Massively Parallel Contact Simulation on Graphics Hardware using NVIDIA CUDA

Bachelor's Thesis

Jens-Fabian Goetzmann

Supervisor: **Prof. Dr. Elmar Schömer**
Institute for Computer Science
Universität Mainz

September 18, 2007

Hiermit versichere ich, Jens-Fabian Goetzmann, dass ich die vorliegende Bachelorarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

I, Jens-Fabian Goetzmann, hereby affirm that I have independently composed this bachelor's thesis and that I have used no sources or aids other than those mentioned.

Jens-Fabian Goetzmann

Mainz, September 18, 2007

Contents

Bibliography

[ALS2006] B. Addis, M. Locatelli and F. Schoen: *Efficiently packing unequal disks in a circle: A computational approach which exploits the continuous and combinatorial structure of the problem*. Optimization Online 1343, 2006.
http://www.optimization-online.org/DB_HTML/2006/03/1343.html.

[BSW2007] T. Baumann, E. Schömer and K. Werth: *Solving geometric packing problems based on physics simulation*. Unpublished.

[DOW2006] J. Doweck: *Inside Intel Core Microarchitecture and Smart Memory Access*. Intel Corporation, 2006.
http://download.intel.com/technology/architecture/sma.pdf.

[EIS2003] F. Eisenbrand, S. Funke, J. Reichel and E. Schömer: *Packing a Trunk*. In 11^{th} European Symposium on Algorithms, ESA'03, LNCS 2832, pp. 617–629, 2003.
http://citeseer.ist.psu.edu/673482.html

[EIS2005] F. Eisenbrand, S. Funke, A. Karrenbauer, J. Reichel and E. Schömer: *Packing a Trunk - now with a Twist!*. In ACM Symposium on Solid and Physical Modeling, SPM'05, pp. 197–206, 2005.
http://www.staff.uni-mainz.de/schoemer/publications/SPM05.pdf

[GOT1996] S. Gottschalk, M. C. Lin and D. Manocha: *OBB-Tree: A Hierarchical Structure for Rapid Interference Detection*, In Computer Graphics (SIGGRAPH '96 Proceedings), pp. 171–180, 1996.
http://citeseer.ist.psu.edu/gottschalk96obbtree.html

[GRE2007] S. Green: *NVIDIA CUDA FAQ Version 1.0*. NVIDIA Corporation, 2007.
http://forums.nvidia.com/index.php?showtopic=36286.

[LEC1999] P. L'Ecuyer: *Tables of Linear Congruential Generators of Different Sizes and Good Lattice Structure*. In Mathematics of Computation 68, 225, pp. 249–260, 1999.
http://citeseer.ist.psu.edu/132363.html.

[NVI2007] NVIDIA Corporation: *NVIDIA CUDA Compute Unified Device Architecture, Programming Guide*. NVIDIA Corporation, 2007.
http://developer.download.nvidia.com/compute/cuda/1_0/NVIDIA_CUDA_Programming_Guide_1.0.pdf.

[POD2007] V. Podlozhnyuk: *Parallel Mersenne Twister*. NVIDIA Corporation, 2007.
http://developer.download.nvidia.com/compute/cuda/sdk/website/projects/MersenneTwister/doc/MersenneTwister.pdf

[ZIM2005] A. Zimmermann: *Al Zimmermann's Circle Packing Contest*. 2005.
http://www.recmath.org/contest/CirclePacking/index.php.

Introduction

Modern graphics hardware provides enormous computing resources. Driven especially by the game market's demand for ever more realistic 3-D graphics, current graphics adapters such as the NVIDIA GeForce 8 series feature theoretical computing performances that are a multiple higher than those of current CPUs, with an even increasing gap.

Until now, these possibilities were hardly utilizable for other purposes than generating real-time 3-D graphics. Graphics hardware was not freely programmable like the CPU, but only through special *shader programs* that had a number of limitations regarding reading and writing data and control over the execution. In June 2007, graphics hardware manufacturer NVIDIA released the CUDA SDK (Software Development Kit) making it possible to freely program (with some limitations) the GPU. The computation power that a GPU offers is always *massively parallel*, i.e. in order to fully take advantage of the resources, a program has to be run in several hundred or thousand threads in parallel.

In this thesis, I will first evaluate and discuss the NVIDIA CUDA SDK in general (in chapters 1 and 2). Afterwards, I will describe a physical contact simulation algorithm for packing problems (in chapter 3) and outline its massively parallel implementation using the NVIDIA CUDA SDK for the specific packing problems of circle packing (chapter 4) and trunk packing (chapter 5). As a conclusion, I will discuss various aspects of the CUDA SDK and the overall advantages and disadvantages of using CUDA to approach non-graphic problems (chapter 6).

The evaluation of using NVIDIA CUDA for massively parallel contact simulation has lead to ambivalent results: For the problem of circle packing, an efficient implementation is presented in this thesis that performs about 50 times as fast as a comparable serial implementation. For the complex problem of trunk packing, the implementation presented in this thesis performs only slightly better than a comparable serial implementation.

Chapter 1

An Introduction to NVIDIA CUDA

1.1 What is NVIDIA CUDA?

NVIDIA CUDA[1] (Compute Unified Device Architecture) is an SDK (Software Development Kit) released by graphics hardware manufacturer NVIDIA with the purpose of making it possible for programmers to accelerate general-purpose computations by using the computational resources available to modern GPUs (Graphic Processing Units). The current 1.0 release was announced in June 2007. The central tool of CUDA is a specialized C Compiler that is able to separate a given file written in C with special extensions into a "host" C file and a binary "device" kernel that contains all functions that are to be run on the graphics hardware. The portions of the program that should be run on the CPU can be written in C/C++ or any other language able to be linked to the object files generated by the system's default C Compiler.

CUDA features a so called "device emulation" mode that does not utilize the GPU for computations but instead emulates the behaviour of the GPU on the CPU, and a hardware-accelerated mode. The latter is only available on machines equipped with the newest generation NVIDIA graphics hardware.[2]

1.1.1 Benefits of CUDA

CUDA is the first SDK to expose the resources on the graphics hardware for general purposes. Some programmers used shader programs to utilize the computing power available to the GPU to accelerate general-purpose applications, but this had several disadvantages: Shader languages are specialized for writing shaders, not general-purpose programming languages, so the programmer had to use tricks in order to achieve the results wanted. Using a shader language instead of a popular language such as C also resulted in a higher learning curve. Shader programs also had very limited abilities to store their results. With CUDA it is possible to perform arbitrary *gather* (read) and *scatter* (write) operations in the graphics memory.

CUDA is easy to learn for anyone already familiar with C-style languages. There are only a few special syntactic constructions in the .cu source files that are easy to understand. Additonally one can easily use existing IDEs to develop CUDA source code and use their advantages such as syntax highlighting or automatic code completion.

The computing resources available on current generation graphics hardware are enormous. NVIDIA's current flagship GPUs offer a theoretical number of 346 GFLOPS[3] (billions of floating point operations

[1] Called "CUDA" in the following for the sake of simplicity.
[2] (Consumer) GeForce 7 or 8 series, (Workstation) Quadro FX 4600 and 5600 or one of the specialized Tesla graphics processors.
[3] According to [GRE2007].

per second), compared to only 48 GFLOPS[4] on a Intel Core 2 Duo running at 3.0 GHz. The gap between GPU and CPU computing power is rather increasing.

CUDA functions can be called *asynchronously*, i.e. the CPU can carry out other computational tasks while a parallel computation is running on the GPU.

1.1.2 Terminology Used in CUDA

The CUDA documentation and code uses a special terminology that has to be understood in order to use CUDA. The following list contains a few of the terms that are important for this thesis.

Host: The *host* is the CPU of the machine running CUDA code. It is of course in no way limited by CUDA and can perform all kinds of calculations in the same way as it would if it was not running a program utilizing CUDA.

Device: The *device* is the graphics hardware. The graphics hardware features its own set of processors[5] and its own memory. Both of them could not be used for general purposes without tricks[6] before the advent of CUDA. With CUDA, it is possible to write special functions in C (with some extensions) that can be loaded onto the device and then be executed parallely.

NVCC: The NVIDIA C Compiler *nvcc* is utilized to separate host code from device code in .cu files containing CUDA-enabled C Code. It then compiles the device code to a binary *device kernel* which is in turn embedded into a C file containing the stripped host code. Finally, the resulting file is compiled using the system's standard C Compiler[7] into a standard object file that can be normally linked to other object files.

Thread: The concept of a *thread* is not special to CUDA programming but is a terminology related to all kinds of parallel computation: A thread is an independent string of computations. More than one thread may run concurrently. Threads in CUDA have several limitations, however, see section 1.2 for a detailed description.

SIMD: SIMD is an acronym for "single instruction, multiple data" and it is the way in which the processors in the GPU work. More on the SIMD processors is detailed in section 1.2.

Kernel: A compiled function that should be executed on the GPU is called a *kernel*. The CUDA SDK automatically creates these kernels and takes care of loading them to the device before execution of the function.

1.2 Understanding the CUDA Programming Model

In order to understand the CUDA programming model, one must have a detailed understanding of the computing resources available on a graphics adapter.

[4] According to [DOW2006].
[5] More correctly: SIMD-Multiprocessors.
[6] I.e., writing shader programs.
[7] E.g. the GCC or the Microsoft C compiler.

1.2.1 Multiprocessors

The graphics adapter has a number (between 2 and 16 for normal graphics cards, up to 64 for the specialized Tesla graphics processors) of SIMD multiprocessors. The SIMD concept means that each of these processors may parallely execute the same operation on a number of different operands.[8]

The current generation of GPUs supports only single precision floating point operations. Future hardware generations will also support double precision floating point arithmetic.[9]

1.2.2 Thread Batching

Threads in CUDA are organized in a one-, two- or three-dimensional *grid* of thread *blocks*. Each of these blocks, in turn, consists of the same number of threads, organized as a one- or two-dimensional array. All threads of a single block are executed on the same multiprocessor via parallel execution and time slicing and share the resources of the multiprocessor, such as registers and local memory. The maximum number of threads per block is 512, the suggested minimum is 64, recommended is a number divisible by 32 and at least 192. The maximum dimension of the grid is 65535 in each dimension, the suggested minimum is a grid of 100 blocks for current hardware. If one wants to ensure to take full advantage of the computing capabilities of future generations of graphics hardware, [NVI2007] recommends at least 1000 blocks. All in all one can easily compute that in order to use the GPU to full capacity, one has to have at least about 20000 threads.[10]

Threads of the same block can be synchronized (unlike threads in different blocks) and also have the ability of using a special shared memory area which is located directly on the multiprocessor and therefore much faster to access. Deciding how to group threads into blocks is one of the most challenging tasks the programmer has to face when designing a CUDA enabled program.

Once started, a parallel execution of a kernel cannot be changed in size (except for some threads `return`ing earlier than others). Particularly, no further parallel threads can be started from within device code. There is also no possibility to run more than one kernel at the same time.

1.2.3 Memory

There is a lot of different types of memory available to functions running on the hardware. Each of them has specific advantages and disadvantages.

Registers: Each multiprocessor features 8192 32-bit registers which may store a single precision floating point or an integer value. These registers are divided among the threads of a block. Registers are the fastest memory available. They can be read and written to only from the thread they belong to. They provide no support for indirect addressing or pointer arithmetic.

Local memory: Larger arrays or structures or variables accessed indirectly or using pointer arithmetic are stored in the local memory area directly on the multiprocessor. Local memory can – like registers – only be read and written from a single thread, but it is much slower.

[8] The number of executions parallely executed is 8 in the current hardware generations. However, the so called "warp size", i.e. the number of threads that have to execute the same operation without leading to performance losses, is 32 according to [NVI2007]. [NVI2007] states no reasons for this discrepancy.

[9] [GRE2007] states that "NVIDIA GPUs supporting double precision in hardware will become available in late 2007".

[10] Thus *massively* parallel.

Shared memory: Shared memory is a special type of memory residing on the multiprocessor that may be read and written to from all threads of a thread block, but not from the host or from threads in other blocks. Because it is located directly on the multiprocessor, it is much faster than global memory. There are 16 KB of shared memory available on each multiprocessor in the current generation of graphics hardware.

Global memory: Global memory is located in the general RAM of the graphics card. It can contain large amounts of data,[11] which may also be dynamically allocated from the host, and it can be read and written from both functions running on the device and from the host. Global memory is the memory type with the highest access times from device code.

Constant memory: Constant memory can be read and written from the host, device code can only read it. Constant memory is cached on the multiprocessors, thus it is very fast to read. Constant memory can not be dynamically allocated, but provides only a fixed space. Its main use is therefore just what it is called: constants. There are 64 KB of constant memory available on current hardware.

Texture memory: Texture memory can be read and written from the host, device code can only read it. It is cached, just like constant memory. It is also optimized for (2-D) spatially localized access, i.e. when threads access memory locations from neighboring memory locations, access times are better. Data in texture memory may be dynamically allocated. There are actually two types of textures: Textures residing in linear memory, which may contain data blocks of a size up two 2^{27} elements (each of which may in turn contain up to 4 32-bit floating point or integer values), or textures residing in specialized one- or two-dimensional arrays, which may contain up to 2^8 resp. $2^{16} \cdot 2^{15}$ elements. The latter also support certain additional functions such as addressing with normalized floats in the interval of $[0,1)$ or bilinear interpolation between values.

1.2.4 Flow of Instructions

As stated above, CUDA features a SIMD processing model. As a result, deviations within the control flow of threads within a so-called *warp* of 32 threads should be avoided (although they are not forbidden). Deviations may result from any control flow instruction such as `if`, `switch`, `for` or `while` if the contained condition evaluates differently in different threads of the warp. If such a deviation occurs, the threads have to be *serialized* (i.e. executed one after the other) and the advantages of the parallel capabilities of the multiprocessor are lost until the different flows of instructions converge again.

An important limitation of the CUDA SDK is that recursive function calls are not allowed. Of course, recursion can always be simulated using iterations, but the administrative overhead for doing so can be quite high.

1.3 Basic Considerations for Writing a CUDA Program

As discussed in section 1.2, there is a number of limitations and constraints one has to keep in mind when writing a program that should use CUDA. A few of them are discussed in detail in this chapter.

[11] [NVI2007] does not state an upper boundary. The data size is of course limited by the size of the RAM available on the hardware, i.e. 768 MB on current top-of-the-line graphics hardware.

1.3.1 Organization Considerations

The most important question that has to be answered when a specific problem is to be solved by a CUDA program is how it can be broken down into blocks and threads. While the threads of a block can be synchronized and have a fast means of communication (the shared memory), threads from two different blocks can not communicate. They could be executed on different multiprocessors, there is not even a guarantee that they will truly be executed parallely – it could be that one block is already finished before another block has started. Therefore, inter-block communication is generally only possible with a round trip to the CPU, which in turn leads to performance losses.

Some problems can be nicely divided into blocks that do not have to communicate with each other. Many of them come from the field of computer graphics (which is no surprise, obviously) – for example image processing and encoding, where each block could take care of one pixel or of a rectangular area of pixels, such as the $8 \cdot 8$ pixel squares used in JPEG and MPEG formats.

The great majority of computation-intensive problems, however, does not does not have this property. Especially the limitations for the block size are serious constraints: There is no possibility to create blocks larger than 512 threads, so if a sub-problem could only be parallelized to more than 512 threads, one has to separate it to more than one block, thus losing the possibility of communication between the threads. Another problem arises if the problem can be split into partial problems that could be assigned to blocks, but that have a different size: As all blocks have to have the same size, one has to accept some threads doing no useful work, or come up with another division without this problem.

There is no general rule for dividing problems into blocks and threads. A few examples of dividing specific problems are discussed in chapters 3 and 4.

1.3.2 Performance Considerations

1.3.2.1 Memory

Memory accesses from within the device code can easily become a bottleneck. For each variable, be it input data, temporary data or output data, one has to carefully consider which of the different memory areas (see section 1.2.3) it should reside in. Even indirect addressing of arrays, which is normally quite a common practice, might lead to slower code, see listing 1.1 for an example. In this simple example, the compiler itself might perform the optimization of unrolling the loop and making the indirect memory access to a direct memory access, but there are more complicated cases in which human optimization may prevent temporary variables from ending up in local memory instead of a register.

Listing 1.1: Indirect Memory Access

```
// May be bad in CUDA
float afArray[3];
for (int iDimension = 0; iDimension < 3; iDimension++) {
  afArray[iDimension] = someComputation(iDimension);
}

// May be better
float3 oFloat3;
oFloat3.x = someComputation(0);
oFloat3.y = someComputation(1);
oFloat3.z = someComputation(2);
```

But not only the memory location of a variable is important, but for shared and global memory also the pattern of how the threads within a warp (i.e. 32 consecutive threads of a block, starting at a thread

number divisible by 32)[12] access the memory.

It is also worth noting that no memory can be allocated from within code executed on the device. All memory must be allocated prior to the actual parallel execution. This is no serious constraint in the most cases, however.

For *global memory* the best performance is given if memory accesses from the threads of a warp fulfil the following requirements:

- The thread with number `N` should access the element `SomeArray[N]`, so that all memory accesses are *coalesced* and can be accessed with one contiguous memory access.
- The size of an element of SomeArray should be 32, 64 or 128 bit and all elements should have their addresses aligned at 4, 8 or 16 byte[13] in order to be read with one load instruction.[14]
- The base address of `SomeArray` should be divisible by `16 * sizeof(SomeArray[0])`.

The CUDA runtime library provides functions that can allocate memory that fulfills these requirements, see section 1.5.2.

The *shared memory* of a block is divided into 16 32-bit *banks*. The first 32 bit of shared memory belong to the first bank, the second 32 bit belong to the second bank and so on. The 17^{th} 32 bit belong to the first bank again. Memory accesses may cause bank conflicts if more than one thread of a *half-warp*[15] accesses the same bank – with the only exception of all threads of the half-warp accessing the same bank. Bank conflicts result in a significant performance loss and thus should be avoided.

The descriptions given in this chapter are not very detailed, refer to [NVI2007], section 5.1.2 for a more in-depth discussion.

1.3.2.2 Flow of Instructions

Like already mentioned in section 1.2.4, deviations within the control flow of threads belonging to the same warp should be avoided. When a problem is to be solved using CUDA, one should carefully write the code in a way that minimizes possible deviations: Statements that control the flow such as **`if`** do not have to be avoided, but should evaluate the same at least in all threads of a warp, if possible.

1.3.2.3 Execution Time

Another problem that might arise is the limitation of the execution time of kernels. Although this is not documented in the manual, the discussions in the NVIDIA forum[16] clearly point out that – no matter what operating system is used – the execution time of kernels is limited by the operating system to a time of about 5 - 10 seconds, if a kernel run takes longer it will fail with an *unspecified launch failure* (see section 1.5.4). This means that large calculations always have to return to the CPU after at least 5 seconds. Some computations do not have a problem with that and can easily go on after the next call to the parallely executed function, this is especially true for all iterative processes. Some other processes might have to be artificially split in order to stay below the time limit – this might cause a loss of performance when compared to a "native" approach without artificial splits.

