Efficient Evaluation of Functionally Represented Volumetric Objects

Tuomo Rinne

A thesis submitted in partial fulfilment of the requirements of Bournemouth University for the degree of Masters by Research

September, 2014

Copyright statement

Contents

Table of contents ii
List of figures iv
List of tables vii
List of Acronyms viii
Abstract ix
Acknowledgements x

1 Introduction **1**
1.1 Hardware for high-performance computing 2
1.2 Problem statement 4
1.3 Contributions 5

2 Related work **6**
2.1 Function representation 6
2.1.1 Tree traversal 7
2.2 Parallel hardware 8
2.2.1 SIMD 10
2.2.2 Development frameworks 13
2.3 Fast Rendering of FRep shapes 15
2.3.1 Auxiliary data-structures for rendering 15
2.3.2 Direct rendering 18
2.4 Survey on efficient FRep function evaluation 19
2.4.1 Hardware methods 20
2.4.2 Algorithmic methods 22
2.5 Conclusion 23

3 Efficient function evaluation **24**

3.1 A multikernel approach to distributed function evaluation 24
3.1.1 The multikernel method 25
3.2 FRep function classifications 27
3.2.1 Geometric objects 27
3.2.2 Geometric operations 29
3.3 Conclusion 34

4 Applications **35**
4.1 Test settings for node evaluations 35
4.1.1 Case study: sweeping by a moving solid 40
4.1.2 Case study: Offset Along a Normal 41
4.1.3 Case study: convolution surface using lines 42
4.1.4 Case study: HyperFun fractal 43
4.2 Complex tree structure generation 44
4.3 Conclusion 44

5 Results **46**
5.1 Tree traversal results 47
5.2 Individual node results 51
5.2.1 Case study: Inside-outside segmentation of polygonal mesh 53
5.2.2 Case study: Offset along a normal 54
5.2.3 Case study: Sweeping by a moving solid 55
5.2.4 Case study: Projection 56
5.2.5 Case Study: Inverse free-form deformation . . . 57
5.2.6 Memory 57
5.3 Conclusion 58

6 Conclusion and future work **60**
6.1 Future work 63

References **63**

Appendices **69**
.1 Sweeping by a moving solid 69

List of Figures

2.1 The first phase of function evaluation. The image is a reproduction from (Kravtsov 2011) 8
2.2 The second phase of function evaluation. Image from (Kravtsov 2011) 9
2.3 On left a scalar ADD instruction executed on two 32-bit registers containing values A[0] and B[0]. To execute four float additions, a scalar instruction is executed for each pair of operands in a sequence. On right, a vector VADD operation executed on two 128-bit registers, each containing a vector of four 32-bit values. To add two vectors together only one instruction is required. Both scalar and vector addition requires one clock cycle per instruction. 11
2.4 Most modern GPUs have a two-level scheduling: user defined thread blocks are first scheduled to streaming processors, whose thread schedulers make sure all the processors are kept busy. 12
2.5 Scheduling of threads of SIMD instructions. SIMD instructions are scheduled to a group of processors. By definition the instructions can be independently, therefore the scheduler fetches an instruction that is ready for evaluation (Hennessy and Patterson 2011). 13
2.6 Bottom-up build process of 2-dimensional HP, adding the values of four texels repeatedly. The last level contains the total number of triangles in the pyramid. The figure is a reproduction as seen in (Dyken *et al.* 2008). 16

2.7 A pyramid is accessed by using a key k. A pointer p, a current level l, and a local key k_l are maintained during traversal. Given $k = 5$, the level below ranges equal to $A = [0, 3)$, $B = [3, 6)$, $C = [6, 7)$, $D = [7, 9)$. In the first step the k_l falls to range B. The p now points to B. In second step $A = [0, 0)$, $B = [0, 1)$, $C = [1, 1)$, $D = [1, 3)$ and $k_l = 5-3$, hence p ends up pointing to a texel D. The figure is a partial reproduction of a figure seen in (Dyken *et al.* 2008). 17

2.8 In ray marching, a ray progresses through the scene until sampling at either of the intervals end points returns a zero or the values have different signs. The boundary will be within the interval if the values have opposite signs. . 19

2.9 An instance of the whole FRep tree is given to each execution unit. An execution unit in case of a GPU is a streaming processor, and in case of a multiprocessor system: a CPU or a core. 21

3.1 When evaluating nodes individually, a node is distributed amongst the execution units and results stored for subsequent nodes that require the results. The first step is to turn the tree into a queue of tasks; In this example Post-order tree traversal is used. An execution unit in case of a GPU is a streaming processor, and in case of a multiprocessor system: a CPU or a core. 26

3.2 The host side instructions for a simple tree. In the evaluation post-order traversal is used to evaluate the tree; on each node, memory is managed and kernels initiated.The keep track of allocated GPU memory the host uses reference counted handles. 27

5.1 Execution of a random tree with 112 nodes on a GPU; When a GPU can not fit all the samples into memory, the average execution time rises. 47

5.2 Execution of a random tree with 112 nodes on a multiprocessor system. 48
5.3 Execution of a random tree with 112 nodes on multiprocessor system ; High number of memory queries results in high cache-miss ratio. 49
5.4 A compilation step in comparison to the evaluation in a single kernel method, is a major cause for execution overhead. 50
5.5 The execution speed calculations made on existing FRep function classifications for geometric objects. 51
5.6 The execution speed calculations made on existing FRep function classifications for geometric operations. 52
5.7 Evaluation times of Mesh on the left-hand side graph. . . 53
5.8 Evaluation times of OffsetAlongNormalNode on the left-hand side graph. A rendered image of operation applied to union of two spheres and a cube on right-hand side. . 54
5.9 Evaluation times of SolidSweepNode on the left-hand side graph. A rendered image of solid sweep operation applied to union of two spheres and a cube on right-hand side. . 55
5.10 Evaluation times of Projection on the left-hand side graph. A rendered image of projection operation applied to twisted cube on right-hand side. The faraway plane is the result of the projection. 56
5.11 Evaluation times of InverseFFD on the left hand side graph. A rendered image of InverseFFD operation applied on torus on right hand side. 57
5.12 A division of Hyperfun fractal evaluation into a memory transfer from host to device, evaluation of the code, and transfer of the results back to host. The timings are done with OpenCL timing events. 58

List of Tables

4.1 A simplification of node classifications in form of a graph. L1 and L2 are classes for geometric objects: L1 classifies functions according to their defining function, L2 classifies the functions according to their requirements for external data. N1, N2, and N3 are classes for geometric operations: N1 classifies operations according to their tree traversal method, N2 classifies the operations according to the complexity of function evaluation, N3 classifies the operations according to their requirements for external data. 36

5.1 The GPU and CPU device info. 46

List of Acronyms

BRep Boundary representation

FRep Function representation

GPU Graphics programming unit

GPGPU General purpose processing on graphics programming units

ISA Instruction set architecture

SIMD Single-Instruction, Multiple-Data

SIMT Single-Instruction, Multiple-Thread

SM Streaming multiprocessor

Abstract

There are several approaches to representing shapes in computer graphics. One of the ways to describe objects and operations is Function Representation (FRep). In FRep, a geometric object is defined by a single continuous real-valued function of point coordinates.

Generally geometric modelling is conducted in order to achieve visual outcome. In FRep the transformation of a function into a visual representation relies on extensive sampling of the function. The computational cost of the sampling can cause adverse effects during applications runtime.

In this thesis the problem of efficient evaluation of the defining function is discussed. An observation is made on wide range of operations and primitives within FRep and their suitability for parallelization. Furthermore, a new novel method is proposed to distribute FReps computational workloads on parallel hardware devices such as graphics programming units and multi-core processors.

Keywords: Scalar fields, heterogeneous computing, Function Representation, High-Performance computing

Acknowledgements

I would like to thank my supervisory team: Prof. Alexander Pasko, Dr. Valery Adzhiev and Dr. Oleg Fryazinov for their advice and guidance during this project. I would also like to thank Dr. Leigh McLoughlin and Mathieu Sanchez for their advice and support.

For code contributions I would like to thank: Dr. Pierre-Alain Fayolle for the genetic algorithms to create random tree structures, Dr. Oleg Fryazinov for the numerous individual node implementations, Dr. Leigh McLoughlin for the initial tree structure implementation, and Mathieu Sanchez for the meshing, inside-outside segmentation, and shading algorithms.

I would also like to thank my reviewers Dr.Ian Stephenson and Dr. Irina Voiculescu for their feedback and suggestions.

Finally, I would like to thank my family and friends for their support.

Chapter 1

Introduction

Geometry of the real-life objects can be represented in a computer systems by a wide range of different ways. Fundamentally, the representation of the geometry of the objects can be distinguished into parametric form, where the geometry of the surface can be obtained explicitly and into implicit form where a predicate is given to distinguish the points which belong to the object and not.

One of the main examples of a parametric form is a polygonal mesh which is widely used for various applications because of its direct support by a computer hardware for visualisation purposes. Recent advances in the hardware and software allows to use wider range of representations in the interactive modelling and visualisation systems. Many methods are using graphical hardware (GPU) for general purposes computations in order to accelerate the processing of the geometry.

Function Representation(FRep) is a way to describe geometric shapes implicitly as a single function of point coordinates or with a scalar field. The shape is constructed from functions that describe geometric objects and operations. The constructed function returns a scalar value that represents the points relation to a surface of the shape.

The defining function can be constructed using wide range of simple functions such as trigonometric and algebraic functions. Very often the model defined with Function Representation is expressed as a tree struc-

ture of geometric operations as internal nodes and geometric primitives as leaves. In such cases modelling of the shape can be done through modifying a tree structure either by adding and removing nodes from it or changing parameters of the existing ones.

In order to object expressed as a function, it has to be sampled in a number of points to derive enough information to determine where the boundary of the shape is. The sampling can be also used to derive other information about object such as material properties. The density of the sampling is directly connected to the quality of approximation that can be derived from the samples. For example, sparse sampling can miss some sharp features. Denser sampling gives better approximations while increasing the computational cost of the sampling process.

Interactive modelling of geometry is one application where quick visual feedback and good quality of surface approximation are crucial for the user. These applications require good sampling performance.

One of the main approaches to efficiently evaluate the function in a number of points is to use specialised hardware where the same operation can be calculated simultaneously by using the resources of the hardware. In the current state of the art specialised hardware usually means graphical hardware (GPU) applied for general purposes computations. However apart from benefits the graphics hardware has its own limitations which should be taken into an account.

1.1 Hardware for high-performance computing

Performance has always been an important factor in computing and its is actively improved by hardware and new parallel algorithms. For many years increasing the clock-speed of a processor was the main method to increase the performance of a computer. However, complex multimedia applications have raised the bar and a uniprocessor struggles to keep up the demands. Increasing the frequency of a processor produces more

heat and consumes more power. Therefore, the feasibility of increasing clock-speed in consumer level hardware is questionable.

In order to improve the performance semiconductor designers have added more processors so the workload can be shared. A multiprocessor design adds more responsibility on the software developer who has to implement parallel algorithms to facilitate to all the processing power. Programming for a parallel system like multiprocessor system requires mental paradigm change from a serial execution model to a parallel execution model. In practice, a re-organization of algorithms is required to utilize the parallel features of the computer.

Computer graphics is a field where the applications are well known for their computational requirements. Dedicated graphics programming units are manufactured to deal with graphical workloads. The computational profile in graphical applications often includes a lot of same computation done on multiple elements. These tasks are generally fairly trivial and deal with vertices and pixels. In many cases the tasks can be executed in parallel. As a consequence, a GPU has evolved to support high-level of parallelism by using a large number of independent processors.

Historically GPUs were mainly designed to speed up the computation of graphics in gaming. However, there are a number of problems unrelated to graphics that can benefit from parallelization, for example complex simulations. The GPUs have beed adopted to these purposes and specialized frameworks are developed to help developers (Luebke *et al.* 2004). This form of computing is often referred as general purpose processing on graphics programming units (GPGPU).

There is a inherent difference between a GPU and a CPU. Therefore, some algorithms may not translate from CPU to GPU. Problems are often caused by restrictions in the model GPUs are developed. The device memory is handled differently in GPUs and CPUs. Hence some algorithms and data structures do not directly translate from CPU to GPU. For example, modern GPU development frameworks do not allow some programming constructs available in popular programming lan-

guages such as function pointers and recursive functions.

An important difference between the devices is the memory management model. A GPU is controlled by a host program that initiates tasks and manages the memory. A host is responsible for allocating a block of memory that will store the results of computation. This model can be difficult for some algorithms. For example in the context of FRep, transforming an object from function to a auxiliary data-structure such as polygonal mesh requires some memory which is not known a priori of computation.A CPU in the other hand has access to computers main memory and can dynamically allocate some memory if needed.

1.2 Problem statement

Several FRep applications, especially interactive modelling, are problematic because of simultaneous need for fast and high-quality visual feedback. Using FRep for a visual application implies two processing stages: sampling of the defining function and the rendering of the shape. A slow execution of either stage can result in interruptions or slowdowns. In geometric modelling, the slowdowns are disturbing and in the worst case, some geometrical features can be missed if an user or application proceeds to next iteration without waiting for the previous results.

From perspective of parallelism FRep function evaluation is simple as generally sampling at a point does not rely on the results at other points. Therefore, each evaluation of the tree can run independently and therefore in parallel. However, one has to keep in mind the limitations of graphics programming units and their development frameworks. For example, some complex models are difficult to implement because of the restrictions.

The goal of this research is to investigate how to achieve efficient function evaluation by using the computing resources present in a modern computer. The focus is on how to efficiently distribute workloads to processors seen in consumer-level hardware. A wide range of FRep prim-

itives and operations are observed in order to determine the feasibility of the used methods.

The initial state of the research consists of:

- The existing FRep function classifications.
- A survey of techniques to improve the efficiency of function evaluation.

A set of operations to achieve the research goal includes:

- Understanding of available computing hardware and their computing capabilities for the purpose of efficient function evaluation.
- A study on how to effectively control the hardware how to distribute work loads on them.

1.3 Contributions

This thesis presents a discussion on the problem of efficient evaluation of the defining function for the model represented with Function Representation. An observations is made on wide range of the nodes and primitives within Function Representation along their suitability for parallelization.

A novel approach to distribute workloads on parallel hardware is presented in order to efficiently evaluate the defining function for a large set of points. The presented method circumvents some problems present in the current methods.

Finally, in this thesis we present empirical studies on application of graphics hardware for a wide range of models defined with Function Representation. The extend of these studies are not seen in previous work where the focus is generally only on a small subset functions. From the empirical result it is possible to observe the suitability of FRep primitive and operation evaluation on parallel devices.

Chapter 2

Related work

This chapter is split into three distinct parts, the first part a defines of function representation along with the tree structure and its traversal. The second part gives a brief description on modern consumer-level hardware and their parallel features.

In the third part, a survey is provided on existing methods of efficient FRep function evaluation. An analysis is made on the strengths and weaknesses of methods. The part also contains brief description on available rendering methods that partly determine how the sampling is conducted. A study on marching cubes polygonization algorithm also provides an insight into some of the problems in the GPU development models.

2.1 Function representation

Function representation (FRep) can be presented as an algebraic system (Pasko and Adzhiev 2004):

$$(M, \Phi, W) \tag{2.1}$$

Where M is a set of geometric objects, Φ is a set of geometric operations and W is a set of relations for the set of objects.

