A framework for easy CUDA integration in C++ applications

Jens Breitbart

Research Group Programming Languages / Methodologies
Dept. of Computer Science and Electrical Engineering
Universität Kassel
Kassel, Germany

Supervisor:

Prof. Dr. Claudia Leopold
Prof. Dr. Lutz Wegner

To myself, as a proof...

ABSTRACT

Big improvements in the performance of graphics processing units (GPUs) and enhancements in the corresponding programming systems turned GPUs into a compelling platform for high performance computing. In this thesis, we present CuPP, a C++ framework built up on NVIDIAs CUDA. CuPP allows easier development of GPU programs even as compared to CUDA, by automating frequent programming tasks e.g. memory management.

The thesis is roughly divided into three parts. We begin with an introduction to the CUDA programming system and discuss difficulties when integrating it into already existing C++ applications. Then we describe the CuPP framework and explain how it solves these difficulties. Afterwards we demonstrate the benefits of CuPP on the example of OpenSteer, a steering library and a demo application. With only a small amount of code changes, the performance was increased by a factor of 42 as compared to the CPU version.

CONTENTS

CHAPTER 1 – INTRODUCTION 1
- 1.1 Structure of this thesis . . . 2

CHAPTER 2 – CUDA 5
- 2.1 Hardware model . . . 5
- 2.2 Software model . . . 6
- 2.3 Performance . . . 8
- 2.4 Why a GPU is no CPU . . . 10

CHAPTER 3 – PROGAMMING WITH CUDA 11
- 3.1 Programming the device . . . 11
- 3.2 Controlling the device . . . 14
- 3.3 CUDA and C++ . . . 16

CHAPTER 4 – CUPP 21
- 4.1 Device management . . . 21
- 4.2 Memory management . . . 22
- 4.3 A C++ kernel function call . . . 23
- 4.4 Use of classes . . . 26
- 4.5 Type transformation . . . 28
- 4.6 CuPP Vector . . . 29

CHAPTER 5 – OPENSTEER FUNDAMENTALS 33
- 5.1 Steering . . . 33
- 5.2 Flocking . . . 34
- 5.3 OpenSteer . . . 38

CHAPTER 6 – GPU STEERING 41
- 6.1 Agent by agent . . . 41
- 6.2 Step by step . . . 42
- 6.3 Performance . . . 46

CHAPTER 7 – CONCLUSION 49

Chapter 1

Introduction

Over the last five years, general-purpose computation on graphics processing units (known as GPGPU) has evolved from a rather unknown research area to a technique used in real world applications [Fol07]. Both memory bandwidth and floating-point performance of graphics processing units (GPUs) outrange their CPU counterparts roughly by a factor of 10. In the beginning of GPGPU development, GPUs had to be programmed with programming systems not designed for general-purpose computations but 3D graphics. This has changed in the last years, when multiple specially designed GPGPU programming systems became available.

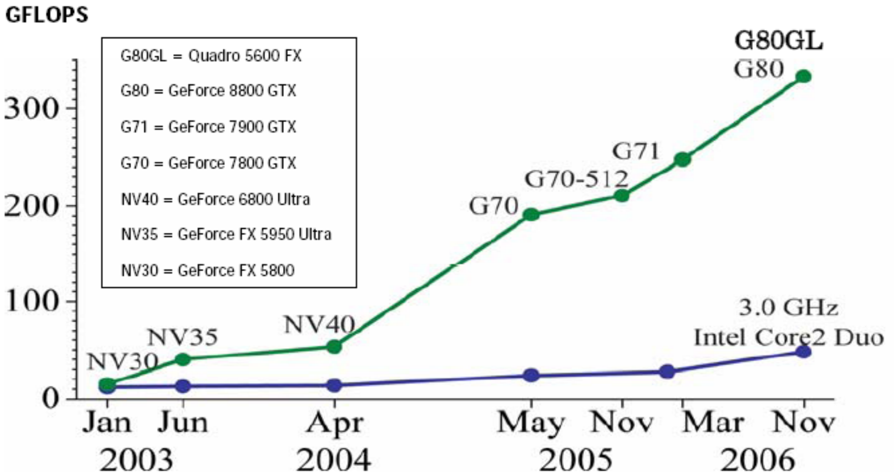

Figure 1.1: A comparison between the floating point performance of the CPU and the GPU [Cor07c].

This thesis is based on a programming system developed by NVIDIA called CUDA. CUDA exposes the GPU processing power in the familiar C programming language. It has been designed to be used together with C or C++ programs, but the integration into an already existing C++ program is rather complex.

In this thesis, we describe a novel C++ framework, called CuPP, which eases the integration of CUDA into C++ programs. CuPP automatizes common operations for the developer, e.g. GPU memory management. To expose the benefits of our approach, we have integrated CUDA/CuPP into OpenSteer as an example. OpenSteer is a game-like application used to simulate and debug some artificial intelligence aspects of games. We decided to concentrate our approach on a specific scenario – the simulation of birds in a flock, called Boids.

1.1 Structure of this thesis

This thesis is divided into three parts. The first part discusses the CUDA architecture (chapter 2) and the CUDA programming environment (chapter 3), including a discussion on why the integration into C++ programs is problematic. The second part of the thesis introduces the CuPP framework (chapter 4), which was designed to solve these problems. The last part demonstrates the benefits of CuPP, by integrating CUDA into the OpenSteer Boids scenario as example application. At first, the theoretical basics of the example application and its architecture are discussed (chapter 5), followed by a description of the integration of CUDA/CuPP in OpenSteer and performance measurements (chapter 6).

The CUDA programming system

Chapter 2

CUDA

This chapter introduces CUDA version 1.0, which was released by NVIDIA in 2007. CUDA is one of the first GPGPU programming systems offering high-level access to the GPU. Previously, GPGPU programming was mostly done, by using OpenGL or DirectX [OLG$^+$05], which are designed for 3D graphics and not for general computations. CUDA consists of both a hardware and a software model allowing the execution of computations on modern NVIDIA GPUs in a data-parallel fashion. We describe both models. Afterwards we discuss the performance of CUDA, especially performance-critical parts. We end the CUDA architecture discussion with a comparison between the CPU concept and CUDA. Chapter 3 introduces the CUDA programming language and the available libraries. The information we provide in chapters 2 and 3 is based on the CUDA programming handbook [Cor07c], if not explicitly stated otherwise.

2.1 Hardware model

CUDA requires a GPU that is based on the so-called G80 architecture, which was released by NVIDIA in 2006. Unlike CPUs, which are designed for high sequential performance, GPUs are designed to execute a high amount of data-parallel work. This fundamental difference is reflected in both the memory system and the way instructions are executed, as discussed next.

GPUs are constructed in a superscalar fashion. A GPU is implemented as an aggregation of multiple so-called *multiprocessors*, which consist of a number of SIMD ALUs. A single ALU is called *processor*. According to the SIMD concept, every processor within a multiprocessor must execute the same instruction at the same time, only the data may vary. We call this build-up, which is shown in figure 2.1, *execution hierarchy*.

Figure 2.1 also shows that each level of the execution hierarchy has a corresponding memory type in what we call the *memory hierarchy*. Each processor has access to local 32-bit *registers*. On multiprocessor level, so-called *shared memory* is available where all processors of a multiprocessor have read/write access to. Additional *device memory* is available to all processors of the device for read/write access. Furthermore, so-called

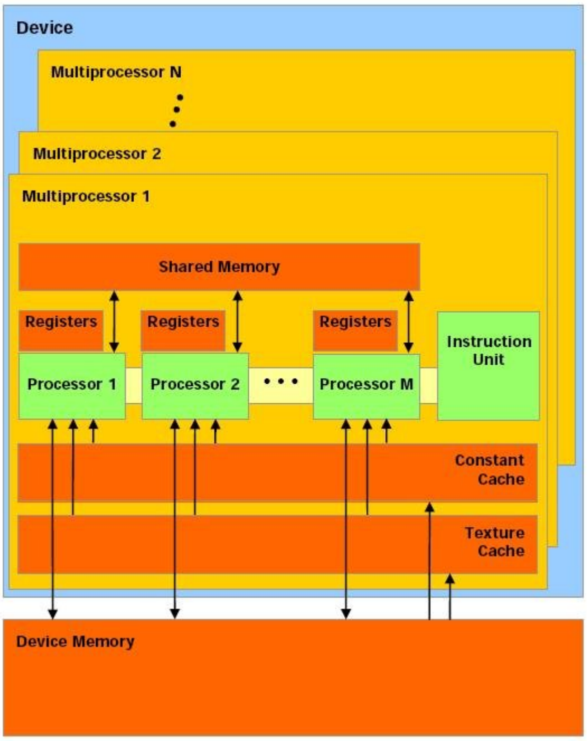
Device
Multiprocessor N
Multiprocessor 2
Multiprocessor 1
Shared Memory
Registers
Registers
Registers
Processor 1
Processor 2
Processor M
Instruction Unit
Constant Cache
Texture Cache
Device Memory

Figure 2.1: The CUDA hardware model: A device consists of a number of multiprocessors, which themselves consist of a number of processors. Correspondingly, there are different types of memory [Cor07c].

texture and constant caches are available on every multiprocessor.

2.2 Software model

The software model of CUDA offers the GPU as data-parallel coprocessor to the CPU. In the CUDA context, the GPU is called *device*, whereas the CPU is called *host*. The device can only access the memory located on the device itself.

A function executed on the device is called *kernel*. Such a kernel is executed in the single program multiple data (SPMD) model, meaning that an user-configured number of threads execute the same program.

A user-defined number of threads ($\leq$ 512) are batched together in so-called *thread blocks*. All threads within the same block can be synchronized by a barrier-like construct, called `__syncthreads()`. The different blocks are organized in a *grid*. A grid can consist of up to 2^{32} blocks, resulting in a total of 2^{41} threads. It is not possible to synchronize blocks within a grid. If synchronization is required between all threads, the work has to be split into two separate kernels, since multiple kernels are not executed in parallel.