[12] For the current hardware generation this is really only the case for a half-warp, i.e. 16 threads, but [NVI2007] states that future generations will have these memory access requirements for whole warps as well.

[13] Compilers generally have special keywords to guarantee a specific alignment, CUDA adds the macro `__align(n)__` that can be used when defining custom datatypes and automatically translates to the compiler-specific alignment keyword.

[14] The CUDA predefined data types such as **`float4`** automatically fulfil this requirement, see section 1.5.1.

[15] 16 consecutive threads, starting at a thread number divisible by 16.

[16] http://forums.nvidia.com/index.php?showtopic=36004.

1.4 CUDA Extensions to C

As already mentioned several times, the central tool of CUDA is NVCC, a special compiler that compiles .cu files containing C code with some extensions. This section describes the most important extensions. A more in-depth description of all the extensions is found in [NVI2007].

CUDA features two API levels: The lower-level *driver API* and the higher-level *runtime API*. The driver API allows for more control over different aspects of the kernel execution, but is also more complicated. The runtime API provides a convenient means of generating, loading and executing kernels, and is the recommended choice for most projects. Only the latter is discussed in this thesis.

1.4.1 Function Type Qualifiers

CUDA separates functions into three categories: The first category are the *host* functions, i.e. functions running normally on the CPU and callable from any other host function or from functions outside the current module. They follow the same syntax as normal C functions. They may be optionally denoted with the keyword **__host__** in front of their definition / declaration, see listing 1.2.

Listing 1.2: Host Functions

```
// A host function
int someHostFunction1(int iPar1, int iPar2) { ... }

// Another host function
__host__ void someHostFunction2(char *pcPar) { ... }
```

The second category are the *global* functions. They are the functions that are executed parallely on the device. Their definition / declaration has to have the keyword **__global__** in front of the function definition, and they must have a **void** return type, as shown in listing 1.3. They can only be executed from the host and the caller must specify an execution configuration as detailed in section 1.4.3.

Global functions are executed *asynchronously*, i.e. the call will return instantly, although the computation is not yet finished.[17] If one wants to get sure that all threads of the global function are finished, one should call the function `cudaThreadSynchronize()` (see section 1.5.5).

Listing 1.3: Global Functions

```
// A global function, that is called from the host and executed parallely
__global__ void globalFunction(int iPar1, float4 *poData) { ... }
```

The last category are *device* functions. They are executed on the device and callable from the device only. As already noted, they can not be called recursively. Typical examples for device functions are small helper functions, such as vector addition or matrix / vector multiplication. Device functions are denoted with the keyword **__device__** in front of their definition / declaration, see listing 1.4.

Listing 1.4: Device Functions

```
// A device function with a return type
__device__ float3 deviceFunction(float3 oPar) { ... }

// A device function using references
__device__ void deviceFunction2(float3 &oOp1, float3 &oOpResult) { ... }
```

[17] Obviously, this is the cause for the need of a **void** return type.

The keywords **__device__** and **__host__** may also be used together resulting in a function compiled for both host and device use.[18] Especially mathematical helper functions such as those mentioned above are useful if the same code can be used for both device and host.

1.4.2 Memory Type Qualifiers

1.4.2.1 Registers and Local Memory

Any variable normally declared in a global or device function will automatically reside in a register or in local memory, depending on the size – large arrays will be assigned to local memory – and the means of access – indirectly addressed arrays or data accessed using pointer arithmetic will also be assigned to local memory. Listing 1.5 shows some examples.

Listing 1.5: Registers and Local Memory

```
__global__ void globalFunction() {
  // The following variables are likely to be assigned to registers:
  float3 oFloat3;
  int iValue;
  float fFloat;
  float afSmallArray[5];

  // The following variable is likely to be assigned to local memory:
  float afLargeArray[20][20];

  // The following variables are assigned to local memory as they are
  // accessed indirectly:
  float afArrayIndirectly[3];
  float4 oFloat4PointerArithmetic;

  iValue = someComputation();

  // Indirect access
  fFloat = afArrayIndirectly[iValue];

  // Access using pointer arithmetic: Works, but will force
  // oFloat4PointerArithmetic to be stored in local memory.
  fFloat = *((float *)&oFloat4PointerArithmetic + iValue);
}
```

1.4.2.2 Shared Memory

Shared memory is declared inside a global function as an array of unspecified size with the keywords **extern** and **__shared__**. The actual size of the shared memory is determined via the execution configuration (see section 1.4.3) at runtime. If one wants shared memory to contain more than one array, one should use the programming pattern shown in listing 1.6.

Listing 1.6: Shared Memory

```
__global__ void globalFunction() {
  // This array contains all shared data. The type is unimportant
  // if the pattern in this listing is used.
  extern __shared__ float afShared[];
```

[18] This is the only real cause for explicitly declaring a function as **__host__**.

```
  // The shared memory is interpreted to contain one arrays of float
  // values and one array of short values, each of the same size as
  // the x dimension of the block.
  float *pfFirstShared = (float *) &afShared[0];
  short *piSecondShared = (short *) &pfFirstShared[blockDim.x];
  ...
}
```

1.4.2.3 Global Memory

Global memory may be declared in two ways: The first way is using the keyword **__device__** preceding the declaration of a variable at file scope. This is used for global variables that do not have to be dynamically allocated. It may be accessed directly from device code. From host code, it may only be accessed using the functions `cudaMemcpyToSymbol()` and `cudaMemcpyFromSymbol()`, see section 1.5.2.

The other, and most important way, is dynamically allocating global memory via the `cudaMalloc()` (see section 1.5.2) function family and passing the pointer to the newly allocated memory as a parameter to the global function.

Listing 1.7 demonstrates both ways.

Listing 1.7: Global Memory

```
// Declaration of a global variable at file scope
__device__ float fStaticGlobal;

__global__ void globalFunction(float *pfDynamicGlobal) {
  float f1 = pfDynamicGlobal[0];
  float f2 = fStaticGlobal;
  ...
}

void hostFunction() {
  float fValueToCopy = 1.0f;

  // Copying a value to the static global variable
  cudaMemcpyToSymbol(fStaticGlobal, &fValueToCopy, sizeof(float),
      cudaMemcpyHostToDevice);

  // Dynamically allocating a global variable
  float *pfGlobal;
  cudaMalloc((void **)&pfGlobal, sizeof(float));
  cudaMemcpy(pfGlobal, &fValueToCopy, sizeof(float), cudaMemcpyHostToDevice);

  // Passing the pointer to the global function
  globalFunction<<<dim3(1), dim3(1)>>>(pfGlobal);
}
```

1.4.2.4 Constant Memory

Variables that should reside in constant memory have to be declared at file scope with the keyword **__constant__** in front of their declaration. It may be accessed directly from device code, from host code, the functions `cudaMemcpyToSymbol()` and `cudaMemcpyFromSymbol()` (see section 1.5.2) have to be used. Listing 1.8 shows some examples.

Listing 1.8: Constant Memory

```
__constant__ float fConstValue;

void hostFunction() {
  float fValueToCopy = 1.0f;
  // Copying a value to the constant variable
  cudaMemcpyToSymbol(fConstValue, &fValueToCopy, sizeof(float),
      cudaMemcpyHostToDevice);
}

__global__ void globalFunction() {
  // The constant value can be directly accessed.
  doSomething(fConstValue);
}
```

1.4.2.5 Texture Memory

Using texture memory is by far the most complicated technique of the different memory types. It contains of three parts: Declaring a texture reference, allocating memory and binding it to the texture, and lastly reading the texture from device code.

Declaring a Texture Reference A texture reference has to be declared at file scope as a variable of the type **texture**, which has to be given one, two or three parameters in a C++-template-style syntax, as shown in listing 1.9. The three parameters have the following meaning:

- The first parameter is the name of the type of data stored in the texture. It can only be a basic integer or floating point type or one of the CUDA specific vector types discussed in section 1.5.1.
- The second, optional, parameter is an integer literal defining the dimensionality of the texture. Only one- and two-dimensional textures are supported. The default value is `1`.
- The third, optional, parameter is one of the keywords `cudaReadModeNormalizedFloat` or `cudaReadModeElementType` and specifies how values read from the texture should be interpreted. The latter is the default and means that all values should be returned as they are stored in the texture. The former only makes sense for textures of 8- or 16-bit integer data types and makes all values be returned as normalized floats in the interval of $[0,1)$ instead of $\{0,\ldots,2^8-1\}$ resp. $\{0,\ldots,2^{16}-1\}$. The default is `cudaReadModeElementType`.

Listing 1.9: Declaring Texture References

```
// A one-dimensional texture of float4 values
texture<float4> oTexFloat4;

// A two-dimensional texture of float values
texture<float, 2> oTexFloat;

// A one-dimensional texture of int values, that will be returned as
// float values in the interval [0,1).
texture<float, 1, cudaReadModeNormalizedFloat> oTexInt;
```

Allocating and Binding There are two types of memory that can be bound to textures: Linear memory, which is – like global memory – allocated using the standard `cudaMalloc()` function, or CUDA arrays, which are allocated using the `cudaMallocArray()` function. The former is easier to understand and implement, the latter provides some useful functions like texture filtering and others as discussed in section 1.5.3.

Textures allocated as linear memory can have a size of up to 2^{27} elements. Memory is allocated and assigned to a pointer via the `cudaMalloc()` function and then bound to the reference using the `cudaBindTexture()`. The latter function takes four arguments:

- A pointer to an integer variable in which an offset can be stored to ensure an alignment for texture memory. As memory allocated using the `cudaMalloc()` function are guaranteed to fulfill the alignment requirements, a `NULL` value may be passed.
- The texture reference.
- A pointer pointing to the allocated device memory.
- The size of memory to be bound to the texture. Most of the time, it will be just the same size as allocated using `cudaMalloc()`.

Textures allocated as CUDA arrays are a bit more complicated. First, two variables of the type `cudaArray*` and `cudaChannelFormatDesc` have to be declared. Then, a channel format description has to be generated and assigned to the latter variable. The channel format descriptor contains a description of the contents of the texture and can be generated by calling the function `cudaCreateChannelDesc<typename>()`, with `typename` replaced by the name of the data type to be stored in the texture. Then, the memory for the array has to be allocated using a call to `cudaMallocArray()`, which takes as parameters a pointer to the first declared variable, a pointer to the channel format description, and the width (and optionally height, for two-dimensional textures) in elements of the texture. Afterwards it can be bound to the texture by a call to `cudaBindTextureToArray()` which takes as parameters the texture reference and the allocated CUDA array. The maximum dimension of textures using CUDA arrays is 2^{13} for one-dimensional and $2^{16} \cdot 2^{15}$ for two-dimensional textures.

More on the mentioned functions is detailed in sections 1.5.2 and 1.5.3. Examples for allocating and binding textures can be found in listing 1.10.

Listing 1.10: Allocating and Binding Textures

```
// The textures from the last listing are being used in this example

// Linear memory
float4 *pfLinearMemory;
// Allocation (100 elements)
cudaMalloc((void **)&pfLinearMemory, 100 * sizeof(float4));
// Binding
cudaBindTexture(NULL, oTexFloat4, pfLinearMemory, 100 * sizeof(float4));

// One-dimensional CUDA array
cudaChannelFormatDesc oFormatInt;
cudaArray *poArrayInt;
// Allocation (100 elements)
oFormatInt = cudaCreateChannelDesc<int>();
cudaMallocArray(&poArrayInt, &oFormatInt, 100);
// Binding
cudaBindTextureToArray(oTexInt, poArrayInt);

// Two-dimensional CUDA array
cudaChannelFormatDesc oFormatFloat;
```

```
cudaArray *poArrayFloat;
// Allocation (100 x 100 elements)
oFormatFloat = cudaCreateChannelDesc<float>();
cudaMallocArray(&poArrayFloat, &oFormatInt, 100, 100);
// Binding
cudaBindTextureToArray(oTexFloat, poArrayFloat);
```

Accessing Textures from Device Code Textures cannot be as easily accessed as the other memory types, but they have to be fetched using special functions.

Texture references bound to linear memory have to be fetched using the `tex1Dfetch()` function. It takes as parameters the texture reference and the coordinate of the access as an integer.

Texture references bound to CUDA arrays have to be fetched using the `tex1D()` and `tex2D()` functions. They take as parameters the texture reference and the coordinate(s) of the access as a float.

All texture fetching functions return the value stored in the element at the given coordinates (with the exception of filtering that can be optionally enabled for textures bound to CUDA arrays). Listing 1.11 shows examples of device code accessing textures.

Listing 1.11: Accessing Textures

```
__global__ void globalFunction() {
  // Read the 73rd element of the texture oTexFloat4
  float4 oFloat4 = tex1Dfetch(oTexFloat4, 73);

  // Read the 25th element of the texture oTexInt
  int iInt = tex1D(oTexInt, 25.0f);

  // Read the element at (15, 52) of the texture oTexFloat
  float fFloat = tex2D(oTexFloat, 15, 52);
}
```

1.4.3 Execution Configuration

Whenever a global function is called from a host function, an *execution configuration* must be given. The execution configuration specifies the dimension of grid and blocks and also the size of shared memory per block, if necessary.

An execution configuration is given by inserting a special expression of the form `<<<GridSize, BlockSize>>>` or `<<<GridSize, BlockSize, SharedMem>>>` between the function name and the opening parenthesis of the argument list at the position of the source code where the global function is called. GridSize and BlockSize both have to be of the type **`dim3`** (see section 1.5.1) and specify the size of the grid resp. of each block. Although they are both three-dimensional vectors, the grid may only be two-dimensional. The optional parameter SharedMem is of the type `size_t` and specifies the amount of shared memory (in bytes) that should be allocated per block.

All parameters of the execution configuration can be dynamically determined. Listing 1.12 shows a few examples.

Listing 1.12: Execution Configurations

```
// 10 x 10 grid of blocks of 256 threads, no shared memory
globalFunction<<<dim3(10, 10), dim3(256)>>>();

// Dynamically computed grid and block sizes, shared memory for
// 8 float values per block
```

```
dim3 oGridSize, oBlockSize;
oGridSize = computeGridSize();
oBlockSize = computeBlockSize();
globalFunction<<<oGridSize, oBlockSize, 8 * sizeof(float)>>>();

// Grid of 10 blocks of dynamically computed size, shared memory
// for 2 float values per thread
dim3 oBlockSize = computeBlockSize();
int iBlockSize = oBlockSize.x * oBlockSize.y * oBlockSize.z;
globalFunction<<<dim3(10), oBlockSize, 2 * iBlockSize * sizeof(float)>>>();
```

1.4.4 Built-in Variables

Within global functions, the following important built-in variables are available: `gridDim`, `blockIdx`, `blockDim` and `threadIdx`. `gridDim` and `blockDim` are of the type **`dim3`** and contain the grid resp. block dimension just as given in the execution configuration. `blockIdx` and `threadIdx` are of the type **`uint3`** (see section 1.5.1) and contain the index of the block inside the grid and resp. the thread inside the block. Listing 1.13 shows some examples.

Listing 1.13: Built-in Variables

```
__global__ void globalFunction(float *pfSomeData) {
  // If grid and block dimension are both one-dimensional (i.e. y / z = 1),
  // then this code will process all of the elements of pfSomeData
  float fValue = pfSomeData[blockDim.x * blockIdx.x + threadIdx.x];
}
```

1.5 The CUDA Runtime Library

Apart from the NVCC-specific extensions to C, CUDA contains a runtime library with many components that are necessary for allocating memory on the device, copying memory contents to and from the device, synchronizing with running device functions, error checking etc. A small subset of them that is essential for understanding the code implemented for this thesis is detailed in this section.

1.5.1 Data Types

1.5.1.1 Vector Types

CUDA provides vector types with two, three or four components for all basic integer and floating point data types (except double precision floating point data types). All of them have a name of the form `typeN` with `N` being one of `2`, `3` or `4` denoting the dimensionality of the vector and `type` being one of **`uchar`**, **`char`**, **`short`**, **`ushort`**, **`int`**, **`uint`**, **`long`**, **`ulong`** or **`float`** denoting the data type.

The components of the vectors are stored in the member variables `x`, `y`, `z` and `w`. Two- or three-dimensional vector types offer only the first two resp. three of them. As already shown in listing 1.5, their contents can also be indirectly accessed using pointer arithmetic, however, this practice is discouraged in device code as it makes the variables be stored in local memory instead of registers.

Each vector type also comes with a convenient construction function called `make_typeN()` which takes two, three or four arguments of the respective type and returns a `typeN`.

A specialized vector type is the type **dim3**, which is basically a version of **uint3**. It comes with constructors taking one, two or three arguments, the other dimensions are automatically initialized as 1.

The CUDA vector types follow all alignment requirements to make them suitable for fast loading, so that for example a **float4** can be read from global memory using one load operation from within device code.

The vector data types are automatically available within files compiled with the NVCC compiler. Other C/C++ files may also use them by including the vector_functions.h header file from the CUDA SDK.

Listing 1.14 demonstrates some of the uses of the vector data types.

Listing 1.14: Vector Data Types

```
void hostFunction() {
  float4 oVector;
  oVector = make_float4(1.0f, 2.0f, 3.0f, 4.0f);
  printf(
    "oVector = (%f; %f; %f; %f)\n",
    oVector.x, oVector.y, oVector.z, oVector.w
  );
  ...
}
```

1.5.1.2 Other Data Types

There are a few other data types that are important for programming with CUDA. Some of them are shortly described in the following.

cudaArray Pointers of this type are used to store CUDA arrays used for texturing, see sections 1.4.2.5, 1.5.2.2 and 1.5.3.

texture<Type, dimension, readMode> Global variables of this type are used to store texture references, see sections 1.4.2.5 and 1.5.3.

cudaChannelFormatDesc Variables of this type are used to store information about data contents of textures, see sections 1.4.2.5 and 1.5.2.2.

1.5.2 Memory Management

1.5.2.1 Linear Memory

Linear memory is the memory used by global memory and by one type of textures. It can be managed using the functions shortly described in the following.

cudaMalloc(void **pvDevPtr, size_t iSize) Allocates iSize bytes of linear device memory and assigns its address to *pvDevPtr.

cudaMallocPitch(void **pvDevPtr, size_t *piPitch, size_t iWidthInBytes, size_t iHeight) This function is useful for allocating two-dimensional arrays. If they were allocated the naive way

(using `cudaMalloc(*pvDevPtr, iWidth * iHeight * sizeof(type))`), accessing them might lead to a bad performance because the base addresses of the different rows would not be well aligned (see section 1.3.2.1). This function allocates `iHeight` at least `iWidthInBytes` bytes per row. If more bytes per row are necessary in order to achieve a better alignment for the rows, then each row is padded with this number of bytes. In `*piPitch`, the resulting number of bytes per row is returned.

