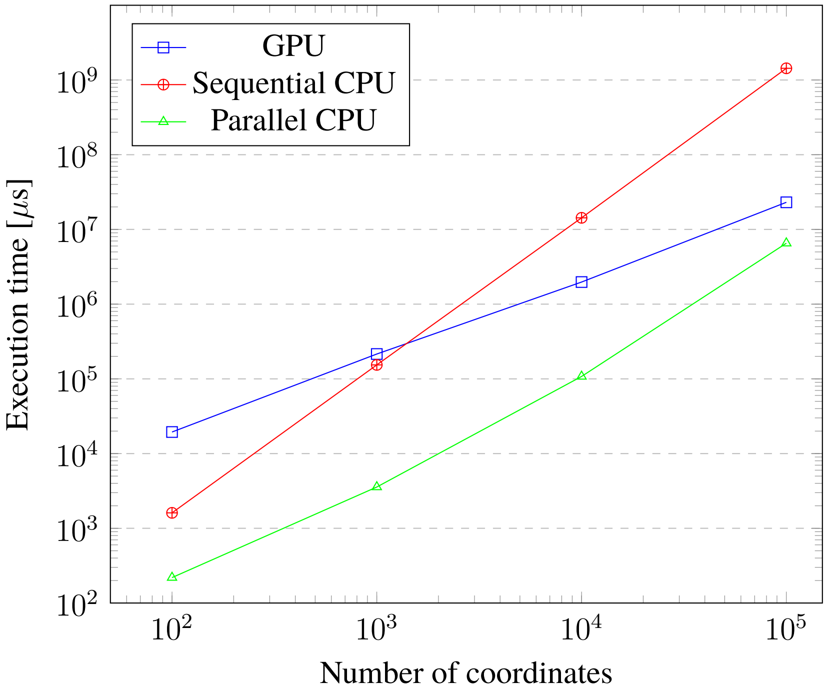


Figure 4.6: Execution time for Ramer-Douglas-Peucker algorithm.

1x indicates that the operator is slower. The problem sizes for which the speedup is shown is the largest problem size for that operator. All operators and algorithms measured achieve a speedup for the GPU implementation compared to the sequential CPU implementation. The Ramer-Douglas-Peucker algorithm achieves a speedup above 60x when the number of points is 10^5. The point-in-polygon operator (with a polygon with 10^3 coordinates) achieves a speedup slightly above 20x when the number of points is 10^7. The offset operator achieves a speedup above 10x for 10^7 coordinates. The line-length operator achieves almost 6x speedup when the total number of coordinates is 10^8. The raster operator achieves a speedup slightly above 3x when the total number of cells is 10^8.

It is clear that the GPU interpreter achieves large speedups compared to the sequential CPU interpreter for large problem sizes. It is also clear that the speedup depends a lot on the algorithms. The results show that the GPU interpreter achieves larger speedups when the number of involved operations are larger, as the GPU overhead gets relatively small. All operators and algorithms evaluated have large sections benefiting from parallel processing, which is a prerequisite to achieve speedups. It is also possible that different Thrust algorithms perform differently compared to sequential