A geometric object in Euclidean space E^n is defined as $f(\mathbf{p}_1, x_2, \cdots, x_n) \geq 0$, where f is a defining function and the inequality is called a function representation of a geometric object. The requirements for the function is to have at least C^0 continuity. The inequality defines a closed n-dimensional object in E^n space with the following characteristics:

- $f(\mathbf{p}) > 0$, for points inside the object,
- $f(\mathbf{p}) = 0$, for points on the object's boundary,
- $f(\mathbf{p}) < 0$, for points outside the object,

where $\mathbf{p} = (\mathbf{p}_1, x_2, \cdots, x_n)$ is a point in E^n. In the context of FRep, the boundary of the object is also known as the zero-set.

In practice an object constructed from geometric operations and geometric objects results to a tree-structure of nodes. Internal nodes of the tree are geometric operations and leaves of the tree are geometric objects. The geometric object expressed as a tree-structure can be modified by adding nodes or removing nodes from the tree.

2.1.1 Tree traversal

An evaluation of the tree consists of initiating an traversal algorithm that visits each node of the tree. The evaluation is divided into two separate phases.

In the first phase the tree is recursively traversed from the root to the leaves; Space mappings, if present, are applied to the input coordinate in which the function is to be evaluated, the space mapping modifies the point coordinates. The succeeding nodes in the branch are evaluated with the modified point coordinates, until either a leaf is reached, or another space mapping node occurs. See Figure 2.1 As a result of the first pass, each node in the tree is associated with point coordinates that the node is to be evaluated in.

After the first pass, the second phase is initiated. The second phase evaluation starts from the leaves and ends at the root node. Each leaf

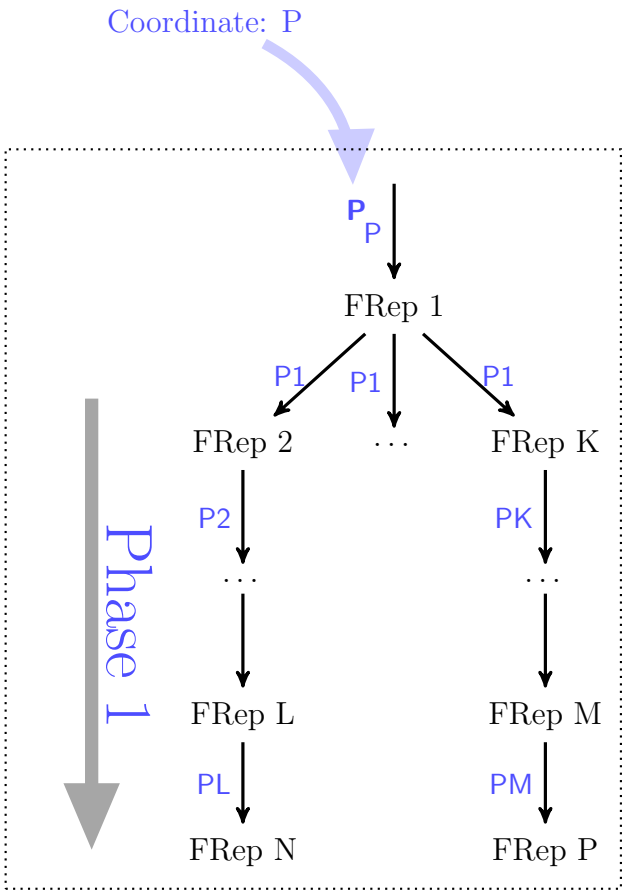


Figure 2.1: *The first phase of function evaluation. The image is a reproduction from (Kravtsov 2011)*

node is evaluated in its associated space, resulting in a scalar value, and the result is passed to the higher level, until root node is reached. A tree may contain function mapping operations that modify the values received from the evaluation of its operands. For example a set-theoretic operation such as union receives scalar values from both its operands and operates the values accordingly and returns a scalar. The procedure is recursed until the root is reached which will yield the result of the tree.

2.2 Parallel hardware

Modern hardware is capable of several types of parallelism, these are Instruction-level parallelism(ILP), Data-level parallelism(DLP), and Task-level parallelism(TLP) (Hennessy and Patterson 2011). From the three

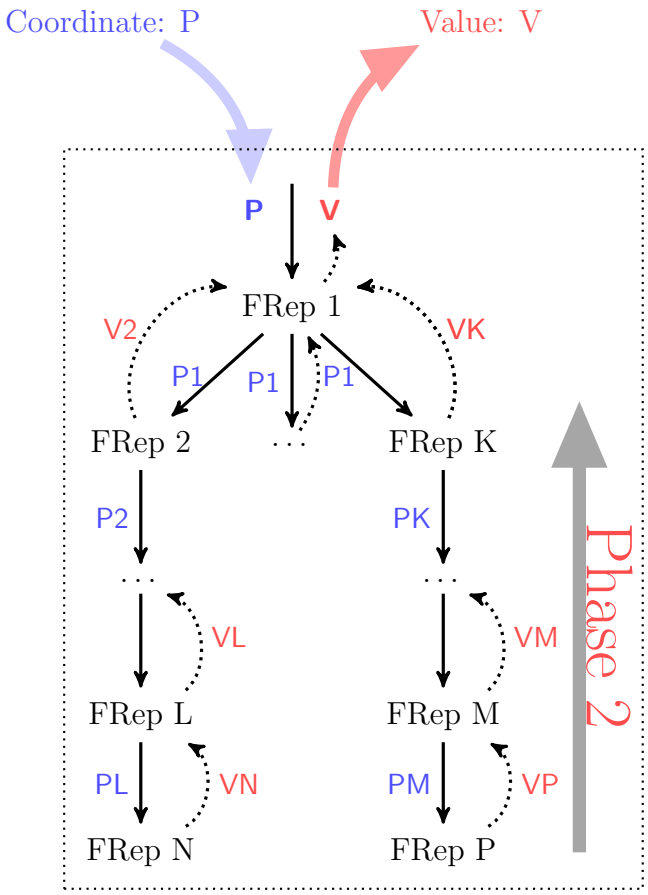


Figure 2.2: *The second phase of function evaluation. Image from (Kravtsov 2011)*

types of parallelism DLP and TLP are most relevant to FRep and these are discussed in the following sections; ILP is mainly a hardware detail which a developer can not actively influence. Some devices are specifically built for certain task. For example, graphics processors heavily uses data-parallelism in order to provide performance benefits.

There are differences between on how different devices execute parallel task and how the devices are controlled. Graphics processors require a host program that manages the device. This includes the memory and program executions on the device. In the context of GPUs the programs executed on the devices are called kernels or shaders. A shader is generally used for a programs that deal with graphical tasks. Kernel is a used in general purpose computing. General purpose computing on graphics processors (GPGPU) is a fairly new approach on facilitating the

devices. This model of using mixed processors types for computation is *heterogeneous computing*.

In contrary to a GPU, a program execution on a CPU executes a program by following a thread of instructions. From the main thread it is possible to instruct the operating system to start execution of secondary threads. The operating system will generally make the decisions on how to run the threads, it can migrate threads to idle processor if present or schedule a single processor. The synchronization between the threads is done via specialized structures and functions. A modern multiprocessor system contains several processors, where each processor is capable of running its own thread in parallel with other processors. Each processor has an access to the main memory, and therefore is capable of allocating and freeing memory from the main memory without intervention from the main thread. This memory model can be difficult as generally the developer has to explicitly make sure threads access certain memory in desired order to avoid unexpected behaviour.

2.2.1 SIMD

Single-Instruction, Multiple-Data (SIMD) is a class of computer architecture in Flynn's taxonomy (Flynn 1972). These architectures provide is a form of data-parallelism. All computational devices are operated by specific instructions, and the instruction sets are generally device specific. However, some more generic instruction sets exists that can operate multiple devices such as Intel's popular x86-instruction set. In a classic instruction set architecture (ISA) an instruction operates singular scalar values. A SIMD instruction on the contrary operates multiple scalar elements simultaneously.

In a simplified view, a processor is capable of executing a single instruction per a clock-tick. A scalar processor processes a single scalar instruction per a clock-tick. A SIMD processor can process multiple elements with a single instruction per a clock tick. Therefore, effective the usage SIMD processors can improve the execution times by reducing the

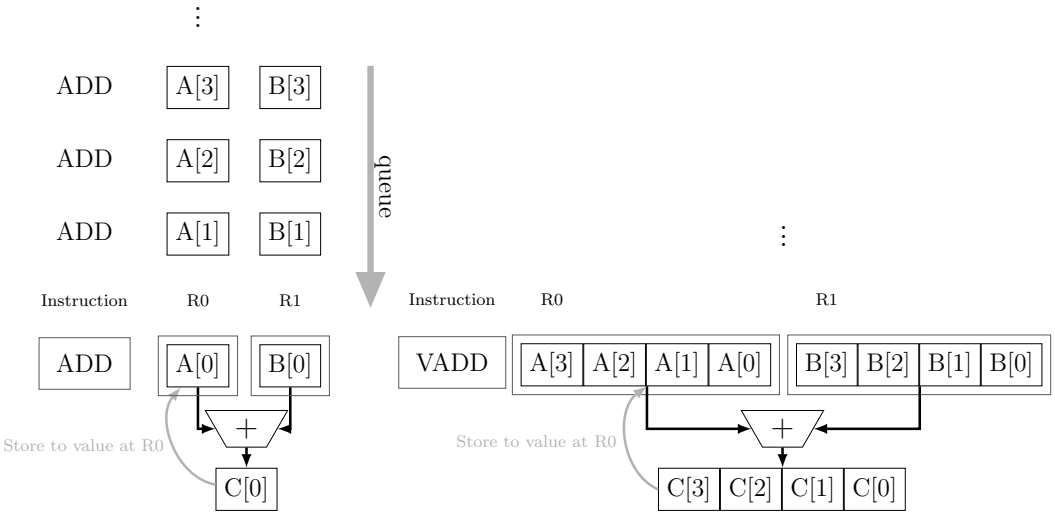


Figure 2.3: *On left a scalar ADD instruction executed on two 32-bit registers containing values A[0] and B[0]. To execute four float additions, a scalar instruction is executed for each pair of operands in a sequence. On right, a vector VADD operation executed on two 128-bit registers, each containing a vector of four 32-bit values. To add two vectors together only one instruction is required. Both scalar and vector addition requires one clock cycle per instruction.*

number of instructions.

SIMD in CPU Generally a CPU operates on scalar values, however especially multimedia applications apply same task to multiple floating point values. Therefore, computations of such profile can benefit from SIMD instructions.

A modern CPU contains multimedia extensions that gives SIMD support. In practice, the scalar instructions are extended with SIMD instructions, and the hardware is extended with special arithmetic units and registers to support the multi-element operations (Stokes 2006). A CPU implements SIMD computation as operations on multi-element *vectors*. See Figure 2.3 for a visual demonstration, in which an add operation is visualised as a scalar and a vector operation.

SIMD in GPU A GPU is a device specifically built for data-parallel problems. The SIMD approach of a GPU and a multiprocessor system

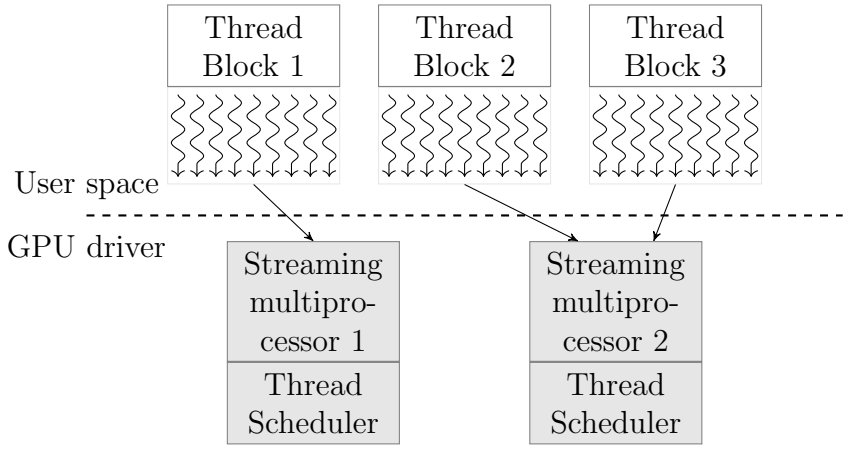


Figure 2.4: *Most modern GPUs have a two-level scheduling: user defined thread blocks are first scheduled to streaming processors, whose thread schedulers make sure all the processors are kept busy.*

differs fundamentally. A single CPU can execute vector instructions using specialized instructions and arithmetic units, whereas modern GPUs have individual *streaming processors* for each element.

In modern development frameworks such as OpenCL, an user generally develops a program considering only a single element. When deploying the program for execution, the user targets program to process a number data elements. Each element requires the thread and the input data to be processed. Before the deployment of execution, an user effectively forms a collection of threads, where each thread is associated with some data. This collections is called a *thread block*. The units are deployed to a device, which will schedule the tasks to the available processors. See Figure 2.4.

A *streaming multiprocessor(SM)* is a collection of streaming processors and has a *thread scheduler* which is responsible for keeping all the processors busy. In program execution, a streaming multiprocessor receives a thread block and divides it to threads of SIMD instructions. For example, a collection of 32 elements and their threads forms a single thread of 32-wide SIMD instructions. The thread scheduler picks individual SIMD instructions and gives them to the collection of processors the SM overlooks. Therefore, a SM is capable of executing multiple SIMD threads at a time by picking instructions from different threads whenever they

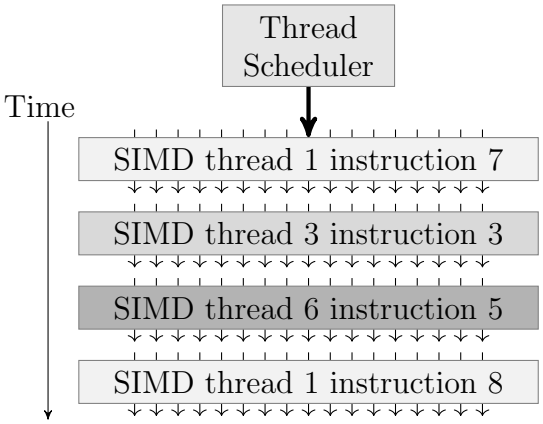


Figure 2.5: *Scheduling of threads of SIMD instructions. SIMD instructions are scheduled to a group of processors. By definition the instructions can be independently, therefore the scheduler fetches an instruction that is ready for evaluation (Hennessy and Patterson 2011).*

are ready for execution. See Figure 2.5.

The number of processors a SM overlooks is dependant on the architecture, for example NVidias Fermi architechture has 16 Streaming multiprocessors each having 32 cores (Nvidia). Because of the difference between traditional vector based SIMD approach and the GPU approach, NVidia refers this type of execution model Single-Instruction, Multiple-Thread (SIMT).

2.2.2 Development frameworks

Often parallel devices are developed using specialized frameworks. OpenCL is a framework for writing programs that execute on highly-parallel devices. From OpenCL framework it is possible to deploy programs to different types of devices and it is used extensively in the work presented.

The framework is split into two separate entities, a host and a device, where the host is responsible for controlling the device. The host initiates task executions and manages the device memory. Communication between the host and the device is done through a command-queue. The commands include memory management commands, kernels deployment, and synchronization commands.

The framework has a specialized language for high-performance programming: OpenCL C. A program written in OpenCL C can be compiled and deployed to different types of execution platforms. This is achieved by having different hardware vendors to provide their own back-end compiler for their device.