Threads within a thread block can be addressed by 1-, 2 or 3-dimensional indexes. Thread blocks within a grid can be addressed by 1- or 2-dimensional indexes. A thread

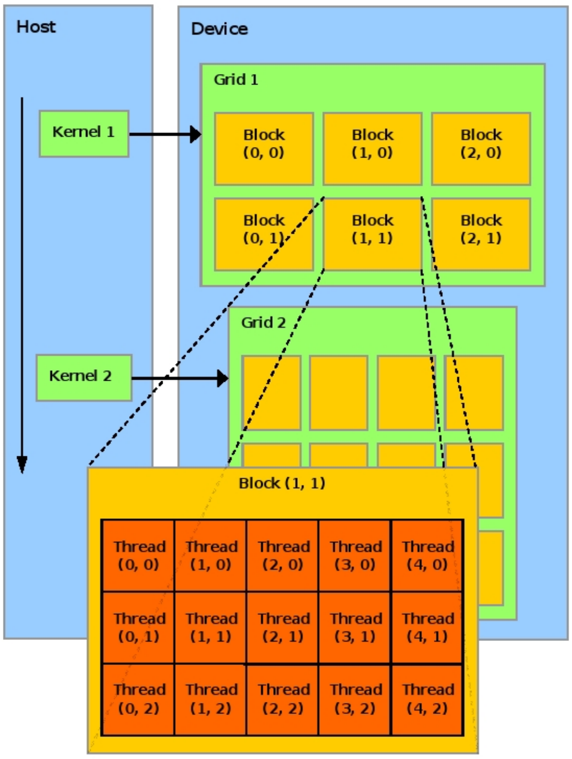
Host
Device
Grid 1
Kernel 1
Block (0, 0)
Block (1, 0)
Block (2, 0)
Block (0, 1)
Block (1, 1)
Block (2, 1)
Grid 2
Kernel 2
Block (1, 1)
Thread (0, 0)
Thread (1, 0)
Thread (2, 0)
Thread (3, 0)
Thread (4, 0)
Thread (0, 1)
Thread (1, 1)
Thread (2, 1)
Thread (3, 1)
Thread (4, 1)
Thread (0, 2)
Thread (1, 2)
Thread (2, 2)
Thread (3, 2)
Thread (4, 2)

Figure 2.2: The CUDA execution model: A number of threads are batched together in blocks, which are again batched together in a grid [Cor07c].

is thereby unambiguously identified by its block-local thread index and its block index. Figure 2.2 gives a graphical overview of this concept. The addressing scheme used influences only the addressing itself, not the runtime behavior of an application. Therefore it is mostly used to simplify the mapping of data elements to threads – e.g. see the matrix-vector multiplication provided by NVIDIA [Cor07b]. When requiring more than 2^{16} thread blocks, 2-dimensional block-indexes have to be used. During the development of our example application, only 1-dimensional indexing was used for both threads and thread blocks.

A grid is executed on the device by scheduling thread blocks onto the multiprocessors. Thereby, each block is mapped to one multiprocessor. Multiple thread blocks can be mapped onto the same multiprocessor and are then executed concurrently. If multiple blocks are mapped onto a multiprocessor, its resources, such as registers and shared memory are split among the thread blocks. This limits the amount of blocks that can be mapped onto the same multiprocessor. A block stays on a multiprocessor until it has completed the execution of its kernel.

Thread blocks are split into SIMD groups called *warps* when executed on the device. Every warp contains the same amount of threads. The number of threads within a warp is defined by a hardware-based constant, called *warp size*. Warps are executed by scheduling them on the processors of a multiprocessor, so that a warp is executed in

software model (memory address space)	hardware model (memory type)	access by	
		device	host
thread local	registers & device	read & write	no
shared	shared	read & write	no
global	device	read & write	read & write

Table 2.1: An overview of the mapping between the hardware model memory types and the software model types including accessibility by device and host.

SIMD fashion. On the currently available hardware, the warp size has a value of 32, whereas 8 processors are available in each multiprocessor. Therefore a warp requires at least 4 clock cycles to execute an instruction. Why a factor of 4 was chosen is not known to public known, but it is expected to be used to reduce the required instruction throughput of the processors, since a new instruction is thereby only required every 4th clock cycle.

The grid and block sizes are defined for every kernel invocation and can differ even if the same kernel is executed multiple times. A kernel invocation does not block the host, so the host and the device can execute code in parallel. Most of the time, the device and the host are not synchronized explicitly – even though this is possible. Synchronization is done implicitly when data is read from or written to memory on the device. Device memory can only be accessed by the host if no kernel is active, so accessing device memory blocks the host until no kernel is executed.

The memory model of the CUDA software model differs slightly from the memory hierarchy discussed in section 2.1. Threads on the device have their own local memory address space to work with. Additional threads within the same block have access to a block-local shared memory address space. All threads in the grid have access to global memory address space and read-only access to both constant and texture memory address spaces.

The different memory address spaces are implemented as shown in table 2.1. Shared and global memory address space implementation is done by their direct counterpart in the hardware. Local memory address space is implemented by using both registers and device memory automatically allocated by the compiler. Device memory is only used when the number of registers exceeds a threshold.

2.3 Performance

The number of clock cycles required by some instructions, can be seen in table 2.2. To note some uncommon facts:

- Synchronizing all threads within a thread block has almost the same cost as an addition.
- Accessing shared memory or registers comes at almost no cost.

Instruction	Cost (clock cycles per warp)
FADD, FMUL, FMAD, IADD	4
bitwise operations, compare, min, max	4
reciprocal, reciprocal square root	16
accessing registers	0
accessing shared memory	≥ 4
reading from device memory	400 - 600
synchronizing all threads within a block	4 + possible waiting time

Table 2.2: An overview of the instruction costs on the G80 architecture

- Reading from device memory costs an order of magnitude more than any other instruction.

The hardware tries to hide the cost of reading from device memory by switching between warps. Nonetheless reading from device memory is expensive and the high costs can not be hidden completely in most cases. Reading data from global memory should therefore be minimized, for instance by manually caching device memory in shared memory. Unlike reading from device memory, writing to device memory requires less clock cycles and should be considered a *fire-and-forget* instruction. The processor does not wait until memory has been written but only forward the instruction to a special memory writing unit for execution.[1]

As said in section 2.2, warps are SIMD blocks requiring all threads within the warp to execute the same instruction. This is problematic when considering any type of control flow instructions (`if`, `switch`, `for`, `do`, `while`), since it would require uniform control flow across all threads within a warp. CUDA offers two ways of handling this problem automatically, so the developer is not forced to have uniform control flow across multiple threads. Branches can be serialized, or executed by predication. Predication is not discussed here since experiments show it is currently not used – even though stated differently in the CUDA Programming Guide [Cor07c].

When multiple execution paths are to be executed by one warp – meaning the control flow diverges – the execution paths are serialized. The different execution paths are then executed one after another. Therefore serialization increases the number of instructions executed by the warp, which effectively reduces the performance. Code serialization is done by the hardware itself in an unspecified but correct way – meaning even though a processor executed an instruction that should not be executed, the result is correct. How much branching can affect the performance is almost impossible to predict in a general case. When the warp does not diverge, only the control flow instruction itself is executed.

---

[1] Writing to device memory itself is not discussed in [Cor07c], but this is a widely accepted fact by the CUDA developer community, see e.g. http://forums.nvidia.com/index.php?showtopic=48687.

2.4 Why a GPU is no CPU

GPUs are designed for executing a high amount of data-parallel work – NVIDIA suggests having at least 6,400 threads on the current hardware, whereas 64,000 to 256,000 threads are expected to run on the upcoming hardware generations. Unfortunately, massive parallelism comes at the cost of both missing compute capacities and bad performance at some scenarios.

- Current GPUs are not capable of issuing any function calls in a kernel. This limitation can be overcome by compilers by inlining all function calls – but only if no recursions are used. Recursion can be implemented by manually managing a function stack. But this is only possible if the developer knows the amount of data required by the function stack, since dynamic memory allocation is also not possible. Again, this limitation may be weakened by the developer – possibly by implementing a memory pool[2] – but currently no work in this area is available to public.
- GPUs are not only designed for a high amount of data-parallel work, but also optimized for work with a high arithmetic intensity. Device memory accesses are more expensive than most calculations. GPUs perform poorly in scenarios with low arithmetic intensity or a low level of parallelism.

These two facts show that unlike currently used CPUs, GPUs are specific and may even be useless at some scenarios. Writing fast GPU program requires

- a high level of arithmetic intensity – meaning much more arithmetic instructions then memory access instructions.
- a high degree of parallelism – so all processors of the device can be utilized.
- predictable memory accesses, so the complex memory hierarchy can be utilized as good as possible, e.g. by manually caching memory accesses in shared memory.

The performance of a CPU program is not affected that badly by memory accesses or by a low degree of parallelism. The cost of memory accesses is rather low due to the different architecture of modern CPUs and GPUs – e.g. the usage of caches. Since current multicore CPUs still offer high sequential performance and the number of cores available is small as compared to the number of processors available on a GPU[3], less independent work is required for good performance.

---

[2] Preallocating a high amount of memory and than hand it out at runtime to simulate dynamic memory allocation.

[3] The hardware used for this thesis offers 96 processors.

Chapter 3

Progamming with CUDA

Based on the software architecture described in the previous chapter, CUDA offers a special C dialect to program the device as well as two libraries. The *host runtime library* offers functions for device memory management and functions to control the device, whereas the *device runtime library* offers functions to ease programming the device. The following sections first introduce the C dialect used to program the device and the device runtime library, followed by the host runtime library. The last section discusses the integration of CUDA into a C++ application.

3.1 Programming the device

The language used to program the device is based on C and extended with

- function type qualifiers,
- variable type qualifiers,
- built-in variables.

We discuss these extensions in the following sections.

3.1.1 Function type qualifiers

Function type qualifiers are used to specify whether the function is callable from the host or the device. There are three different qualifiers available:

- The `__device__` qualifier defines a function that is executed on the device and callable from the device only. Such functions are always inlined and therefore neither function pointers are supported nor is it possible to take the address of such a function. This is a direct result from limitations we discussed in section 2.4.

function type qualifier	callable	runs on
`__host__`	host	host
`__device__`	device	device
`__global__`	host	device

Table 3.1: An overview of the CUDA function type qualifiers

- `__host__` defines a function that is executed and callable from the host only. This is the default behavior of functions without a qualifier.
- The `__global__` qualifier defines a kernel, meaning it is callable by the host only and executed on the device – so to speak a main function for the program executed by the device. A `__global__` function must have the return value `void` and may only be called as described in section 3.2.2.

`__device__` and `__host__` can be used together, so the function is compiled for both the host and the device. See listing 3.1 for a small example of how function qualifiers are used.

Listing 3.1: Function type qualifiers example

```
// callable from both device and host
__host__ __device__ int pass(int input) {
    return input;
}

// callable from the host, but executed on the device
__global__ void kernel (int input) {
    int a = pass(input);
}
```

3.1.2 Variable type qualifiers

Variable type qualifiers are used to define the memory storage class of the variable. Two of the available qualifiers are discussed next.

- `__device__` declares a variable residing on the device. If no additional variable type qualifier is used, such a variable is stored in global memory, has the lifetime of the application and can be accessed by all threads in the grid and the host by using the functions which are described in section 3.2.3. `__device__` variables can only be defined at file scope.
- Variables stored in shared memory are qualified with both the `__device__` and the `__shared__` qualifier. These variables are accessible only by the threads within a thread block and have the lifetime of the block, meaning after the thread block has finished executing, the memory is no longer accessible.

Variables defined without a qualifier, but inside a function executing on the device, are declared in thread local memory. Shared variables may be defined without the `__device__` qualifier in these functions. Pointers to both global and shared memory are supported, as long as it is possible to identify the memory address space the pointer belongs to at compile time, otherwise the pointer is expected to point to global memory. See listing 3.2 for an example.

Listing 3.2: Variable type qualifiers example

```
// an array allocated in global memory
__device__ int a[100];

__global__ void kernel (bool sh_mem) {
    // an array allocated in shared memory
    __shared__ int sh_a[100];
    int *pa;

    if (sh_mem) {
        pa = sh_a;
    } else {
        pa = a;
    }

    // this is undefined if sh_mem == true
    pa[50] = 1;
}
```

3.1.3 Built-in variables

There are four built-in variables available. The types of these variables are defined in the *common runtime library*[1]. The built-in variables `blockIdx` and `threadIdx` are of type `uint3`. `uint3` is a 3-dimensional vector type containing three unsigned integer values, which are stored in fields called `x`, `y` and `z`. The built-in variables `gridDim` and `blockDim` are of the type `dim3`. `dim3` is identical to `uint3`, except that all components left unspecified when creating have the value 1. The built-in variables store the following information:

- `gridDim` contains the dimensions of the grid. The dimension of the grid can only be defined in two dimensions (see section 2.2), therefore `gridDim.z` is always 1.
- `blockDim` contains the dimensions of the block.
- `blockIdx` contains the index of the block within the grid.