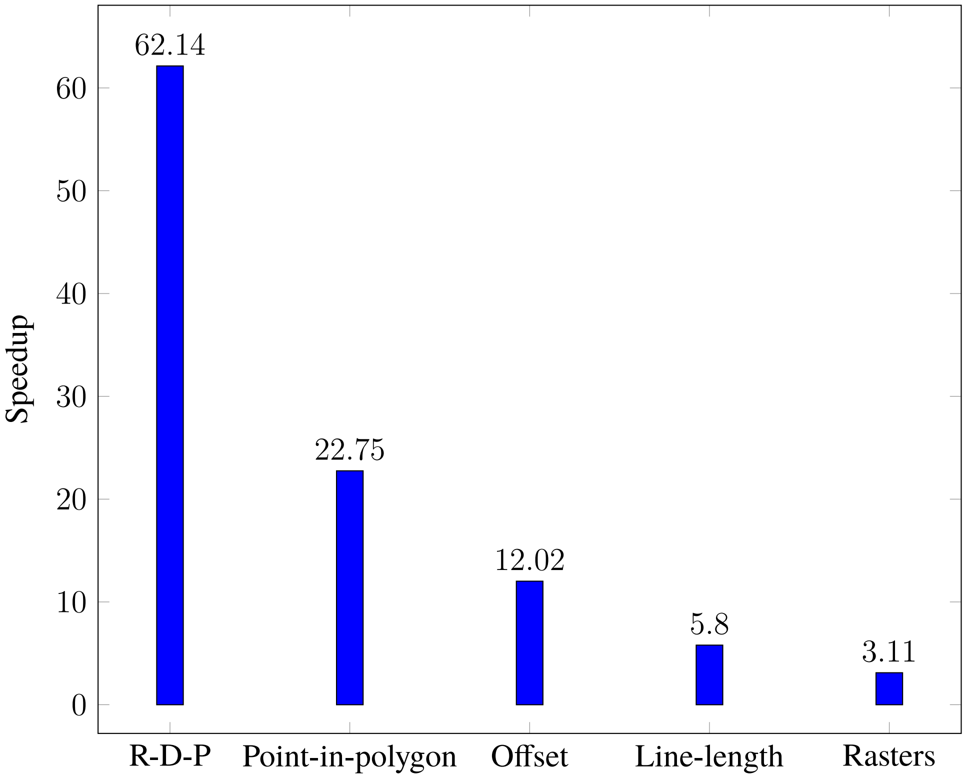


Figure 4.7: Speedup for GPU compared to sequential CPU

implementations.

Figure 4.8 shows the speedup for the GPU implementation compared to the parallel CPU interpreter for the different operators. The number of coordinates is the same as in Figure 4.7. For the Ramer-Douglas-Peucker algorithm, the speedup is 0.35x, e.g. the parallel CPU implementation has 3.5x shorter execution time than the GPU implementation. For the point-in-polygon operator, the speedup for the GPU implementation compared to the parallel CPU implementation is about 3x. The offset-operator achieves a small speedup for the GPU implementation, only 1.1x. For the line-length operator, the GPU implementation achieves a speedup of 6x. For the raster operation, the speedup is close to 1.5x.

The GPU interpreter achieves speedups compared to the parallel CPU interpreter for most operators. The speedups are generally smaller compared to the speedups when compared to the sequential CPU interpreter. This is expected, since the parallel CPU interpreter also can benefit from data parallelism. The speedup differs between different algorithms. One reason could be that some Thrust algorithms better utilizes the GPU than others.