Each device has its own implementation of the OpenCL specification. A *platform* construct contains a data and services of a specific OpenCL implementation and provides a compiler to translate OpenCL C to instructions that comply with platform specific Instruction Set Architechture(ISA). Each connected device that implements the same ISA is recognized as a same platform. As an example, a GPU and CPU execute code with a different ISAs, hence OpenCL recognizes them as separate platforms. A platform is currently vendor specific: a GPU from a vendor is not compatible with a GPU from another vendor.

A device is associated with a context, which manages its program and memory objects. Complex system can contain several contexts and several devices per context. Also, a device can be part of several contexts, which can be of benefit in a case where one context is preparing data, and another context is processing the data. In a regular consumer setting the usual case is to have one context to handle a single CPU or GPU device.

One segregating feature between the supported devices is their memory model: a regular desktop GPU has its own device memory, a CPU in the other hand uses the main memory of the computer. The device specific memory requires memory transfers from the host to the device memory. A CPU however is capable of accessing the same memory as the host program. All command initiations contains small overhead costs, therefore the model OpenCL implies may not be the most optimal way to control a device.

2.3 Fast Rendering of FRep shapes

A boundary of a three-dimensional solid object separates space into interior and exterior. In order to visualise the solid, sampling of the function is performed in E^3 space. The sampling process is often related to the rendering method. For example, in direct rendering methods the samples are generally placed on rays that are cast from a camera overlooking the scene. Indirect methods often transform the defining function into an auxiliary data-structure, such as polygons.

2.3.1 Auxiliary data-structures for rendering

A boundary representation (BRep) of a solid object is a way of representing an object using only the limits of a shape; the boundary between solid and non-solid. The polygonal representation is the most widespread form of BRep.

A polygonal representation provides a way to approximate surfaces and curves with a small set of topological elements: lines, edges and faces. A shell is a set of connected faces that can be fitted to resemble the surface of an object. By increasing the density of the faces in the shell, a better approximation of the surface is achieved.

Polygonal representation has a prominent status in many fields of computer graphics. Polygons provide relatively easy means for a user to modify the boundary of an object, and good control over localized details.

The polygonal mesh is used as an approximation for the shape defined by a function where the quality of the approximation depends on the density of sampling. Denser sampling results in better approximation and generally more polygons. However, adding the sample count also increases the computational costs. If visual appearance is of concern it is possible to achieve seemingly smooth surface using shading algorithms (Phong 1975). Therefore, adding polygons is may not be the most efficient way to achieve some visual results.

Marching Cubes To extract a boundary from a FRep object, a zero-set of the function is found by sampling the function in E^3 space. A popular method to extract the boundary is to use spatial partitioning algorithm: marching cubes (MC) (Lorensen and Cline 1987).

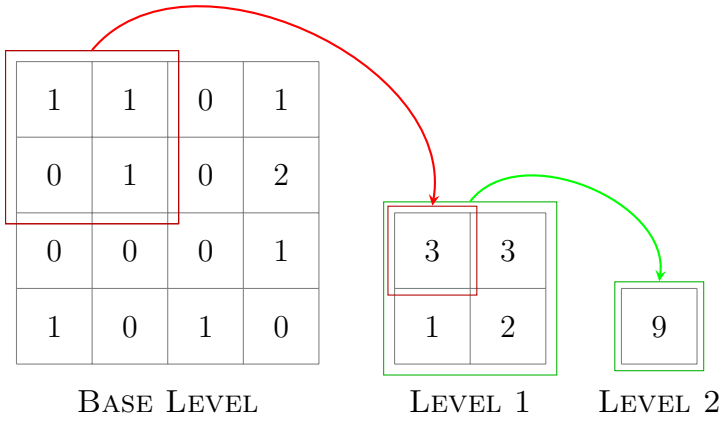


Figure 2.6: *Bottom-up build process of 2-dimensional HP, adding the values of four texels repeatedly. The last level contains the total number of triangles in the pyramid. The figure is a reproduction as seen in (Dyken* et al. *2008).*

Other polygonization algorithms exists (Pasko A. A. 1995). However, MC has gained attention as an efficient way to polygonize boundaries of functionally represented objects. Proposed optimization methods to MC generally use different parallel architectures for better performance (Shirazian *et al.* 2012) (Dyken *et al.* 2008).

The MC algorithm partitions space into axis-aligned n-cubes, referred as voxels. Overall, the spatial partitioning results in a lattice called a voxel grid. The corners of the n-cubes are the sampling locations. After the grid has been sampled, the MC marches from a voxel to voxel and for all voxels containing the boundary, polygons are created.

The MC algorithm is a parallelizable as each of the voxels can be analyzed separately. However, the algorithm does not know the number of triangles a priori of computation. This is problematic for the current development frameworks as memory can only be managed by a host program. Because of the model, a new step has to be introduced to MC

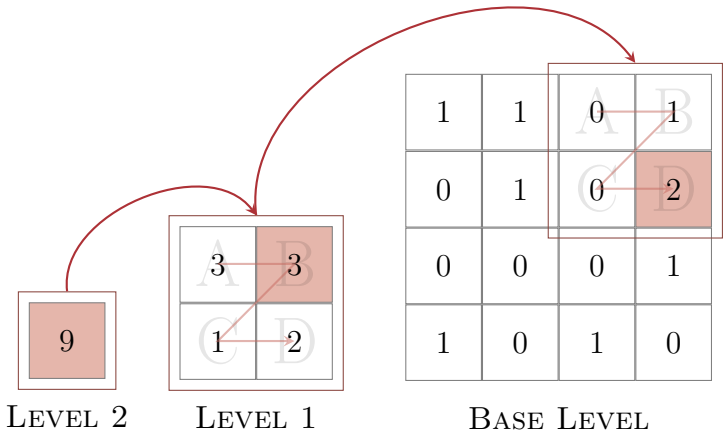


Figure 2.7: *A pyramid is accessed by using a key k. A pointer p, a current level l, and a local key k_l are maintained during traversal. Given $k = 5$, the level below ranges equal to $A = [0, 3)$, $B = [3, 6)$, $C = [6, 7)$, $D = [7, 9)$. In the first step the k_l falls to range B. The p now points to B. In second step $A = [0, 0)$, $B = [0, 1)$, $C = [1, 1)$, $D = [1, 3)$ and $k_l = 5 - 3$, hence p ends up pointing to a texel D. The figure is a partial reproduction of a figure seen in (Dyken* et al. *2008).*

algorithm that calculates the number of triangles that will be generated. Only then right amount of memory can be allocated.

The MC algorithm creates triangles relative to the respective voxel. Therefore, to correctly place the triangle, a polygon has to be associated with a correct voxel. This further complicates the newly introduced pre-processing step, not only the amount of triangles is required but also information to which voxel the triangles belong.

To accommodate the above requirements Dyken *et al.* use a HistoPyramid data-structure (Ziegler *et al.* 2006). The HP algorithm is divided into two stages. In the first stage a HistoPyramid is constructed as a bottom-up process. The construction starts by laying a base level, which contains the number of triangles per voxel. In this step the voxels are sampled.

An overall count of the triangles is calculated by constructing a pyramid structure. The structure is constructed level-by-level, where each new level is constructed from elements that sum values from values of a

previous level. This is done recursively until a level only contains singular scalar value. The figure 2.6 illustrates this procedure. The scalar value represents the number of triangles the MC algorithm will produce.

After memory has been allocated for the triangles, the pyramid is traversed from top-down to associate the triangle with a correct voxel. See Figure 2.7.

2.3.2 Direct rendering

In direct rendering methods, a functionally defined object is generally visualised by casting rays at it (Hart 1993). In FRep, ray casting can be used to find the zero-set surface either analytically or numerically. The analytical solution is possible only for a very limited set of algebraic functions hence a numerical solution is widely used.

A popular method to visualise the boundary of a solid object is volume ray casting. Ray casting is often implemented as ray marching algorithm where the defining function is sampled along the rays in a step by step basis. In the zero-set extraction, a ray progresses through the scene until a step interval contains a zero-value. To determine whether a step interval contains a zero value, the end points of the interval are analyzed. If either of the sample values at the end points is zero, the boundary is at that location. If the function values at the endpoints of the step interval change from negative to positive or vice versa, the boundary of the solid lies within the step. See Figure 2.8.

Ray casting methods are usually executed in a screen-space where a ray is per a pixel. In practice a virtual camera is placed in E^3 space, and according to the size of screen resolution, a number of rays are cast from proximity of the camera. Each a ray is stepped until an interval is marked to contain a boundary. In case a ray has a marked interval, the respective pixel is shaded.

A ray marching algorithm results to a camera dependent approximation of the surface location. Higher quality approximation is achieved by increasing the number of steps, for example by reducing the step size. In

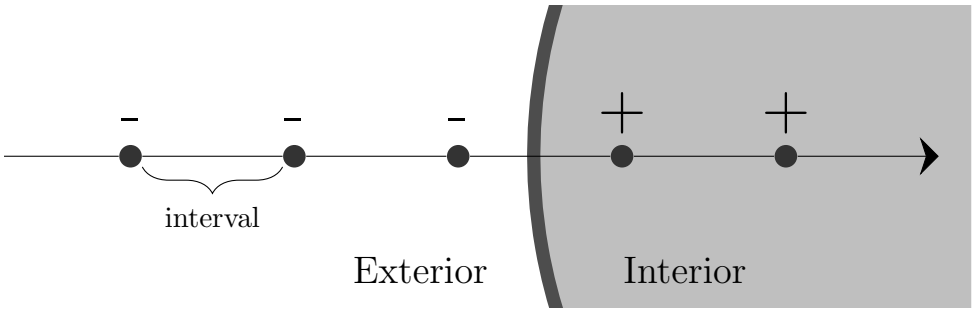


Figure 2.8: *In ray marching, a ray progresses through the scene until sampling at either of the intervals end points returns a zero or the values have different signs. The boundary will be within the interval if the values have opposite signs.*

adaptive approach the marked interval's endpoints are used to construct a start point and an end point for a new ray, and the new ray is stepped with a reduced step, and the procedure recursed until pre-defined exit conditions are met.

2.4 Survey on efficient FRep function evaluation

A modelling process of an object is an iterative trial-and-error process. Therefore, it is crucial for a modeling software to provide quick visual feedback. To attain interactive update rates, a combination of both efficient function evaluation and fast rendering method is required.

A number of papers tackle the problem of improving efficiency of sampling. All methods can be roughly classified into:

- Algorithmic methods, where the efficiency is obtained by minimizing the number of queries by using characteristics of the method itself;
- Hardware methods, where the algorithm is implemented efficiently when specialised hardware is used;

The following subsections give an survey on both of these methods.

2.4.1 Hardware methods

Hardware methods achieve efficiency by facilitating hardware features. These methods generally use the fixed graphics pipeline that contain several programmable stages. In the last stage where that outputs the final image, objects can be visualised by sending rays from pixels. If a ray hits and object, that pixel is shaded (Fryazinov and Pasko 2008).

A tool based on the mechanism has been proposed by (Reiner *et al.* 2011): a modeller based on the direct rendering method using a GPU for acceleration. In Reiner *et al.* (2011)'s work the render algorithm is fully executed on a GPU. See Figure 2.9. This is achieved by constructing a shader that expresses the tree structure. There are several drawbacks on the fully GPU-based evaluation:

- Whenever a structural modification to the function tree is made a recompilation of a new shader is required.
- In modern development frameworks, the memory has to be allocated a priori of computation. Therefore, memory requirements of a complex tree containing multiple nodes with big memory needs could exceed the available memory.

In Reiner *et al.* (2011); Fryazinov and Pasko (2008)'s work he transformation of a defining function into a GPU executable shader consists of following steps:

- The host application has a modifiable tree structure that represents the current state of the object. When the structure is modified, the host application traverses it and constructs a textual representation. As an example, G_3 is an object that is a result of a binary operation using objects G_1 and G_2 as operands. The mathematical notation for such an object is:

  $G_3 = \Phi_i(G_1, G_2)$,

  Therefore, the defining function of G_3 is:

  $f_3 = \Psi(f_1(\mathbf{p}), f_2(\mathbf{p})) \geq 0$

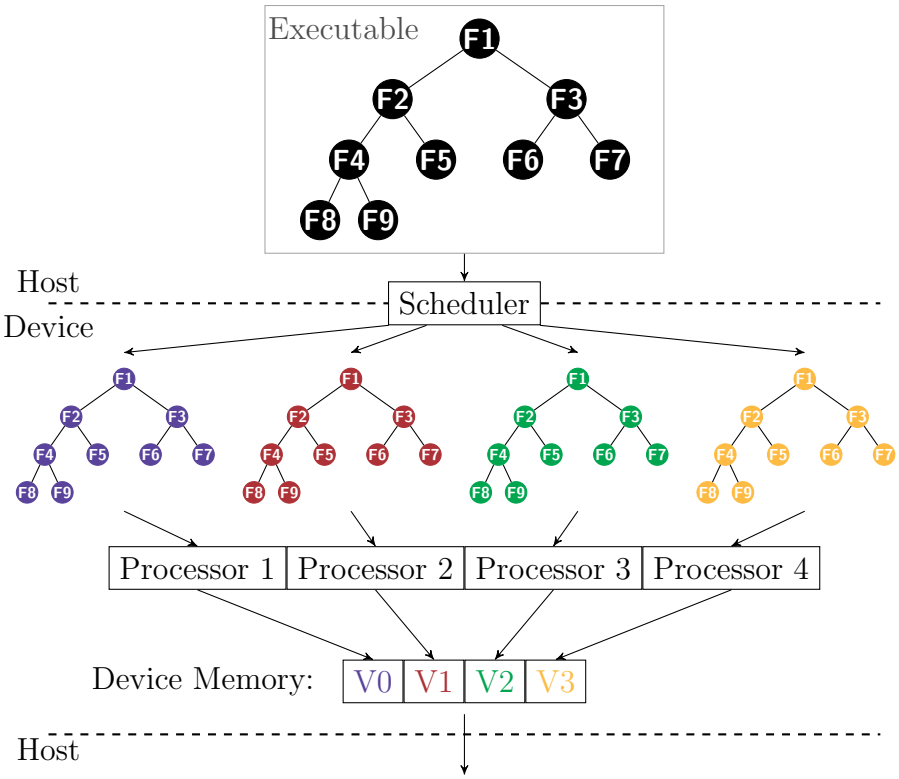


Figure 2.9: *An instance of the whole FRep tree is given to each execution unit. An execution unit in case of a GPU is a streaming processor, and in case of a multiprocessor system: a CPU or a core.*

where Ψ is a continuous real function of two variables. Consider a case of a union of two primitive shapes, a sphere and a torus. In imperative programming language, the representation would look similar to:

```
f3= Union( Sphere( p ), Torus( p ) );
```

- This function string is appended to a pre-existing text that contains the textual definitions for the primitives and operations. In this case, the pre-existing file would contain definitions for a sphere, a torus and a union.
- The full textual representation gets compiled into the device executable program.
- Finally, to evaluate the function, the host program initiates the evaluation with a set of sample points, which a scheduler distributes to the available processing resources.

The method has few problems. Every time a mutation on the tree structure happens, a new shader has to be created and compiled. Parameters of the nodes can be stored read from memory so re-compilation is only needed if there is a structural change in the tree. The compilation can cause slowdowns during applications runtime, which results in annoyances and in the worst case some geometrical features can be missed because of the slowdowns.

Another problem is caused by the memory management model of current development frameworks. All the data the tree requires has to be allocated and transferred to the device before the evaluation step is initiated. In case of a tree that contains multiple complex nodes that have large memory requirements the memory could run out before the evaluation resulting in a failure.