---

[1] The common runtime library defines texture types and vector types. Both are not used in this thesis, therefore the common runtime library is not discussed any further. The interested reader may find more information in [Cor07c] on pages 22 - 25.

- `threadIdx` contains the index of the thread within the block.

Values are assigned to these variables when invoking a kernel (see section 3.2.2).

3.1.4 Device runtime library

The device runtime library contains:

- mathematical functions,
- synchronization function,
- type conversion functions,
- type casting functions,
- texture functions,
- atomic functions.

Only the synchronization function is discussed here, see [Cor07c] for an overview of the other functionality. The synchronization function `__syncthreads()` is used to coordinate threads within a thread block. `__syncthreads()` blocks the calling threads until all other threads of the same thread block have called the function too – similar to the barrier construct of OpenMP [Ope05]. Using this synchronization function in conditional code is only well defined, if the condition evaluates identical over all threads of the same block.

3.2 Controlling the device

The functions to control the device by the host are available through the host runtime library. This library offers functions to handle

- device management,
- memory management,
- execution control,
- texture reference management,
- interoperability with both OpenGL and Direct3D.

Device management, execution control and memory management are discussed in the following sections; the other functionality is explained in [Cor07c] pages 28 - 40.

3.2.1 Device management

The CUDA device management requires that one host thread is bound to at most one device. Multiple threads can be used to control multiple devices. `cudaSetDevice( int dev )` associates the calling thread with device number `dev`. Multiple functions are available to get the device number, e.g. `cudaChooseDevice( int* device, const cudaDeviceProp* prop )` which returns the device number best matching the requested properties `prop`. The property may define the size of memory required or the support for special compute capabilities of the hardware, e.g. atomic operations. If no device has been selected before the first kernel call, device 0 is automatically selected.

3.2.2 Execution control

Execution control refers to the execution of a kernel on the device. There are two possibilities of which one is only supported by the NVIDIA `nvcc` compiler and is therefore not discussed here. See [Cor07c] section 4.2.3 for details. The other possibility is to use the host runtime library. This is done in three steps:

1. Call `cudaConfigureCall(dim3 gridDim, dim3 blockDim)` which configures the next kernel launch, by defining the size of the grid and the size of the thread blocks.

2. Push the kernel call parameters on the kernel stack. The kernel stack is similar to a function stack, but located on the device. The transfer is done by calling `cudaSetupArgument(T arg, size_t offset)` for each parameter to be pushed on the stack. `offset` defines the position on the stack, where the argument should be stored.

3. Start the kernel by calling `cudaLaunch(T kernel_p)`. The passed argument must be a function pointer pointing to the `__global__` function to be executed.

3.2.3 Memory management

Device memory can be allocated as two different memory types, but only *linear memory* is discussed next, since it is the only memory type used in this thesis. The interested reader may find more information in [Cor07c]. *Linear memory* is memory as it is well known from the general CPU realm, meaning it is allocated in a 32-bit memory address space and referable by pointers.

Linear memory management is similar to the classical C style memory management using `free()` and `malloc()`. `count` bytes of linear memory are allocated by calling `cudaMalloc( void** devPtr, size_t count )`. The address of the memory allocated is stored in `*devPtr`. Memory is freed by calling `cudaFree( void* devPrt )`. Deferring a pointer, returned by `cudaMalloc` on the host side is undefined.

Transferring data between host and device is done by calling `cudaMemcpy( void* dest, const void* source, size_t count, cudaMemcpyKind kind )`. `count` specifies the number of bytes to be transferred. `dest` and `source` define the destination and the source address and `kind` specifies the direction of the copy. Data can be copied in any possible direction, e.g. from device to host or device to device. The `cudaMemcpy()` function blocks the calling thread, if a kernel is currently executing on the device, since device memory can not be accessed by the host while a kernel is active (see section 2.2).

3.3 CUDA and C++

As stated before, CUDAs design is based on the C programming language, and the CUDA compiler (called `nvcc`) only supports C. Therefore the integration of CUDA into a C++ application results in several problems due to the lack of C++ support by the CUDA compiler. NVIDIA provides a simple example of how CUDA can be integrated into a C++ application [Cor07a], but this basic approach has some drawbacks. NVIDIA suggests issuing the kernel call in a `__host__`-function, which is then compiled by `nvcc`; thereby the function calling the kernel is a pure C function. To execute the kernel, this function is called from the C++ application and all data passed to the kernel must be passed to this function as well. The integration of C in a C++ program may sound trivial, but as it has been discussed by Meyers [Mey95] in item 34 and Cline et al. [CGL98] in chapter 36, a wide range of problems may occur. The major problem regarding the CUDA integration approach suggested by NVIDIA is that not all data types or data structures used in C++ can be used in C. As C does not support classes it is expected that most user-defined data types or data structures can not be used without code changes. But as the memory layout rules for both C structures and C++ classes not using virtual functions are identical, it is possible to write code that allows the access to C++ classes in C. An example how this code might be written is given in [CGL98] section 36.05, but this requires adding a C structure with the identical memory layout as the C++ class and two new functions for every member function of the class. The amount of required code changes may therefore be high. For example when considering the access of a STL container in C, all required functions must be duplicated and a structure with the identical memory layout must be defined, which requires STL platform-specific code, since the memory layout of the STL container is not defined [Mey01].

Even if the general C++/C mixing problems are solved, passing an object to a kernel can not be done the same way as it is done by passing an object to a function. As mentioned in section 2.2, the device can not access host memory, so passing any object using pointers to a kernel results in invalid pointers when using the automatically generated copy constructor. This derives from the fact that the automatically generated copy constructor does only copy the value of the pointer (*shallow copy*) and not the memory the pointer points at (*deep copy*). Developers may write their own copy constructor and use it to copy all required data to the device, but they must know that this copy is used by the device and not by the host. This information is not provided by the C++ copy constructor, so additional work is required by the developer to ensure the copy is used in the correct memory address space.

The CuPP framework, which is discussed in the following chapter, offers a solution to the problems outlined above. The framework calls member functions of the objects passed to the device, which can be used to transfer all required data from the host to the device and correct the value of used pointers. Additionally it is possible to define two independent structures, where one is only used by the device and the other by the host. The structure used by the device must satisfy the C rules and the other structure can use all C++ functionality, including virtual functions and inheritance. The interface between both types is a transformation function, which is executed by the host.

The CuPP framework

Chapter 4

CuPP

As discussed in section 3.3, CUDAs design is based on the C programming language, and integrating CUDA into a C++ application results in problems, e.g. when using classes. The CuPP framework as described in this chapter has been developed to ease the integration of CUDA into C++ applications. At first, we introduce the CuPP device and memory management followed by the new C++ compliant kernel call, including the possibility to pass classes to a kernel or use references. The support for classes allows for a new approach to reflect the differences between the device and the host on programming level, which is described in section 4.5. For brevity, a complete overview of all CuPP functions has been omitted. The interested reader may find a documentation generated by doxygen [Dox07] at [Bre07]. An overview of the whole CuPP functionality – including possible additions not implemented during this thesis – can be found in figure 4.1.

4.1 Device management

Device management is no longer done implicitly when associating a thread with a device as it was done by CUDA. Instead, the developer is forced to create a device handle (`cupp::device`), which is passed to all CuPP functions using the device, e.g. kernel calls and memory allocation. Currently, only one device handle per thread is supported, but the CuPP framework itself is designed to offer multiple devices to the same host thread with only minor interface changes. Due to the lack of hardware with multiple devices no further work was done in this area.

The creation of a device handle can be done by specifying properties (similar to the original CUDA concept), or without any specification, which uses the default device (see listing 4.1). The device handle can be queried to get information about the device, e.g. supported functionality or total amount of memory. When the device handle is destroyed, all memory allocated on this device is freed as well.

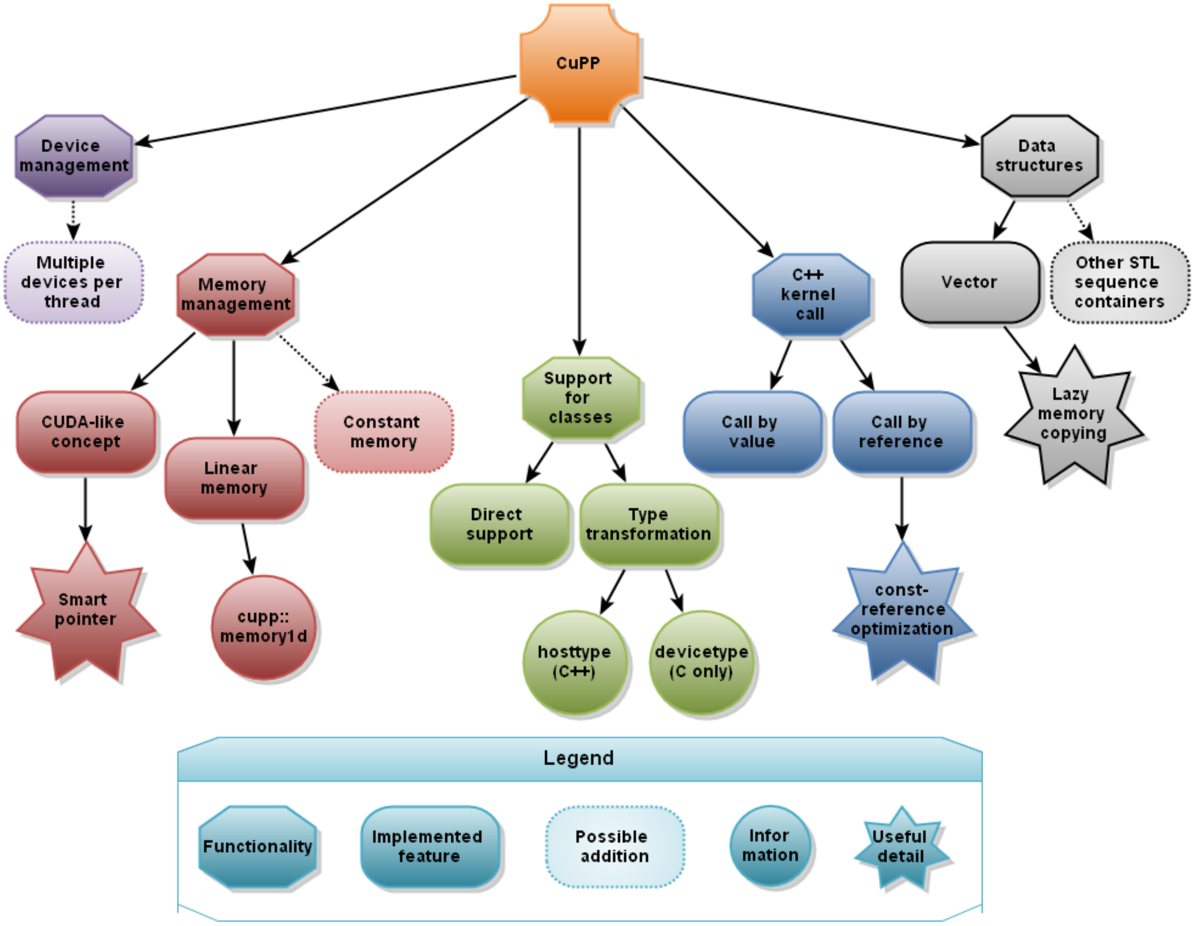
CuPP
Device management
Multiple devices per thread
Memory management
CUDA-like concept
Smart pointer
Linear memory
cupp:: memory1d
Constant memory
Support for classes
Direct support
Type transformation
hosttype (C++)
devicetype (C only)
C++ kernel call
Call by value
Call by reference
const-reference optimization
Data structures
Vector
Lazy memory copying
Other STL sequence containers
Legend
Functionality
Implemented feature
Possible addition
Infor mation
Useful detail

Figure 4.1: CuPP overview

Listing 4.1: CuPP device handle example

```cpp
#include <cupp/device.h>

int main() {
    // creates a default device
    cupp::device device_hdl();

    // ...
}
```

4.2 Memory management

The CuPP framework currently only supports linear memory (see section 3.2.3). Two different memory management concepts are available. One is identical to the one offered by CUDA, unless that exceptions are thrown when an error occurs instead of returning an error code. To ease the development with this basic approach, a boost library-compliant shared pointer [GBD07] for global memory is supplied. Shared pointers are a variant of smart pointers. The memory is freed automatically after the last smart pointer pointing to a specific memory address is destroyed, so resource leaks can hardly occur.