`cudaMemcpy(void *pvDest, void *pvSrc, size_t iSize, enum cudaMemcpyKind iKind)` Copies `iSize` bytes of memory from `pvSrc` to `pvDest`. The last parameter must be one of `cudaMemcpyHostToDevice`, `cudaMemcpyDeviceToHost`, `cudaMemcpyDeviceToDevice` or `cudaMemcpyHostToHost` and determines the type of source and target memory.

`cudaMemcpy2D(void *pvDst, size_t iDPitch, void *pvSrc, size_t iSPitch, size_t iWidth, size_t iHeight, enum cudaMemcpyKind iKind)` Copies `iHeight` rows of `iWidth` bytes from one two-dimensional array to another, where the length of a row in the destination array `pvDst` is `iDPitch` bytes and the length of a row in the source array `pvSrc` is `iSPitch`. The last parameter is the same as for the `cudaMemcpy()` function.

`cudaFree(void *pvMem)` Frees memory allocated with `cudaMalloc()` or `cudaMallocPitch()`.

1.5.2.2 CUDA Arrays

CUDA arrays are used exclusively for textures as described in section 1.4.2.5. All of these functions use pointers to `cudaArray` data structures, see 1.5.1.

`cudaCreateChannelDesc<type>()` Returns a texture channel format description as a `cudaChannelFormatDesc` (see section 1.5.1) for the data type `type`, which must be one of the built-in vector data types.

`cudaMallocArray(cudaArray **poArray, cudaChannelFormatDesc *poFormat, size_t iWidth, size_t iHeight)` Allocates memory for the CUDA array `*poArray` of `iWidth` * `iHeight` elements. Each element is of the type described in `poFormat`.

`cudaMemcpyToArray(cudaArray* poArray, size_t iDstX, size_t iDstY, void *pvSrc, size_t iCount, enum cudaMemcpyKind iKind)` Copies `iCount` bytes of memory from `pvSrc` to the CUDA array `poArray`, starting at position (`iDstX`, `iDstY`) of the array. The last parameter describes the type of the copy as in `cudaMemcpy()`.

`cudaFreeArray(cudaArray *poArray)` Frees memory allocated with `cudaMallocArray()`.

1.5.2.3 Symbols

Device variables declared at file scope and residing in global or constant memory can only be accessed with these functions.

`cudaMemcpyToSymbol(Type &oSymbol, void* pvSrc, size_t iCount, size_t iOffset, enum cudaMemcpyKind iKind)` Copies `iCount` bytes of memory from `pvSrc` to the memory `iOffset` bytes after the start of `oSymbol`. `iKind` describes the direction of the copy as in `cudaMemcpy()`.

`cudaMemcpyFromSymbol(void* pvSrc, Type &oSymbol, size_t iCount, size_t iOffset, enum cudaMemcpyKind iKind)` Copies `iCount` bytes of memory from the memory `iOffset` bytes after the start of `oSymbol` to `pvSrc`. `iKind` describes the direction of the copy as in `cudaMemcpy()`.

1.5.2.4 Host Memory

The CUDA runtime library features functions to allocate and free page-locked host memory, i.e. memory in the main memory of the computer that may not be swapped. Using page-locked memory allows the device to directly read the memory contents via DMA, resulting in higher memory bandwidths for copy operations. As page-locked memory may be a rare resource, these functions should be used with care.

`cudaMallocHost(void **pvData, size_t iSize)` Allocates `iSize` bytes of page-locked host memory and stores its address in `*pvData`.

`cudaFreeHost(void *pvData)` Frees memory allocated with `cudaMallocHost()`.

1.5.3 Texture Management

As already described in sections 1.4.2.5 and 1.5.1, texture references are stored in global variables of the type **`texture`**. The functions that are used to bind and unbind a texture reference to a memory area are described in the following.

`cudaBindTexture(size_t *piOffset, struct texture &oTex, void *poData, size_t iSize)` Binds `iSize` bytes of linear device memory located at `poData` to the texture reference `oTex`. In `*piOffset`, a necessary offset to be used when accessing the texture might be stored, if the device memory was allocated using `cudaMalloc()`, this is unnecessary and `NULL` might be used as the first parameter.

`cudaBindTextureToArray(texture &oTex, cudaArray *poArray)` Binds the CUDA array `*poArray` to the texture reference `oTex`.

`cudaUnbindTexture(texture &oTex)` Unbinds the texture reference `oTex`.

1.5.4 Error Checking

Most functions of the CUDA runtime library return a value of the type `cudaError_t`. There is also a function `cudaGetLastError()` that returns the last error occurred (and clears it, so a subsequent call to the function will not return the same error again) or `cudaSuccess` if no error occurred since the last call to `cudaGetLastError()`.

There are constants of the form `cudaErrorDescription` defined for all possible error types. Most of the time, the function `cudaGetErrorString()` is of more use. It takes one argument of the type `cudaError_t` and returns a **`char`** `*` pointing to a string with a description of the error.

Some error types that are of particular interest when searching for bugs are discussed a bit more in-depth in the following.

Invalid Configuration Argument This error occurs if the specified grid or block size is zero or larger than allowed the in at least one dimension. Of course, this error can only show up if the sizes are determined dynamically

Too Many Resources Requested for Launch This error will occur if a block has less than 512 threads, but the number of threads multiplied by the number of registers required for each thread is larger than the number of registers available on the multiprocessor.[19] If this error occurs, the only solution is using a smaller block size (or trying to reduce the number of registers needed by each thread).

Unspecified launch failure The unspecified launch failure is the most dreaded of all errors that can occur. It can mean a lot of things: Maybe the execution time was too long, maybe some other error occurred during the execution of the thread. If this error occurs, one can only guess where to search for the problem.

1.5.5 Other Useful Functions

`cudaThreadSynchronize()` As mentioned earlier, the call to global functions will instantly return as the real execution is performed asynchronously. A call to the function `cudaThreadSynchronize()` will wait until all threads on the device have finished execution. Most of the time, this call is not really necessary, as all functions that access memory on the device also implicitly wait until all threads have finished – if one wants to measure the execution time of a call to a global function, however, an explicit synchronization is needed.

[19] 8192 on current generation hardware.

Chapter 2

Common Programming Patterns

In this chapter some programming patterns that were found to be useful when programming parallel algorithms (in the special instance of CUDA programs) are discussed. There are of course probably a lot more, and some of those described here may be of no use for other problems. Still, these patterns deserve a discussion that is separate from the discussion of the contact simulation algorithms itself, as they were used in different contexts of the contact simulation algorithms.

2.1 Parallel Aggregate Function

It often occurs that an aggregate function such as max or sum has to be computed over n values, each of which has been computed in one thread of the same block.

When done serially, computing an aggregate function over n values takes $O(n)$. In parallel, this time can be reduced to n parallel threads each running $O(\log(n))$ using a tree-like reduction scheme as shown in figure 2.1.

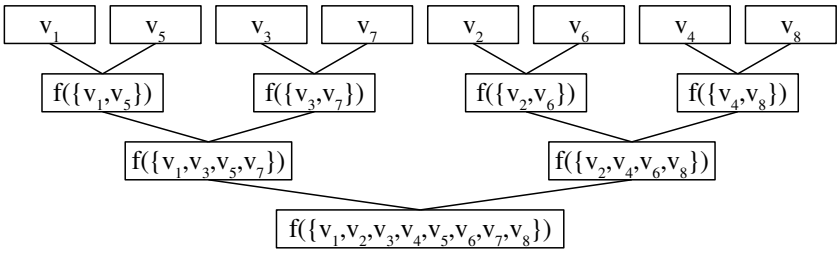

Figure 2.1: Parallel Computation of the aggregate function f over the values $\{v_1,\ldots,v_8\}$

The code shown in listing 2.1 can be used for parallel computation of an aggregate function and is contained in different variations in many places of the source code produced during the work for this thesis. It works for all numbers of threads, not only for powers of 2. Blocks are assumed to have a one-dimensional size (though it could be easily adapted for two-dimensional thread blocks).

Listing 2.1: Parallel Computation of an Aggregate Function

```
// In this function, float values are used. This can of course be adapted for other
    data types.

// Aggregate function, may of course be computed inline instead of in a function
```

```
__device__ float devAggregate(float fV1, fV2) { ... }

__global__ void devGlobal() {
  // Shared memory used for the aggregate function, must be at least blockDim.x *
     sizeof(float)
  extern __shared__ float afShared[];

  // Each thread's value
  float fValue;

  // Each thread computes its value and stores it in shared memory
  fValue = ... // Complex computation
  afShared[iThread] = fValue;

  int iOffset = blockDim.x;
  __syncthreads();

  // Parallel computation of the aggregate function
  while (iOffset > 1) {
    iOffset = iOffset / 2 + iOffset % 2;
    if (threadIdx.x < iOffset && threadIdx.x + iOffset < blockDim.x) {
      fValue = devAggregate(fValue, afShared[threadIdx.x + iOffset]);
      afShared[threadIdx.x] = fValue;
    }
    __syncthreads();
  }

  // Now thread 0 has the function value stored in fValue
  ...
}
```

2.2 Using Thread 0 for Special Tasks

Although most tasks should of course be performed fully parallel, there are always some "special" tasks that have to be performed only once for each block, e.g. storing the result of the computation of an aggregate function to global memory or choosing what to do in the next step. This can be easily done by doing this in the first block (with number 0) only. This does not lead to any performance penalties (aside from performing only one action per block for a limited time) as the threads converge again after the conditional code is finished. An example is shown in listing 2.2.

Listing 2.2: Using Thread 0 for Special Tasks

```
__global__ void devGlobal() {
  // ...

  if (threadIdx.x == 0) {
    // Perform special task
  }

  // All threads go on together
  __syncthreads();
}
```

2.3 Pair Sieve

Often (at least for the example of contact simulation) each block should perform computations for a number of pairs of elements where one of the elements is the element that this block is responsible for. While it works fine to have thread j of block i simply look at the pair of elements (i,j), this fails when there are more than 512 elements. The boundary might be even lower if the global function uses to many registers to be run in blocks of 512 threads.

The pair sieve uses an upstream kernel that filters out pairs that are negligible (in the case of contact simulation for example because the respective objects are so far from each other that they can not collide). This kernel should only perform minor computations per pair. They may be performed serially to overcome the limitation to 512 threads. Eligible pairs are then stored in global memory.

In the main computation function, thread i of block j would then do no longer process the pair (i,j), but the pair $\left(i, P_j^{(i)}\right)$, where $P^{(i)}$ is the set of all eligible partners of i found by the filter kernel. The only problem that arises when using the pair sieve is the fact that not all $P^{(i)}$ necessarily have the same number of elements, so that the number of threads of each block must be picked as $\max\left\{\left|P^{(i)}\right|\right\}$. This leads to some threads having no eligible partner to deal with. This condition must be checked and explicitly dealt with.

Listing 2.3 demonstrates this algorithm.

Listing 2.3: Pair Sieve

```
__global__ void devFilter(int *piDevEligiblePartners, int *piDevPartnerCount) {
  extern __shared__ int aiShrPartnerCount[];

  // First element is threadIdx.x, what is the partner?
  int iPartner = threadIdx.x;
  if (i2ndBox >= iBox) i2ndBox++;

  // This will store the sum of all partners for this element
  int iPartnerCount;

  // Is the pair (threadIdx.x, iPartner) eligible or negligible?
  if (devPairIsEligible(threadIdx.x, iPartner)) {
    // Eligible: Store number of partner
    piDevEligiblePartners[blockDim.x * blockIdx.x + threadIdx.x] = iPartner;
    iPartnerCount = 1;
  } else {
    // Negligible: Store -1
    piDevEligiblePartners[blockDim.x * blockIdx.x + threadIdx.x] = -1;
    iPartnerCount = 0;
  }

  // Parallely compute sum of iPartnerCount over all threads
  aiShrPartnerCount[threadIdx.x] = iPartnerCount;

  int iOffset = blockDim.x;
  __syncthreads();
  while (iOffset > 1) {
    iOffset = iOffset / 2 + iOffset % 2;
    if (threadIdx.x < iOffset && threadIdx.x + iOffset < blockDim.x) {
      aiShrPartnerCount[threadIdx.x] = iPartnerCount = iPartnerCount +
          aiShrPartnerCount[threadIdx.x + iOffset];
    }
    __syncthreads();
  }
```

```
  // Now thread 0 knows the number of all eligible partners
  if (threadIdx.x == 0) {
    piDevPartnerCount[iBox] = iPartnerCount;
  }
}

__global__ void devComplexTask(int iNumElements, int *piDevEligiblePartners) {

  // Find the partner for this thread
  int iElement = blockIdx.x;
  int iPartner = -1;

  // Look which partner should be picked
  int iX = threadIdx.x;
  for (int iIndex = 0; iIndex < iNumBoxes - 1; iIndex++) {
    int iPotentialPartner = piDevEligiblePartners[iPartner * (iNumElements - 1) +
        iIndex];
    if (iPotentialPartner >= 0)
      iX--;
    if (iX < 0) {
      iPartner = iPotentialPartner;
      // Ensure that iX will never get to 0 again
      iX = 0xffff;
    }
  }

  // Now we have to check whether this thread actually found a partner
  if (iPartner < 0) {
    // No partner found, explicitly deal with it
  }

  // Perform computation for the pair (iElement, iPartner)
}

void hostFunction(int iNumElements) {
  // Allocate memory
  int *piDevEligiblePartners, *piDevPartnerCount;
  cudaMalloc((void **)&piDevEligiblePartners, iNumElements * (iNumElements - 1) *
      sizeof(int));
  cudaMalloc((void **)&piDevPartnerCount, iNumElements * sizeof(int));

  // Execute filter
  devFilter<<<dim3(iNumElements), dim3(iNumElements - 1), (iNumElements - 1) *
      sizeof(int)>>>(piDevEligiblePartners, piDevPartnerCount);

  // Copy numbers of eligible partners for each element to host memory
  int *piPartnerCount = new int[iNumElements];
  cudaMemcpy(piPartnerCount, piDevPartnerCount, iNumElements * sizeof(int),
      cudaMemcpyDeviceToHost);

  // Find maximum number of eligible partners
  int iMaxPartners = 0;
  for (int iElement = 0; iElement < iNumElements; iElement++) {
    if (piPartnerCount[iElement] > iMaxPartners) iMaxPartners =
        piPartnerCount[iElement];
  }
```

```
  // Free unnecessary memory
  delete[] piPartnerCount;
  cudaFree(piDevPartnerCount);

  if (piMaxPartners > 0) {
    // Execute complex kernel
    devComplexTask<<<dim3(iNumElements), dim3(iMaxPartners)>>>(iNumElements,
        piDevEligiblePartners);
  }

  // Free memory
  cudaFree(piDevEligiblePartners);
}
```

2.4 Using Texture Memory for Input and Global Memory for Output

It often occurs that kernels perform a basic input - processing - output task. An easy approach for this is using a pointer to a global memory area as a parameter for the global function, from which the input data is read and to which the results are stored after finishing the processing. For some cases, this will be the fastest possible solution, particularly if the memory reads are properly coalesced.

In other cases, a global function may be significantly sped up if memory reads address texture memory and not global memory. The interesting fact is that a global memory area may both be used as a function parameter (that can be written to) and be bound to a texture reference for faster reading. In this case, the global function should of course not read from the texture after having written to the global memory area, because that would lead to wrong values in the texture cache. For the basic input - processing - output tasks mentioned above, this is no problem most of the time, so trying to replace reading operations by texture fetching operations is always worth a try – especially at it takes no effort except declaring, binding and unbinding a texture reference and replacing read operations by texture fetch operations.

Chapter 3

Solving Geometric Packing Problems Using Contact Simulation

3.1 An Introduction to Packing Problems

Packing problems are a well-studied class of problems in computer science. They are usually defined as either decision problems ("Given a set of items, do they fit into a specified container?") or as optimization problems ("How many items fit into a given container?" or "What is the minimum container for a given set of items?").

A large sub-class of packing problems are of geometrical type – whether two- or three-dimensional. This leads to certain tests that have to be performed in order to solve a packing problems. Typical tests are the following:

- Can a given object be placed into a given packing with a certain position and orientation without overlapping other objects of the packing?
- Where and with what orientation can a given object be placed into a packing without overlapping other objects of the packing?
- Is a given packing *conflict free*, i.e. are there no objects that overlap other objects?

There are several approaches for solving geometric packing problems. *Combinatorial approaches* work with a discretization of the solution space, i.e. of the possible translations and rotations for the items to be packed. Then a conflict graph can be precomputed which contains the information which configurations of pairs of items are conflict-free. This way, the problem can be reduced to the graph theoretic problem of finding the maximum stable set. As this problem is NP-complete, the conflict graph may only be of limited size to achieve reasonable computation times thus leading to a coarse discretization of the solution space.[1]

3.2 Physical Contact Simulation as a Non-Discretized Solution Attempt

If the solution space is not discretized, but all possible translations and rotations have to be allowed, a precomputation of conflicting configurations is no longer possible, and so they have to be explicitly dealt

[1] See [EIS2003].

with. One possible approach for doing so is the use of a real-world physics simulation. The purpose of this is to simulate the mechanical behaviour of rigid bodies that exchange impulses when they collide with each other. After all, this is what you would do if you would solve a packing problem in real life – you would stuff the items to pack into the container and they would shove each other around as they collide and exchange impulses.

Physics simulations are getting ever more popular in the field of computer games, as 3-D games provide an increasingly realistic simulation of physical behaviour. For different types of geometrical figures, efficient algorithms exist for deciding whether a collision between two figures has occurred and if so, where they have made contact and how deep they penetrate each other. These algorithms make it possible to compute forces that have to be applied to the figures in order to separate them again.

Contact simulations can be used for optimization problems in a number of ways. Consider the first subtype of optimization problems given above: "How many items fit into a given container?" Suppose we already have a packing of $n-1$ items in the container. We can then simply put the n^{th} item into the container in an arbitrary position and orientation and start the physics simulation. There is a high possibility that if n items actually fit into the container, the possibly *illegal* starting configuration will later be *legalized* by the physics simulation. This is particularly interesting as an add-on to combinatorial approaches, that may seriously underestimate the best possible solution because of their coarse discretization of the solution space. See chapter 5 for an example of this type of optimization problem. This approach might be combined with a Monte Carlo simulation to achieve good results.

The other subtype of optimization problems ("What is the minimum container for a given set of items?") can also benefit from physics simulation. If the shape of the container is known, and only the size of the container is of interest, one possible approach is to start with a container sized large enough for all items to comfortably fit inside and subsequently continuously decrease it using the physics simulation after each step to move all items that might now be partially outside the container into the inside again and to ensure the conflict freedom of the packing. See chapter 4 for an example of this type of optimization problem.