Additionally some nodes, as seen in following chapters, have evaluation profile which can be difficult to express in a single shader.

2.4.2 Algorithmic methods

A range of modelling tools polygonize the zero-value isosurface of the defining function (Schmidt *et al.* 2005a; Schmidt and Singh 2010; Alexe *et al.* 2004). The papers describing the tools typically do not provide in-depth technical discussion on the function evaluation procedure. However, several techniques are discussed to reduce redundant function evaluation: a modelling process generally modifies only a local area of a function tree, hence only modified subtree and affected nodes require function re-evaluation. To avoid redundant re-evaluation of a subtree the intermediate results can be stored in cache for further queries (Schmidt *et al.* 2005b; Reiner *et al.* 2012).

Adaptive sampling is another technique where sample placement is driven by previous sample results (**?**). However, the technique does not necessarily reduce the number of function evaluations but usually leads to better results with the same number of samples than exhaustive enumeration. However, implementation of these techniques is often complex

and may not suit the GPU development frameworks.

The tools use only a small sub-set of shapes and operations. These tools are often specialized on a single method of modelling, such as modelling using sculpt-like methods.

2.5 Conclusion

From the discussion above, it is clear that FRep function evaluation can benefit from using parallel hardware and many industry and academic tools already facilitate these features. However, many of the tools focus only on a subset of FRep operations and no extensive study exist whether the tools are able to handle arbitrary shapes.

The GPGPU is a fairly new approach for parallel software development, therefore the previous work is mostly done using the graphical pipelines that to some extent restrict the methods used. For example, the direct rendering methods generally use the fragment shader stage that produces visual output, which is efficient but limit the use only to visual applications.

Even with the general purpose frameworks, some limitation still exist for the ways the GPU can be used. This is apparent in the parallel MC algorithm, which requires an additional step in order to determine memory requirements.

In this work, an extensive study is done using wide range of operations and primitives to determine the suitability of using state of the parallel hardware and frameworks for function representation.

The existing hardware methods have few problems, for example an explicit shader creation and compilation that can cause slowdowns during applications runtime. Implementation of complex nodes is also difficult using these methods. In this work, a solution is presented that overcomes these problems.

Chapter 3

Efficient function evaluation

In the first part of this chapter, a proposal of a new novel way to distribute sampling workloads to parallel devices is described. This method overcomes some of the problems seen in the previous methods, such as abolishing the need for explicit compilation step.

In the previous work, the methods often specialize on a small subset of nodes and no proof is given whether they are suitable for wider range of functions. Therefore, a goal of this work is to build a method that can handle majority of the existing primitives and operations in FRep.

The second part of this chapter describes a classification for a existing FRep functions. The variety of FRep functions are classified based on (Pasko and Adzhiev 2004). In this chapter, a formalization of classes is presented. The classifications is used as a basis for empirical results, which can be used to identify whether a class of primitives or operations is suitable for GPU evaluation.

3.1 A multikernel approach to distributed function evaluation

The problem of transforming the mathematical formulation into a parallel program currently has an solution where a program is constructed

that represents the whole FRep tree structure (Fryazinov and Pasko 2008; Reiner *et al.* 2011). However, this approach has few problems. During applications runtime, a compilation step is required for each new tree structure and memory management in current development frameworks has to be done before execution of the program. In the following text, this method is called *single kernel* method.

In this work an alternative method is proposed where each geometric operation and primitive is as a small program or multiple programs. Similarly to the previous work, the host application contains a modifiable tree structure that represents the shape. During evaluation, the structure is traversed and on each node a program respective to the node is deployed to execution device. In the context of FRep modelling, this evaluation model has similarities to the interpreted HyperFun language used for modelling (Adzhiev *et al.* 1999). The proposed method is called *multikernel* method. A more detailed description is given below.

3.1.1 The multikernel method

In the proposed multikernel method, a set of operations and primitives are pre-defined and pre-compiled into a set of small programs. The host application contains a tree structure constructed from elements that contain evaluation instructions for the nodes. During function evaluation, the tree is traversed and the host takes appropriate measures to evaluate each node with the instructions within. In pratice when visiting a node: memory can be managed and a small program is deployed to a parallel device. See Figure 3.2.

Each node can be implemented agnostic the general structure. Therefore, the tree can contain nodes that have different evaluation profiles. In practice, the multikernel method allows partial distribution of the tree to different types of devices. For example, some parts of the tree can be executed on a multiprocessor device, and other parts on a GPU device. Therefore, if certain node is deemed GPU-unfriendly, it can be evaluated on a CPU instead. One example of such node is a complex mesh that

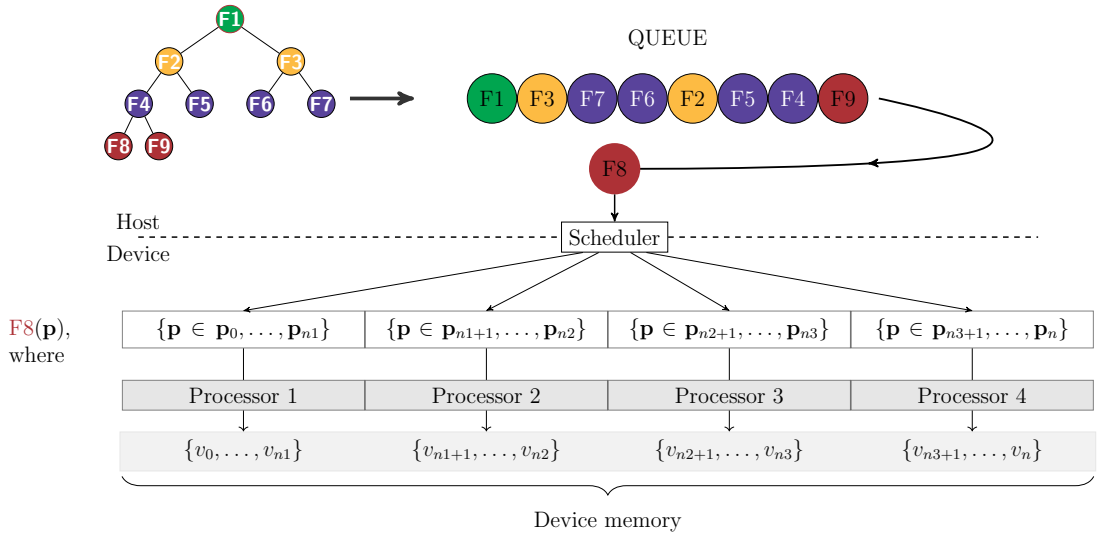


Figure 3.1: *When evaluating nodes individually, a node is distributed amongst the execution units and results stored for subsequent nodes that require the results. The first step is to turn the tree into a queue of tasks; In this example Post-order tree traversal is used. An execution unit in case of a GPU is a streaming processor, and in case of a multiprocessor system: a CPU or a core.*

could too big memory requirements for a GPU. It is therefore important to determine whether a certain node is a good fit for GPU evaluation.

Compared to a single kernel method, in which the host application controls memory only before initiation of the program, the multikernel method gives control to host application in each node. This is important for memory management purposes: temporary memory that is required by the node can be allocated when needed and freed after use. Also, the parameters to specific nodes are defined on a node basis in contrary to single kernel method that defines node parameters before the function evaluation.

In the multikernel method each node is evaluated in all sample locations before continuing to next node. Essentially the results of each node evaluation are temporarily stored as a cache, similarly to (Reiner *et al.* 2012). Caching can be used to avoid unnecessary evaluation of nodes that are not affected by modification. However, temporary storage of node results can be problematic in systems with limited memory, especially when conducting dense sampling of the function.

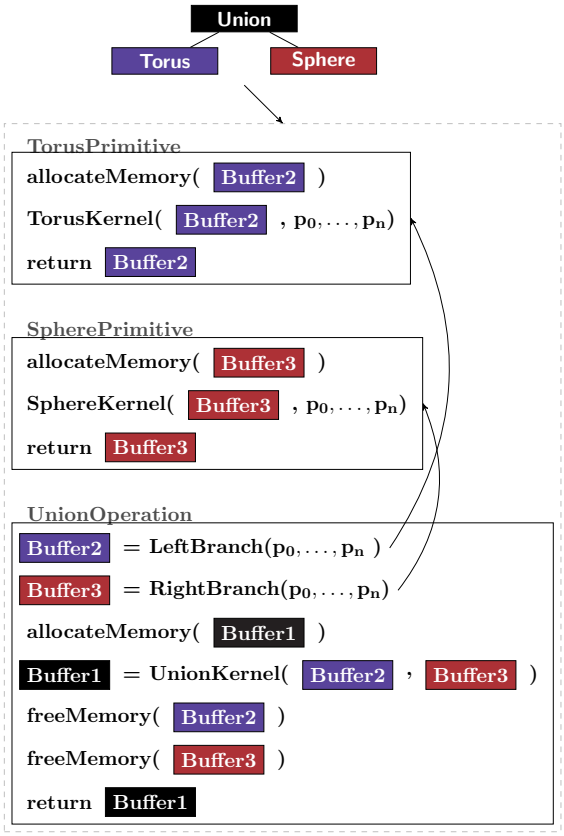


Figure 3.2: *The host side instructions for a simple tree. In the evaluation post-order traversal is used to evaluate the tree; on each node, memory is managed and kernels initiated.The keep track of allocated GPU memory the host uses reference counted handles.*

3.2 FRep function classifications

The set of geometric objects M, and the set of geometric operations Φ can be classified from multiple perspectives. This section describes the current classifications and explains the distinct characteristics of each category.

3.2.1 Geometric objects

The leaf functions are currently classified two separate classifications. The first classification divides leaves according to their defining function:

- Algebraic function: The function can be defined by a polynomial equation. Polynomial equation is expressed as finite sequence of

terms involving only algebraic operations: addition, subtraction, multiplication, and raising to a fractional power. A wide variety of geometric primitives can be modeled using simple operations. Torus in E^3:

$$f(\mathbf{p}) = (R - \sqrt{x^2 + y^2})^2 + z^2 - r^2, \tag{3.1}$$

where R is the distance from the center of the torus to the center of ring shape defining a torus, r is the radius of the ring, and x, y, z are the coordinates of a sample $\mathbf{p}$.

- Analytical function with arbitrary closed-form functions and expressions: In closed-form solutions a function is expressed in terms of functions and mathematical operations from an predefined set. In geometric modelling the predefined set extends to cover trigonometric functions such as sine and cosine functions. An example of such function is the Blob (Bourke) which defines the object as:

$$f(\mathbf{p}) = x^2 + y^2 + z^2 + sin(4x) + sin(4y) + sin(4z) - 1,$$

where x, y, z are the coordinates of a sample $\mathbf{p}$.

- Procedural function, non purely analytical involving at least some constructs of imperative programming: Typical constructs of imperative language includes if-statements, switch-statements, for-loops, and while-loops. Iterative primitives that require exit conditions are defined by using if-statements to branch the execution if needed. As an example, HyperFun fractal(Hyp) as seen in Algorithm 4 in Chapter 4.

The second classification of leaves distinguishes the primitives according to their data requirements:

- No external data required: Simple algebraic and analytical functions do not require external data for the function definition.

- Use of restricted external data: As an example a complex geometric object can be modeled using a skeleton structure. The defining

function $f(\mathbf{p})$ is obtained via a convoluting a geometry function $g(\mathbf{p})$, with a kernel function $h(\mathbf{p})$ (Sherstyuk 1999):

$$f(\mathbf{p}) = g(\mathbf{p}) \star h(\mathbf{p}) \tag{3.2}$$

The kernel function defines the distribution of some potential, that is produced by each point on the object. A geometry function is defined as $g = \sum_{i=1}^{N} g_i$, where the skeletal elements g_i are generally: points, line segments, arcs, triangles and planes.

- Essential data of large size: A complex geometric object that is defined by a large data set. As an example, a manifold geometric object represented as a triangle mesh requires an inside-outside segmentation to be used in conjunction with the rest of FRep functions. Several algorithms providing inside-outside segmentation are available (Jacobson *et al.* 2013).

3.2.2 Geometric operations

The set of geometric operations Φ contains unary, binary or k-ary operations:

$$\Phi_i : M^1 + M^2 + \cdots + M^n \rightarrow M, \tag{3.3}$$

where n is a number of operands on the operation. A unary operation on object G_1, with a function of $F_1(\mathbf{p}) \geq 0$, is described as $G_2 = \Phi_i(G_1)$. A binary operation on objects G_1 and G_2 is described as $G_3 = \Phi_i(G_1, G_2)$.

The set of Geometric operations Φ can be roughly divided into two main categories, geometric space and function mappings, however extension to this categorisation exists (Savchenko and Pasko 1998). Essentially all operations are transformations of an initial object or objects that can

be described by a system of equations:

$$\mathbf{p}^{'} = \phi_1(\mathbf{p}, \xi)$$
$$\xi^{'} = \phi_2(\mathbf{p}, \xi)$$
$$\xi = f(\mathbf{p}),$$

where $\mathbf{p}^{'} = \phi_1(\mathbf{p}, \xi)$ is a space mapping operation and $\xi^{'} = \phi_2(\mathbf{p}, \xi)$ a function mapping operation. The zero-set $\xi = 0$ is the surface of a geometric object.

The first classification of operations classifies the operations from the tree traversal and geometric transformation point of view:

- Geometric space mappings, evaluated on the top-down traversal pass: In space mapping the geometric object is modified by displacing the coordinate parameter instead of deforming the resulting scalar field. A space mapping, in E^n, defines a relationship between original point coordinates and deformed point coordinates:

  $$\mathbf{p}^{'} = \phi_1(\mathbf{p}) \tag{3.4}$$

  Where ϕ_1 is one-to-one invertible mapping of the subspace E^n. The transformed geometric object is therefore defined as:

  $$\xi^{'} = f(\phi_1^{-1}(\mathbf{p}^{'})) \tag{3.5}$$

  An example of such operation is a Twist (Barr 1984). A render of a twisting of a cube is seen in in Chapter 4 Figure 5.10.

- Function mappings, evaluated on bottom-up traversal pass. The transformed geometric object is defined by:

  $$\xi^{'} = \phi_2(\mathbf{p}, f(\mathbf{p})), \tag{3.6}$$

  Where $\phi_2(\mathbf{p}, f(\mathbf{p}))$ is the mapping of the scalar returned by the

function f. A more practical way to define the function is:

$$\xi^{'} = f(\mathbf{p}) + d(\mathbf{p}), \tag{3.7}$$

where $d(\mathbf{p})$ is a continuous real displacement function. An example of such operation is metamorphosis (Hughes 1992).

- Function-dependent space mappings, evaluated on both traversal passes: Let inverse space mapping function of Eq. 3.5 be defined as :

$$\phi_1^{-1}(\mathbf{p}) = \psi(\mathbf{p}, f(\mathbf{p})) \tag{3.8}$$

where $\psi = (\mathbf{p}_1, x_2, \cdots, x_n)$ is a function generating point coordinates in E^n. The transformed object is therefore defined as:

$$\xi^{'} = f(\psi(\mathbf{p}, f(\mathbf{p}))), \tag{3.9}$$

Hence, as Eq. 3.8 states, the reult of function-dependent space mapping is requires the operand to be evaluated first. Evaluation of a function-dependent space mapping operation therefore requires instructions to be carried out on both tree traveral passes. See 2.1.1.

An example of such operation is offsetting along a normal. To calculate a normal at a given point, the operand function has to be evaluated and from result a normal vector derived. The given point is then offset along the normal vector. A render of the operation applied on a NoiseSphere can be seen in Figure 5.8.