The second type of memory management uses a class called `cupp::memory1d`. Objects of this class represent a linear block of global memory. The memory is allocated when the object is created and freed when the object is destroyed. When the object is copied, the copy allocates new memory and copies the data from the original memory to the newly allocated one. Data can be transferred from the host to the device by two kinds of functions:

- functions using pointers
- functions using C++ iterators

The functions using pointers require a specific memory layout for the data to be transferred from the host to the device. The data must be stored in a linear memory block, so the memory to be transferred can be defined by a starting address and the amount of data. The functions using iterators, in contrast, can be used to transfer any data structure supporting iterators to or from host memory. As already explained `cupp::memory1d` is always a linear memory block on the device, so the transferred data structure must be linearized in some way. The CuPP framework linearizes the data structure based on the way it is traversed by its iterators, meaning the value of the iterator passed to the function is the first value in the memory block, the value the iterator points to when incrementing is the next value in the memory block and so on.

Both memory management concepts relieve the developer from manually freeing global memory. The second concept additionally simplifies the usage of global memory as a member of an object. If `cupp::memory1d` is used as a member of class and an object of this class is copied, the memory on the device is copied too. This is useful, when implementing a data structure like the vector described in section 4.6. When a vector is copied, the copy is expected to have its own dataset, so if a vector uses global memory, it must be copied as well. This is done automatically if `cupp::memory1d` is used.

4.3 A C++ kernel function call

CuPP supports CUDA kernel calls by offering a so called functor called `cupp::kernel`. In C++, functors are classes supporting `operator()` and thereby function-like calls, e.g. `f()` can be both a function call, and `f.operator()`. The call of `operator()` of `cupp::kernel` calls the kernel and issues all instructions described in section 3.2.2. A `cupp::kernel` object is created by passing a function pointer pointing to a CUDA `__global__` function to the constructor. Grid and block dimension of the kernel calls can be passed as an optional parameter to the constructor or may be changed later with set-methods.

Calling kernels by using `cupp::kernel::operator()` mimics a C++ function call and supports both call-by-reference and call-by-value semantic. The syntax for writing the CUDA kernel is identical to the C++ syntax, e.g. reference types are postfixed by an ampersand (`&`). This syntax is only possible because, when compiling with `nvcc`, C++-like references are automatically transformed into automatically derefered pointers[1]. Listings 4.2 and 4.3 offer a simple example of how `cupp::kernel` can be used, including the call-by-reference semantic.

Listing 4.2: CuPP kernel example: CUDA file

```
// CUDA kernel
__global__ void kernel (int i, int& j) {
    // ...
    j = i/2;
}

// typedef of the function pointer type of 'kernel' to kernelT
typedef void (*kernelT) (int i, int& j);

// returns a function pointer pointing to 'kernel'
kernelT get_kernel_ptr() {
    return kernel;
}
```

Listing 4.3: CuPP kernel example: C++ file

```
#include <cupp/device.h>
#include <cupp/kernel.h>

typedef void (*kernelT) (int i, int& j);

// prototype, implemented in CUDA file
kernelT get_kernel_ptr();

int main() {
    int j = 0;

    // define the dimension of the kernel
    // 10 * 10 = 100 thread blocks
    //  8 *  8 =  64 threads per block
    dim3 grid_dim  = make_dim3 (10, 10);
    dim3 block_dim = make_dim3 ( 8,  8);

```

---

[1] This behavior is not officially supported in CUDA 1.0, but by using it, a more C++-like interface can be offered. Instead of relying on this behavior, a reference-like structure could be implemented using C macros.

```
    // create the kernel functor
    cupp::kernel f( get_kernel_ptr(), grid_dim, block_dim );

    // call the kernel by using operator()
    // 1. parameter: the device the kernel runs on
    // all following parameters are passed to 'kernel'
    f (device_hdl, 10, j);

    // j == 5
}
```

The next sections describe the default implementation of call-by-value and call-by-reference. It can be enhanced for classes using pointers, as described in section 4.4. The following two sections refer to objects only, but the behavior described is identical for primitive data types (so called PODs).

4.3.1 Call-by-value

The implementation of call-by-value semantic is almost identical to that of a normal C++ function. When an object is passed by value to a C++ function:

1. A copy of the object is generated by calling its copy constructor.
2. The copy is passed to the function.
3. The function is executed.
4. After the function call, the copy calls its destructor.

When an object is passed to a kernel using CuPP:

1. A copy of the object is generated by calling its copy constructor on the host.
2. The copy is passed to the kernel, by calling `cudaSetupArgument()`, which generates a byte-wise copy of the object on the kernel stack (see section 3.2.2 for details).
3. The kernel is executed.
4. After the kernel has started, the generated host copy calls its destructor.

Both methods differ only in the time the destructor is called. The destructor of the host side copy is not called after the kernel has finished, but may be called earlier. This is done to prevent an unnecessary synchronization between host and device. If for some reason the synchronization is required, it can be done explicitly in the destructor. In the development done during this thesis, it was never required to synchronize the host and the device, especially since global memory accesses implicitly synchronize host and device. So the destructor can not influence the currently running kernel. Even when it would try to change the memory on the device, these changes would be carried out after the kernel execution is complete.

4.3.2 Call-by-reference

The CuPP kernel call-by-reference semantic is again implemented similar to that of C++. References refer to an object, and if a reference is passed to a function, it directly refers to an object created outside of the function, and changes done to a reference change this object. In C++, references are typically implemented as automatically derefered pointers. So the reference itself is no more than the address of an object. A C++ function call using call-by-reference may be implemented as follows:

1. The reference is created by taking the address of the object to be passed to the function.
2. This reference is passed to the function.
3. The function is executed.
4. The reference is destroyed after the functions execution.

CuPP call-by-reference is implemented as follows:

1. The object to be passed to a kernel is copied byte-wise to the global memory on the device.
2. The address of the object in global memory is passed to the kernel.
3. The kernel is executed.
4. The copy in global memory is byte-wise copied back to host memory overwriting the original object.

Step 4 includes a synchronization, since global memory is accessed. If the device has not changed the object, there would be no need to copy the data back to the host. To reduce the amount of unneeded copies, the CuPP framework analyzes the kernel declaration, and if a reference is defined as constant (using the `const` keyword), the last step is skipped. This analysis is done using the boost function traits [ADS$^+$07] and self-written template metaprogramming code. Template metaprogramming code is carried out at compile time and therefore can increase the time required for compilation.

4.4 Use of classes

The CuPP framework also supports passing of classes to a kernel. The technique described in this section does not support `virtual` functions. `virtual` functions are used as member functions of classes, and can be overwritten in a derived class[2]. `virtual` functions change the internal data representation of an object, so using them as a member function is not possible, since an object is byte-wise copied to global memory.

---

[2] There are exceptions to this rule. See e.g. [Str00] for a more complete overview of virtual functions and inheritance in C++.

Using the default behavior for both call-by-value and call-by-reference semantics described in section 4.3 can work fine for all PODs and even some classes, but it is problematic when pointers or dynamic memory allocation are used by a class. Both call semantics use a byte-wise copy to transfer the object to the kernel stack or the global memory, so the problems discussed in section 3.3 have not been solved so far. To solve the problem regarding the usage of pointers, it is possible to enhance the two calling behaviors by specifying the following three functions as member functions of a struct[3].

Listing 4.4: CuPP kernel call traits definition

```
struct example {
    #if !defined(NVCC)
    device_type transform ( const cupp::device &d ) const;

    cupp::device_reference < device_type >
       get_device_reference ( const cupp::device &d ) const;

    void dirty ( device_reference <device_type> device_ref );
    #endif
};
```

`device_type` is discussed in section 4.5. For now, it should be considered to be of the same type as the class in which the function is defined (here `example`). The function `transform()` is called by the framework, when an object of this class is passed by value to a kernel and has to return an object, which can be byte-wise copied to the device. The function is called on the copy created in step 1 of the call-by-value semantic (see section 4.3.2). It may be used to copy additional data to the global memory and to set the pointers to the correct values.

`get_device_reference` is called by the framework when the object is passed by reference and has to return an object of the type `cupp::device_reference <device_type>`. `cupp::device_reference <T>` is a reference to an object of type `T` located on the device. When created, it automatically copies the object passed to its constructor to global memory. The member function `get()` can be used to transfer the object from global memory back to the host memory.

Objects passed to a kernel as a non-`const` reference can be changed by the device. These changes have to be transferred back to the object on the host side; therefore the object stored in the host memory has to be notified. This notification is done by calling the `dirty()` function. A `device_reference` to the object stored in global memory is passed to this function, so the changed data on the device can be accessed by calling the member function `get()` and can be used to update the old host data.

It is optional to implement any of these three functions. The CuPP framework employs template metaprogramming to detect whether a function is declared or not. If it is not declared, the default implementation shown in listing 4.5 is used.

---

[3] Note, that defining functions inside a struct is not C compliant, therefore the functions should only be defined if compiled with the C++ compiler – as done in the example.

Listing 4.5: Default implementation

```
struct example {
    device_type transform (const cupp::device &d) const {
        // cast *this to device type
        return static_cast<device_type>(*this);
    }

    cupp::device_reference < device_type >
       get_device_reference (const cupp::device &d) const {
        // copy the 'transformed' object to global memory
        return cupp::device_reference < device_type > (d,
           transform(d));
    }

    void dirty (device_reference <device_type> device_ref) {
        // replace *this with updated device data
        *this = static_cast<host_type> ( device_ref.get() );
    }
};
```

It is expected, that the `transform()` and `dirty()` functions must only be implemented, when pointers or type transformations – as described in the next section – are used. `get_device_reference()` must only be implemented if `transform()` and `dirty()` are implemented and the default behavior is not satisfying.