3.2.1 Efficiently Implementing Contact Simulation

To efficiently implement contact simulation for the purposes of solving packing problems, some limitations are imposed onto the system. First, it is supposed that the packing is always in the state of rest, i.e. the items do not have linear or angular velocities > 0. This means that impulses that are exchanged when a collision occurs between two items only lead to a movement just far enough to separate those two items from each other again. This makes the computation of contact forces much easier and does not impose limitations onto our packing algorithms.

The second limitation is much more severe: If a contact simulation step would be required to legalize each possible configuration, it would have to solve a non-linear optimization problem with a target function of dimension $O(n)$ and with $O(n^2)$ inequations. Obviously, this is far too complicated for each step of a contact simulation that should be run hundreds or thousands of times. Instead, we just look at pairwise collisions and compute an adaptation of the positions and orientations of the two items colliding that would legalize their configuration. After all pairs have been processed, the adaptations are added up and applied to the items. Of course, this will not necessarily lead to a legal configuration after this step: The adaptation might have moved one item so that it now collides with another object it did not collide with before, or the summation of the adaptations may have lead to the conflicts not being resolved after this step. See figures 3.1 and 3.2 examples.

Nevertheless, after each contact simulation step, the sum of all penetration depths will be significantly lower than before the step. Thus, it is obvious that for any given positive threshold ε there is a k so that after k steps of the contact simulation, the sum of all penetration depths will be below ε. It is not guaranteed, however, that the packing will be entirely conflict free after any number of contact simulation steps, see the example in figure 3.3, in which the penetration decreases by a fraction of $\frac{1}{2}$ for

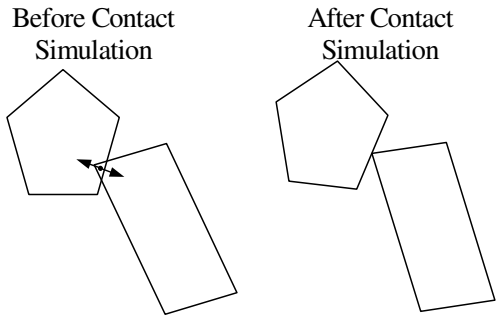

Figure 3.1: Simple collision which is legalized after one step

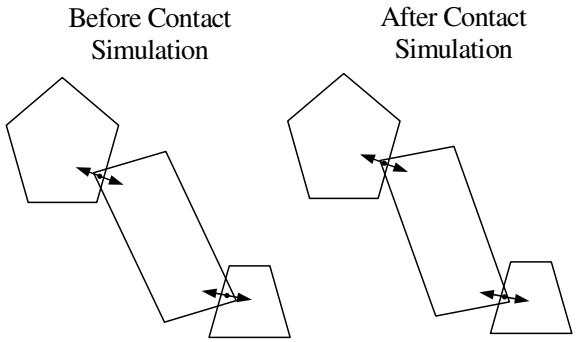

Figure 3.2: Complex collision which is not legalized after one step

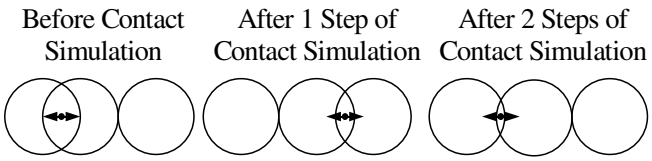

Figure 3.3: Complex collision which is not legalizable using the simplified contact simulation

each contact simulation step and thus will never reach 0.

This limitation means that compression steps must always be run repetitively in order to ensure a high possibility of a conflict free packing afterwards, and the resulting packing should be tested for being free of conflicts.

3.2.2 The Abstract Contact Simulation Algorithm

With the two limitations detailed in the last section, it is fairly easy to give a (serial) pseudocode algorithm of an abstract contact simulation algorithm.

1: $O \leftarrow \{o_1, \ldots, o_n \mid \forall 1 \leq j \leq n : o_j \text{ is a geometric object}\}$
2: **for** $i \leftarrow 1$ to n **do**
3: $\quad l_i \leftarrow 0$
4: $\quad r_i \leftarrow 0$
5: **end for**
6: **for** i from 1 to $n-1$ **do**
7: $\quad$ **for** j from $i+1$ to n **do**
8: $\quad\quad$ **if** objects o_i and o_j *collide* **then**
9: $\quad\quad\quad p' \leftarrow o_i$
10: $\quad\quad\quad p'' \leftarrow o_j$
11: $\quad\quad\quad$ *Compute* rotatory forces r' and r'' and linear forces l' and l'' necessary to separate objects p' and p''.
12: $\quad\quad\quad l_i \leftarrow l_i + l'$
13: $\quad\quad\quad r_i \leftarrow r_i + r'$
14: $\quad\quad\quad l_j \leftarrow l_j + l''$
15: $\quad\quad\quad r_j \leftarrow r_j + r''$
16: $\quad\quad$ **end if**
17: $\quad$ **end for**
18: **end for**
19: **for** i from 1 to n **do**
20: $\quad$ Apply forces l_i and r_i to object o_i
21: **end for**

There are only two procedures that have to be implemented in order to adapt this algorithm to different classes of geometric objects: The collision test of two objects (line 8) and the computation of the forces necessary to separate two colliding objects (line 11).

This abstract contact simulation algorithm can be run repetitively to achieve a packing that is conflict free with high probability. In chapters 4 and 5, applications of this algorithm to two different packing problems are shown.

Chapter 4

Circle Packing

4.1 An Introduction to the Problem of Circle Packing

The problem that is dealt with in this chapter was described in the programming contest [ZIM2005]. The problem is the following:

Given n circles with radii $\{1,\ldots,n\}$*, what is the packing with the smallest radius of the enclosing circle of the packing?*

During the programming contest, different solution attempts were conceived, especially noteworthy is the method used by the contest winners and described in [ALS2006], which uses the local optimizer SNOPT[1] and *monotonic basin hopping* to jump from one local optimum to another. A team from the University of Mainz developed the contact simulation approach to the problem and also found the best packing found in the course of the contest for one of the problem instances given in the contest.

This problem is a good starting point for discussing contact simulation based packing algorithms because of the following properties:

- The number of the degrees of freedom is only 2 because the rotatory degree of freedom does not matter.
- All impulses and forces applying to the circles are likewise only linear, not rotatory.
- Collision tests for two circles are especially easy; so is the computation of correcting forces.

4.1.1 Applying the Technique of Contact Simulation to the Circle Packing Problem

Given the properties of the circle packing given above, it is quite easy to design the geometric algorithms necessary to fully implement the contact simulation algorithm following the abstract simulation step given in 3.2.2. The two necessary test are easily implemented.

[1] http://www.sbsi-sol-optimize.com/asp/sol_product_snopt.htm.

4.1.1.1 Collision Test

Implementing the collision test between two circles C_1 and C_2 with centers $c_1 = (x_1;y_1)$ and $c_2 = (x_2;y_2)$ and radii r_1 and r_2 means simply comparing the distance of the circle centers with the sum of the radii:

$$\text{Collide}(C_1 = (c_1;r_1), C_2 = (c_2;r_2)) \quad \Leftrightarrow \quad \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} < r_1 + r_2 \tag{4.1}$$

$$\Leftrightarrow \quad (x_1 - x_2)^2 + (y_1 - y_2)^2 < (r_1 + r_2)^2 \tag{4.2}$$

4.1.1.2 Contact Forces

As rotatory forces do not occur in this problem and there are only two dimensions mattering for this problem, the contact force algorithm has to compute two linear force vectors $f_1 = (\tilde{x}_1, \tilde{y}_1)$, $f_2 = (\tilde{x}_2, \tilde{y}_2)$ necessary to separate two colliding circles, i.e. $\neg\text{Collide}((c_1 + f_1;r_1);(c_2 + f_2;r_2))$.

Although other moves would be possible to separate two colliding circles, the natural and physically correct way of separating them is moving the circles away from the *contact area* along the *contact normal* $n = (c_1 - c_2)$ as shown in figure 4.1.

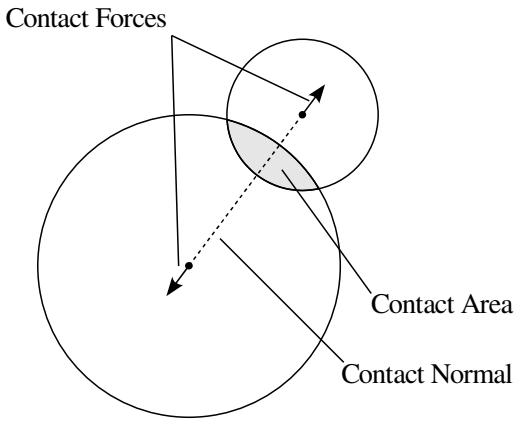

Figure 4.1: Collision between two circles

If the circles are moved only along the contact normal just as far as to separate them, then it is quite obvious that the following equation must apply:

$$f_1 - f_2 = \left(1 - \frac{\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}}{r_1 + r_c s}\right)(c_1 - c_2) \tag{4.3}$$

Since f_1 and f_2 are collinear, f_2 may be substituted by ρf_1:

$$f_1 + \rho f_1 = \left(1 - \frac{\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}}{r_1 + r_c s}\right)(c_1 - c_2) \tag{4.4}$$

The only thing left to solve the equation is the ratio of the forces applying to the two circles, ρ. Physically, the forces would apply to the circles according to their masses, i.e. the circle with the higher mass would move less than the circle with the lower mass. There are at least two possible ways to assign masses to the circles and thus choose ρ:

- A circle C with the radius r has a mass corresponding to its area, i.e. $m(C) = \pi r^2$. This approach would yield $\rho = \frac{m(C_1)}{m(C_2)} = \frac{{r_1}^2}{{r_2}^2}$.
- Each circle has the same mass $m(C) = 1$. In this case the result would be $\rho = 1$.

Both attempts yield good results for the contact simulation, I used the latter one as it makes the computation a bit easier. The resulting equation is the following:

$$f_1 = -f_2 = \frac{1}{2}\left(1 - \frac{\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}}{r_1 + r_c s}\right)(c_1 - c_2) \tag{4.5}$$

4.1.2 Solving the Circle Packing Problem Using Contact Simulation

Given a starting packing, it is easy perform an improvement trial using contact simulation: The outmost circle is picked and moved a distance of δ towards the origin (thus reducing the enclosing radius of the packing), possibly creating an illegal packing. Then, a number of contact simulation steps is run until the packing is legalized again. This sequence of actions is called a *compression step*. After a number of compression steps, every starting packing will be turned into a *stable packing* the radius of which is not getting smaller through more compression steps.

4.1.2.1 Generating a Starting Packing

Generating a starting packing is a simple task. A few possibilities are:

- Start with n circles at random positions between $-\frac{\sqrt{n}}{2} \cdot n$ and $\frac{\sqrt{n}}{2} \cdot n$ (or any other interval) and legalize the packing by performing contact simulation steps until no circle moves more than a given positive ε.
- Start with n circles at random positions between $-\frac{\sqrt{n}}{2} \cdot n$ and $\frac{\sqrt{n}}{2} \cdot n$ (or any other interval) and then scale the circles (resp. the coordinates) so that no circles overlap anymore.
- Start with the circles at positions $(i \cdot 2n, j \cdot 2n)$ with $i, j \in \mathbb{Z}; -\left\lceil \frac{\sqrt{n}}{2} \right\rceil \leq i, j \leq \left\lceil \frac{\sqrt{n}}{2} \right\rceil$. i and j may be selected randomly for each circle in a way so that no two circles are assigned the same i and j. In this case, no circles may overlap because each pair of circles has a distance of at least $2n$.

All of these possibilities have different advantages and disadvantages. The first one is easy to implement and can generate as many different packings as the random number generator allows. It might, however, result in illegal packings (as detailed in section 3.2.1). The second one is a bit more difficult to implement and can also generate as many different packings as the random number generator allows, the resulting packing will always be legal. Since the third one allows only a finite number of starting positions, it can only generate a limited number $\binom{\lceil\sqrt{n}\rceil^2}{n}\lceil\sqrt{n}\rceil^2! = \left(\frac{\left(\lceil\sqrt{n}\rceil^2\right)!}{\left(\lceil\sqrt{n}\rceil^2 - n\right)!}\right)$ of packings, but will also always result in a legal packing.

As the problem of illegal resulting packings will appear only in a very few cases and – as detailed above – illegal packings may also always occur during the compression due to the limitations of the contact simulation, the disadvantage of the first possibility does not have much relevance and it was selected as a generator of starting packings.

4.1.2.2 Further Refining Packings

Packings that have become stable through compression are not necessarily optimal, but most packings will be far less than optimal. Thus, after a stable packing has been found, it may be altered in a Monte Carlo manner and afterwards compressed again. There is an unlimited number of possible alterations. The following are used in this implementation:

- Swap pairs of circles: One or more pairs of circles are swapped. This may be done either directly, or in small steps (changing the radius of each circle by a given δ each step) performing a number of contact simulation steps after each change.
- Swap circles with the next larger circles: This is exactly the same alteration as the former one, except that a circle is always exchanged with the next larger circle. As this alteration often leads to good results, it is assigned the highest probability.
- Move a random circle outside of the enclosing circle of the packing.

After the random improvement trial and the following compression, the enclosing radius of the packing before the improvement trial is compared to the radius of the resulting packing. The better one was kept and used as the basis for the next improvement trial, the other one is discarded.

Using these random improvement trials it is possible to generate quite good packings. Experiments showed that after a while, most packings turn into a *meta-stable* state, i.e. that no improvement trials result in a better radius. This meta-stable state does not have the same radius for different starting packings, but some are better, some worse.

4.1.2.3 Serial Implementation

The algorithm described here had already been implemented in a normal, serial manner for the programming contest and had also been improved afterwards. However, because of the Monte Carlo simulation, running a high number of instances of the same problem is essential. This is where CUDA comes into play: Offering much more processing power (when properly parallelized) than a CPU, it might be possible to simulate more instances from starting packing to meta-stable state in a given time and thus find a packing near the optimum faster than the serial implementation.

4.2 Parallelization of the Circle Packing Problem

There are a few possible approaches to parallelizing the circle packing problem. I have implemented three of them and will discuss them in detail in this section. The step that consumes most of the running time of the serial implementation is the contact simulation as it has to look at each pair of circles and thus has a running time of $O(n^2)$. Thus most attention was given to how this step could be efficiently implemented in parallel.

4.2.1 One Instance, Multiple Improvement Trials

In this approach, one starting packing is generated and tried to be refined by trying several, parallely executed improvement trials followed by a compression at once, afterwards discarding all but the best resulting packing.

The reason for doing this is that after a packing has been compressed and improved for a while, it often takes a high number (a few thousand) random improvement trials to find one that actually improves the packing.

4.2.1.1 Division into Blocks and Threads

To achieve this, the problem was broken down into threads and blocks in the following way: Each block represents one packing, each thread represents one circle of the packing. The starting packing is loaded onto the device, then several blocks of threads are started. Each block then performs one or more improvement trials followed by compression steps, always checking the resulting radius, comparing it to the starting radius and keeping only the better one.

Because one packing is situated in one block, it can be fully synchronized and has a fast means of communication through the shared memory.

Contact Simulation In this approach, the essential contact simulation step can be executed in n parallel threads in time $O(n)$, because for all $1 \leq i \leq n$, thread i processes all pairs of circles (i, j) for $1 \leq j \leq n$. This computation is done *serially*.

Computation of the Enclosing Radius The computing radius can be easily computed parallely: Each thread computes the minimum radius enclosing its circle, then the maximum over all threads is computed using a parallel aggregate function, see section 2.1.

Monte Carlo Step The Monte Carlo step can also be performed on the device: The first thread of each block randomly selects an improvement trial and performs it (See section 2.2). This also means that any number of full improvement trials followed by compression steps can be performed on the device as long as the total kernel execution time stays below any existing limits (see section 1.3.2.3).

4.2.1.2 Implementation Difficulties

The only major difficulty in this implementation was the implementation of pseudo random number generators that would produce different random numbers per thread. [POD2007] provides an example implementation of a Mersenne Twister pseudo random number generator for CUDA. Although the Mersenne Twister is one of the best pseudo random number generators known, the implementation has a number of drawbacks: The configurations for each thread have to be generated in advance, and the Mersenne Twister is also very memory consuming.

As an alternative, a simple linear congruential generator was used. Good coefficients for linear congruential generators can be found in [LEC1999]. In the implementation, one linear congruential generator[2] is used to generate different seeds for each block. On the device another linear congruential generator[3] then generates the pseudo random numbers for each block.

4.2.1.3 Performance

The raw performance, i.e. the number of contact simulation steps per time unit, is stunning. Although each contact simulation step takes longer on the GPU, the number of parallely executed contact simulation steps is so high that the CUDA implementation can perform almost 50 times as many contact simulation steps per time unit as the GPU implementation. The CUDA implementation of this approach performed 486 contact simulation steps per second, while the serial implementation performed only 11 steps per second.[4]

[2] Using the parameters $m = 65521$, $a = 17364$.
[3] Using the parameters $m = 65521$, $a = 33285$.
[4] Measured on a GeForce 8800 GTX in comparison to a Intel Core 2 Duo at 2.13 GHz.

The interesting question was how the size of the best known packing would develop when using the algorithm described in this section compared to the serial version. The expectation would be that in the beginning the serial version would develop faster, because the choice of the improvement trial would not matter very much – the decrease of the radius of the enclosing circle would mainly be caused by the compression, not by the improvement trials. Then after a while, the parallel algorithm should overtake the serial one because it could carry out many more improvement trials per time unit.

Experiments were run on the same starting packing[5] with many configurations. The most successful was trying 200 improvement instances at once, while each instance would do only one improvement trial per kernel run. Although each run is different because of different generated random numbers, the expected behaviour showed up quite clearly in each run, as shown in figure 4.2. The figure also clearly shows that the packing of the parallel algorithm has come to a meta-stable state at about 175 sec, while the serial algorithm is still getting better up to the end of the test. The only difference between the observed behaviour and the expected behaviour is that the parallel algorithm has a very fast start in comparison to the serial algorithm. The cause for this is unclear, maybe one of the 200 improvement trials found an improvement that swapped a very large circle from the outside to the inner region of the packing thus drastically reducing the size of the enclosing circle. However, this behaviour seemed to be very common as all results showed this pattern.