- Combined mapping, evaluated on both traversal passes defined as a usage of geometric space mappings as parameters to function mapping:

$$\xi^{'} = \psi_2(\psi_i^{-1}(\mathbf{p}), f(\psi_1^{-1}(\mathbf{p}))), \tag{3.10}$$

The second classification of operations is based on the complexity of the subtree evaluation:

- Single point function evaluation: The most general case of geomet-

ric operation evaluation, where the function is evaluated using the point coordinates given as a parameter.

- Fixed number of multiple points: The initial object G_1 in E^n is defined by function $f_1(\mathbf{p}_n) \geq 0$. As an example, a geometric object G_2 is a projection of G_1 to E^{n-1} with the defining function $f_2(\mathbf{p}_{n-1})$, which is more or equal zero in the given point $\mathbf{p}_{n-1} = (x_1, x_2, \cdots, x_{i-1}, x_{i+1}, \cdots, x_n)$, only if the point $\mathbf{p}_n$ exists where $f_1(\mathbf{p}_n) \geq 0$ (Pasko and Savchenko 1997). The object G_2 can be thought as a union of cross-sections of G_1 by the infinite set of hyperplanes $x_i = C$. Another way to define f_2 is:

$$f_2(\mathbf{p}_{n-1}) = \cup_{x_i}(f_1(\mathbf{p}_n)) \tag{3.11}$$

  In practice the function result is approximated by selecting only a finite set of hyperplanes. In case of union of the cross-sections of the hyperplanes, $x_i = C_j$, the function can be defined as:

$$f_2(\mathbf{p}_{n-1}) = (((f_{11} \cup f_{12}) \cdots \cup f_{1j}) \cdots \cup f_{1N}) \tag{3.12}$$

  Where $f_{1j} = f_1(x_1, x_2, \cdots, x_{i-1}, C_j, x_{i-1}, \cdots, x_n)$, and N is the number of cross sections. See Figure 5.2.3 in Chapter 4 for a render of a sweep operation applied onto two spheres and a cube.

- Iterative evaluation with exit conditions: As an example, an HyperTextures(Perlin and Hoffert 1989) can be seen as an operation on a shape used to produce rough surfaces. Hart (1997) presents a practical FRep formulation of HyperTextures as:

$$\xi^{'} = f(\mathbf{p}) + 1/v^{\beta} - noise(\mathbf{p}), \tag{3.13}$$

  where v is the frequency and, β controls roughness of the surface. In order to achieve fractal-like shapes through HyperTextures the function can be reformulated to:

$$\xi^{'} = f(\mathbf{p}) + \sum_{i=0}^{N-1} 2^{-\beta i} noise(2^i \mathbf{p}), \tag{3.14}$$

where $noise(\mathbf{p})$ is a scalar valued function: $noise : R^n \rightarrow R$, and N is the number of intervals. By iterating, successive refinement is applied to the surface.

- Recursive evaluation: The evaluation procedure of the function calls itself. No practical examples exist for this class.

The third classification categorises the operations according to external data requirements:

- No external data is required: Operations that only require the definition of the function parameters do not require any external data.

- Use of some restricted external data:

  As an example, consider warping operation that distorts a shape by modifying coordinates in its neighbourhood. The warping can be influenced by using primitives as skeletal elements, for example points (Wyvill and van Overveld 1997) or curves (Sugihara *et al.* 2010). The defining function is described as:

$$f^{'}(\mathbf{p}) = f(\mathbf{p} + f_{bounding}(\mathbf{p}) * d(\mathbf{p})), \tag{3.15}$$

  where $d(\mathbf{p})$ returns the displacement of a point $\mathbf{p}$ and $f_{bounding}(\mathbf{p})$ is the bounding field, generated by convolving skeletal primitives with a kernel function. The result is a scalar value per $\mathbf{p}$ that modulates the displacement. The skeletal primitives require external data for their definitions.

- Essential data of large size: As an example, Free-form deformations (FFD) in general are implemented to require two large lattices. An object is first embedded into a reference lattice, deformations are then achieved by modifying a deformation lattice which reflects as modifications to the object. Hua and Qin (2003) propose a scalar field as the embedding space, where deformation of the scalar field will modify the embedded object.

3.3 Conclusion

In this chapter a novel work distribution, the *multikernel*, method is proposed that overcomes some problems present in methods. In multikernel method the need for runtime compilation is abolished by compiling smaller set of nodes either off-line or at application start-up.

The multikernel method also provides better memory management model to the previous methods. The host application receives control to manage memory on each node. As a consequence the tree can contain complex nodes. Another benefit of splitting the function evaluation into smaller execution units is that it is easier to implement nodes with complex evaluation profiles.

In the second part of the chapter a formalization of the classifications is made to be used as a basis for testing.

Chapter 4

Applications

In this chapter, a description of all the test cases are given. For the test cases both multikernel method and single kernel method are implemented and compared where appropriate. Moreover, an implementation of wide range of FRep operations and primitives are made using the multikernel method. Some of the nodes are complex for single kernel method, therefore only multikernel method is considered for the single node evaluations. As a basis for the test cases the classifications from the Section 3.2 are used.

In the following text, the test for a each class is described. Some more complex nodes are described in more detail in case study subsections. The case studies demonstrates the complexity why some nodes might not be suitable for the *single kernel* method.

In the final part of the chapter a description is given how test case for a complex tree structure is built. In the next chapter the complex tree structures are evaluated using on both *single kernel* and *multikernel* methods.

4.1 Test settings for node evaluations

FRep contains several classes of functions for both geometric operations and geometric primitives as shown in Chapter 3. In order to test how

Leaves	
L1A	Algebraic function
L1B	Analytical function, with closed form functions
L1C	Procedural function
L2A	No external data
L2B	Use of restricted data
L2C	Essential data of large size
Nodes	
N1A	Geometric space mapping
N1B	Function mapping
N1C	Function-dependent space mapping
N1D	Combined mapping
N2A	Single point function evaluation
N2B	Fixed number of multiple points
N2C	Iterative evaluation with exit conditions
N2D	Recursive evaluation
N3A	No external data
N3B	Use of restricted data
N3C	Essential data of large size

Table 4.1: *A simplification of node classifications in form of a graph. L1 and L2 are classes for geometric objects: L1 classifies functions according to their defining function, L2 classifies the functions according to their requirements for external data. N1, N2, and N3 are classes for geometric operations: N1 classifies operations according to their tree traversal method, N2 classifies the operations according to the complexity of function evaluation, N3 classifies the operations according to their requirements for external data.*

different devices handle functions from varying classes, for each class in Table 4.1, a test case was chosen and implemented. The test cases were implemented using OpenCL framework, therefore each implementation includes host side instructions and device instructions written as kernels. To target different devices, a specific back-end compiler compiles the kernel instructions into a device executable program.

In the following listing an implementation for an example from each class is described, with notes and references to related work.

L1A: Torus A simple algebraic function defined in Section 3.2.

L1B: NoiseSphere A function that uses of some closed form functions, such as trigonometric functions. Perlin and Hoffert (1989) describes HyperTextures which can be turned into implicit form and used to create rough surfaces.

L1C: HyperFun fractal The algorithm for HyperFun fractal is defined in (Hyp) and in Section 4.1.4.

L2A: Sphere An algebraic function.

L2B: Convolution surface using lines A convolution surface is constructed from skeletal primitives where the definition of the skeletal primitives generally require some external data. For example, a line segment can be defined by the start and end points (Sherstyuk 1999). The host side instructions for this function are listed in Algorithm 3 later in this Chapter.

L2C: Inside-outside segmentation of a polygonal mesh Some primitives require data of large size, for this test case a polygonal mesh is used. A polygonal mesh requires inside-outside segmentation before it can be used as a primitive. Several solutions to determine whether a point lies within a mesh exists (Sanchez *et al.* 2012), the test case implementation is based on winding numbers and is defined in (Jacobson *et al.* 2013).

N1A: Twist A twist is an operation where a space is twisted using a pair of global basis vectors without altering the third (Barr 1984). A render of a twist operation applied on a cube can be seen in Figure 5.10.

N1B: Metamorphosis The metamorphosis is an operation that allows smooth transition from one volumetric model to another. With two volumetric models, linear interpolation can be directly used for this effect (Hughes 1992).

N1C: Offset along a normal An offset along a normal is a function-dependent space mapping. In our test case the operation first evaluates the subtree to determine a normal vector per point. Secondly, the original sample point is offset in direction of the normal. Finally, the subtree is evaluated in the modified sample point (Savchenko and Pasko 1998). The host side instructions for this node is shown in Algorithm 2 and a render seen in Figure 5.8.

N1D: Not implemented No specific implementation for this class is made. Such operation can be seen as a combination of geometric space mapping and function mapping, however in context of this paper, these are treated as separate entities. Therefore, they are tested on their own (Savchenko and Pasko 1998).

N2A: Union A simple max operation on two scalar values is used.

N2B: Sweeping by a moving solid The host side algorithm for sweeping by a moving solid is shown in Algorithm 1, the kernels are defined in Appendix .1. The operation is defined in (Sourin and Pasko 1996).

N2C: Projection The implemented test case uses Monte Carlo Method to find the global maximum. A random points are generated on a line segments defined within bounded volume. The exit conditions are: the number of sampling points exceeds a predefined number, or the values of last n sample points are within a found range. The n is predefined.

N2D: Not implemented No practical examples of functions of class exist. Moreover, a GPU does not support recursive functions.

N3A: R-Union R-functions operate on real-valued inequalities as differentiable logic operations. (Shapiro 2007)

N3A: Warp Point deformation Warping point is defined in Section 3.2.

N3C: Inverse free form deformation In inverse free form deformation, is implemented as two lattices: reference and deformation. First an object is embedded into a reference lattice, succeeding with modifications to the deformation lattice, which will reflect on the embedded object (Comninos *et al.* 2014). See Figure 5.11 for results and a render.

4.1.1 Case study: sweeping by a moving solid

Algorithm 1: Sweeping by a moving solid Host instructions

Data: Sample set S

Number of subdivisions N

Result: Values $V3$ for the samples in S

Allocate memory for new sample set $S2$, and for sample result set V;

`// Calculate a new sample set` $S2$

$S2 \leftarrow$ SolidSweepStep1(S, N);

$V \leftarrow$ Evaluate(SolidSweep.child$(S2)$);

Free memory of $S2$;

Allocate memory for new sample set $S3$, and for sample result set $V2$;

`// Calculate a new sample set, by interpolating values of` V

$S3 \leftarrow$ SolidSweepStep2(S, N, V);

Free memory of V;

$V2 \leftarrow$ Evaluate(SolidSweep.child$(S3)$);

Free memory of $S3$;

Allocate memory for sample result set$V3$;

`// Finally, calculate the final results for the sample set` S `by finding the maximum value of samples on a trajectory`

$V3 \leftarrow$ SolidSweepStep3$(V2, N)$;

Free memory of $V2$;

return $V3$;

A sweeping by a moving solid operation is one of the complex test cases implemented. In the classification, it is classified according to its complexity of function evaluation. See Algorithm 1 for the host side instructions of the operation. The $S2$ is a set of samples on trajectories; For each sample in S a trajectory is created and N number of samples placed on it. Therefore, the size of $S2$ equals $SizeOf(S) * N$. The subtree of the solid sweep operation is evaluated in $S2$ space resulting to V. To get smoother result the values in V are interpolated according to Newton's interpolation method, the interpolants are used to create new

sample points. The procedure results to a sample set $S3$, which contains N interpolated point coordinates on trajectories for each sample in S. The evaluation of solid sweeps sub-tree at $S3$ results to $V2$. Finally, to get values for $V3$: For each trajectory, union all the samples on it, and append the result to $V3$.

See Section 5.2.3 for a render. See Appendix .1 for the kernels used: SolidSweepStep1, SolidSweepStep2, SolidSweepStep3.

4.1.2 Case study: Offset Along a Normal

Algorithm 2: The host instructions of offset along normal operation

Data: Sample set S
Offset O
Result: Values V for the samples in S
$V \leftarrow$ Evaluate(OffsetAlongNormal.child(S));
Allocate memory for set of normals N;
`// Compute a normal per a sample in` S
$N \leftarrow$ GetNormalsKernel(V);
Free memory of V;
Allocate memory for set of samples $S2$;
$S2 \leftarrow$ OffsetAlongNormalKernel(N, 0);
Allocate memory for set of values V;
$V \leftarrow$ Evaluate(OffsetAlongNormal.child($S2$));
Free memory of $S2$;
return V;

Offset along a normal is an operation classified according to its tree traversal method, but it can also be classified as a geometric space mapping. In practice, the operation calculates a normal vector at a sample position, then the vector is used to offset the original sample point along the normal vector. See Algorithm 2 for the host side instructions.

4.1.3 Case study: convolution surface using lines

Algorithm 3: The host instructions for Convolution Surface Using Lines

Data: Sample set S

Set of lines L defined by the user

A set of coefficients C_h, one per line in L_h

Result: Values V for the samples in S

Allocate memory for C_d and L_d;

`// Transfer memory from host to device`

Transfer C_h to C_d, and L_h to L_d;

$V \leftarrow$ ConvolutionLinesKernel(S, L_d, C_d);

return V;

A convolution surface is classified according to its requirements for external data. When using lines as skeletal elements, the required data consists start and end points for the lines and coefficients to control a kernel width (McCormack and Sherstyuk 1998). The kernel function and the coefficients associated with a line describe the final shape.

4.1.4 Case study: HyperFun fractal

Algorithm 4: The kernel instructions for one type of HyperFun fractal

Data: Sample set S

Coefficients $c1, c2, c3, c4$

bailout b

number of iterations n

Result: Values V for the samples in S

$lvalue \leftarrow 0.0$;

$bailoutsq \leftarrow b^2$;

for $i \leftarrow 0$ **to** $SizeofS$ **do**

  Fetch $sample_i$ from sample set S;

  for $k \leftarrow 0$ **to** n **do**

    if *!bSkip* **then**

      $length \leftarrow sample_i.x^2 + sample_i.y^2 + sample_i.z^2 + lvalue^2$;

    if $length > bailoutsq$ **then**

      $bSkip \leftarrow True$;

      $k \leftarrow n$;

  $f_i = \sqrt{length} - bailout$;

  Append f_i to V;

The HyperFun fractal is classified according to its defining function. The implementation contains constructs from imperative programming such as if and for loops. An *if* statement is generally problematic when using GPUs (Zhang *et al.* 2010), as SIMD-instructions are implemented as a group of threads, if a single or more threads within a group diverge in their execution because of a conditional statement, all the threads in that group execute the conditional instructions. In order to achieve maximum performance gains, conditional statements should be generally avoided.

4.2 Complex tree structure generation

A tree structure to compare a complex tree structure evaluation is constructed from a set of predefined primitives and operations. The trees are built using genetic programming as a basis (Koza 1992). In the context of this thesis, the initial creation of a random population is sufficient. From a terminal set and functions the genetic algorithm creates random population.

For purpose of FRep, the terminal set contains geometric primitives and set of functions contains geometric operations. The created population can be seen as a FRep tree structure that expresses complex geometric shape. However, in the context of this work the constructed trees do not necessarily describe sensible geometric shapes. They are suitable for simulating structures that could be seen in a modelling application, and therefore are sufficient for performance measurement.

The constructed test trees are built from primitives: sphere, plane, cone, cylinder and torus; and operations: union, intersection, subtraction and negation. The results in Section 5.1 are measured with trees construced in this manner.