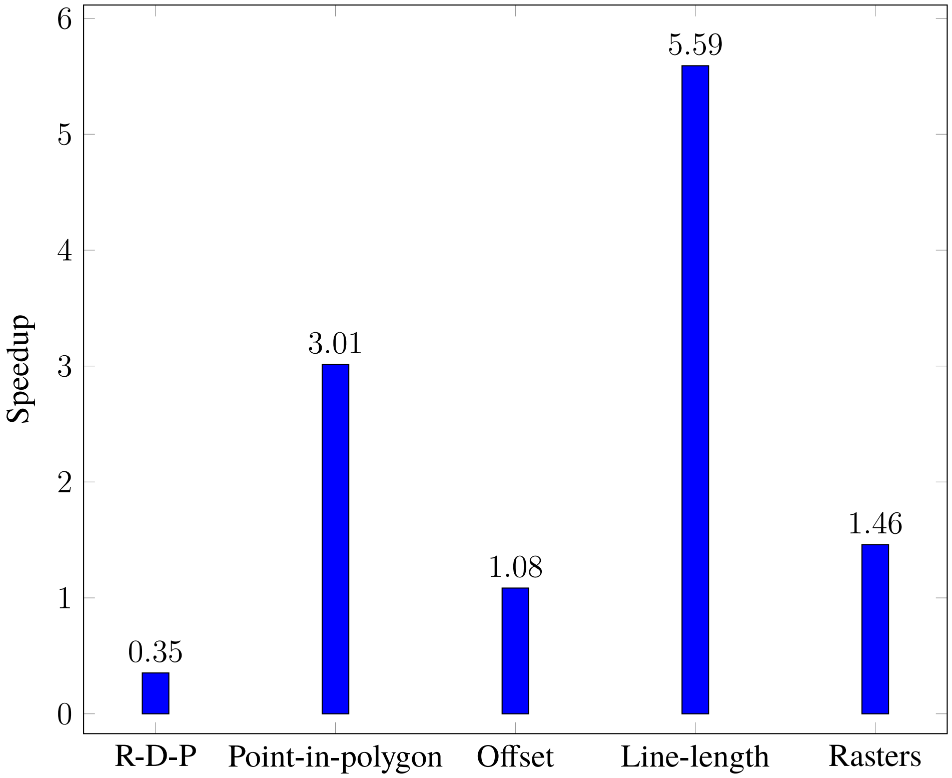


Figure 4.8: Speedup for GPU compared to parallel CPU

Chapter 5

Conclusion

Section 5.1 presents the conclusions of the project, based on the research questions. Section 5.2 presents the limitations of the project. In Section 5.3, future work is presented. Section 5.4 reflects on the project and how it is related to sustainability.

5.1 Conclusion

5.1.1 Level of functionality

- What is the level of functionality of a domain-specific language for simple geospatial operators on a GPU?

The level of functionality of the DSL was evaluated based on what geospatial operators were supported and an FQAD evaluation. Three out of five categories of operators on entities have at least some support. Two out of four categories of operators on fields have at least some support. The research question focused on simple geospatial operators, which all belong to the categories of analysis which has at least some support. The FQAD evaluation showed that the DSL was effective since all sub-characteristics had at least the support level required by the characteristics' importance degree.

5.1.2 Performance

- What are the performance gains for a domain-specific language for simple geospatial operators on a NVIDIA GPU?

The performance of the DSL was evaluated by comparing the execution time for three different interpreters for the DSL, one sequential on the CPU, one parallel on the CPU and one on the GPU. The results showed speedups for the GPU interpreter for all operators and algorithms compared to the sequential CPU interpreter for large problem sizes. The speedup ranged from 3x-60x. The result also showed speedups for the GPU interpreter for four out of five algorithm and operators compared to the parallel CPU interpreter for large problem sizes. The speedup ranged from 1.1x-5x for the four algorithms and operators which had positive speedups. The performance gains are highly affected by the problem size and the algorithms. The input size needs to be large enough to be able to effectively utilize the GPU. The algorithm must be able to exploit data parallelism.

5.1.3 Summary

Overall, this thesis demonstrates the potential of a domain-specific language for geospatial computations on the GPU. The thesis shows that such a DSL can achieve a high level of functionality as well as speedups compared to both a sequential and a parallel implementation for large problem sizes.

5.2 Limitations

One limitation of the project is the lack of a user study. A user study would provide more accurate results regarding the level of functionality of the DSL. Currently, the evaluation does not include any users using the DSL.

Another limitation of the project is the use of Thrust algorithms. It is possible that the geospatial operators could achieve even larger speedups from the GPU with hand-tuned CUDA.

5.3 Future work

Table 4.1 and Table 4.2 shows what spatial analysis is supported by the DSL. The next step would be to add support for the categories of analysis currently not supported and increase the support for the categories that currently only have some support. Spatial queries and network analysis would be a suitable next step.

5.4 Reflections

Utilizing the GPU for operators on large datasets is more energy-efficient than executing the same operators on the CPU. This projects enables computation to be moved from the CPU to the GPU without large efforts, resulting in reduced energy consumption. Reducing energy consumption is one important step towards more sustainable computing.