4.5 Type transformation

As described in section 2.4, CPUs and GPUs differ in a lot of details, e.g. the way memory should be accessed or the missing compute capabilities. Based on these differences, different approaches to solve a problem may be required to achieve good performance. On the host side, using a balanced tree may be a good choice to store data, in which searching is a regular operation. But this concept requires a high amount of rather unpredictable memory accesses, so the memory hierarchy of the device can not be used effectively. A simple brute force approach using shared memory as a cache may even perform better; as long as the factor between accessing global memory and shared memory is not outranged by the algorithmic benefit of the search done in a tree. But not only the memory access pattern when using a tree is problematic, also are the used algorithms. As discussed in section 2.4, using recursion on the device is rather problematic, due to the lack of function call capabilities and dynamic memory allocation.

The CuPP framework offers a way to reflect the differences between the two architectures. The developer can define two independent types, which get transformed into one another when transferred from one memory domain to the other. The type used by the host is called *hosttype*, whereas the type used by the device is called *devicetype*. The types can be optimized for the architecture they are used on. The matching is defined as shown in the following listing.

Listing 4.6: CuPP host/device type definition

```
// prototype
struct device_example;

// host type
struct host_example {
    typedef device_example  device_type;
    typedef host_example    host_type;
};

// device type
struct dev_example {
    typedef device_example  device_type;
    typedef host_example    host_type;
};
```

The matching between the two types has to be a 1:1 relation. The transformation of the two types is not only done if the objects are passed by value, as it is done by C++ function calls, but also if the objects are passed by reference. The transformation between the two types has to be done by the `transform()`, `get_device_reference()` and `dirty()` functions described in section 4.4. The next section gives an example of how the type transformation and these functions can be used.

4.6 CuPP Vector

The class `cupp::vector` (in the following just called `vector`) is supplied by the CuPP framework as a wrapper class of the STL vector [Str00]. It provides for an example implementation of a feature called *lazy memory copying*. The C++ STL vector is best described as an array, which can grow and shrink at runtime and fulfills the STL container requirements.

The CuPP vector implementation uses the host/device type bindings. The host offers almost[4] the same functionality as the C++ STL vector. The device type suffers from the problem that it is not possible to allocate memory on the device. Therefore the size of the vector cannot be changed on the device. The type transformation is not only done to the vector itself, but also to the type of the values stored by the vector. Therefore `vector<T>::device_type` is identical to `deviceT::vector<T::device_type>`, with `deviceT::vector` being the device-type of `vector`. This kind of transformation makes it possible to pass e.g. a two dimensional vector (`vector< vector<T> >`) to a kernel.

---

[4] The CuPP vector uses a technique called proxy classes to detect whether a value of the vector is just read or changed. Due to limitations of the C++ language itself, it is not possible to get this information without the usage of proxy classes. Proxy classes mimic the classes they are representing, but are not identical. Therefore they behave differently in some rather rare situations. Scott Meyers discusses this problem in detail in [Mey95] item 30.

Vectors are often used to store a large amount of data. Therefore transferring the data of the vector from or to global memory should be minimized. CuPP vectors use lazy memory copying to reduce the amount of copies done. The technique is based on some special behavior of the functions `transform()`, `get_device_reference()`, `dirty()` and monitors the values on the host side.

- `transform()` and `get_device_reference()` copy the vector data to global memory, if the data is out of date or no data has been copied to the device before.
- `dirty()` marks the host data of the device to be out of date.
- Any host functions or operators that read or write data, check if the host data is up to date and if not, copy the data from the device.
- Any host functions or operators changing the state of the vector, marks the data on the device to be out of date.

Using this concept, the developer may pass a vector directly to one or multiple kernels, without the need to think about how memory transfers may be minimized, since the memory is only transferred if it is really needed. The example application described in chapter 6 uses this technique to improve performance, e.g. see section 6.3.2.

Example application

Chapter 5

OpenSteer fundamentals

In this part of the thesis we demonstrate the benefits of the CuPP framework by integrating CUDA into the Boids scenario of OpenSteer. This chapter lays out the foundation of the simulation. We start with an overview of steering itself, followed by its usage in the scenario. After that the OpenSteer architecture and the performance of the existing CPU Boids simulation are discussed.

5.1 Steering

The term *steering* refers to life-like navigation of autonomous characters. Autonomous characters are special types of autonomous agents used in games or other interactive media and computer animations. In the following those autonomous characters are called *agents*. In most cases all agents are simulated by executing a so called *steering behavior*, which defines the way an agent behaves e.g. flee from another agent. The term *behavior* can refer to many different meanings, but throughout this thesis it refers to improvisational and lifelike actions of agents. Steering behaviors do not define the behavior of a group of agents, but the behavior of a single agent and are based on its local environment. The group behavior itself is an emergent phenomenon and results in intelligent behavior of the group.

The Boids scenario is simulating flocks of bird. One bird is simulated by one agent. The only action an agent can take in this scenario is moving to a specific direction; therefore all steering behaviors are represented by a 3-dimensional vector. These vectors are called *steering vectors*. A steering vector is interpreted in two different ways. The direction of the vector defines the direction in which the agent wants to move, whereas the length of the vector defines the acceleration.

An agent in the Boids simulation is represented by a sphere. The radius of the sphere is identical for all agents, so the representation of all agents is identical. The simulation takes place in a spherical world. An agent leaving the world is put back into the world at the diametric opposite point.

5.2 Flocking

The Boids scenario uses a steering behavior called *flocking*. This behavior is not only used to simulate flocks of birds but also herds or schools. Flocking as it is used here has been proposed by Reynhold [Rey99] and is built up from three basic behaviors: separation, cohesion and alignment. The final steering vector is a weighted addition of the steering vectors which is returned by these basic behaviors, as shown in listing 5.1.

Listing 5.1: Flocking

```
Vec3 flocking() {
    Vec3 SeparationW = weightA * normalize( separation() );
    Vec3 AlignmentW = weightB * normalize( alignment() );
    Vec3 CohesionW = weightC * normalize( cohesion() );
    Vec3 Flocking =  SeperationW + AlignmentW + CohesionW;
    return Flocking;
}
```

The following sections introduce these three basic steering behaviors, but at first the neighbor search is explained in detail. Steering behaviors in general are defined on the local environment of the agent. In the Boids scenario considered during this thesis, the local environment of an agent is the agents surrounding it – all other aspects of the world are uniform and never influence the behavior of an agent. Therefore all behaviors discussed next are using the neighbor search.

5.2.1 Neighbor search

The neighbor search is used to find agents within a given radius. This radius is called *neighbor search radius*. The number of agents to be found may be unlimited, but in most cases the number of agents is limited by a fixed value. The limitation is used as real individuals are not capable of reacting to an unlimited amount of other individuals surrounding it. We only consider the 7 nearest neighbors in the simulation, discussed in the following sections. The following pseudo code demonstrates the used algorithm.

Listing 5.2: Neighbor search

```
Agent[] neighbor_search() {
    Agent neighbors[7]

    for each (Agent A in World) {
        dist := distance(Me, A)

        if (dist < search radius) {
            // already the maximum nb of agents found?
            if (neighbors found < 7) {
                // no
                add A to neighbors
            } else {
                // yes, only store the 7 nearest
                B := Agent with the biggest distance in
                   neighbors
                if (distance (Me, B) > distance (Me, A)) {
                    overwrite B in neighbors with A
                }
            }
        }
    }

    return neighbors
}
```

The algorithm in listing 5.2 is used to find the 7 nearest agents within a given radius. As long as there are found less than 7 neighbors, all agents within the neighbor search radius are added as a neighbor. If there are already 7 neighbors found, the furthest neighbor is replaced by the new found one, if the new one is closer than the original one. This algorithm has a complexity of $O(n)$, with n being the number of agents within the world.

5.2.2 Separation

The separation steering behavior refers to the fact that one character within a flock tries to keep a certain distance to the other characters in the flock. To calculate this behavior a neighbor search is done. A repulsive force is calculated based on the distance vector between the simulated agent and every neighbor. At first the distance between a neighbor and the simulated agent is calculated. The distance vector is normalized, so the vector has the length 1. Agents further away should have a lower impact on the separation behavior, so the distance vector gets divided through its original length a second time. By this second division, the newly calculated value is smaller if the neighbor is further away. The value which is calculated for every agent is summed up and returned as the steering vector. See figure 5.1 for a graphical representation. The red vector is the steering vector, whereas the green lines show the offset. The following code shows the algorithm used to calculate this behavior.

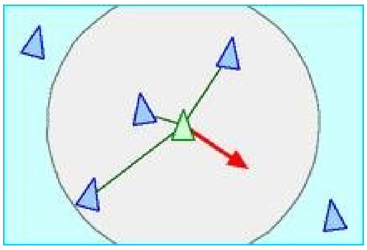

Figure 5.1: Graphical representation of the separation steering behavior. The red vector is the calculated steering vector [Rey99].

Listing 5.3: Separation steering behavior

```
Vec3 separation () {
    Agent[] neighborhood = neighbor_search()
    Vec3 steering = (0.0, 0.0, 0.0)

    for each (Agent A in neighborhood) {
        // opposite of the offset direction
        Vec3 distance = A.position - me.position

        // divided to get 1/d falloff
        steering -= (distance.normalize()/distance.length())
    }

    return steering
}
```

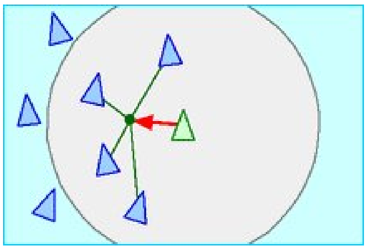

Figure 5.2: Graphical representation of the Cohesion steering behavior. The red vector is the calculated steering vector [Rey99].

5.2.3 Cohesion

The cohesion steering behavior tries to steer an agent to the center of other agents nearby, so multiple agents can start forming groups. This is done by finding all neighbors and then calculating the center of these agents. The resulting steering force is the difference between the current position of an agent and the calculated center. See figure 5.2 for a graphical representation. The red vector in the figure is the returning steering vector.

Listing 5.4: Cohesion steering behavior

```
Vector cohesion () {
    Agent[] neighborhood = neighbor_search()
    Vector3 steering = (0.0, 0.0, 0.0)

    for each (Agent A in neighborhood) {
        // accumulate sum of distance between neighbor
        // and local agent position
        steering += A.position - Me.position
    }

    return steering
}
```

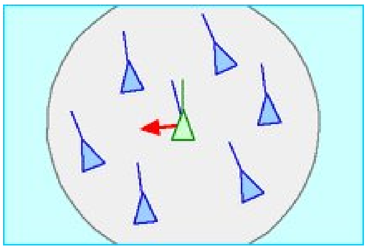

Figure 5.3: Graphical representation of the alignment steering behavior. The red vector is the calculated steering vector [Rey99].

5.2.4 Alignment

Alignment is used to have all agents of a group flying in the same direction. This is done by finding the neighbors and then averaging the forward vector of these agents. The forward vector points to the direction an agent is heading. The result is the difference between the forward vector of the agent and the calculated average forward vector. See figure 5.3 for a graphical representation. The red vector is the calculated steering vector.