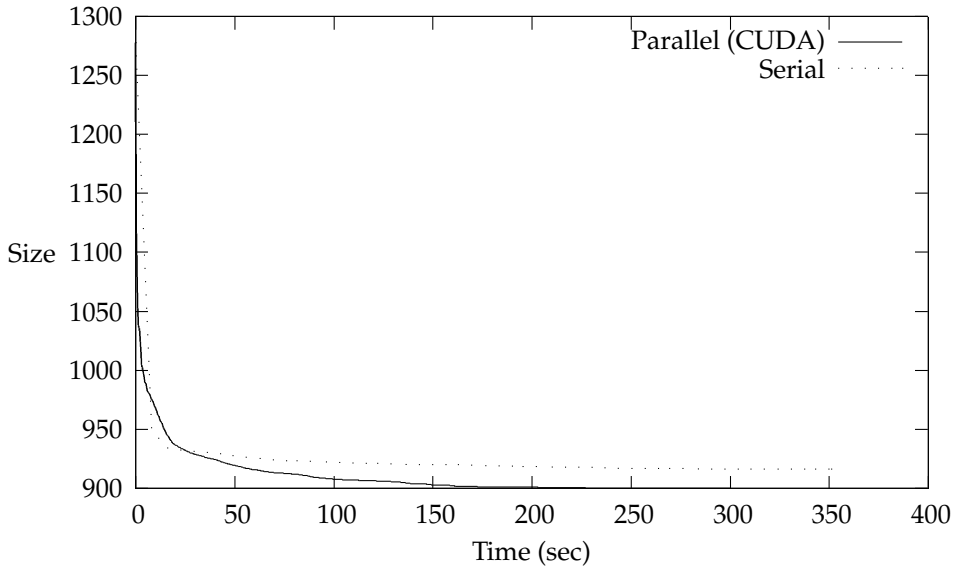

Figure 4.2: Packing size of CUDA implementation compared to serial implementation

4.2.1.4 Advantages and Disadvantages of this Approach

This approach is very good and fast for refining one starting packing.

It has two main disadvantages: First, it was not designed to be run for a long time. If it is run for a long time, the packing will become meta-stable and will not get any better. Second, this approach works best if only one improvement trial is performed per kernel run (because otherwise one block might have already found an improving move while the others do not yet know about it). This means that the kernel runs will be very short and a lot of copying results between host and device occurs, thus losing performance.

[5]The starting packing was a randomly generated packing of 128 circles.

4.2.2 Multiple Independent Instances

The second approach was specifically designed for a long-term search of good packings. As already stated, the packings enter a meta-stable state after a number of random improvement trials followed by compressions. It would thus be of no use to simply start with one packing and try to improve it for a very long time, as it would possibly not become better once it has entered a meta-stable state.[6] Because of this, the idea for a long term packing program was to parallely simulate a number of instances monitoring each of them for changes. When one of the instances did not change anymore, it would be deemed meta-stable and either stored (if it was the best known) or discarded (if it was worse than the best known). A new packing would then be generated.

4.2.2.1 Division into Threads and Blocks

The algorithm is divided into threads and blocks in the same way as in the former algorithm. The only difference is that the blocks do not start with the same starting instance, but each block has its "own" instance. As a result, the same advantages as discussed above also apply to this approach.

4.2.2.2 Detecting a Meta-Stable State

The host part of this approach has to implement a meta-stable state detection. Of course, one can never know whether a given packing will be improved by any number of improvement trials, so a heuristic was developed that simply checks how many improvement trials have not managed to reduce the enclosing circle of the packing. In the test runs, values of about 7500 unsuccessful improvement trials before a packing is considered meta-stable were used.

4.2.2.3 Advantages and Disadvantages of this Approach

This approach has the advantage that, in contrast to the former approach, long kernel runs are possible and do not impose any limitations on the performance of the algorithm, as the different instances are totally independent.

The disadvantage of this approach is, that new packings never benefit from good packings that have already been found. New packings are always randomly generated, and it just depends on luck (resp. the generated pseudo random numbers) when a generated packing will be turned into a packing that is better than the best known so far.

4.2.3 Fully Parallelized Contact Simulation

In the last approach, the contact simulation is run fully parallel, i.e. no loops are present (except for loops that perform parallel computation of aggregate functions, see chapter 2.1). Also, the computation of the enclosing radius is performed in a separate, parallel kernel run.

4.2.3.1 Division into Threads and Blocks

Each block performs collision testing and impulse computation for one circle. Each thread of the block checks for collisions with one partner. This approach therefore needs n blocks of $n - 1$ threads. Because inter-block communication is not possible, only one contact simulation step can be performed per kernel

[6] Of course, a meta-stable state might be left after any number of random improvement trials if the pseudo random number generator keeps producing "new" numbers. This has never been observed, however.

run, because the next contact simulation step will need the updated positions from all blocks of the first kernel run.

The computation of the enclosing radius is done in 1 block of n threads, where each thread computes the enclosing radius of one circle; afterwards, the maximum of the computed radii is computed parallely.

4.2.3.2 Advantages and Disadvantages of this Approach

This approach has the advantage that it fully parallelizes the contact simulation step that is consuming most of the runtime. This could be expected to reduce the total runtime of each improvement trial.

Its main disadvantage is that it can only perform very short kernel runs, as only one contact simulation step can be executed per kernel run. Improvement trials and checking the new radius against the best known radius has to be performed on the CPU. This approach can not make use of the fast shared memory in the contact simulation step, but has to perform many slow global memory accesses.

4.2.3.3 Performance

The performance of this approach turns out to be quite bad. On average it is even a bit slower as the serial CPU implementation, as the chart shown in figure 4.3 shows. On a GeForce 8800 GTX, this approach could only perform 8.4 improvement trials per second, while the serial implementation performed 11 trials per second.

The causes for this are obviously the short kernel runtimes, the high amount of global memory accesses from the GPU, and the high amount of operations that have to be performed on the CPU. While this approach could seem like the best at first, because it utilizes the parallelizability of the contact simulation to its maximum extent, it turns out to be the worst approach.

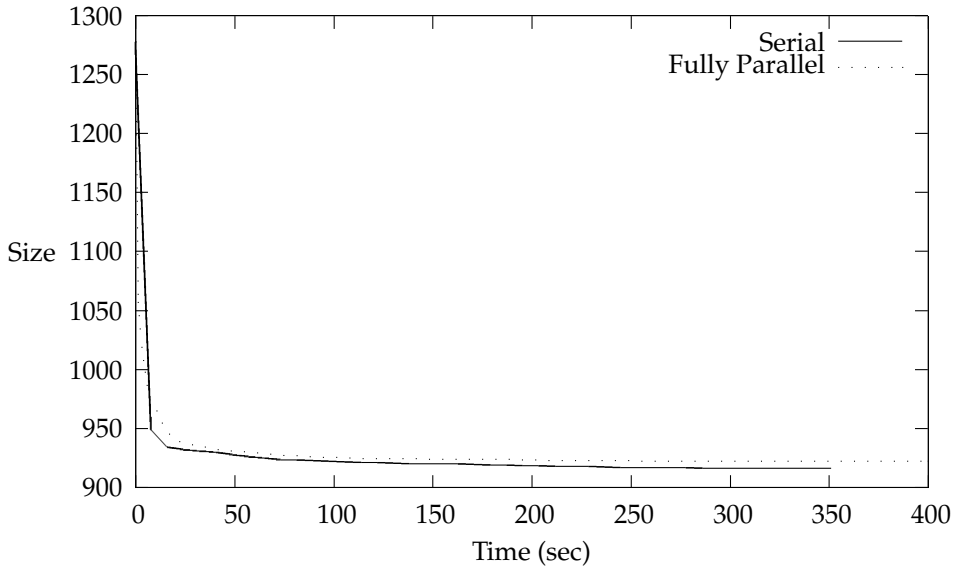

Figure 4.3: Packing size of fully parallel CUDA implementation compared to serial implementation

Chapter 5

Trunk Packing

5.1 An Introduction to the Problem of Trunk Packing

The problem that is dealt with in this chapter is originated in the measurement of loading volumes of car trunks according to the German standard DIN 70020. According to this standard, the loading volume has to be measured by putting as many boxes of the dimension $200 \times 100 \times 50$ mm into the trunk as possible. The total number of boxes that fit is the packing volume (in liters) according to this standard.

Because it is important for car manufacturers to be able to estimate the results of measuring the loading volume of a newly designed car before actually having built a physical prototype, it is essential for them to have a possibility for simulating the packing process using only a CAD model of the car.

Before automatic solutions were conceived, this was done by an expert who manually and using visual judgement only placed boxes into the CAD model of the trunk.

5.1.1 Approaches to Solving the Trunk Packing Problem

As shown in [EIS2003], deciding whether n boxes of the dimension $200 \times 100 \times 50$ mm fit into a given trunk is NP-complete. If each box is given its full 6 degrees of freedom (3 translatory and 3 rotatory), the problem is $6 \cdot n$-dimensional. It is therefore obvious that approaches to this problem must work with some restraints.

5.1.1.1 Discretization to a Uniform Grid

[EIS2003] proposes a discretization of the problem in the following way:

- All boxes must be axis-aligned, yielding only 6 possible orientations for each box.
- Box positions are discretized to a uniform grid.

Using this discretization, a *conflict graph* can be created containing the information about conflicting configurations of boxes, thus reducing the packing problem to the graph problem of a *maximum independent set*. This problem is still NP-complete, but [EIS2003] discusses various approaches to efficiently approximating a very good solution.

The solutions obtained with this approach fall short of solutions obtained by human experts.

5.1.1.2 Specialized Grand Canonical Simulated Annealing

[EIS2005] uses a specialized *simulated annealing* approach that allows arbitrary rotations and translations for the boxes. It has the following specialized properties:

- It can use a solution obtained by a combinatorial approach as a starting configuration.
- It can create new boxes in promising positions and destroy prematurely created boxes.

Using this approach, it was possible to give industry-standard solutions to trunk packing problems in a reasonable working time.[1]

5.1.1.3 Using Physical Contact Simulation

Another approach might be physical contact simulation as a layer to a Monte-Carlo approach such as simulated annealing. The contact simulation may be used in between Monte Carlo steps to resolve possibly occurring conflicts.

In this thesis, I will only describe the implementation of the contact simulation itself, not the embedding of it into an algorithm solving the trunk packing problem.

5.2 Applying the Technique of Contact Simulation to the Trunk Packing Problem

The trunk packing problem has a number of difficulties making it more complicated than the circle packing problem:

- The problem realm is $\mathbb{R}^3$, not $\mathbb{R}^2$.
- The rotatory degrees of freedom of the boxes have to be taken into account.
- The shape of the container is very complicated.

While in the circle packing problem the enclosing shape was only indirectly taken into account by repetitively pushing the outmost circle to the inside, in the trunk packing problem, the enclosing shape has to be part of the contact simulation. There are therefore two possible combinations: Boxes can collide with other boxes, or boxes can collide with the trunk. In the former case, impulses are exchanged and both boxes will be applied a force to resolve the conflict. In the latter case, the trunk remains fixed and only the box is applied a force to move it fully out of the conflict.

There are a few possibilities how the trunk should be represented in the contact simulation. The data from the CAD comes as a set of triangles, therefore taking triangles as geometric primitives to perform the collision test with is one possibility. Other possibilities include approximating the surface of the trunk with points sampled on the surface sufficiently dense or even approximating the shape of the trunk with other, 3-dimensional primitives such as spheres or pyramids.

For the CUDA implementation, a point-sample approximation of the trunks was used. Creating a good approximation of the trunks using point samples is also a complex problem, but will not be discussed here, as different point models of trunks were already available. There was also a comparable serial implementation of a contact simulation algorithm using the point data.

[1] Reasonable working time means: Less than about one day.

5.2.1 Impulse Exchange for 3-Dimensional Objects with Arbitrary Rotations

In order to be able to compute translatory and rotatory movements necessary to separate two rigid bodies from each other, a physically based method must be conceived.

5.2.1.1 Physical Properties of Rigid Bodies

The configuration of each rigid body is fully specified by the *position* of its center of mass, denoted as a vector $\mathbf{c} \in \mathbb{R}^3$, and its *orientation*, denoted by a quaternion $\mathfrak{q} \in \mathbb{R}^4$. A rigid body has a *linear velocity* $\mathbf{v} \in \mathbb{R}^3$ and an *angular velocity* $\omega \in \mathbb{R}^3$. The equations of motion are then

$$\frac{d\mathbf{c}}{dt} = \dot{\mathbf{c}} = \mathbf{v} \tag{5.1}$$

$$\frac{d\mathfrak{q}}{dt} = \dot{\mathfrak{q}} = \frac{1}{2}\mathfrak{Q}(\omega) \cdot \mathfrak{q} \tag{5.2}$$

with $\mathfrak{Q}(\omega)$ denoting the embedding of $\omega \in \mathbb{R}^3$ into the $\mathbb{R}^4$ (i.e., a quaternion with a real component of 0). Using the explicit Euler scheme, these differentials can be discretized in the following way:

$$\mathbf{c}(t + \Delta t) = \mathbf{c}(t) + \Delta t \cdot \mathbf{v} \tag{5.3}$$

$$\mathfrak{q}(t + \Delta t) = \mathfrak{q}(t) + \frac{1}{2} \cdot \mathfrak{Q}\left(\Delta t \cdot \omega\right) \cdot \mathfrak{q}(t) \tag{5.4}$$

In order to fully describe the movement of a rigid body, one also has to take into account the *mass* $m \in \mathbb{R}$ and the moment of inertia, which can be described as a *inertia tensor* $I \in \mathbb{R}^{3\times3}$. An axis-aligned box of the dimension (l_1, l_2, l_3) has the inertia tensor shown in equation 5.5.

$$I_0 = \frac{m}{3}\begin{pmatrix} l_2^2 + l_3^2 & 0 & 0 \\ 0 & l_1^2 + l_3^2 & 0 \\ 0 & 0 & l_1^2 + l_2^2 \end{pmatrix} \tag{5.5}$$

If the box is not axis-aligned, but rotated according to a quaternion $\mathfrak{q}$, then its inertia tensor can be computed using the equation given in 5.6, with $\mathrm{DCM}(\mathfrak{q})$ denoting the conversion of the rotation quaternion to a direct cosine matrix as given in equation 5.7.

$$I(\mathfrak{q}) = \mathrm{DCM}(\mathfrak{q}) \cdot I_0 \cdot \mathrm{DCM}(\mathfrak{q})^T \tag{5.6}$$

$$\begin{aligned} \mathrm{DCM}(\mathfrak{q}) &:= (q_0^2 - \mathbf{q}^T\mathbf{q}) \cdot E + 2\mathbf{q}\mathbf{q}^T + 2q_o\mathbf{q}^\times \quad \text{(with } \mathbf{q}^\times \text{ denoting the skew symmetric matrix of } \mathbf{q}\text{)} \\ &= \begin{pmatrix} q_0^2 + q_1^2 - q_2^2 - q_3^2 & 2q_1q_2 - 2q_0q_3 & 2q_0q_2 + 2q_1q_3 \\ 2q_0q_3 + 2q_1q_2 & q_0^2 - q_1^2 + q_2^2 - q_3^2 & 2q_2q_3 - 2q_0q_1 \\ 2q_1q_3 - 2q_0q_2 & 2q_0q_1 + 2q_2q_3 & q_0^2 - q_1^2 - q_2^2 + q_3^2 \end{pmatrix} \end{aligned} \tag{5.7}$$

5.2.1.2 Impulse Exchange of Colliding Bodies

As in the problem of circle packing, it is supposed that at all times all bodies are at rest ($\mathbf{v} = \omega = 0$). When two bodies A and B collide at time t_0, an impulse p just enough to separate the objects at time $t_0 + \Delta t$ is applied. Therefore a *contact point* $\mathbf{z}$ inside the conflict region $A \cap B$ and a *contact normal* $\mathbf{n}$ pointing from A to B are identified. The exchange of impulses will take place at $\mathbf{z}$.

The impulses $-p\mathbf{n}$ applying to A and $p\mathbf{n}$ applying to B instantaneously change the linear and angular velocities of the bodies from 0 to $\mathbf{v}_A$, $\mathbf{v}_B$, ω_A and ω_B, respectively.

The linear velocity is collinear with the contact normal and depends on the impulse and the mass of each body:

$$m_A \mathbf{v}_A = -p\mathbf{n}, \quad m_B \mathbf{v}_B = p\mathbf{n} \tag{5.8}$$

The angular velocity is caused by an angular momentum forcing both objects to rotate around their center. The momentum applying to a body is perpendicular to the contact normal and the lever $\mathbf{r}_A := \mathbf{z} - \mathbf{c}_A$ ($\mathbf{r}_B := \mathbf{z} - \mathbf{c}_B$) from the center of mass to the contact point, and depends on the inertia tensor of the body:

$$I_A \omega_A = -p\mathbf{r}_A \times \mathbf{n}, \quad I_B \omega_B = p\mathbf{r}_B \times \mathbf{n} \tag{5.9}$$

From equations 5.8 and 5.9 the resulting linear and angular velocities can easily be computed:

$$\mathbf{v}_A = -\frac{p}{m_A}\mathbf{n}, \quad \mathbf{v}_B = \frac{p}{m_B}\mathbf{n} \tag{5.10}$$

$$\omega_A = -pI_A^{-1}(\mathbf{r}_A \times \mathbf{n}), \quad \omega_B = pI_B^{-1}(\mathbf{r}_B \times \mathbf{n}) \tag{5.11}$$

By linearly approximating the relative velocity of the bodies at the contact point $\mathbf{z}$ in the direction normal $\mathbf{n}$, one can easily compute a function of the penetration depth (where $d(t_0)$ is the penetration depth at time t_0):

$$d(t) = d(t_0) - (t - t_0)\mathbf{n}^T(\mathbf{v}_B + \omega_B \times \mathbf{r}_B - \mathbf{v}_A - \omega_A \times \mathbf{r}_A) \tag{5.12}$$

At time $t_0 + \Delta t$, the penetration depth should be 0, so that the bodies no longer collide:

$$0 = d(t_0 + \Delta_t) = d(t_0) - \Delta t\mathbf{n}^T(\mathbf{v}_B + \omega_B \times \mathbf{r}_B - \mathbf{v}_A - \omega_A \times \mathbf{r}_A) \tag{5.13}$$

$$\Leftrightarrow$$

$$d(t_0) = \Delta t\mathbf{n}^T(\mathbf{v}_B + \omega_B \times \mathbf{r}_B - \mathbf{v}_A - \omega_A \times \mathbf{r}_A) \tag{5.14}$$