4.3 Conclusion

In this chapter the test cases are described. The cases for individual node evaluations are based on the existing FRep classifications. Overall, two classifications exists for the primitives and three for the operations. The primitives are classified according to the defining function and requirements for external data. The operations are classified according to: the geometric transformation and tree traversal method, the complexity point of view in terms of the function evaluation for subtrees, and from the point of view of external data requirements. Each classification contains several classes of functions which are shown in Table 4.1.

For complex tree traversal random trees are constructed from pre-

defined set of primitives and operations. With randomness it is possible to quickly build trees that avoid some caching and optimization behaviour that might happen within the devices if the test cases are predictable.

Chapter 5

Results

This chapter presents results for tree traversal and individual node evaluation. The implementation of the test cases is described in Chapter 4. The tests are conducted on a hardware listed in Table 5.1.

CPU	
Device:	Intel(R) Xeon(R) CPU E5-1650 0 @ 3.20GHz
Vendor:	Intel(R) Corporation
CL Profile:	FULL_PROFILE
OpenCL device version:	OpenCL 1.2 (Build 82248)
OpenCL C version:	OpenCL C 1.2
Overall memory:	33587380224 B ≈ 32 GB
Max block allocation size:	8396845056 B ≈ 8 GB
Compute Units:	12
GPU	
Device:	Quadro K2000
Vendor:	NVIDIA Corporation
Device OpenCL Profile:	FULL_PROFILE
OpenCL device version:	OpenCL 1.1 CUDA
OpenCL C version:	OpenCL C 1.1
Overall memory:	2146762752 B ≈ 2 GB
Max block allocation size :	536690688 B ≈ 500 MB
Compute Units(=Streaming Multiprocessor):	2
Streaming Processors per Compute Unit:	192

Table 5.1: *The GPU and CPU device info.*

5.1 Tree traversal results

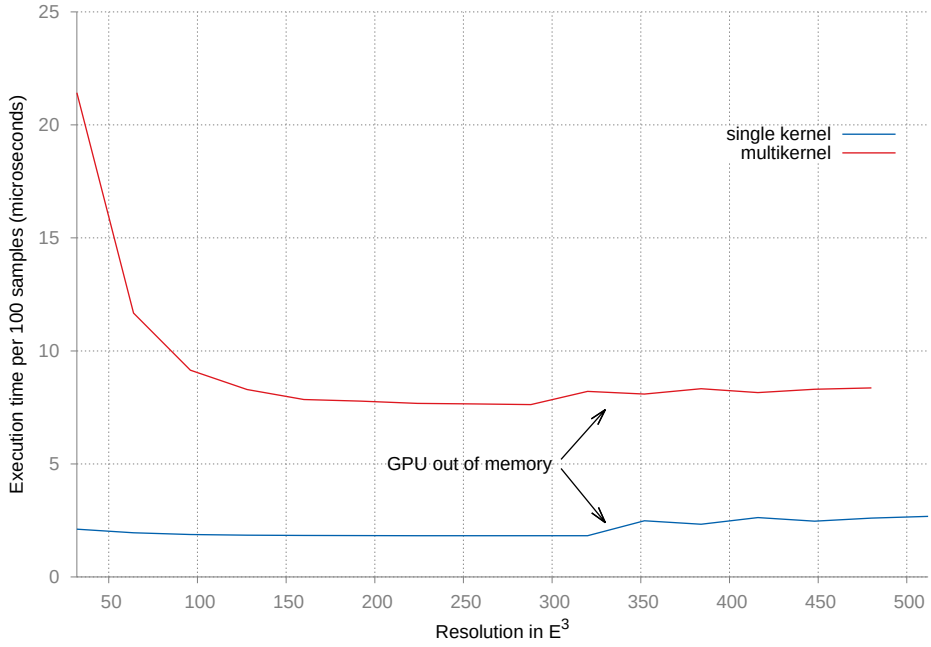


Figure 5.1: *Execution of a random tree with 112 nodes on a GPU; When a GPU can not fit all the samples into memory, the average execution time rises.*

In Figure 5.1, a comparison between multi- and single kernel methods shows the initialization cost of small kernels. An initialization of small kernels without sufficient amount of data diminishes overall execution speed because of the overhead caused by the command initiations. The overall performance plateaus when enough samples are used.

Both single- and multikernel executions suffer minor speed reductions when sample set is larger than the maximum memory allocation size for GPU. When limit is reached the sample set is separated into two memory blocks instead of one. In batch-execution mode, the tree evaluation is initiated with two separate batches of samples one after each other.

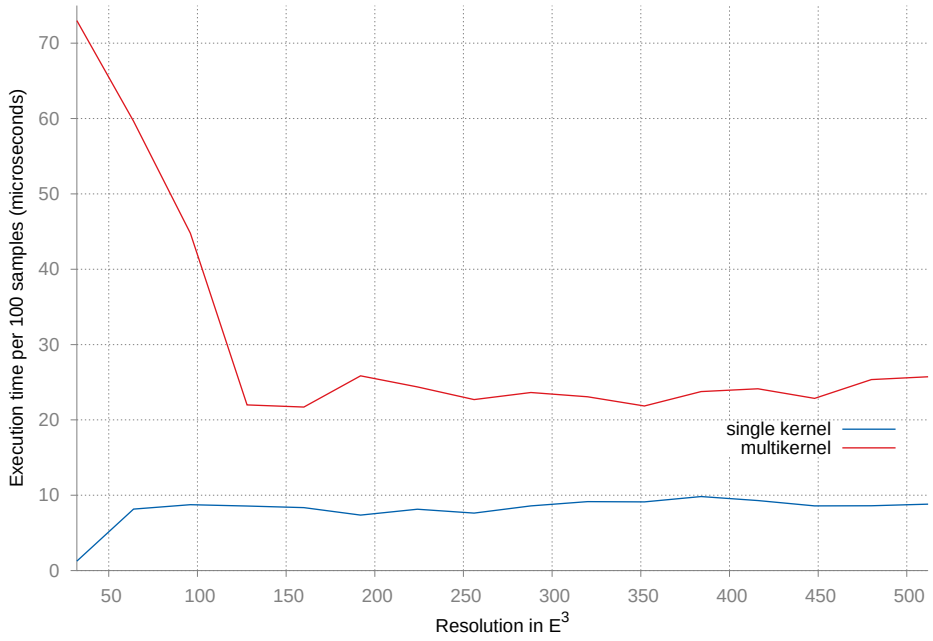


Figure 5.2: *Execution of a random tree with 112 nodes on a multiprocessor system.*

In multiprocessor system, each processor is capable of using vector-instructions, which improves the performance in form of reducing scalar instructions. However, as Figures 5.2 and 5.1 show both multiprocessor and GPU execution suffer from the same overhead caused by initiating small kernels with small sample sets. One factor contributing to the overhead on a multiprocessor system is thread migration which happens when a idle CPU starts executing thread, such thread also suffers from a small overhead when the informs about a completed thread. The cache of the idle CPU is empty, therefore few cycles are spent to fill the cache. With small sample sets and small kernels, this overhead is significant enough to drastically reduce the overall performance (Gummaraju *et al.* 2010).

In a single kernel method executing the thread of instructions that describes the whole tree, gives better cache prediction behaviour than small fragmented threads of a multikernel method. Therefore, when initiating the kernel with a sufficiently small sample set, the work can be executed on available processors with smaller number of thread migra-

tions and smaller number of thread-ready notifications. With larger sets, the problem is not that apparent.

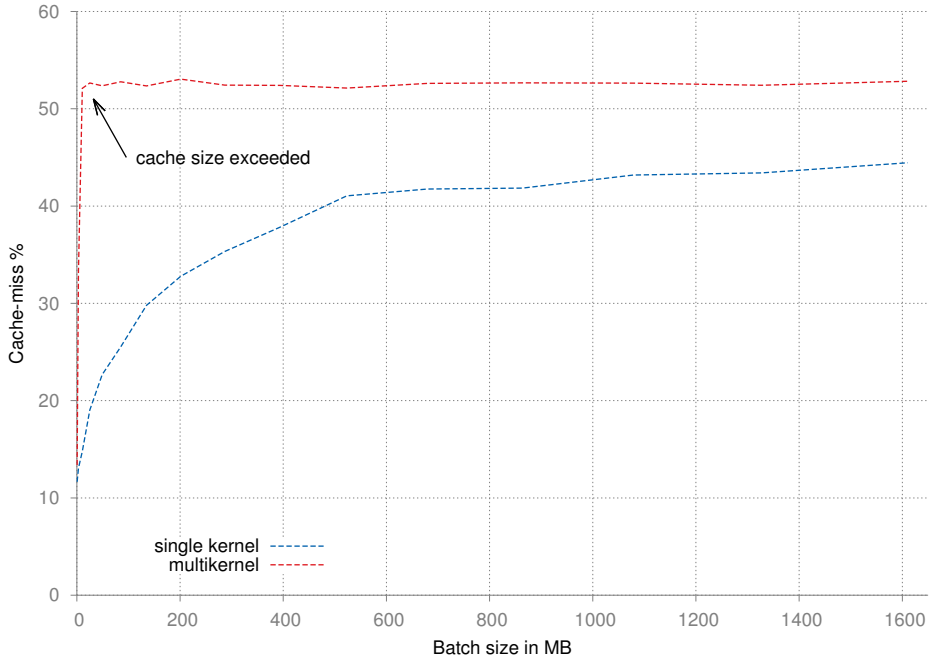


Figure 5.3: *Execution of a random tree with 112 nodes on multiprocessor system ; High number of memory queries results in high cache-miss ratio.*

The Figure 5.3 shows a comparison between cache-miss rate between a single kernel and multikernel methods. The data is collected using *perf-tools*. A curve for a multikernel method clearly shows when a cache memory is depleted, this is partly because of fragmented execution of small functions the cache prediction becomes difficult and the cache-miss rate plateaus.

In the single kernel method, the cache miss rate rises linearly while the amount of samples increases exponentially. The cache-miss rate can be partly explained by the required number of thread migrations to another CPU. When a migration occurs the local cache of the target CPU doesn't necessarily contain relevant data, so cache-misses happen (Tanenbaum 2007).

The overall cache prediction is significantly better in single kernel execution than in multikernel one. The cache prediction is conducted

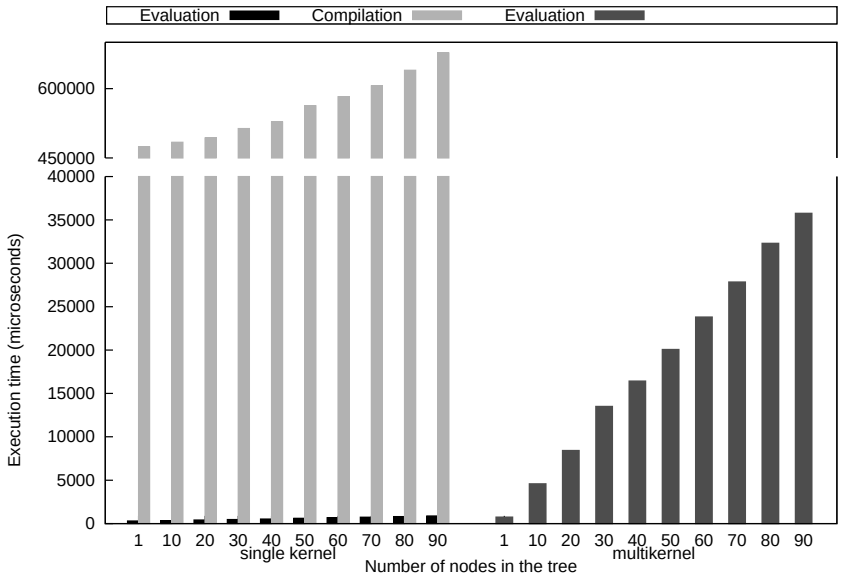


Figure 5.4: *A compilation step in comparison to the evaluation in a single kernel method, is a major cause for execution overhead.*

by the hardware and with small fragmented function executions, the prediction is difficult. Therefore, a single long thread of instructions is better for cache-prediction rather than small fragmented threads of instructions.

Whenever a structural modification to a function tree is made, the tree requires re-evaluation, and in case of single kernel method, re-compilation before. The Figure 5.4 shows the time spent on evaluation of a tree and time spent on compiling the tree.

With a complex tree, the time spent in compilation is around half a second. When conducting an iterative trial-and-error process, the compilation time can cause severe distractions to the process. However, only when a structural change to a tree is made a re-compilation is required. When a change of an argument value on a function made, only re-evaluation is required.

	A		B		C	
L1	64x64x64 S: 0.0011s M: 0.0001s G: 0.0004s	128x128x128 S: 0.0061s M: 0.0069s G: 0.0012s	64x64x64 S: 0.0071s M: 0.0020s G: 0.0007s	128x128x128 S: 0.0548s M: 0.0091s G: 0.0031s	64x64x64 S: 0.0013s M: 0.0019s G: 0.0005s	128x128x128 S: 0.0782s M: 0.0120s G: 0.0016s
	256x256x256 S: 0.0449s M: 0.0899s G: 0.0067s	512x512x512 S: 0.3615s M: 0.7372s G: 0.0520s	256x256x256 S: 0.4346s M: 0.1178s G: 0.0225s	512x512x512 S: 3.4719s M: 0.9462s G: 0.0822s	256x256x256 S: 0.0591s M: 0.0912s G: 0.0102s	512x512x512 S: 0.4689s M: 0.8397s G: 0.1274s
L2	64x64x64 S: 0.0010s M: 0.0025s G: 0.0004s	128x128x128 S: 0.0062s M: 0.0139s G: 0.0016s	64x64x64 S: 0.0154s M: 0.0042s G: 0.0011s	128x128x128 S: 0.1142s M: 0.0296s G: 0.0052s	64x64x64 S: 33s M: 3.3931s G: 1.9643s	128x128x128 S: 266s M: 30s G: 13.7s
	256x256x256 S: 0.0460s M: 0.1016s G: 0.0066s	512x512x512 S: 0.3656s M: 0.8180s G: 0.0522s	256x256x256 S: 0.9108s M: 0.1551s G: 0.0374s	512x512x512 S: 7.2840s M: 1.4504s G: 0.2989s	256x256x256 S: 36m M: 6m G: na	512x512x512 S: 5h M: 1h G: na

S = Sequential Execution, M = Multicore Execution, G = GPU execution, ■ = Out of Memory

Figure 5.5: *The execution speed calculations made on existing FRep function classifications for geometric objects.*

5.2 Individual node results

These results presented in this section are done using the multikernel method. Most of the test cases are simple tree structures containing only an individual node or few nodes. For example the operations are applied on simple primitives.

Chapter 4 defines all the test cases used for collecting results. The Figures 5.5, 5.6 show the results for each category. Unless otherwise stated, the results in the Figures 5.5 and 5.6 are implemented using OpenCL Framework.