Listing 5.5: Alignment steering behavior

```
Vector cohesion () {
    Agent[] neighborhood = neighbor_search()
    Vector3 steering = (0.0, 0.0, 0.0)

    for each (Agent A in neighborhood) {
        // accumulate sum of neighbor's heading
        steering += A.forward
    }

    steering -= neighborhood.size * Me.forward
    return steering
}
```

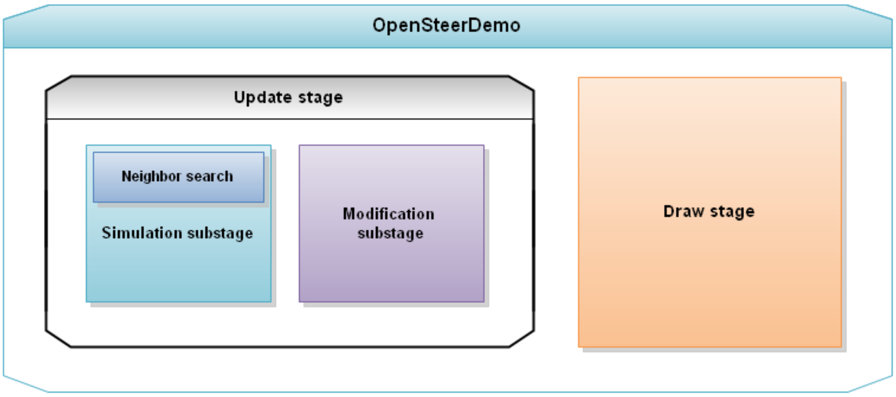

Figure 5.4: An architecture overview of OpenSteerDemo, showing all (sub)stages.

5.3 OpenSteer

Whereas the previous sections discussed the theoretical foundations of the Boids scenario, this section introduces OpenSteer in general and its architecture, followed by a performance overview of the already existing CPU version.

OpenSteer is an open-source library written in C++ by Reynolds in 2002. It provides simple steering behaviors and a basic agent implementation. OpenSteerDemo currently offers different scenarios – among others the Boids scenario.

The design of OpenSteerDemo is similar to the ones of games. It runs a main loop, which first recalculates all agent states and then draws the new states to the screen. The recalculation phase is called *update stage*, followed by the so called *graphics stage*. The update stage is again split into two substages. Within the *simulation substage* all agents compute their steering vectors, but do not change their state. These changes are carried out in a second substage called *modification substage*. See figure 5.4 for a graphical overview of the architecture.

To effectively port the scenario to the GPU, the performance of the already existing version must be exposed. The version used for the performance exposure is based on a version by Knafla and Leopold [KLar] in which only minor modifications were done as part of this thesis. Given below the hardware for all following measurements:

- CPU: AMD Athlon 64 3700+
- GPU: GeForce 8800 GTS - 640 MB
- RAM: 3 GB

The GPU offers a total number of 12 multiprocessor, each offering 8 processors. This results in a total of 96 processors. The GPU is running at 500 MHz, whereas the processors itself are running at 1200 MHz. The CPU is a single core CPU running at 2200 MHz.

Figure 5.5 gives an overview of how CPU cycles are spent when simulating the Boids scenario. The neighbor search is the performance bottleneck, with about 82% of the

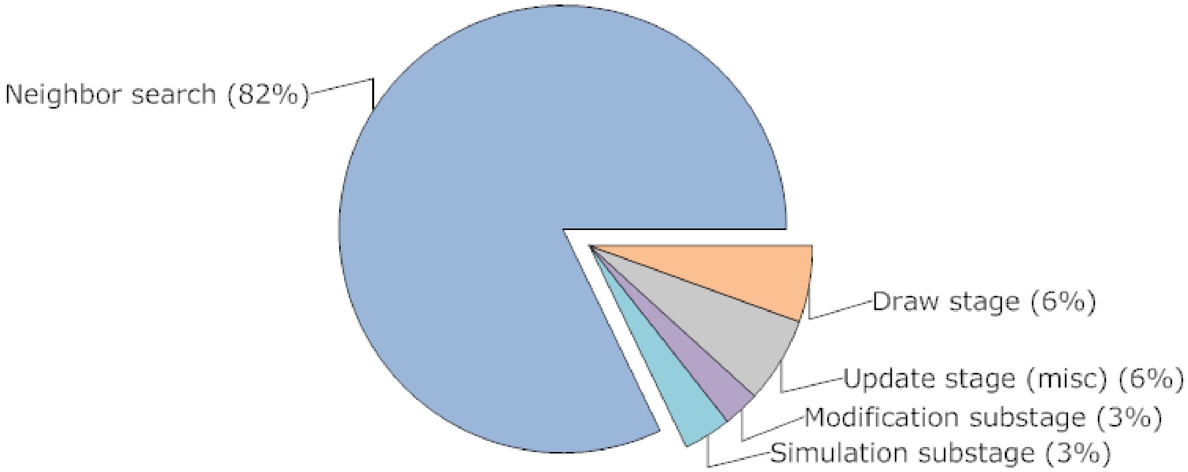

Figure 5.5: Overview of how much time the different stages require.

used CPU cycles. The neighbor search is done once for every calculation of the resulting steering vector and not once for every basic steering behavior, as it was described in section 5.2. The calculation of the steering vector (simulation substage) or any other work requires less than 20% of the overall performance. The neighbor search for all agents is a $O(n^2)$ problem, since the $O(n)$ neighbor search algorithm described in section 5.2.1 is executed to for all agents.

A technique called *think frequency* reduces the amount of time required by the all agents neighbor search, even though it is not necessarily implemented to achieve this. Frame rates of about 30 to 60 frames per second (fps) are required to produce a smooth animation, but the steering vector can be recalculated at a slower rate [Rey06]. A think frequency of $\frac{1}{10}$ refers to that the simulation substage of a specific agent is executed every ten simulation time steps. In one simulation time step only $\frac{1}{10}$th of the agents execute the simulation substage. Reynolds [Rey06] calls this technique *skipThink*.

Figure 5.6 shows how the simulation scales with the addition of new agents. The performance is measured in the number of times the update stage is executed in one second. With the addition of new agents, the performance without think frequency suffers greatly due to the $O(n^2)$ all agent neighbor search problem. The performance with think frequency does not suffer that much from the $O(n^2)$ nature of the problem, since the think frequency factor is introduced as a constant reducing the number of times the neighbor search algorithm is executed. This does not change the complexity of the simulation substage, but has an obvious impact on the performance.

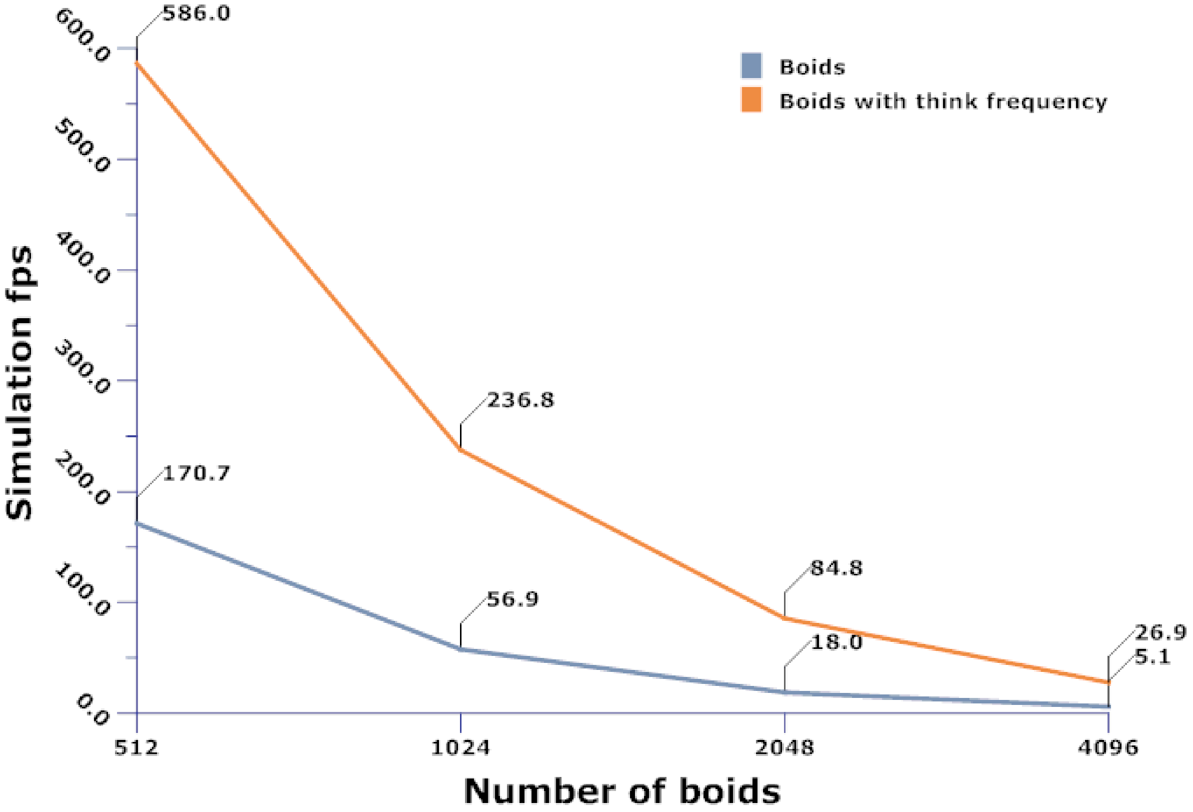

Figure 5.6: Performance measurement of the original OpenSteer Boids simulation with and without think frequency

CHAPTER 6

GPU Steering

This chapter gives an overview of the development of the GPU based Boids scenario and its performance. The simulation uses the CuPP framework to integrate CUDA. The code running on the device is based on the CPU version of OpenSteer, which was introduced in the previous chapter.

At first we describe how the simulation substages are mapped to different kernels and how the agents are mapped to the threads. This is followed by an overview of the development process that led to the simulation itself. At the end different performance aspects are discussed in detail including if the branching issue discussed in section 2.3 is a major problem.

6.1 Agent by agent

The first step to be done when parallelizing an application is to identify independent parts. Considering the concept discussed in section 5.2 and 5.3, this is fairly simple. On the one side there are the two substages, which depend on one another. The simulation substage has to be completed before the modification substage can be started, since the simulation substage requires the current agent state and the modification substage updates this state. But in both substages, the calculation done for one agent is independent from that of the others. Base on these two facts a fairly simple parallelization approach arises:

- Synchronization is required between the two substages. Therefore the two substages have to be implemented as two separated kernels, since it is impossible to synchronize all threads within a kernel as said in section 2.2.

- The calculation of one agent is independent of the calculations of the other agents, but the instructions are the same. With mapping one agent to one thread, it is almost a perfect match to the CUDA programming model.