Substituting equations 5.10 and 5.11 into equation 5.14, the following equation results:

$$\begin{aligned}
d(t_0) &= \Delta t\mathbf{n}^T\left(\frac{p}{m_A}\mathbf{n} + \frac{p}{m_B}\mathbf{n} + (pI_A^{-1}(\mathbf{r}_A \times \mathbf{n})) \times \mathbf{r}_A + (pI_B^{-1}(\mathbf{r}_B \times \mathbf{n})) \times \mathbf{r}_B\right) \\
&= \Delta tp\mathbf{n}^T\left(m_A^{-1}\mathbf{n} + m_B^{-1}\mathbf{n} + (I_A^{-1}(\mathbf{r}_A \times \mathbf{n})) \times \mathbf{r}_A + (I_B^{-1}(\mathbf{r}_B \times \mathbf{n})) \times \mathbf{r}_B\right) \\
&= \Delta tp\left(m_A^{-1} + m_B^{-1} + \mathbf{n}^T((I_A^{-1}(\mathbf{r}_A \times \mathbf{n})) \times \mathbf{r}_A) + \mathbf{n}^T((I_B^{-1}(\mathbf{r}_B \times \mathbf{n})) \times \mathbf{r}_B)\right) \\
&= \Delta tp\left(m_A^{-1} + m_B^{-1} + \mathbf{n}^T(-\mathbf{r}_A \times (I_A^{-1}(\mathbf{r}_A \times \mathbf{n}))) + \mathbf{n}^T(-\mathbf{r}_B \times (I_B^{-1}(\mathbf{r}_B \times \mathbf{n})))\right) \\
&= \Delta tp\left(m_A^{-1} + m_B^{-1} + (\mathbf{n} \times -\mathbf{r}_A)^T(I_A^{-1}(\mathbf{r}_A \times \mathbf{n})) + (\mathbf{n} \times -\mathbf{r}_B)^T(I_B^{-1}(\mathbf{r}_B \times \mathbf{n}))\right) \\
&= \Delta tp\left(m_A^{-1} + m_B^{-1} + (\mathbf{r}_A \times \mathbf{n})^T(I_A^{-1}(\mathbf{r}_A \times \mathbf{n})) + (\mathbf{r}_B \times \mathbf{n})^T(I_B^{-1}(\mathbf{r}_B \times \mathbf{n}))\right)
\end{aligned} \tag{5.15}$$

The parenthesized term is called μ:

$$\mu := \left(m_A^{-1} + m_B^{-1} + (\mathbf{r}_A \times \mathbf{n})^T(I_A^{-1}(\mathbf{r}_A \times \mathbf{n})) + (\mathbf{r}_B \times \mathbf{n})^T(I_B^{-1}(\mathbf{r}_B \times \mathbf{n}))\right) \tag{5.16}$$

Using equation 5.15, the impulse p necessary to separate the bodies A and B at time $t_0 + \Delta t$ can be computed, if the penetration depth at time t_0, $d(t_0)$, is known:

$$p = \frac{d(t_0)\mu^{-1}}{\Delta t} \tag{5.17}$$

Substituting equation 5.17 into equations 5.8 and 5.9, $\Delta\mathbf{c} = \Delta t\mathbf{v}$ and $\Delta\varphi = \Delta t\omega$ that can be used to separate the bodies at time $t_0 + \Delta t$ can be computed:

$$\begin{aligned} m_A\mathbf{v}_A &= -\frac{d(t_0)\mu^{-1}}{\Delta t}\mathbf{n} \\ &\Leftrightarrow \\ \Delta\mathbf{c_A} &= \Delta t\mathbf{v}_A = -\frac{d(t_0)\mu^{-1}}{m_A}\mathbf{n} \end{aligned} \tag{5.18}$$

$$\begin{aligned} m_B\mathbf{v}_B &= \frac{d(t_0)\mu^{-1}}{\Delta t}\mathbf{n} \\ &\Leftrightarrow \\ \Delta\mathbf{c_B} &= \Delta t\mathbf{v}_B = \frac{d(t_0)\mu^{-1}}{m_B}\mathbf{n} \end{aligned} \tag{5.19}$$

$$\begin{aligned} I_A\omega_A &= -\frac{d(t_0)\mu^{-1}}{\Delta t}\mathbf{r}_A \times \mathbf{n} \\ &\Leftrightarrow \\ \Delta\varphi_A &= \Delta t\omega_A = -d(t_0)\mu^{-1}I_A^{-1}(\mathbf{r}_A \times \mathbf{n}) \end{aligned} \tag{5.20}$$

$$\begin{aligned} I_B\omega_B &= -\frac{d(t_0)\mu^{-1}}{\Delta t}\mathbf{r}_B \times \mathbf{n} \\ &\Leftrightarrow \\ \Delta\varphi_B &= \Delta t\omega_B = d(t_0)\mu^{-1}I_B^{-1}(\mathbf{r}_B \times \mathbf{n}) \end{aligned} \tag{5.21}$$

This way, computing the forces necessary to separate two bodies penetrating each other can be reduced to a computation of the following three properties:

- A contact normal $\mathbf{n}$ matching the shortest way of overcoming the penetration,
- A conflict center $\mathbf{z}$ inside the overlapping region, and
- The penetration depth $d(t_0)$.

5.2.2 Collision of Boxes and Points

5.2.2.1 General Considerations

A box can be unambiguously defined in its configuration by a center point $\mathbf{c} \in \mathbb{R}^3$, an orientation quaternion $\mathfrak{q} \in \mathbb{R}^4$ and a dimension $\mathbf{l} \in \mathbb{R}^3$ (Although the implementation has to deal only with boxes of the dimension $200 \times 100 \times 50$ mm in the end, the algorithms described here also work for other box dimensions, so the box dimension is included as a parameter). Thus the set of all boxes can be defined as

$$\mathcal{B} := \left\{ (\mathbf{c}, \mathfrak{q}, \mathbf{l}) \mid \mathbf{c}, \mathbf{l} \in \mathbb{R}^3,\ \mathfrak{q} \in \mathbb{R}^4,\ |\mathfrak{q}| = 1 \right\} \tag{5.22}$$

A point, of course, is defined as a vector $\mathbf{P} \in \mathbb{R}^3$.

5.2.2.2 Collision Test

Testing the collision between a point and a box is fairly easy: One simply has to compute the projection of the point to the face normals (without taking into account the sign, i.e. checking only in three directions), that can be computed by simply converting the rotation quaternion of the box to a direct cosine matrix

(using equation 5.7) and taking the column vectors $\mathbf{d}_i$ of the resulting matrix. Then, box center and point are projected onto each of the normals and the absolute difference of the projection values is compared to half the dimension of the box in the respective dimension. If the absolute difference is larger than or equal to half the dimension in one of the normal directions, then no collision is present (a separating axis was found). If the absolute difference is less than half the dimension in all direction, a collision has occurred.

In figure 5.1, the procedure is illustrated for the two-dimensional case.

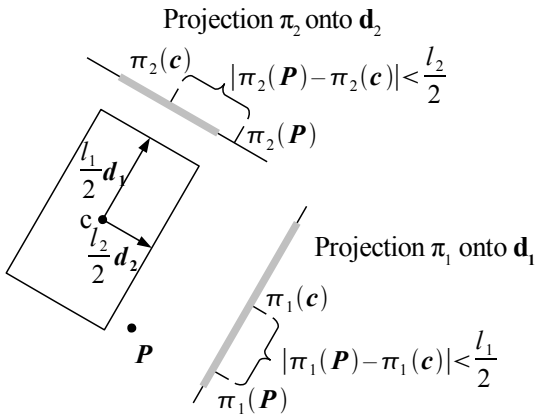

Figure 5.1: Collision test between a (two-dimensional) box and a point **P**

5.2.2.3 Contact Forces

The contact forces can be computed with the approach shown in section 5.2.1.2 using information gathered in the collision test. The penetration depth for each direction in the collision test is the difference of the length in the direction and the absolute difference of the projected center of the box and the point:

$$\delta_i = l_i - \left|\mathbf{d}_i^T\mathbf{c} - \mathbf{d}_i^T\mathbf{P}\right| \tag{5.23}$$

$d(t_0)$ is chosen as the minimum of the penetration depths of all directions $\min\{\delta_i\}$. The contact normal $\mathbf{n}$ is the direction of the collision test with the minimum penetration depth. The only thing that has to be taken care of is the sign: The contact normal should point from the box center in the direction of the point. If this is not the case, it has to be flipped. The conflict center $\mathbf{z}$ can be simply chosen as the point that is checked.

The only difficulty is that the point should not move (boxes will never be able to push aside a trunk boundary). This can be achieved by assigning the point an infinite mass, leading to the following equations for Δc and $\Delta\varphi$ of the box adapted from equations 5.16, 5.18 and 5.20, with m denoting the mass of the box, I denoting the inertia tensor of the box and $\mathbf{r}$ denoting the lever from the center of the box to the point:

$$\mu = \left(m^{-1} + (\mathbf{r}\times\mathbf{n})^T(I^{-1}(\mathbf{r}\times\mathbf{n}))\right) \tag{5.24}$$

$$\Delta\mathbf{c} = -\frac{d(t_0)\mu^{-1}}{m}\mathbf{n} \tag{5.25}$$

$$\Delta\varphi = -d(t_0)\mu^{-1}I^{-1}(\mathbf{r}\times\mathbf{n}) \tag{5.26}$$

5.2.3 Collision of Boxes among Each Other

5.2.3.1 Collision Test

For arbitrary aligned boxes an efficient collision test exists that is discussed in [GOT1996]. The test is known as *separating axis test*. The test is based on the fact that two convex polyhedra do collide if and only if there exists no separating axis, i.e. no direction in which the projections of both polyhedra are disjoint.

In the case of two boxes $A = (\mathbf{c}_A, \mathfrak{q}_A, \mathbf{l}_A)$ and $B = (\mathbf{c}_B, \mathfrak{q}_B, \mathbf{l}_B)$, if a separating axis exists, then one of the cross products of two edge directions of the boxes must also be a separating axis. See [GOT1996] for the proof. For the (two-dimensional) case of two rectangles, in which there are only four possible directions for the separating axis, an example is shown in 5.2.

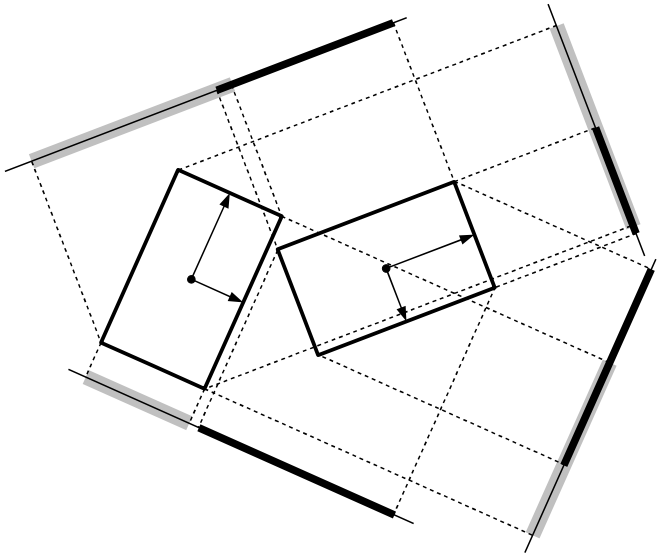

Figure 5.2: Separating axis test for two rectangles showing the projections of the rectangles onto each of the possibly separating axes

For two boxes, there are only 15 possible separating axes: one of the edge directions of A or B or a cross product of an edge direction of A and B. The edge directions of A and B can again be easily computed by converting the rotation quaternion to a direct cosine matrix and using the column vectors of the resulting matrix. Then, the penetration depth δ_i for each direction $\mathbf{d}_i$ can be computed in the following way:

$$\delta_i = \left| \gamma \left(\left[\min_{\mathbf{v} \in \mathrm{V}(A)} \{\mathbf{d}^T \mathbf{v}\}, \max_{\mathbf{v} \in \mathrm{V}(A)} \{\mathbf{d}^T \mathbf{v}\} \right], \left[\min_{\mathbf{v} \in \mathrm{V}(B)} \{\mathbf{d}^T \mathbf{v}\}, \max_{\mathbf{v} \in \mathrm{V}(B)} \{\mathbf{d}^T \mathbf{v}\} \right] \right) \right| - \left| \mathbf{d}_i^T \mathbf{c}_A - \mathbf{d}_i^T \mathbf{c}_B \right| \tag{5.27}$$

with $\gamma : \{(I_1, I_2) \mid I_1, I_2 \text{ closed intervals}\} \mapsto \mathbb{R}$ (the overlap between two intervals) and $\mathrm{V} : \mathcal{B} \mapsto \wp(\mathbb{R}^3)$ (the corner vertices of a box) being defined as:

$$\gamma([l_1, u_1], [l_2, u_2]) := -\left| \frac{l_1 + u_1 - (l_2 + u_2)}{2} \right| + \frac{u_1 - l_1 + u_2 - l_2}{2} \tag{5.28}$$

$$\mathrm{V}\Big((\mathbf{c}, \mathfrak{q}, \mathbf{l})\Big) := \left\{ \mathbf{c} + \mathfrak{Q}^{-1} \left(\mathfrak{q} \cdot \mathfrak{Q} \begin{pmatrix} \circ_1 \mathbf{l}_1 \\ \circ_2 \mathbf{l}_2 \\ \circ_3 \mathbf{l}_3 \end{pmatrix} \cdot \mathfrak{q}^{-1} \right) \;\middle|\; \circ_i \in \{+, -\} \right\} \tag{5.29}$$

In practice, this can be done more rapidly by first rotating and translating the coordinate system so that box A is located at the origin with the edge directions equalling the x-, y- and z-axis of the global coordinate system.

If the δ_i are > 0 for all directions d_i, then no separating axis was found and the boxes A and B collide. If one or more δ_i is ≤ 0, then a separating axis was found and the boxes A and B do not collide.

5.2.3.2 Contact Forces

For the contact forces between two colliding boxes, the approach from 5.2.1.2 is used again. As in the case of point / box contacts, information gathered in the collision test can be used to get the penetration depth and the contact normal: The penetration depth is the minimum of all δ_i and the contact normal is the direction $\mathbf{d}_i$ in that this minimum penetration depth occurred. Once again, the normal might have to be flipped in order to point from A to B and not the other way round.

Computing the conflict center $\mathbf{z}$ is a bit more difficult though. Depending on how the direction of the contact normal was computed (one of the edge directions of A, one of the edge directions of B or a cross product of two edge directions of A and B), different computations must be performed.

Case 1: $\mathbf{n} = \pm\text{DCM}(\mathfrak{q}_A)_{(i)}$ for some i If the contact normal is one of the edge directions (resp. face normals) of A, then a vertex of B is penetrating a face of A. Thus the penetrating vertex has to be searched and a point between this vertex and the penetrated face has to be computed in order to find a conflict center. Finding the penetrating vertex p is easy: It is the vertex the projection of which onto the contact normal has the minimum distance to the projection of the center of A onto the contact normal. The conflict center can then be computed by moving along the contact normal out of A for half the penetration depth:

$$\mathbf{z} = \mathbf{p} + d(t_0)\mathbf{n} \tag{5.30}$$

Case 2: $\mathbf{n} = \pm\text{DCM}(\mathfrak{q}_B)_{(i)}$ for some i This case is dealt with analogously to case 1. In equation 5.30, the sign has to be flipped in this case.

Case 3: $\mathbf{n} = \pm\text{DCM}(\mathfrak{q}_A)_{(i)} \times \text{DCM}(\mathfrak{q}_B)_{(i)}$ for some i, j In this case, the collision has occurred between an edge of A and an edge of B. The question is then which of the four edges of A that point in the direction $\text{DCM}(\mathfrak{q}_A)_{(i)}$ is outmost in the direction of the contact normal. This can be done by simply projecting one point of the edge onto the contact normal – because the contact normal is perpendicular to all of the edges, it does not matter which point of the edge is picked – and using the edge with the maximum projection value. The edge of B is found analogously, except that the edge producing the minimum of all projection values is used. Once the two edges have been computed, the conflict center $\mathbf{z}$ can be computed as the center of the shortest straight line connecting both edges.

5.2.3.3 Degenerate Cases

In the trunk packing problem degenerate cases of collisions often occur, i.e. cases where two faces or a face and an edge of the boxes collide. These cases are not explicitly handled in this implementation, because the penetration depths are typically so little that it does not matter that the boxes are in fact not pushed in the absolutely right direction – after a few contact simulation steps, the configuration will have been perfectly legalized with the boxes neatly aligned. If the contact simulation would be implemented for another purpose, those cases might need more attention.

5.3 Parallelization of the Box / Point Problem

5.3.1 Problem Dimension

The dimension that had to be dealt with in this case were about 40,000 points and about 250 boxes. From the raw numbers it is easy to see that simply testing all box / point pairs is not an option. In the serial version, a *k*d-tree was used to extract all points within the AABB (axis aligned bounding box) of a box to find eligible collision partners.

The use of a *k*d-Tree is far too complicated on the device:

- If not implemented very carefully, it will cause many divergent instruction paths in different threads.
- Recursion is not possible, thus all operations would have to be implemented iteratively.
- It will most possibly lead to many uncoalesced memory operations.

Thus a different solution had to be conceived, that uses a variant of the pair sieve pattern as described in section 2.3: The points are ordered into a uniform, axis oriented grid.[2] Then, the grid cells overlapped by the AABB of each box are computed in the filter step. In the following step, only points of the overlapped grid cells are considered for each box.

5.3.2 Generating the Regular Grid

Sorting points into a regular grid is an easy task, if the size c of each cell is known. In this implementation, the cell size is definable by the user and defaults to 20 mm.[3] What has do be considered, though, is an efficient storage of the grid. For this problem, the following solution was conceived:

- First, the dimension $\mathbf{d} = (d_1, d_2, d_3)$ (i.e. the number of cells in each direction) of the grid is computed. This can be done by simply searching the AABB of all points and dividing its dimensions by the cell size.
- Every grid cell is assigned a unique integer index: The cell at the grid position (x, y, z) is assigned the index $x + d_1 \cdot (y + d_2 \cdot z)$.
- For each point, the cell index is computed.
- The array of points is ordered according to ascending grid cell indices.
- For each grid cell, the index of the first point and the number of points are stored.[4]

The re-ordered point array and the grid information are then stored as textures on the device. The textures use linear memory, because CUDA arrays do not allow one dimension to be as large as the number of points in the existing data sets.

The generation of the regular grid is a preparatory step that has do be conducted only once. Thus it was not implemented parallely.

[2] Henceforth called *regular grid*.

[3] Experiments have shown that a grid size of 20 mm offers the best balance between being able to compute a relevant set of possibly overlapped points and the overall number of grid cells.

[4] The latter is in fact redundant, because it could also be computed by subtracting the former from the starting index of the next grid cell.

5.3.3 Selecting Possibly Colliding Points for each Box

In order to select eligible points for each box, the AABB for each box has to be computed. Then the grid cells overlapped by the AABB are stored and the points of these will be checked in the next step.

This step was parallelized in a way that every thread is responsible for one box. The division of the threads into blocks is done arbitrarily, as different threads do not need to be synchronized or to communicate.