References

[1] P. A. Longley, M. F. Goodchild, D. J. Maguire, and D. W. Rhind, *Geographic Information Systems and Science*, 2. ed. Wiley, 2005, ISBN: 0-470-87000-1.

[2] R. Membarth, O. Reiche, F. Hannig, J. Teich, M. Körner, and W. Eckert, "Hipacc: A domain-specific language and compiler for image processing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 1, pp. 210–224, 2015.

[3] Z. DeVito *et al.*, "Liszt: A domain specific language for building portable mesh-based pde solvers," in *Proceedings of 2011 international conference for high performance computing, networking, storage and analysis*, 2011, pp. 1–12.

[4] M. Fowler, *Domain-specific languages*. Pearson Education, 2010.

[5] *Cuda c++ programming guide*, https://docs.nvidia.com/cuda/cuda-c-programming-guide/, Accessed: 2024-01-29.

[6] M. Mernik, J. Heering, and A. M. Sloane, "When and how to develop domain-specific languages," *ACM Comput. Surv.*, vol. 37, no. 4, pp. 316–344, 2005, ISSN: 0360-0300. DOI: 10.1145/1118890.1118892. [Online]. Available: https://doi-org.focus.lib.kth.se/10.1145/1118890.1118892.

[7] D. M. Weiss and C. T. R. Lai, *Software product-line engineering: a family-based software development process*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[8] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. Peterson, "Feature-oriented domain analysis (foda) feasibility study," 1990.

[9] A. Wasowski and T. Berger, *Domain-Specific Languages: Effective modeling, automation, and reuse*, English. Springer International Publishing, 2023, ISBN: 9783031236686.

[10] G. Kahraman and S. Bilgen, “A framework for qualitative assessment of domain-specific languages,” *Software & Systems Modeling*, vol. 14, pp. 1505–1526, 2015.

[11] *Thrust*, https://docs.nvidia.com/cuda/thrust/, Accessed: 2024-01-29.

[12] N. Corporation, *Thrust quick start guide*, https://docs.nvidia.com/cuda/archive/9.0/pdf/Thrust_Quick_Start_Guide.pdf, Accessed: 2024-01-29, 2018.

[13] C. D. Lloyd, *Spatial data analysis : an introduction for GIS users*. Oxford University Press, 2010, ISBN: 978-0-19-955432-4.

[14] P. A. Burrough and R. A. McDonell, *Principles of Geographical Information Systems*. Oxford University Press, 1998, ISBN: 0-19-823365-5.

[15] M. J. De Smith, M. F. Goodchild, and P. A. Longley, *Geospatial analysis : a comprehensive guide to principles, techniques and software tools*. Matador, 2007, ISBN: 1905886608.

[16] J. Ragan-Kelley, A. Adams, S. Paris, M. Levoy, S. Amarasinghe, and F. Durand, “Decoupling algorithms from schedules for easy optimization of image processing pipelines,” *ACM Transactions on Graphics (TOG)*, vol. 31, no. 4, pp. 1–12, 2012.

[17] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, “Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines,” eng, *SIGPLAN notices*, vol. 48, no. 6, pp. 519–530, 2013, ISSN: 0362-1340.

[18] T. D. Han and T. S. Abdelrahman, “Hicuda: High-level gpgpu programming,” *IEEE Transactions on Parallel and Distributed systems*, vol. 22, no. 1, pp. 78–90, 2010.

[19] D. Unat, X. Cai, and S. B. Baden, “Mint: Realizing cuda performance in 3d stencil methods with annotated c,” in *Proceedings of the international conference on Supercomputing*, 2011, pp. 214–224.

[20] C. Nugteren and H. Corporaal, “Introducing’bones’ a parallelizing source-to-source compiler based on algorithmic skeletons,” in *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, 2012, pp. 1–10.