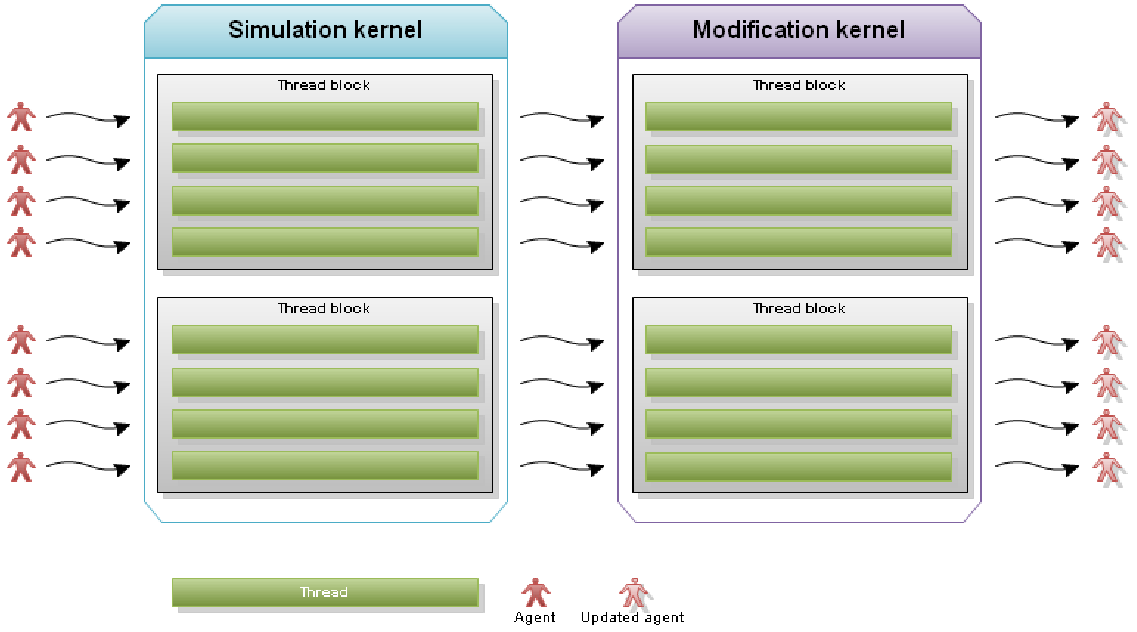
Simulation kernel
Modification kernel
Thread block
Thread block
Thread block
Thread block
Thread
Agent
Updated agent

Figure 6.1: The mapping of agents to threads/blocks.

Version	substage executed by the device		
	update		modification
	neighbor search	steering vector calculation	
1	✓		
2	✓		
3	✓	✓	
4	✓	✓	
5	✓	✓	✓

Table 6.1: Overview of the different development versions

This approach does not take the shared memory into account, but it still offers a good possibility to use it. Every agent looks at all other agents to find its neighbors, so the data requirement for all threads is uniform – at least during the neighbor search. This uniform data requirement makes the usage of the shared memory both efficient and easy. Details on shared memory usage are discussed in section 6.2.1. See figure 6.1 for a graphical overview of this first approach.

6.2 Step by step

The development of the simulation has been divided into three steps. First only the neighbor search is executed by the device, whereas later the complete simulation substage and finally the complete update stage are executed by the device. Figure 6.2 gives an

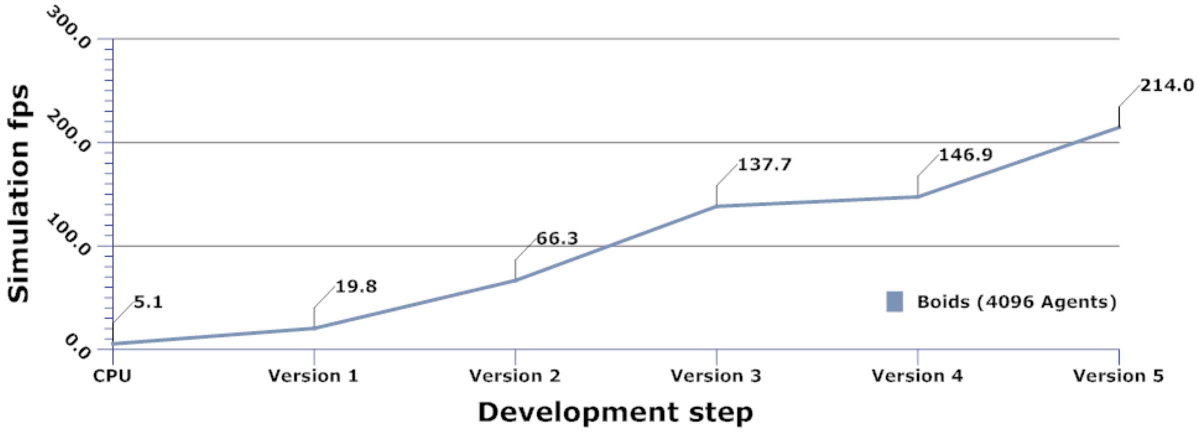

Figure 6.2: Simulation frames per second at every development step when simulating 4096 agents.

overview of how the performance increased during the development stages.

6.2.1 Neighbor search

Considering the massive amount of runtime required by the neighbor search – 82% – porting it to CUDA, was the first task to be done. The algorithm itself remains unchanged. The first program code executed by the device was hardly more than a copy and paste work of the code running on the CPU. The only thing needed to be done was extracting the positions of all agents, store them in a `cupp::vector` and call the neighbor search kernel. The kernel call is done by using `cupp::kernel`, so all memory management happens automatically.

Listing 6.1: Calling the neighbor search kernel

```
cupp::vector <Vec3> positions
cupp::vector <int>  results;

// copy the positions of all agents in position_data
for (int i=0; i<flock.size(); ++i) {
    positions.push_back(flock[i]);
}

// allocate memory to store the neighbors
result.resize(7 * flock.size());

find_neighbors_kernel (device_hdl, positions, search_radius,
   results);

// result now contains the indexes of the neighbors found
// ... continue with the old CPU simulation
```

`positions` is passed as a const reference to the kernel, whereas result is passed as a non-const reference. This results in optimized memory transfer and no copy constructor calls. `positions` could be passed by value, but this would create a copy of all its containing elements. `result` must be passed as a reference, since the kernel modifies it and uses it to store the indexes of the found neighbors. Changes done by the kernel are only reflected back, when an argument is passed as a reference.

This simple approach already improved performance by a factor of about 3.9 compared to the CPU version. This performance improvement is surprising, since the program does not utilize any other memory than global and local memory. But the performance improvement is the result of the massive amount of processing power available on the device. We refer to this version as *version 1*.

The next step was to improve the usage of the GPU memory hierarchy to reduce the number of device memory accesses, since accessing device memory is expensive (see table 2.2). This was done by using shared memory as a cache for position data. The kernel was modified as shown below; no modifications to the host code were required.

Listing 6.2: Neighbor search with shared memory usage

```
 1 // allocate shared memory to cache positions
 2 __shared__ Vec3 s_positions[threads_per_block];
 3
 4 // iterator through all agents
 5 for (int base=0; base < number_of_agents; base+=
      threads_per_block) {
 6     // read data from device memory to shared memory
 7     // every threads reads one element
 8     s_positions[threadIdx.x] = positions[base + threadIdx.x];
 9
10     // wait until all reads are completed
11     __syncthreads();
12
13     for (int i=0; i<threads_per_block; ++i {
14         // execute the neighbor search on
15         // the data stored in s_positions
16         // see listing 6.3
17     }
18
19     // wait until all threads have finished
20     // the neighbor search on s_position
21     __syncthreads();
22 }
```

As said in sections 3.1.3 and 3.1.4 `threadIdx.x` is a block local thread index and `__syncthreads()` synchronizes all threads within a block. `threads_per_block` is the number of threads per block. The kernel shown in listing 6.2 is executed with one thread per agent, whereas the number of agents has to be a multiply of `threads_per_block`. In line 8 all threads issue a read from global memory to store a position of an agent into an array allocated in shared memory, which is used as a cache. This technique reduces the number of values read from global memory for every block from $threads\ per\ block * number\ of\ agents$ to $number\ of\ agents$. By that the performance was improved to almost a factor of 3.3 compared to version 1, which does not use shared memory, and about 12.9 times the performance of the original CPU solution. We refer to this version as *version 2*.

6.2.2 Simulation substage

Reprofiling the modified OpenSteerDemo showed that the neighbor search now occupies about 2% of the runtime and is therefore no longer the performance bottleneck. As a next step, the rest of the simulation substage is executed by the device. The code is again based on the CPU code, but is adapted to the GPU for higher performance. The additional data, which is required to execute the simulation substage, is extracted and transferred the same way as described in section 6.2.1.

We present two different versions executing the complete substage at the device. *Version 3* uses thread local memory to cache values calculated during the neighbor search e.g. the distance of the simulated agent to its neighbors. This version turned out to be slower compared to the next version. *Version 4* does not cache elements in local memory, but recalculates them when the values are required. The performance improvement results from the fact that these values have been stored in slow device memory, which is a valid way of implementing thread local memory (see table 2.1). Version 3 improved performance by a factor of 27 of the CPU version. Version 4 achieved even factor of 28.8 compared to the CPU version and thereby more than doubled the performance compared to version 2.

6.2.3 Modification substage

With the modification substage being the last part of the update stage still running on the host, we decided to move this one to the device as well. Shared memory is used in this stage, but not as it may have been intended to be, since the calculations of one thread do not use data required by another thread. Shared memory has been used as an extension to thread local memory, so local variables are not stored in device memory. This has to be done manually by the developer, as the variables stored in device memory can only be identified by reading the compiler generated assembler code (known as PTX code). Details on how variables stored in device memory are identified can be found in Parallel Thread Execution ISA [Cor07d] in section 5.

The performance has been further improved to about 42 times of the CPU version. Parts of this improvement are due to the optimized memory management done by the CuPP framework, by which only the required information to draw the agents[1] is moved from the device to the host memory. All other data stays on the device. We refer to this version as *version 5*.

---

[1] A 4x4 matrix containing 16 float values.

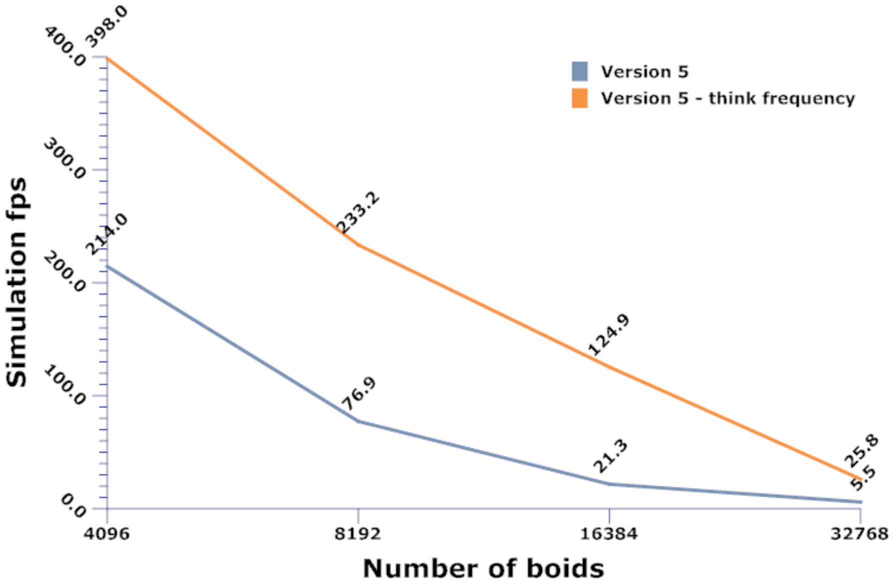

Figure 6.3: Performance of the version 5 at different numbers of simulated agents

6.3 Performance

Figure 6.3 demonstrates the scalability of version 5 with the addition of new agents. The variant without think frequency shows a similar behavior then the CPU version shown in figure 5.6, the $O(n^2)$ nature of the problem is clearly visible.