Before the parallel computation can actually be performed, global memory for the return values has to be allocated. In this case, the size of the return value is not known, because the number of grid cells overlapped by the AABB of a box differs depending on how the box is positioned and oriented. Nevertheless, an upper boundary can be easily computed by assuming a dimension of the AABB of the diagonal of a box (in our case 229.13) in each direction and then adding one grid cell of "padding" in each direction.

The actual parallel function then computes the AABB – which only means converting the rotation quaternion to a direct cosine matrix, multiplying it with half the box dimension and then adding it to or subtracting it from the box center – and walks over all overlapped grid cells, storing the index in the return buffer and adding up the number of points of each grid cell. The sum of all potentially overlapped points is also stored in a separate return buffer as it turned out to be essential for the next step.

5.3.4 Computing Contact Forces

In the next step, the idea is to run one block per box and let each thread of a block compute the penetration depth and the contact forces (if any) for the box and one of the possibly overlapped points found through the means of the sieve step. Then, using a variant of a parallel aggregate function as described in section 2.1, the contact forces of the point with the maximum penetration are searched and applied. Only the contact forces of the point with the largest penetration are applied because applying all contact forces would lead to anomalies if a box is neatly aligned with a planar boundary of the trunk (such as the bottom) and then pushed a small distance outside of the trunk by boxes further inside: It would instantly collide with many points, thus leading to many contact forces pushing it in the direction of the interior of the trunk. If all these forces would simply be added together, the box would be moved much too far into the interior of the trunk.

Although this approach seems very promising at first, it has a few difficulties that have to be dealt with:

- The various boxes have a greatly differing number of possibly overlapped points. If one would simply use the maximum number of possibly overlapped points as the block dimension and the number of boxes as the grid dimension, many threads would be forced to be idle. An illustration of this problem is shown in figure 5.3, with the gray columns representing the number of busy threads per box, and the hatched columns representing the number of idle threads per box. Box B_5 has the fewest, B_3 the most possibly overlapped points.
- One or more boxes might have more then 512 possibly overlapped points. Depending on the quality of the point sample model, this is very likely to occur.

To approach the first problem, the following fact was taken into account: The number of boxes is far higher then the minimum recommended number of blocks. It could therefore be wise to perform more then one kernel run by dividing the boxes into more then one group, ordered by ascending number of possibly overlapped points. Thus the block dimension could be less for the first groups of boxes. Figure 5.4 illustrates the procedure.

The second problem could be solved in two different ways:

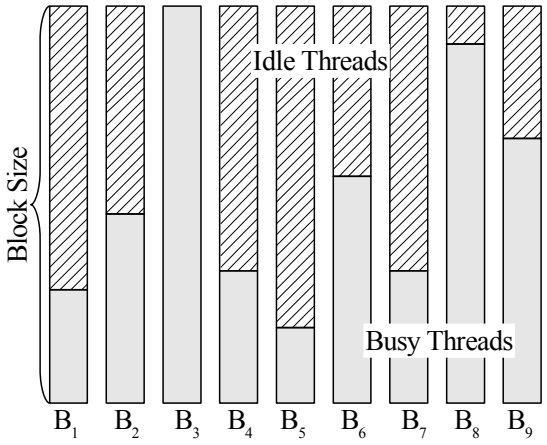

Figure 5.3: Different numbers of possibly overlapped points of the boxes $\{B_1, \ldots, B_9\}$ leading to idle threads

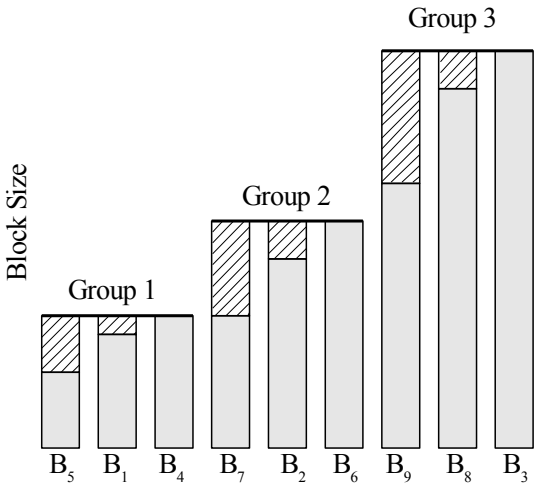

Figure 5.4: Grouping boxes to reduce the overall number of idle threads

- One could (if necessary) dedicate more then one block of the same kernel run to the same box, each checking a different set of possibly overlapped points. This would mean that every block would have to be given an offset at which point its threads should start to check the points.
- One could (if necessary) perform more then one kernel run, if one or more blocks of the kernel have not yet checked all points. In this case, only one offset would be valid for all blocks of the block.

In both cases, a final comparison of the penetration depths would have to be performed on the CPU, because different blocks can in no way communicate.

Because of the low number of boxes that actually have a high number of possibly overlapped points, the second approach was implemented because it is easier in most cases (It is not necessary to allocate and set an offset for each block). The application of the contact forces was implemented as its own kernel, so that this step could still be done parallely, although a round-trip to the CPU was necessary because of the reasons mentioned above.

5.4 Parallelization of the Box / Box Problem

For the collisions of boxes among each other, the first approach was to simply use one block for each box and one thread for each other box, leading to n blocks of $n-1$ threads each (with n being the number of boxes). If only the problem dimension (about 250 boxes) and the maximum block size (512 threads) are taken into account, this seems perfectly feasible. However, it turned out that the separating axis test uses so many registers that there are not enough for 250 threads. Thus a pair sieve (see section 2.3) approach had to be taken again.

5.4.1 Selecting Eligible Box Pairs

In the first part of the pair sieve, one block is used per box and one thread per possible partner. Each thread then tests whether the bounding spheres of the two boxes intersect. This is achieved by comparing the squared distance between both box centers and the squared sum of the radii – the latter being simply the dot product of the box dimension vector with itself.

Each thread then writes either the number of the partner (if there is a collision) or -1 (if there is no collision) to the result array. Using a parallel aggregate function (see section 2.1) each block can then compute how many possible partners for this box were found. The maximum of all these numbers is then used as the block dimension for the second step.

In an additional step, the result array (that in the beginning contained a mix of -1s and feasible partners in an arbitrary order) is compressed (i.e. the feasible partners moved to the beginning of the field) by parallely searching the field for relevant entries and moving them to the front of the field (in shared memory) before writing it out to global memory, as this turned out to be faster than walking the field in the second step.

5.4.2 Computing Contact Forces

Each block performs all collision tests for one box. Because of the use of the pair sieve technique, each thread first has to find the box it should perform the collision test with. Because of the compression mentioned above, thread i simply has to look at the i^{th} element of the partner list. Box positions and orientations reside in textures to allow for quicker access. After each thread has read the position and orientation of its box and the box to be tested, the separating axis test is performed.

As stated above, the first box is rotated and translated into the origin for this purpose, because that makes most computations much easier. Then, the penetration depth is computed in all directions. If a separating axis is found, the computation is nonetheless continued in order to avoid divergent threads.

After the computation of the minimum penetration depth (which may be 0 if a separating axis was found), a conflict center may have to be computed (if no separating axis was found). Unfortunately, all three cases mentioned above have to be dealt with separately and thus the threads of a block most possibly diverge at this time. After the conflict center has been computed, the contact forces can be easily computed from them as shown in equations 5.18, 5.19, 5.20 and 5.21.

If no penetration is detected, the contact forces are set to 0.

Using a parallel aggregate function (see section 2.1), the impulses are summed up and thread 0 of each block applies them to the box position and orientation and stores the new values into a separate result buffer. Back on the CPU, these values are then copied over the original values.

5.5 Performance

Although much care was taken in the implementation to avoid unnecessary overhead and divergent thread flows where possible, the performance of the CUDA implementation was only very little faster than the serial implementation that was already present. In fact, at first the CUDA implementation was even slower – the advance that it has gained in the end was only achievable through heavy optimizations such as unrolling loops, replacing indirect memory accesses or using texture memory for input parameters (for the latter see section 2.4).

Table 5.1 shows the average runtime for a contact simulation step in the CUDA implementation compared to the runtime of a contact simulation step in the serial CPU implementation. The problem instance was a trunk defined by 38693 points with 242 boxes in it that were randomly perturbed from a legal, axis oriented packing so they collided with other boxes and trunk boundaries in the beginning.

	Overall Time	Allocation / Copying	Computing Eligible Box/Box Pairs	Box / Box Forces	Computing Eligible Box / Point Pairs	Box / Point Forces	Force Application
CUDA	4.2238 ms	0.20831 ms	1.0015 ms	1.1652 ms	0.51613 ms	1.0554 ms	0.068326 ms
CPU (Intel Core 2 Duo 2.13 GHz)	4.7802 ms	-	-	-	-	-	-
CPU (Athlon XP 2400+)	10.1000 ms	-	-	-	-	-	-

Table 5.1: Average runtime per contact simulation step

As the table shows, there is no real bottleneck – most of the time is spent in the computation of the contact forces between two boxes, but not much more as in the computation of the computation of the forces between boxes and points. Since there are many more point / box contact simulations then box / box computations, it is obvious that each box / box computation is using much more runtime. Nevertheless, the separating axis test cannot be implemented more efficiently. Another problem that cannot be avoided is the divergence of threads in the second part of the computation of the contact forces between two boxes.

To identify the causes leading to the low performance, different tests were undertaken. Some tests were performed to profile the code and see what parts of the algorithms cause the biggest performance losses. In the course of these tests, some parts of the code that were the reason for divergent threads could be replaced by slightly longer, but non-diverging code constructions. In other places, different types of memory were compared. Thus the code got faster, but not by orders of magnitude.

Especially insightful was replacing the complicated physical algorithms with much simpler alternatives that could work only on axis-oriented boxes. The results were staggering: The average runtime of a contact simulation step only dropped by 25%. Although the code could probably be heavily optimized because it had only draft quality (since it was run only as a test), the experiences show that even this would not make it faster by an order of magnitude. This means that although the complicated collision tests and computations of contact forces surely slow down the algorithm somewhat, they are not the main reason for the relatively poor performance.

The main problem seems to be that the trunk packing problem is very data-intensive. Although not much data is used, the data that is used is accessed in irregular patterns and the amount of memory reads from different threads is also very high. Although data performance was optimized in the course of programming the CUDA implementation of the trunk packing contact simulator, there is possibly more room for performance gains on this space.

Additionaly, many switches between CPU and GPU are necessary to implement the contact simulation. This fact causes further performance losses.

Chapter 6

Discussion and Evaluation

The experiments showed that NVIDIA CUDA is a highly potential technology that can give a performance boost of up to 5000% if a problem can be parallelized in a feasible way. In this chapter I will discuss some advantages and disadvantages of CUDA that appeared during the experiments.

6.1 Ease of Use

CUDA is very easy to use, but difficult to master. The conversion of an algorithm to a CUDA program is performed very quickly if the algorithm offers potential for being parallelized, but a naive parallelization will often bring no or only little performance gain – thus giving no reason for buying a 500 € graphics card for the computations. If an algorithm does the same thing for a high number of elements, such as image effects or graphics encoders, then a CUDA program can bring enormous performance gains with almost no additional learning needed – all relevant code parts are coded in almost plain C.

Mastering all the subtleties of how the performance of a CUDA program is affected by the runtime configuration and the code quality is very difficult. Simple things like a loop containing an `if` statement can cause an enormous performance penalty and are hardly ever noticed as a possible cause for slowness of the program in the beginning. If one is used to optimizing code for serial execution, one will most likely write loops in a way that they are exited as soon as possible by using complex queries and computations in each iteration resulting in a lower total number of iterations. In SIMD programs, the opposite will give performance gains: Loops with many iterations, but only simple computations that can be executed fully parallel by all threads.

Another thing that is hard to get used to is the performance loss caused by indirect memory access: On the CPU, using pointer arithmetic to walk arrays or to perform vector computations is a very C specific way of speeding up programs – in CUDA code it will lead to variables being stored in local memory instead of registers leading to slower code.

6.2 Debugging and Profiling

A major disadvantage of CUDA code is the lack of debuggability and profileability. Device code is almost impossible to debug – there is the device emulation that can be helpful when trying to debug a kernel (in device emulation mode, the system debugger may be used or device code may print information to standard output), but it has a lot of drawbacks:

- It is impossibly slow: A kernel running a few seconds is impossible to debug in device emulation

mode because it will run minutes or hours.

- Some bugs only occur when executed on the device: Synchronization or memory access errors are very difficult to spot using device emulation.

The effect is that one will always switch between device emulation and generation of device code when looking for bugs or testing newly written code, thus leading to a much slower development cycle.

Profiling is an even bigger problem. There is a tool called the CUDA profiler that can be enabled by simply setting a shell variable – but it will only measure the time of each kernel run and memory copy operation from the host and dump them to a file in a non-aggregated way. These results are also easily achievable (but with aggregation) by using timers in the source code to measure average kernel runtimes.

What is much more important in many cases is how much time is spent in different parts of a single kernel. There is no way to find that out, though. The only workaround is commenting out relevant parts of the code and replacing them with stubs that output invalid, though possibly fitting results and computing the difference of the runtime between the original and the shortened code. However, as the results will always be incorrect (because a chain of computation was removed), these measures can only be taken as indications, not as evidence for the cause of slowly running code.

Device emulation, of course, is of no help for profiling, as running the code serially on the CPU will completely alter the runtime behaviour.

The only way provided by the CUDA runtime library is reading out processor clock cycles using the runtime library. However, these figures might also be biased by time sharing of the multiprocessor.

As long as GPUs don't provide more sophisticated on-chip profiling capabilities, profileability will probably remain a problem.

6.3 Performance

As the circle packing problem showed, the performance gains that can be achieved for certain kinds of problems are stunning (see section 4.2.1). Unfortunately, it is very difficult to estimate if a specific problem can be efficiently implemented in a massively parallel way fitting for a CUDA program or not – some approaches that seemed to be very good in the first place like fully parallelizing the contact simulation in the circle problem turned out to perform worse than the serial implementation.

Using CUDA means throwing overboard most optimization schemes that are neccessary to achieve the best performance on serial CPUs and letting the massive parallel power perform many un-optimized tasks at once.

After a while, one can get a feeling for how the GPU reacts to specific code patterns. Some patterns seem to be simply unfeasible for being massively parallelly executed. The question remains in these cases whether there are algorithms that would perform worse in the serial, but better in the massively parallel SIMD case. The computation of forces between boxes (section 5.4) for example could well be achieved better using another way of computation that would have a longer runtime when run serially but lead to fewer divergent paths of execution yielding faster parallel execution. More research on an algorithm level could maybe come up with new ideas that would erase some of the performance penalties imposed by using algorithms originally designed for quick serial execution.

6.4 Flexibility

CUDA provides the flexibility of easily using the GPU for general-purpose computations in a way that was impossible to achieve before. It also allows for very flexible access of device memory from both host and device. It even provides interfaces to OpenGL and DirectX, though those were not used or evaluated in the course of the work for this thesis. This unleashes the enormous computing power of the GPU before only usable for generating ever more realistic 3D real time graphics. The gain of flexibility in using the hardware is enormous, of course.

The constraints of flexibility imposed by the limitations of the device code keep within reasonable limits. Though there is no possibility for recursion, this fact means no loss of flexibility the most time, as recursive functions would possibly mean a loss of performance anyway.

The most severe constraints of flexibility in my eyes are the limitations for the execution configuration. Although I don't know the hardware implementation, I estimate that it could easily be possible to allow for differently sized blocks in a grid. This would, of course, mean more effort to specify the execution configuration, but would in many cases that occured in my solution attempts be able to eliminate idle threads and thus lead to a higher performance. There is no obvious reason why that should be impossible.

Another limitation of flexibility is the limit of 512 threads per block. However, this limit is at least partially caused by the number of registers per multiprocessor and that will possibly rise in future hardware generations. This limitation is also not a cause for much loss of performance in most cases.

6.5 Conclusion

Having worked and experimented with CUDA and GPU accelerated programs for about three months now, the conclusions remain unclear. The technology offers enormous performance potentials that are not expected to significantly decrease in the future – as long as gamers keep shelling out for ever more realistic graphics, the speed of graphics hardware will possibly keep developing at its current pace, leaving CPUs even further behind.

On the other hand, problems like the trunk packing problem simply cannot seem to benefit from these possibilities. It is possibly an interesting field of research to find massively parallel alternatives to known fast serial algorithms. It may be that doing a bit of fundamental research on these topics backed by actual experiments on CUDA hardware might lead to promising results – it may also be that some problems remain only inefficently solveable on a massively parallel SIMD platform like CUDA. For these problems, using graphics hardware is simply not worth it – for the 500 € price of a GeForce 8800 GTX one can get easily the fastest desktop processor currently available together with fitting motherboard and RAM, all of which would have to be bought in addition to the graphics adapter in the former case as well.

All in all, CUDA is a very interesting new way of accelerating computations needing high performance. Fully using the possibilities of the graphics hardware is not so easy in many cases. Some problems, however, may achieve levels of performance otherwise only achievable through clusters at a fraction of the cost and administrative expense.

Appendix A

Contents of the CD-ROM

A.1 Directories

The implementations of the algorithms discussed in this thesis are contained on the CD-ROM accompanying this thesis. The CD-ROM contains the following directories:

CirclePackingParallel: This directory contains the implementation of the circle packing algorithm as detailed in section 4.2.1.

CirclePackingLongTerm: This directory contains the implementation of the circle packing algorithm as detailed in section 4.2.2.

CirclePackingFullyParallel: This directory contains the implementation of the circle packing algorithm as detailed in section 4.2.3.

PackingsCircles: This directory contains sample starting packings for 24, 26, 50 and 128 circles. The latter was used in the performance comparisons as shown in figures 4.2 and 4.3.

Cudacs: This directory contains the implementation of the trunk packing algorithm *Cudacs* as detailed in chapter 5.

PackingsTrunks: This directory contains a sample trunk data set (w168-6.point) and two packings, one legal (w168-6-25o-242-best.box) and one illegal (w168-rnd.box). The latter was used as a starting packing to measure performances as given in table 5.1.

All source code directories also contain a makefile feasible for use with GNU make. The makefiles suppose that CUDA is installed in `/usr/local/cuda`.

A.2 Command Line Syntax of the Programs

A.2.1 CirclePackingParallel

The executable generated by the makefile is called `ParallelCirclePacking`. It has to be given at least two command line parameters: The first mandatory parameter is the name of the starting packing, the second is the name of the file the resulting packing should be written to. Before the mandatory one or more of the following optional parameters may be given:

-statfile [filename] Specifies the name of a file to which statistics should be dumped.

-time [seconds] Specifies how long the program should be run.

-instances [number] Specifies the number of instances (= blocks).

-trials [number] Specifies the number of improvement trials per kernel run.

A.2.2 CirclePackingLongTime

The executable generated by the makefile is called `LongTimePacking`. It has to be given at least one command line parameter, that specifies the name of the file which the resulting packing should be written to. Before the mandatory one or more of the following optional parameters may be given:

-logfile [filename] Specifies the name of a file a log should be written to.