[21] M. Steuwer, P. Kegel, and S. Gorlatch, “Skelcl-a portable skeleton library for high-level gpu programming,” in *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, IEEE, 2011, pp. 1176–1182.

[22] S. Ernsting and H. Kuchen, “Algorithmic skeletons for multi-core, multi-gpu systems and clusters,” eng, *International journal of high performance computing and networking*, vol. 7, no. 2, pp. 129–138, 2012, ISSN: 1740-0562.

[23] OpenACC-Standard.org, *The openacc application programming interface*, https://www.openacc.org/sites/default/files/inline-images/Specification/OpenACC-3.3-final.pdf, Version 3.3, 2022.

[24] OpenMP, *Openmp application programming interface*, https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf, Version 5.2, 2021.

[25] Y.-j. Xia, L. Kuang, and X.-m. Li, “Accelerating geospatial analysis on gpus using cuda,” *Journal of Zhejiang University SCIENCE C*, vol. 12, no. 12, pp. 990–999, 2011.

[26] N. Stojanovic and D. Stojanovic, “High–performance computing in gis: Techniques and applications,” *International Journal of Reasoning-based Intelligent Systems*, vol. 5, no. 1, pp. 42–49, 2013.

[27] J. Zhang, “Speeding up large-scale geospatial polygon rasterization on gpgpus,” in *Proceedings of the ACM SIGSPATIAL second international workshop on high performance and distributed geographic information systems*, 2011, pp. 10–17.

[28] S. You, J. Zhang, and L. Gruenwald, “Parallel spatial query processing on gpus using r-trees,” eng, in *Proceedings of the 2nd ACM SIGSPATIAL International Workshop on analytics for big geospatial data*, ACM, 2013, pp. 23–31, ISBN: 1450325343.

[29] J. Zhang and S. You, “Cudagis: Report on the design and realization of a massive data parallel gis on gpus,” in *Proceedings of the 3rd ACM SIGSPATIAL International Workshop on GeoStreaming*, 2012, pp. 101–108.

[30] NVIDIA Corporation, *Cuspatial: Gpu-accelerated geospatial and spatiotemporal algorithms*, https://github.com/rapidsai/cuspatial, Accessed: 2024-02-01, 2023.

TRITA-EECS-EX- 2023:0000

€€€€ For DIVA €€€€

{
”Author1”: { ”Last name”: ”Tidestav”,
”First name”: ”Louise”,
”Local User Id”: ”louiseti”,
”E-mail”: ”louiseti@kth.se”,
”organisation”: {”L1”: ”School of Electrical Engineering and Computer Science”,
}
},
”Cycle”: ”2”,
”Course code”: ”DA231X”,
”Credits”: ”30.0”,
”Degree1”: {”Educational program”: ”Degree Programme in Computer Science and Engineering”
,”programcode”: ”CDATE”
,”Degree”: ”Degree of Master of Science in Engineering”
,”subjectArea”: ”Computer Science and Engineering”
},
”Title”: {
”Main title”: ”A domain-specific language for geospatial computations on the GPU”,
”Subtitle”: ”A study on the level of functionality and the performance gains”,
”Language”: ”eng” },
”Alternative title”: {
”Main title”: ”Ett domänspecifikt språk för geospatiala beräkningar på GPU:n”,
”Subtitle”: ”En studie på användbarhet och prestandaförbättringar”,
”Language”: ”swe”
},
”Supervisor1”: { ”Last name”: ”Peng”,
”First name”: ”Ivy Bo”,
”Local User Id”: ”bopeng”,
”E-mail”: ”bopeng@kth.se”,
”organisation”: {”L1”: ”School of Electrical Engineering and Computer Science”,
”L2”: ”Computer Science” }
},
”Supervisor2”: { ”Last name”: ”Aasa”,
”First name”: ”Jakob”,
”E-mail”: ”jakob.aasa@carmenta.com”,
”Other organisation”: ”Carmenta Geospatial Technology”
},
”Examiner1”: { ”Last name”: ”Markidis”,
”First name”: ”Stefano”,
”Local User Id”: ”markidis”,
”E-mail”: ”markidis@kth.se”,
”organisation”: {”L1”: ”School of Electrical Engineering and Computer Science”,
”L2”: ”Computer Science” }
},
”Cooperation”: { ”Partner_name”: ”Carmenta Geospatial Technology”},
"National Subject Categories": "10201, 10206",
”Other information”: {”Year”: ”2024”, ”Number of pages”: ”1,53”},
”Copyrightleft”: ”copyright”,
”Series”: { ”Title of series”: ”TRITA-EECS-EX” , ”No. in series”: ”2023:0000” },
”Opponents”: { ”Name”: ”A. B. Normal & A. X. E. Normalè”},
”Presentation”: { ”Date”: ”2022-03-15 13:00”
,”Language”:”eng”
,”Room”: ”via Zoom https://kth-se.zoom.us/j/ddddddddddd”
,”Address”: ”Isafjordsgatan 22 (Kistagången 16)”
,”City”: ”Stockholm” },
”Number of lang instances”: ”2”,
”Abstract[eng]”: €€€€