On the other side the variant using think frequency scales linear up to 16384 agents (the performance is less then halved, when the number of agents is doubled) and looses a high amount of performance when the number of agents is doubled to 32768 (the performance is reduced by a factor of about 4.8 when the number of agents is doubled). The linear loss of performance is because the instructions with highest cost – accessing global memory, which is mostly used at the neighbor search – are not only reduced by the factor of the think frequency, but also are executed once only per block, as said in section 6.2.1. Device memory is of course still accessed $O(n^2)$ times, but these two factors are enough to practically prevent a quadrate performance loss when increasing the number of agents up to 16384. The high performance loss when increasing the number of agents further is not only based on the complexity of the neighbor search, but also on the number of times a warp diverges increases, as discussed in the next section.

6.3.1 SIMD branching issue

As described in section 2.3, a branch that does not evaluate equal on all threads within the same warp reduces the performance. The used code does not have any special optimization to reduce this problem, but the performance still looks good – even when compared with similar work, e.g. the N-body system implemented by NVIDIA [NHP07], which does not suffer of divergent warps.

The modification kernel uses no loops and only a small amount of branches – one used to prevent calculation not needed in the first simulation time step and others to prevent division through zero – and is therefore not the important factor considering the SIMD branching issue.

The simulation kernel on the other side uses both loops and branches, which can roughly be divided into two groups. The one used inside the neighbor search and the one needed, because the number of neighbors found may vary on a per agent basis. The second group has no practical relevance, since almost all agents find the maximum number of neighbors in the simulated scenario. The threads, which found fewer agents, execute the same amount of instructions as the agents that found the maximum number of agents.

By contrast the first group may be a real problem. The neighbor search code can be viewed in listing 6.2 and 6.3. Both loop conditions shown in listing 6.2 are based on variables that evaluate identically over all threads, so the instruction flow of a warp cannot diverge.

Listing 6.3: Neighbor search done in shared memory

```
for (int i=0; i<threads_per_block; ++i) {
    const Vec3 offset = position - s_positions[i];
    const float d2 = offset.lengthSquared();
    const float r2 = search_radius * search_radius;
    const int global_index = base + i;

    if (d2 < r2 && global_index != my_index) {
        // neighbor found
        if (neighbors_found < 7) {
            // less then 7 neighbors found
            // just add new found neighbor to neighborhood
            ++neighbors_found;
        } else {
            // already found 7 neighbors
            // loop through all 7 neighbors
            // and find farthes 'old' neighbor
            // replace with new if 'new' found one is closer
        }
    }
}
```

As far, as the two `if`/`else` statements in listing 6.3 are considered, there is no guarantee that they evaluate identical over all threads. There is no order within the way the agents are stored in the vector `positions` respectively `s_positions`, so it is expected that only a single thread executes a branch most of the time and possible performance is lost. The lost performance increases with the amount of added agents, since with more agents the number of agents within the neighbor search radius increases and therefore the times the warp diverges. How much performance is lost in this case, can only be

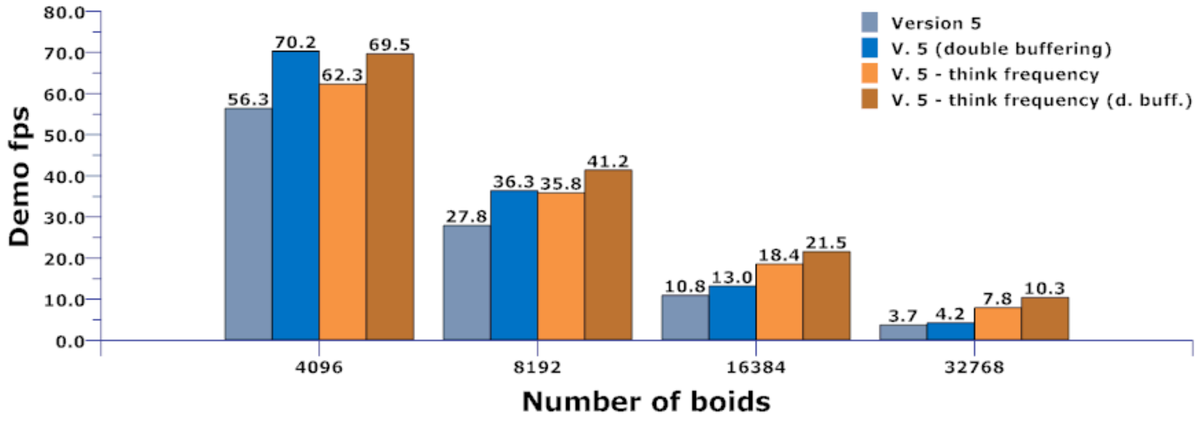

Figure 6.4: Performance improvement by using double buffering

speculated, as no profiling tool is available offering this information. The device handles divergent warps rather efficient and comparing the performance with NVIDIAs N-body system, which does not suffer from any SIMD branching issues, therefore the amount of lost performance is expected not to be a serious issue.

6.3.2 Double buffering

This chapter discusses an optimization, which does not have any impact on the number of times the update stage is executed in one second, but the overall performance of the demo. Since the complete update stage is running on the device and kernel calls are asynchronous function calls (see section 2.2) it is possible to overlap the draw and the update stage. When the host is executing the draw stage for simulation step n, simulation step n+1 can already be calculated by the device. We call this technique *double buffering*.

Using the CuPP framework, the implementation was fairly easy. We only had to add an additional CuPP vector, so we have two vectors available to store the data required to draw the agents. At simulation step n the first vector is used to store the results and the second vector is used to draw step n-1, whereas at step n+1 the second vector is used to store the results and the first one is used at the draw stage.

The performance improvement is between 12% and 32% compared to the demo not using double buffering, as it can be seen in figure 6.4. The performance improvement is at the highest level, when device and host finish their work at the same time, so the components do not have to wait for each other. Experiments show that this is the case when simulating 8192 agents without think frequency or 32768 agents with think frequency. When simulating 4096 agents, the update stage is executed so fast, that is does not matter if think frequency is used or not – the performance of 4096 agents is only limited by the draw stage.

Chapter 7

Conclusion

In this thesis, we have developed the CuPP framework to ease the integration of CUDA into C++ applications and demonstrated its abilities with enhancing the Boids scenario of OpenSteer. The usability of CUDA in combination with the CuPP framework is superior to solely using CUDA – at least in the scenario considered. The CuPP framework automatically supports the developer at all memory management operations, e.g. transferring data from the device to the host or allocating enough device memory. But this higher level of abstraction also comes at the cost of lost direct control of the tasks done automatically. The framework obviously cannot outperform an experienced developer, as the developer may simply have more knowledge of the application, which may lead to a more efficient memory management or at least to specially designed code, which should outperform the general code of CuPP.

As far as we experienced at the development of the new OpenSteer plugin, we would have used exactly the same memory management as it was automatically done by the CuPP framework. Also, the high abstraction level of the CuPP framework offered an easy way to implement the double buffering.

But even small mistakes can harm performance badly. Using a non-const reference instead of a const one harms performance since additional memory transfers are done. Passing a vector by value results in a high amount of copy constructor calls, because all elements of the vector must be copied. These problems occur as memory transfers and copy constructor calls are done implicit with calling the kernel and only small details define the behavior. But this problem is strongly related to C++, since the same problems occur there as well. Therefore every C++ developer should be aware of these problems and should be capable of choosing the correct possibility of passing an object to a kernel.

Another problem of CuPP is based on the high amount of template metaprogramming usage. Since template metaprogramming is carried out at compile time, the compile time increases. In the case of the Boids scenario, the compile time was more than double with the introduction of CuPP – from about 3.1 seconds to 7.3 seconds. Due to restrictions of the standard C++ language, e.g. missing reflections support, the usage of template metaprogramming is inevitable to analyze the kernel definitions, so there is currently no way to avoid this problem.

Future work on the CuPP framework could refer to currently missing CUDA functionality, like support for texture or constant memory. Both memory types could be used to enhance the CuPP vector, e.g. if it is known that the vector is passed as a const reference to a kernel, texture of constant memory could automatically be used to offer even better performance. Also the CuPP framework currently misses support for multiple devices in one thread.

Regarding the example application, it does not only offer an almost perfect possibility to use the CuPP vector, but could also benefit from the CuPP type transformations, e.g. spatial data structures could improve the neighbor search performance. Data structures must be constructed at the host, due to the low arithmetic intensity of such a process, and then be transferred to the GPU. With CuPP it would be easy to use two different data representations, the host data structure could be designed for fast construction, whereas the device data structure could be designed for fast memory transfer to device memory and fast neighborhood lookup.

REFERENCES

[ADS^{+}07] Adobe Systems Inc, David Abrahams, Steve Cleary, Beman Dawes, Aleksey Gurtovoy, Howard Hinnant, Jesse Jones, Mat Marcus, Itay Maman, John Maddock, Thorsten Ottosen, Robert Ramey, and Jeremy Siek. Boost type traits library. http://www.boost.org/doc/html/boost_typetraits.html, 2007.

[Bre07] Jens Breitbart. CuPP website. http://cupp.gpuified.de, 2007.

[CGL98] Marshall P. Cline, Mike Girou, and Greg Lomow. *C++ FAQs*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[Cor07a] NVIDIA Corporation. CUDA C++ Integration sample. http://developer.download.nvidia.com/compute/cuda/1_1/Website/samples.html, 2007.

[Cor07b] NVIDIA Corporation. CUDA matrix multiplication sample. http://developer.download.nvidia.com/compute/cuda/1_1/Website/samples.html, 2007.

[Cor07c] NVIDIA Corporation. NVIDIA CUDA Compute Unified Device Architecture Programming Guide Version 1.0, 2007.

[Cor07d] NVIDIA Corporation. PTX: Parallel Thread Execution ISA Version 1.0, 2007.

[Dox07] Doxygen website. http://sourceforge.net/projects/doxygen/, 2007.

[Fol07] Folding@Home website. http://folding.stanford.edu, 2007.

[GBD07] Greg Colvin, Beman Dawes, and Darin Adler. Boost smart pointers library. http://www.boost.org/libs/smart_ptr/smart_ptr.htm, 2007.

[KLar] Bjoern Knafla and Claudia Leopold. Parallelizing a Real-Time Steering Simulation for Computer Games with OpenMP. ParCo2007, to appear.

[Mey95] Scott Meyers. *More effective C++: 35 new ways to improve your programs and designs*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[Mey01] Scott Meyers. *Effective STL: 50 specific ways to improve your use of the standard template library.* Addison-Wesley Longman Ltd., Essex, UK, UK, 2001.

[NHP07] Lars Nyland, Mark Harris, and Jan Prins. Fast n-body simulation with CUDA. In Hubert Nguyen, editor, *GPU Gems 3*, chapter 31, pages 677–695. Addison Wesley, 2007.

[OLG$^+$05] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. In *Eurographics 2005, State of the Art Reports*, pages 21–51, August 2005.

[Ope05] OpenMP Application Program Interface, Mai 2005. Version 2.5.

[Rey99] Craig W. Reynolds. Steering Behaviors For Autonomous Characters. In *Proc. Game Developer Conference*, pages 763–782, 1999.

[Rey06] Craig W. Reynolds. Big Fast Crowds on PS3. In *Proc. ACM SIGGRAPH Symp. on Videogames*, 2006.

[Str00] Bjarne Stroustrup. *The C++ Programming Language, Fourth Edition.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.