-time [seconds] Specifies how long the program should be run.

-instances [number] Specifies the number of instances (= blocks).

-trials [number] Specifies the number of improvement trials per kernel run.

-circles [number] Specifies the number of circles.

-stuck [number] Specifies the number of kernel runs that a packing instance may not get better without being deemed meta-stable.

A.2.3 CirclePackingFullyParallel

The executable generated by the makefile is called `ParallelCirclePacking`. It has to be given at least two command line parameters: The first mandatory parameter is the name of the starting packing, the second is the name of the file the resulting packing should be written to. Before the mandatory one or more of the following optional parameters may be given:

-statfile [filename] Specifies the name of a file to which statistics should be dumped.

-logfile [filename] Specifies the name of a file a log should be written to.

-time [seconds] Specifies how long the program should be run.

-trials [number] Specifies the number of improvement trials per invocation of the host packing function.

A.2.4 TrunkPacking

A.2.4.1 Makefile Parameters

The makefile in this directory may be run with or without each of the following parameters:

EMULATION=1 If this parameter is given, the executable uses CUDA device emulation mode.

VERBOSE=1 If this parameter is given, the command line switch -verbose may be used.

NOVISUALIZATION=1 If this parameter is given, the executable is compiled without visualization support.

A.2.4.2 Command Line Parameters

The executable generated by the makefile is called `TrunkPacking`. It has to be given at least three command line parameters: The first mandatory parameter is the name of the point file, the second is the name of the file containing the starting packing, and the last is the name of the file the resulting packing should be written to. Before the mandatory one or more of the following optional parameters may be given:

-time [seconds] Specifies how long the program should be run.

-steps [number] Specifies the number of contact simulation steps per invocation of the host packing function.

-cellsize [number] Specifies the size of a cell of the regular grid.

-boxdim [number] [number] [number] Specifies the dimension of a box (x,y,z) in mm. The default value is $200 \times 100 \times 50$ mm.

-visualize Enables the visualization mode. May not be used together with -time or -steps. See section A.2.4.3 for keyboard commands in the visualization mode.

-verbose Outputs status information during each contact simulation step. May significantly slow down the program.

A.2.4.3 Keyboard Shortcuts in the Visualization Mode

The following keyboard shortcuts are valid in the visualization mode:

q: Quits the program.

w: Toggles wireframe mode.

+,-: Zoom.

Tab: Changes the active box.

4,6,2,8,1,7: Move the active box (Makes most sense when the keys on the num pad are used).

u,j,i,k,o,l: Rotate the active box.

a: Print average run time of a contact simulation step to stdout.

s: Save the current packing.

s: Reload the last saved packing.

Enter: Toggle continuous contact simulation.

Appendix B

The Cudacs Library

The algorithm for the trunk packing problem was constructed in a way that makes it suitable to be used by other programs, such as a simulated annealing algorithm or similar. The library is called cudacs, short for CUDA contact simulation. Any programming language able to link to standard object files could be used to develop a program using cudacs. The program would not need to deal explicitely with CUDA, as all CUDA functions are properly encapsulated.

Cudacs also offers convenience functions for loading and saving point and box files. These do not have to be used, of course. An alternative would be to write a special conversion function that converts boxes or points from a program specific representation into the representation understood by cudacs, runs the contact simulation, and converts the results back.

For a complete reference, see the Doxygen documentation available for the cudacs library in appendix C.

B.1 General Procedure for Using Cudacs

If a program would want to use the cudacs library, the following would be neccessary:

- (If wanted, load points from file using the function `cudacsReadPoints()`.)
- (If wanted, load boxes from file using the function `cudacsReadBoxes()`.)
- Load points onto the device using the function `cudacsLoadPointsToDevice()`. As the points do not change, they stay on the device for more than one contact simulation step.
- Perform one or more contact simulation steps by calling `cudacsContactSimulation`. This function updates the box positions and orientations given as parameters. It can also perform more consecutive contact simulation steps leading to slightly better performance as values don't have to be copied back from the device between iterations.
- If of interest, read the cudacs performance counters that give a measure of the performance in different cudacs parts using the function `cudacsGetPerformanceCounters()`.
- Unload the points from the device using the function `cudacsUnloadPointsFromDevice()`.
- (If points and/or boxes were loaded using the cudacs functions, free the allocated memory using `cudacsFreePoints()` resp. `cudacsFreeBoxes()`.)

B.2 Cudacs Files

This is only intended as a brief overview. For the full reference, see the Doxygen documentation.

cudacsCore.h This is the main cudacs header file allowing the program to load and unload points and to perform the contact simulation. It also provides a means for cudacs configuration via the `oCudacsConfig` struct.

cudacsIO.h This file provides a means for conveniently loading and saving box and point files.

cudacsUtil.h This file provides utility functions for measuring performance and retrieving the current date and time.

cudacsDatatypes.h As the CUDA data types (see section 1.5.1) are not available in code not including the CUDA runtime library, this header defines the most important data types neccessary to use cudacs.

Appendix C

Doxygen Documentation of the Cudacs Library

C.1 cudacsCore.h File Reference

```
#include "cudacsDatatypes.h"
```

Classes

- struct CCudacsConfig

  Contains the configuration for a cudacs instance.

Functions

- void cudacsLoadPoints (int iNumPoints, float4 *const poPoints, float fCellSize=20)

  Loads a set of points into the CUDA CS engine.

- void cudacsUnloadPoints ()

  Unloads the points from the CUDA CS engine.

- void cudacsContactSimulation (int iNumBoxes, float3 oBoxDimension, float4 *poBoxTranslations, float4 *poBoxRotations)

  Performs a single contact simulation step.

- void cudacsContactSimulation (int iNumBoxes, float3 oBoxDimension, float4 *poBoxTranslations, float4 *poBoxRotations, int iSteps)

  Performs one or more contact simulation steps.

- void cudacsGetPerformanceCounters (double *pdCounters)

  Returns the average run time for the different phases of the contact simulation.

Variables

- CCudacsConfig oCudacsConfig
  Contains the configuration for cudacs.

C.1.1 Detailed Description

Contains the basic functions to perform CUDA based contact simulation.

C.1.2 Function Documentation

C.1.2.1 void cudacsContactSimulation (int *iNumBoxes*, float3 *oBoxDimension*, float4 * *poBoxTranslations*, float4 * *poBoxRotations*, int *iSteps*)

Performs one or more contact simulation step. The box positions and orientations may be changed by this function. Calling this function with a number of steps n $>$ 1 is faster than calling cudacsContactSimulation(int, float3, float4 *, float4 *) n times because the intermediate results can stay on the graphics hardware between steps.

Parameters:

- ← ***iNumBoxes*** The number of boxes.
- ← ***oBoxDimension*** The dimension of a box.
- ↔ ***poBoxTranslations*** A pointer to an array of float4 containing the box positions (with any value in the w component).
- ↔ ***poBoxRotations*** A pointer to an array of float4 containing the box orientations as normalized quaternions.
- ← ***iSteps*** The number of contact simulation steps to perform.

C.1.2.2 void cudacsContactSimulation (int *iNumBoxes*, float3 *oBoxDimension*, float4 * *poBoxTranslations*, float4 * *poBoxRotations*)

Performs a single contact simulation step. The box positions and orientations may be changed by this function.

Parameters:

- ← ***iNumBoxes*** The number of boxes.
- ← ***oBoxDimension*** The dimension of a box.
- ↔ ***poBoxTranslations*** A pointer to an array of float4 containing the box positions (with any value in the w component).
- ↔ ***poBoxRotations*** A pointer to an array of float4 containing the box orientations as normalized quaternions.

C.1.2.3 void cudacsGetPerformanceCounters (double * *pdCounters*)

Returns the average run time for the different phases of the contact simulation. The array pdCounters will contain the following information:

pdCounters[0] Time to allocate memory and to load boxes to / from the device.

pdCounters[1] Time to compute possibly overlapping pairs of boxes.

pdCounters[2] Time to compute overlapping boxes and their contact forces.

pdCounters[3] Time to compute the points possibly overlapped by boxes.

pdCounters[4] Time to compute box / point overlaps and the resulting contact forces.

pdCounters[5] Time to apply the contact forces to the boxes.

Parameters:

→ ***pdCounters*** An array with at least 6 elements, that will be assigned the average run times for the different phases of the contact simulation as detailed above.

C.1.2.4 void cudacsLoadPoints (int *iNumPoints*, float4 *const *poPoints*, float *fCellSize* = 20)

Loads a set of points into the CUDA CS engine. Before any contact simulations can be run, this function has to be called. The point set has to be given as a pointer to an array of float4 values with any value in the w component. It builds a regular grid with the cell size fCellSize * fCellSize * fCellSize and loads it onto the CUDA compatible graphics adapter. The grid and the points stay in graphics memory until cudacsUnloadPoints() is called or the application terminates.

Parameters:

← ***iNumPoints*** The number of points.

← ***poPoints*** A pointer to an array of float4 containing the point coordinates.

← ***fCellSize*** The size of the cells of the regular grid, defaults to 20.

C.1.2.5 void cudacsUnloadPoints ()

Unloads a set of points from the CUDA CS engine after it had before been loaded using cudacsLoadPoints(int, float4*, float)}.

C.2 cudacsDatatypes.h File Reference

Classes

- struct float2
- struct float3
- struct float4
- struct int2
- struct int3
- struct int4

Defines

- #define __A8__ __align__(8)

  Enforces a variable to be aligned at an 8-byte boundary.

- #define __A16__ __align__(16)

  Enforces a variable to be aligned at a 16-byte boundary.

C.2.1 Detailed Description

Contains the some CUDA data types that are neccessary to run cudacs. If the CUDA runtime library is already included in the program using cudacs, then _CUDACS_NO_DATATYPES should be #define'd before including any cudacs header files.

C.3 cudacsIO.h File Reference

```
#include <fstream>
#include <string>
#include "cudacsDatatypes.h"
```

Functions

- void cudacsReadPoints (const char *const pcFileName, int &iNumPoints, float4 *&poPoints)

  Reads a set of points from the file pcFileName.

- void cudacsReadBoxes (const char *const pcFileName, int &iNumBoxes, float4 *&poTranslations, float4 *&poRotations, int4 *&poParameters)

  Reads a set of boxes from the file pcFileName.

- void cudacsWriteBoxes (const char *const pcFileName, int iNumBoxes, float4 *poTranslations, float4 *poRotations, int4 *poParameters=NULL, int iVersion=2, std::string sGenerator=std::string("Trunk Packing Lib by JF.Goetzmann"))

  Saves the boxes to the file pcFileName.

- void cudacsFreeBoxes (float4 *&poTranslations, float4 *&poRotations, int4 *&poParameters)

  Frees memory allocated for boxes.

- void cudacsFreePoints (float4 *&poPoints)

  Frees memory allocated for points.

C.3.1 Detailed Description

Contains the convenience functions to load and save POINT and BOX files.

C.3.2 Function Documentation

C.3.2.1 void cudacsFreeBoxes (float4 *& *poTranslations*, float4 *& *poRotations*, int4 *& *poParameters*)

Frees memory allocated for boxes.

Parameters:

← ***poTranslations*** Pointer to the memory area for translations of boxes.

← ***poRotations*** Pointer to the memory area for rotations ofboxes.

← ***poParameters*** Pointer to the memory area for parameters boxes.

C.3.2.2 void cudacsFreePoints (float4 *& *poPoints*)

Frees memory allocated for points.

Parameters:

← ***poPoints*** Pointer to the memory area for points.

C.3.2.3 void cudacsReadBoxes (const char *const *pcFileName*, int & *iNumBoxes*, float4 *& *poTranslations*, float4 *& *poRotations*, int4 *& *poParameters*)

Reads a set of boxes from the file pcFileName.

The file has to be in BOX format: Any line starting with # is considered a comment, also all characters from a # occurring in the middle of a line up to the end of the line are considered a comment. Blank lines are ignored. The first non-comment non-whitespace characters must be the number of boxes in the file. The descriptions of the points follow, each box description must be on a single, separate line, the describing numbers separated by white space. Comments may occur between lines describing point or at the end of a line describing a point, but not between the coordinates of a point.

Each box description consists of the following numbers:

- The x, y, and z coordinates of the translation of the box.
- The first two columns of the rotation matrix (DCM). The remaining column can be computed with the cross product.
- Two integer parameters, which may be used for arbitrary purposes.

The number of boxes is returned in the reference parameter iNumBoxes. The translations are returned in a newly allocated float4 array of the length iNumBoxes with only the x, y and z component set. The rotations are returned in a newly allocated float4 array of the length iNumBoxes in quaternion representation, with x,y,z representing the vector part and w representing the real part. The integer parameters are returned in a newly allocated int4 array with only the x, y and z components set.

The returned pointers must be freed with the cudacsFreeBoxes(float4*&, float4*&, int4*&) function.

Parameters:

← ***pcFileName*** A zero-terminated string containing the name of the POINT file.

→ ***iNumBoxes*** The number of boxes in the file.

→ ***poTranslations*** The positions of the boxes read from the file.

→ ***poRotations*** The translations of the boxes read from the file.

→ ***poParameters*** The auxillary parameters of the boxes read from the file.

C.3.2.4 void cudacsReadPoints (const char *const *pcFileName*, int & *iNumPoints*, float4 *& *poPoints*)

Reads a set of point from the file pcFileName.

The file has to be in POINT format: Any line starting with # is considered a comment, also all characters from a # occurring in the middle of a line up to the end of the line are considered a comment. Blank lines are ignored. The first non-comment non-whitespace characters must be the number of points in the file. The coordinates of the points follow, each triple of coordinates must be on a single, separate line, separated by white space. Comments may occur between lines describing point or at the end of a line describing a point, but not between the coordinates of a point.

The coordinates given in the file have do be in 1/10 millimeters. The coordinates returned by this function will be in millimeters.

The number of points is returned in the reference parameter iNumPoints. The points are returned in a newly allocated float4 array of the length iNumPoints. Each element of the array contains a single point in the x, y and z components, the w component of each float4 element is undefined.

The returned pointer must be freed with the cudacsFreePoints(float4*&) function.

Parameters:

← ***pcFileName*** A zero-terminated string containing the name of the POINT file.

→ ***iNumPoints*** The number of points in the file.

→ ***poPoints*** The points of the file.

C.3.2.5 void cudacsWriteBoxes (const char *const *pcFileName*, int *iNumBoxes*, float4 * *poTranslations*, float4 * *poRotations*, int4 * *poParameters* = `NULL`, int *iVersion* = `2`, std::string *sGenerator* = `std::string("Trunk Packing Lib by JF.Goetzmann")`)

Saves the boxes to the file pcFileName. The boxes may be stored with additional parameters given in poParameters, if poParameters is NULL then 0 values will be saved as parameters. The file will be in box 0.0.2 format by default, if it should be stored in box 0.0.3 format, then 3 has to be given as the version parameter.

Parameters:

← ***pcFileName*** Name of the file to store the boxes in.

← ***iNumBoxes*** Number of the boxes.

← ***poTranslations*** Translations of the boxes.

← ***poRotations*** Rotations of the boxes.

← ***poParameters*** Additional parameters. If null, 0 values are written to the file.

← ***iVersion*** Version of the file: 2 means 0.0.2, 3 means 0.0.3.

← ***sGenerator*** Name of the generating program

C.4 cudacsUtil.h File Reference

`#include <string>`

Functions

- double cudacsTimer ()

  Returns a high precision timer value representing the number of seconds passed since an arbitrary, but fixed, moment in time.

- std::string cudacsFormattedTime ()

  Returns a string showing the current local time and date.

C.4.1 Detailed Description

Contains some utility functions that can be used with cudacs. This file can also be used without the rest of the cudacs library.

C.4.2 Function Documentation

C.4.2.1 std::string cudacsFormattedTime ()

Returns a string showing the current local time and date. Format is YYYY-MM-DD hh:mm:ss.

Returns:

The current local time and date, formatted as a string.

C.4.2.2 double cudacsTimer ()

Returns a high precision timer value representing the number of seconds passed since an arbitrary, but fixed, moment in time. This function is meant to provide a means for performance tests.

Returns:

A high precision timer value representing the number of seconds passed since an arbitrary, but fixed, moment in time.

C.5 CCudacsConfig Struct Reference

Contains the configuration for a cudacs instance.

`#include <cudacsCore.h>`

Public Attributes

- float fEpsilon

The regular grid is enlarged by fEpsilon to ensure that the point with the largest components does not fall outside the grid.

- bool bVerbose

  If set to true, the cudacs functions will output status information to stdout.

C.5.1 Member Data Documentation

C.5.1.1 bool CCudacsConfig::bVerbose

If set to true, the cudacs functions will output status information to stdout. This may significantly slow down execution of the contact simulation.

The documentation for this struct was generated from the following file:

- cudacsCore.h

C.6 float2 Struct Reference

`#include <cudacsDatatypes.h>`

Public Attributes

- float x

  The x component.

- float y

  The y component.

C.6.1 Detailed Description

A 2-component float vector for use with cudacs.

The documentation for this struct was generated from the following file:

- cudacsDatatypes.h

C.7 float3 Struct Reference

`#include <cudacsDatatypes.h>`

Public Attributes

- float x

  The x component.

- float y
  The y component.
- float z
  The z component.

C.7.1 Detailed Description

A 3-component float vector for use with cudacs.

The documentation for this struct was generated from the following file:

- cudacsDatatypes.h

C.8 float4 Struct Reference

`#include <cudacsDatatypes.h>`

Public Attributes

- float x
  The x component.
- float y
  The y component.
- float z
  The z component.
- float w
  The w (=4th) component.

C.8.1 Detailed Description

A 4-component float vector for use with cudacs.

The documentation for this struct was generated from the following file:

- cudacsDatatypes.h

C.9 int2 Struct Reference

`#include <cudacsDatatypes.h>`

Public Attributes

- int x

  The x component.

- int y

  The y component.

C.9.1 Detailed Description

A 2-component int vector for use with cudacs.

The documentation for this struct was generated from the following file:

- cudacsDatatypes.h

C.10 int3 Struct Reference

`#include <cudacsDatatypes.h>`

Public Attributes

- int x

  The x component.

- int y

  The y component.

- int z

  The z component.

C.10.1 Detailed Description

A 3-component int vector for use with cudacs.

The documentation for this struct was generated from the following file:

- cudacsDatatypes.h

C.11 int4 Struct Reference

`#include <cudacsDatatypes.h>`

Public Attributes

- int x
  The x component.
- int y
  The y component.
- int z
  The z component.
- int w
  The w (=4th) component.

C.11.1 Detailed Description

A 4-component int vector for use with cudacs.

The documentation for this struct was generated from the following file:

- cudacsDatatypes.h