To simulate several execution environments the same OpenCL C code is compiled with different compiler back-ends to two targets: x86 instructions for a CPU and PTX instructions for NVidia GPU. The compilation stage into x86 instructions also include AVX Three different execution targets for instructions are: an uniprocessor, a multiprocessor system, and a GPU. An uniprocessor is simulated by restricting the openCL runtime to execute instructions using only one thread. Most modern processors have vector extensions, therefore vector-instructions are en-

	A		B		C	
N1	64x64x64 S: 0.0011s M: 0.0001s G: 0.0017s	128x128x128 S: 0.0061s M: 0.0069s G: 0.0117s	64x64x64 S: 0.0071s M: 0.0020s G: 0.0020	128x128x128 S: 0.0548s M: 0.0091s G: 0.0130s	64x64x64 S: 0.0489s M: 0.0025s G: 0.0034s	128x128x128 S: 0.3888s M: 0.0283s G: 0.0304
	256x256x256 S: 0.0449s M: 0.0899s G: 0.0885s	512x512x512 S: 0.3615s M: 0.7372s G: 0.7040s	256x256x256 S: 0.4346s M: 0.1178s G: 0.0994s	512x512x512 S: 3.4719s M: 0.9462s G: 0.7927s	256x256x256 S: 3.3752s M: 0.4716s G: 0.2524s	512x512x512 S: 26.388s M: 3.3229s G: na
N2	64x64x64 S: 0.2052s M: 0.0107s G: 0.0018s	128x128x128 S: 1.6134s M: 0.1798s G: 0.0109s	64x64x64 S: 58.591s M: 5.588s G: 0.763612	128x128x128 S: 8m M: 52.76s G: na	64x64x64 S: 0.4245s M: 0.0197s G: 0.0053s	128x128x128 S: 1.6981s M: 0.1927s G: 0.0213s
	256x256x256 S: 12.889s M: 1.4489s G: 0.0847s	512x512x512 S: 103.329s M: 11.544s G: 0.6751s	256x256x256 S: na M: na G: na	512x512x512 S: na M: na G: na	256x256x256 S: 6.7872s M: 0.6487s G: 0.0824s	512x512x512 S: 27.179s M: 3.6000s G: na
N3	64x64x64 S: 0.2070s M: 0.0249s G: 0.0017s	128x128x128 S: 1.3094s M: 0.1888s G: 0.0115s	64x64x64 S: 0.0293s M: 0.0052s G: 0.0017s	128x128x128 S: 0.2214s M: 0.0277s G: 0.0115s	64x64x64 S: 0.3369s M: 0.1074s G: 0.0762s	128x128x128 S: 2.2074s M: 0.3129s G: 0.1298s
	256x256x256 S: 13.095s M: 1.3564s G: 0.0881s	512x512x512 S: 104.87s M: 11.479s G: 0.7040s	256x256x256 S: 1.7119s M: 0.3754s G: 0.0882s	512x512x512 S: 13.665s M: 2.9594s G: 0.5875s	256x256x256 S: 15.811s M: 1.9105s G: 0.5698s	512x512x512 S: 125.69s M: 14.622s G: na

S = Sequential Execution, M = Multicore Execution, G = GPU execution, = Out of Memory

Figure 5.6: *The execution speed calculations made on existing FRep function classifications for geometric operations.*

abled for both: an uniprocessor and a multiprocessor system.

In the following subsections, a hand picked selection is made according to the execution results for further analysis. A comprehensive conclusion is given in the end of the chapter.

5.2.1 Case study: Inside-outside segmentation of polygonal mesh

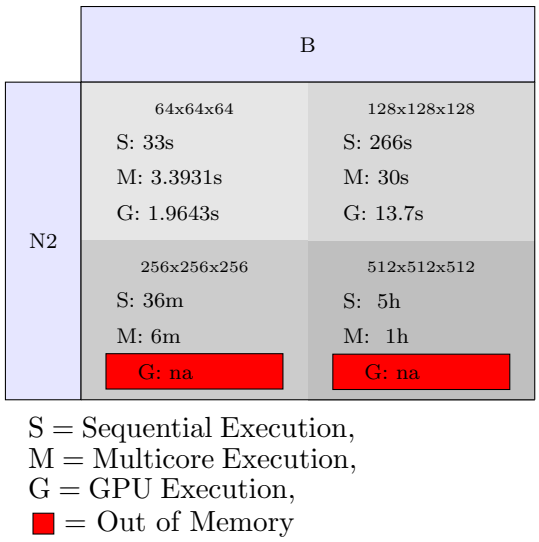


Figure 5.7: *Evaluation times of Mesh on the left-hand side graph.*

Operation on a polygonal mesh is a computationally expensive operation, which leads to better performance on a GPU than a CPU. However, as a result of memory constraints the GPU evaluation is not an optimal solution for a complex mesh.

5.2.2 Case study: Offset along a normal

	C	
N1	64x64x64 S: 0.0489s M: 0.0025s G: 0.0034s	128x128x128 S: 0.3888s M: 0.0283s G: 0.0304
	256x256x256 S: 3.3752s M: 0.4716s G: 0.2524s	512x512x512 S: 26.388s M: 3.3229s G: na

S = Sequential Execution,
M = Multicore Execution,
G = GPU Execution,
■ = Out of Memory

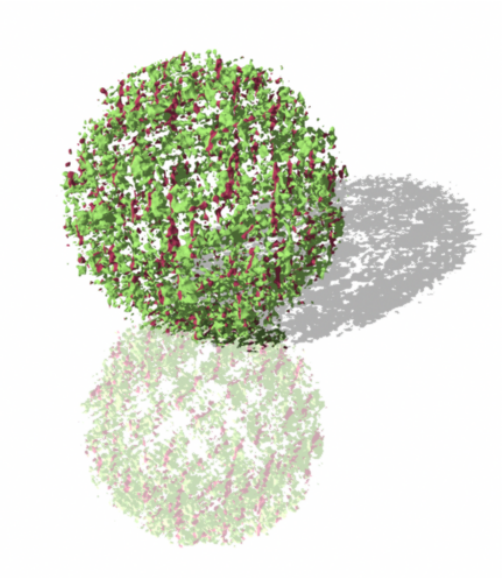
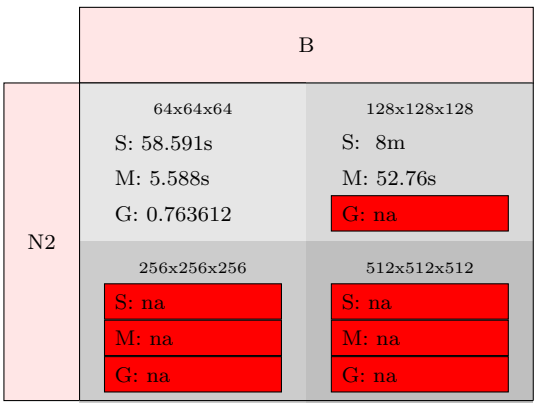


Figure 5.8: *Evaluation times of OffsetAlongNormalNode on the left-hand side graph. A rendered image of operation applied to union of two spheres and a cube on right-hand side.*

An offset along a normal operation requires evaluation of its subtree before the operation can take effect. The subtree evaluation calculates a gradient per a sample point. The gradient is used to offset the points of the original evaluation space.

Concurrently the memory has to store the original samples, the normals and the new point coordinates. After the new coordinates have been calculated, the memory holding the normals is released.

5.2.3 Case study: Sweeping by a moving solid



S = Sequential Execution,
M = Multicore Execution,
G = GPU Execution,
■ = Out of Memory

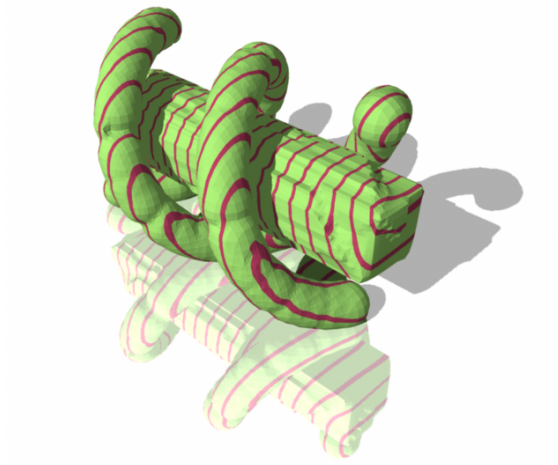


Figure 5.9: *Evaluation times of SolidSweepNode on the left-hand side graph. A rendered image of solid sweep operation applied to union of two spheres and a cube on right-hand side.*

A sweep operation multiplies the evaluation space with the number of subdivisions on the trajectory. Greater number of subdivisions, generally leads to better quality of the sweep. The operation can also be seen as a projection from $E^4 \rightarrow E^3$ and therefore as an union of cross-sections of the operand object in E^4. In practice a single hyperplane in E^3 is duplication of the original evaluation space, with a trajectory defined offset. In a batch-mode evaluation, even with a small batch-size the operation quickly out memory as the coordinate-batch given as parameter is to be multiplied with the number of subdivisions.

As an example, if the operation is given 64x64x64 sample coordinates and 128 subdivisions as parameter, the required memory is:

$$RequiredMemory = BatchSize * sizeof(Coordinate) * subdivisions$$
$$RequiredMemory = 64 * 64 * 64 * 3 * 4 * 128$$
$$RequiredMemory = 402653184\,bytes \approx 400\,MB$$

If evaluation is done with per-coordinate execution rather than per-batch

execution the memory requirements are far less; The memory is allocated on coordinate basis rather than batch basis. See Figure 5.9 for the execution speed comparison.

5.2.4 Case study: Projection

	C	
	64x64x64	128x128x128
N2	S: 0.4245s M: 0.0197s G: 0.0053s	S: 1.6981s M: 0.1927s G: 0.0213s
	256x256x256	512x512x512
	S: 6.7872s M: 0.6487s G: 0.0824s	S: 27.179s M: 3.6000s G: na

S = Sequential Execution,
M = Multicore Execution,
G = GPU Execution,
■ = Out of Memory

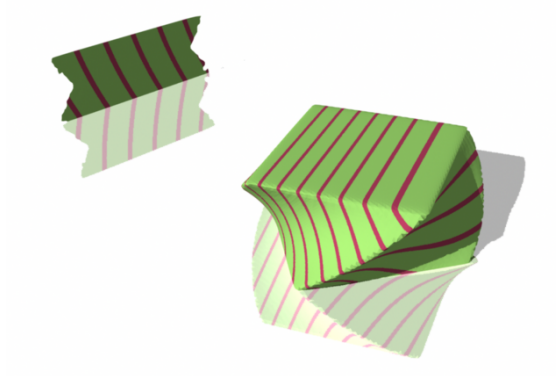


Figure 5.10: *Evaluation times of Projection on the left-hand side graph. A rendered image of projection operation applied to twisted cube on right-hand side. The faraway plane is the result of the projection.*

A projection operation from $E^3 \rightarrow E^2$ is an operation that requires extensive amount of memory. As with solid sweep in section5.2.3, the projection is a union of cross sections of an object in E^3. The hyperplanes therefore are in E^2 space. From memory requirement point of view the E^2 space is duplicated with the desired number of subdivisions. This directly correlates with the amount of memory required.

Along with the memory requirement caveat, a projection from $E^3 \rightarrow E^2$ results to a two dimensional entity that is used in three-dimensional framework. Therefore, for a projection to be used with rest of the operations, the E^2 space of the projection has to be padded with empty data for the projection to be used with operations working in E^3. This results in redundant computation when computing the empty cells.

5.2.5 Case Study: Inverse free-form deformation

	C	
N3	64x64x64 S: 0.3369s M: 0.1074s G: 0.0762s	128x128x128 S: 2.2074s M: 0.3129s G: 0.1298s
	256x256x256 S: 15.811s M: 1.9105s G: 0.5698s	512x512x512 S: 125.69s M: 14.622s G: na

S = Sequential Execution,
M = Multicore Execution,
G = GPU Execution,
■ = Out of Memory

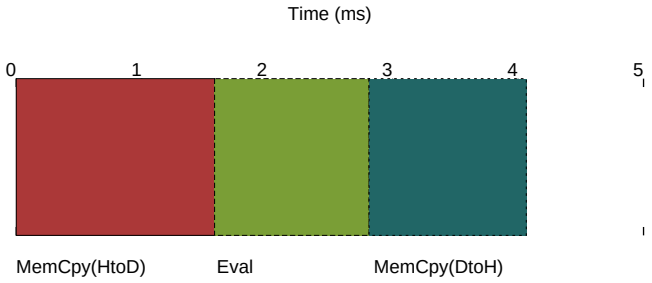
Figure 5.11: *Evaluation times of InverseFFD on the left hand side graph. A rendered image of InverseFFD operation applied on torus on right hand side.*

Inverse free-form deformation is defined by an offset between a reference lattice and a deformation lattice. A geometric object is embedded to the reference lattice. After the object is embedded, a modification of the deformation lattice modifies it. The result of the operation is a new evaluation space. The memory requirements of the function are directly related to the density of the defining and deformation lattice (Comninos *et al.* 2014).

5.2.6 Memory

Memory transfer between the host and the device is slow. The Figure 5.12 shows the time spent executing the tasks: *clEnqueueMapBuffer* and *clEnqueueNDRangeKernel*, which initiates the kernel execution. A CPU in a multiprocessor system is able to access the main memory without an explicit memory transfer, therefore *clEnqueueMapBuffer* does not move memory, and allows usage of the memory block where data was originally initialized. However, in case of a GPU the memory has to be transferred

GPU:



CPU:

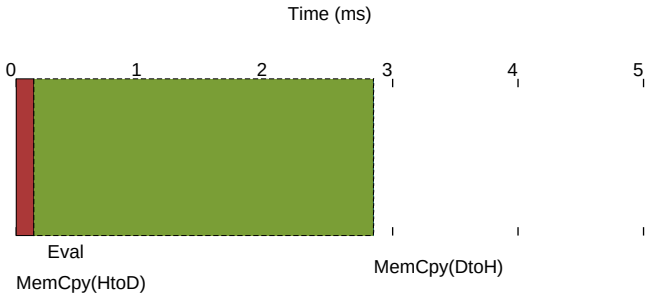


Figure 5.12: *A division of Hyperfun fractal evaluation into a memory transfer from host to device, evaluation of the code, and transfer of the results back to host. The timings are done with OpenCL timing events.*

from the main memory to the device memory. The memory transfer for the sample points is only done at initialization, so subsequent node evaluations using the same points can use the same data. Therefore from perspective of a user, a complex tree has better ratio between memory transfer and computation.

5.3 Conclusion

As seen in Figures 5.5 and 5.6 certain functions are memory intensive, making some functions problematic to be executed with big sample sets. An easy solution is to execute the function with smaller sample sets and compose results as a post-process, however as seen in Figure 5.1 when using GPUs, the overall performance slightly decreases when an execution is split into multiple sets.

A closer analysis of 5.6 reveal that operations with severe memory

issues are following operations:

- N2B: sweeping by a moving solid: $R^4 \rightarrow R^3$,
- N2C: projection: $R^3 \rightarrow R^2$,
- N1C: Offset along a normal,
- N3C: Inverse free-form deformation.

These operations are classified as geometric space mappings or their implementation contain duplication of the original sample set. All geometric space mapping operations duplicate the original sample set so the duplicate can be modified to express the effect of the operation. In such case, the memory contains two big sets of samples in memory subsequently, the initial set and the modified set. If a tree contains multiple elements that require such duplication, it can cause failure in the evaluation because of insufficient memory. This can be mitigated by in the traversal by always visiting the longer branch first, finally resulting in a case where all branches of the visited node have the length of one. This traversal ensures only a single operation and its operands are currently accessing the memory.

In the complex tree traversal results, *multikernel* is compared against *single kernel*. From the results it is evident that in terms of pure evaluation speed the single kernel method provides better performance. However, the overhead of the compilation step of the single kernel method can make it insufficient for some work where a complex tree is constantly under mutation.