This thesis explores how a domain-specific language (DSL) for simple geospatial operators on the GPU can be developed, and evaluates the level of functionality and performance of such a DSL. The purpose of such a DSL is to simplify implementation of geospatial operators on the GPU, in order to increase productivity and performance.

A DSL was designed and implemented according to the geospatial domain. The level of functionality of the DSL was evaluated based on what geospatial operators are supported and with the Framework for Qualitative Assessment of DSLs (FQAD). The level of functionality was evaluated to high for the DSL, as simple geospatial operators are supported and the FQAD evaluation determined that the DSL was effective. To evaluate the performance gains, three interpreters for the DSL were implemented, one sequential on the CPU, one parallel on the CPU and one on the GPU. The execution time for five geospatial operators and algorithms was measured for different problem sizes for the three interpreters. The GPU interpreter achieved large speedups for large problem sizes compared to the sequential CPU interpreter. The GPU interpreter achieved speedups for four out of five operators compared to the parallel CPU interpreter for large problem sizes. In conclusion, this thesis demonstrates the potential of a domain-specific language for geospatial computations on the GPU.

€€€€,

”Keywords[eng]”: €€€€
Domain-specific language, Geospatial, Graphics processing unit €€€€,
”Abstract[swe]”: €€€€

Denna uppsats undersöker hur ett domänspecifikt språk (DSL) för enkla geospatiala operatorer på GPU:n kan utvecklas, och utvärderar användbarheten och prestandan för ett sådant DSL. Syftet med ett sådant DSL är att förenkla implementationen av geospatiala operatorer på GPU:n, för att öka produktivitet och prestanda.

Ett DSL designades och implementerades utefter den geospatiala domänen. Användbarheten av DSL:en utvärderades baserat på vilka geospatiala operatorer som stöds och med Framework for Qualitative Assessment of DSLs (FQAD). Användbarheten utvärderades till hög för DSLen, eftersom enkla geospatiala operatorer stöds och FQAD utvärderingen kom fram till att DSLen var effektiv. För att utvärdera prestandaförbättringar implementerades tre interpreterare, en sekventiell på CPUn, en parallel på CPUn och en på GPU:n. Körtiden för fem geospatiala operatorer och algorithmer mättes för olika problemstorlekar för de tre interpreterarna. GPU implementationen hade bättre prestanda i jämförelse med den sekventiella CPU implementationen för stora problemstorlekar. GPU implementationen hade bättre prestanda för fyra av fem operatorer i jämförelse med den parallela CPU implementationen för stora problemstorlekar. Sammanfattningsvis visar detta projekt på potentialen för ett domänspecifikt språk för geospatiala beräkningar på GPU:n.

€€€€,
”Keywords[swe]”: €€€€
Domänspecifikt språk, Geospatial, Grafikprocessor €€€€,
}