Overall from the results it is clear that the GPU gives the best performance in almost all of the cases. In the tree traversal and in individual node evaluations, the GPU the performance gained by using the GPU is significant. However, optimisations in a OpenCL kernel for specific type of microarchitecture do not necessarily translate well for other microarchitectures (Gummaraju *et al.* 2010). In order to get maximum performance improvements from a multiprocessor system, slightly better results are likely be achieved by optimizing directly for that hardware.

Chapter 6

Conclusion and future work

In the previous academic work efficient FRep function evaluation has been researched from perspective of special cases where the focus is only on a small subset of shapes and operations. In this thesis, an extensive survey on efficient function evaluation is done by using wide range FRep functions and operations. The test cases cover evaluation of a complex FRep tree structures containing a large number nodes and evaluations of individual FRep functions that cover the existing FRep function classifications.

One of the problems in achieving efficient function evaluation especially in case of a GPU is how to transform a tree structure into an executable program. A GPU requires an explicit compilation step for all instructions given to it. This is evident in the previous methods where the tree structure is compiled into executable program during applications runtime.

In an established solution for the transformation, a tree structure is first transformed into a textual representation of the tree, secondly the text is compiled into a device executable program. This procedure happens every time there is a structural change to the tree. Along with a compilation step, an actual evaluation step is conducted which yields the final results. In this thesis, this approach is called a single kernel method.

In the single kernel method, the compilation step can lead to interruptions in the overall program execution because of the time spent compiling the kernel. Another caveat in the single kernel method, is that the memory required by the tree has to be initialized before the initiating the evaluation. This is problematic in cases when the tree contains multiple complex nodes that have large memory requirements.

To overcome some of the problems present in the single kernel method, this thesis proposes an alternative method called a multikernel method. A tree structure is composed from elements that represent operations and primitives. Each element contains instructions on how to deploy a program representing the node to a parallel device. The programs are pre-compiled during either offline or application start-up. During the evaluation the structure is interpreted and each element computes its associated program on device.

In multikernel method the tree structure can contain nodes that have implementations for mixed evaluation devices. For example some nodes can be evaluated on a multiprocessor system and some on a GPU. This allows for *heterogeneous* evaluation, where parts of the tree can be evaluated on a multiprocessor system and others parts on a GPU. As an example a node that requires large amounts of memory that is not evaluable on a GPU can be evaluated on a CPU.

Some important features in the multikernel method:

- Memory management can be done during tree traversal. This is important tree structures that contain a number of complex nodes with large memory requirements.
- Elements from which the tree is built are are compiled at the applications startup or offline.
- The evaluation procedure is device independent and allows implementation of nodes with complex evaluation profiles. For example sweeping by a moving solid has several steps in the evaluation.

The results in Chapter 5 provide some general guidelines. In case of a static tree-structure constructed from functions with a small memory

requirements the single kernel methods provides the best performance. However, when an object goes through iterative mutations, the compilation stage required in single kernel method can cause interruptions.

In case of a complex tree structure with recurring mutations, the multikernel method is most likely to provide the best overall performance. Such cases can be for example, interactive modelling and simulations. Furthermore, from a perspective of an application design, the multikernel method makes a plug-in system easy to implement. Through plug-ins, a user is able extend the system by writing her own functions, from which the tree will eventually be constructed. In the single kernel method, extending the system would require the user to have knowledge about the parallel framework the application uses, and have access to the kernel containing the function definitions.

The results in Chapter 5 show that by using the parallel features of a GPU provides significant performance improvements. However, GPUs memory is limited and the operations to move memory between the host and the device are slow this is a problem when evaluating memory intensive FRep functions. The simple algebraic and procedural functions are a good fit for a GPU but complex operations require a lot of memory. With a small sample sets they are able to provide good performance improvements, but a dense sample set can cause the GPU to run out of memory. In such cases the evaluation can be split into smaller sets, which will sligthly decrease the performance and more importantly add to the complexity of application.

A multiprocessor system also provides parallel capabilities that can be used to improve the performance of the function evaluation. The memory model in such a system is more flexible and generally has more memory capacity than a GPU. Thus, some nodes such as large meshes could be a better fit for CPU evaluation than for graphics programming units.

6.1 Future work

Some micro-optimizations to further improve the performance gained from efficient use of hardware could use the future models of GPU development. The current GPU development model applies some restrictions on the computation. The main restrictions are the memory management model. A host program manages the memory of the device and deploys program executions on it. In the future models of GPU evaluation, the restrictions of the execution model are partially solved through Dynamic Parallelism, which allows a kernel to create and synchronize nested work. This is done without involvement of a CPU, which in case of a multikernel method could provide some performance improvements by eliminating need for a host to initiate and manage all the nodes in the tree structure. Dynamic Parallelism also adds support for recursive functions, hence a kernel may call itself NVIDIA (2012).

The multikernel method can be further developed to perform some optimizations automatically. For example, if a structure contains a collection of nodes that could be presented as a single shader, it should be done as it provides better performance. Also, using the empirical results presented in this paper, a multikernel method could determine the best configuration for sample set splitting automatically by analyzing the tree structure and comparing it against the memory limits.

An interesting direction for future efforts would be to further parallelize the tree evaluation by using configuration of multiple GPUs mixed with multicore CPUs. In a multi-device setting parallelization of subtrees is possible. For example, in case of a binary operation, one branch could be deployed to a GPU and the second to another GPU. The memory transfer between the devices is likely to be a problem.

Bibliography

Hyperfun fractal. `http://hyperfun.org/wiki/doku.php?id=gallery:fractals`. Accessed: 2014-06-19.

Adzhiev V., Cartwright R., Fausett E., Ossipov A., Pasko A. and Savchenko V., 1999. Hyperfun project: a framework for collaborative multidimensional f-rep modeling.

Alexe A., Gaildrat V. and Barthe L., 2004. Interactive modelling from sketches using spherical implicit functions. In *Proceedings of the 3rd International Conference on Computer Graphics, Virtual Reality, Visualisation and Interaction in Africa*, AFRIGRAPH '04, New York, NY, USA. ACM, 25–34.

Barr A. H., 1984. Global and local deformations of solid primitives. In *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '84, New York, NY, USA. ACM, 21–30.

Bourke P. The blob-shape. `http://paulbourke.net/geometry/blob/`. Accessed: 2014-06-16.

Comninos P., Fryazinov O. and Pasko A., 2014. Free-form deformations for function representation.

Dyken C., Ziegler G., Theobalt C. and Seidel H.-P., 2008. High-speed marching cubes using histopyramids. *Computer Graphics Forum*, **27** (8), 2028–2039.

Flynn M. J., September 1972. Some computer organizations and their effectiveness. *IEEE Trans. Comput.*, **21**(9), 948–960.

Fryazinov O. and Pasko A., 2008. Interactive ray shading of frep objects. In Cunningham S. and Skala V., editors, *WSCG' 2008, Communications Papers proceedings*, Plzen, Czech Republic. University of West Bohemia, 145–152.

Gummaraju J., Morichetti L., Houston M., Sander B., Gaster B. R. and Zheng B., 2010. Twin peaks: A software platform for heterogeneous computing on general-purpose and graphics processors. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, New York, NY, USA. ACM, 205–216.

Hart J. C., 1993. Ray tracing implicit surfaces. *WSU Technical Report EECS-93-014*.

Hart J. C., 1997. Implicit representations of rough surfaces. *Computer Graphics Forum*, **16**(2), 91–99.

Hennessy J. L. and Patterson D. A., 2011. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition.

Hua J. and Qin H., 2003. Free-form deformations via sketching and manipulating scalar fields. In *Proceedings of the Eighth ACM Symposium on Solid Modeling and Applications*, SM '03, New York, NY, USA. ACM, 328–333.

Hughes J. F., 1992. Scheduled fourier volume morphing. In *Proceedings of the 19th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '92, New York, NY, USA. ACM, 43–46.

Jacobson A., Kavan L., and Sorkine-Hornung O., 2013. Robust inside-outside segmentation using generalized winding numbers. *ACM Transactions on Graphics (proceedings of ACM SIGGRAPH)*, **32**(4), 33:1–33:12.

Koza J. R., 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA.

Kravtsov D. *Hybrid modelling of time-variant heterogeneous objects*. PhD thesis, Bournemouth University, 2011.

Lorensen W. E. and Cline H. E., 1987. Marching cubes: A high resolution 3d surface construction algorithm. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '87, New York, NY, USA. ACM, 163–169.

Luebke D., Harris M., Krüger J., Purcell T., Govindaraju N., Buck I., Woolley C. and Lefohn A., 2004. Gpgpu: General purpose computation on graphics hardware. In *ACM SIGGRAPH 2004 Course Notes*, SIGGRAPH '04, New York, NY, USA. ACM.

McCormack J. and Sherstyuk A., 1998. Creating and rendering convolution surfaces. *Computer Graphics Forum*, **17**(2), 113–120.

Nvidia . Fermi Compute Architecture Whitepaper. Technical report.

NVIDIA , 2012. Kepler GK110 whitepaper.

Pasko A. and Savchenko V. 1997. 197–205. Projection operation for multidimensional geometric modeling with real functions. In Strasser W., Klein R. and Rau R., editors, *Geometric Modeling: Theory and Practice*, Focus on Computer Graphics, Springer Berlin Heidelberg.

Pasko A. and Adzhiev V. 2004. 132–160. Function-based shape modeling: Mathematical framework and specialized language. In Winkler F., editor, *Automated Deduction in Geometry*, volume 2930 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg.

Pasko A. A. P. V. N., Pilyugin V. V., 1995. Geometric modeling in the analysis of trivariate functions. *Computers and Graphics*, **12**(3/4), 457–465.

Perlin K. and Hoffert E. M., 1989. Hypertexture. In *Proceedings of the 16th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '89, New York, NY, USA. ACM, 253–262.

Phong B. T., June 1975. Illumination for computer generated pictures. *Commun. ACM*, **18**(6), 311–317.

Reiner T., Lefebvre S., Diener L., GarcíA I., Jobard B. and Dachsbacher C., August 2012. Smi 2012: Full a runtime cache for interactive procedural modeling. *Comput. Graph.*, **36**(5), 366–375.

Reiner T., Mückl G. and Dachsbacher C., June 2011. Interactive modeling of implicit surfaces using a direct visualization approach with signed distance functions. *Comput. Graph.*, **35**, 596–603.

Sanchez M., Fryazinov O. and Pasko A., 2012. Efficient evaluation of continuous signed distance to a polygonal mesh. In *Proceedings of the 28th Spring Conference on Computer Graphics*, SCCG '12, New York, NY, USA. ACM, 101–108.

Savchenko V. and Pasko A., 1998. Transformation of functionally defined shapes by extended space mappings. *The Visual Computer*, **14**(5-6), 257–270.

Schmidt R., Wyvill B., Sousa M. C. and Jorge J. A., 2005a. Shapeshop: Sketch-based solid modeling with blobtrees. In *2nd Eurographics Workshop on Sketch-Based Interfaces and Modeling*, 53–62.

Schmidt R. and Singh K., 2010. Meshmixer: An interface for rapid mesh composition. In *ACM SIGGRAPH 2010 Talks*, SIGGRAPH '10, New York, NY, USA. ACM, 6:1–6:1.

Schmidt R., Wyvill B. and Galin E., 2005b. Interactive implicit modeling with hierarchical spatial caching. In *SMI*, 104–113.

Shapiro V., 5 2007. Semi-analytic geometry with r-functions. *Acta Numerica*, **16**, 239–303.

Sherstyuk A., Mar 1999. Interactive shape design with convolution surfaces. In *Shape Modeling and Applications, 1999. Proceedings. Shape Modeling International '99. International Conference on*, 56–65, 270.

Shirazian P., Wyvill B. and Duprat J.-L. 2012. 89–98. Polygonization of implicit surfaces on multi-core architectures with simd instructions. In Childs H., Kuhlen T. and Marton F., editors, *Eurographics Symposium on Parallel Graphics and Visualization, EGPGV 2012, Cagliari, Italy, May 13-14, 2012: Proceedings*, Eurographics Symposium on Parallel

Graphics and Visualization, Eurographics: European Association for Computer Graphics.

Sourin A. I. and Pasko A. A., March 1996. Function representation for sweeping by a moving solid. *IEEE Transactions on Visualization and Computer Graphics*, **2**(1), 11–18.

Stokes J., 2006. *Inside the Machine: An Illustrated Introduction to Microprocessors and Computer Architecture*. No Starch Press, Inc., San Francisco, CA, USA, 1st edition.

Sugihara M., Wyvill B. and Schmidt R., 2010. WarpCurves: A tool for explicit manipulation of implicit surfaces. *Computers and Graphics*, **34**(3), 282–291. Shape Modeling International (SMI) 2010.

Tanenbaum A. S. *Modern Operating Systems*, Multiple Processor Systems. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2007.

Wyvill B. and van Overveld K., Mar 1997. Warping as a modelling tool for csg/implicit models. In *Shape Modeling and Applications, 1997. Proceedings., 1997 International Conference on*, 205–213, 248.

Zhang E. Z., Jiang Y., Guo Z. and Shen X., 2010. Streamlining gpu applications on the fly: Thread divergence elimination through runtime thread-data remapping. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, New York, NY, USA. ACM, 115–126.

Ziegler G., Tevs A., Theobalt C. and Seidel H.-P., June 2006. Gpu point list generation through histogram pyramids. Research Report MPI-I-2006-4-002, Max-Planck-Institut für Informatik, Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany.

.1 Sweeping by a moving solid

Algorithm 5: Step 1: Calculate samples that lie on trajectory

Data: Sample set S
Number of subdivisions N
Result: A new sample set $S2$
for $i \leftarrow 0$ **to** $SizeofS$ **do**
    for $k \leftarrow 0$ **to** N **do**
        calculate sample $sample_{ik}$ that lies on a trajectory ;
        Append $sample_{ik}$ **to** $S2$;

Algorithm 6: Step 2: Calculate interpolated sample positions

Data: Sample set S
Number of subdivisions N
A set of sampling results V
Result: A new sample set $S3$
for $i \leftarrow 0$ **to** $SizeofS$ **do**
    for $k \leftarrow 0$ **to** N **do**
        Read f_v from V using $i * N + k$ as a key;
        Read f_{v-1} from V using $i * N + k - 1$ as a key;
        Read f_{v+1} from V using $i * N + k + 1$ as a key;
        $a \leftarrow f_v - f_{v-1}$;
        $b \leftarrow (f_{v+1} - f_v) - (f_v - f_{v-1})$;
        $ts \leftarrow 0.5 - a/b$;
        if $ts > 2$ **then**
            $ts \leftarrow 2$;
        if $ts < 0$ **then**
            $ts \leftarrow 0$;
        $delta \leftarrow 1.0/N$;
        $t \leftarrow delta * k + delta * ts$;
        use value t, s_{ik} and the trajectory to process $sample_v$;
        Append $sample_v$ **to** $S3$;

Algorithm 7: Step 3: Find the maximum value from the sampling results of interpolated space coordinates for each original sample

Data: Sampling results $V2$ for sample set $S3$//Number of subdivisions N

Result: Result $V3$ for samples S

for $i \leftarrow 0$ **to** $SizeofS$ **do**

    $k \leftarrow 0$;

    Fetch f_{i0} from $V2$ using $i * N$ as key ;

    for $k \leftarrow 1$ **to** N **do**

        Fetch f_{ik} from $V2$ using $i * N + k$ as key ;

        $f_{vi} \leftarrow Union(f_{ik-1}, f_{ik})$;

    Append f_{vi} to $V3$;