Scientific Computation for Simulations on Programmable Graphics Hardware

Robert Strzodka [a] Michael Doggett [b] Andreas Kolb [c]

[a] *Caesar Research Center, Bonn*

[b] *ATI Research*

[c] *Computer Graphics Group, University of Siegen*

Abstract

Graphics Processor Units (GPUs) have emerged as powerful parallel processors in recent years. Although floating point computations and high level programming languages are now available, the efficient use of the enormous computing power of GPUs still requires a significant amount of graphics specific knowledge.

The paper explains how to use GPUs for scientific computations without graphics specific terminology. It offers an algorithmic view on GPUs with comparisons to cache aware and parallel programming of CPUs. Two typical simulation techniques, namely grid based and particle based methods are discussed.

1 Introduction

Three major factors make the development of graphics hardware based on commodity PCs truly outstanding in recent years. First, the *computational power* of graphics processing units (GPUs) for commodity PC hardware has grown much faster than for CPUs. Second, the high performance is available at a *very good cost/performance ratio*. Finally, within the last 2-3 years, GPUs have become *programmable* by high level languages.

From an abstract point of view, the GPU is a *parallel streaming processor*, particularly suitable for the fast processing of large arrays. Thus, many researchers have started utilizing graphics processors to enhance the performance of their specific, in many cases, non-graphics applications and simulations. The special field of "General-Purpose computation on GPU (GPGPU)" has evolved (see GPGPU (2005)), and Owens et al. (2005) offers a survey of this emerging research area. Although performance gains depend strongly on the application, one can say that speedup factors around 5 against algorithms on the CPU are commonly reported.

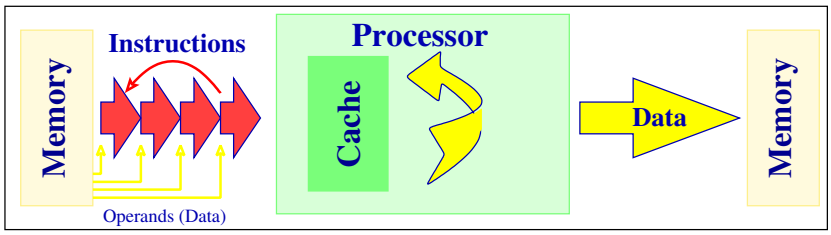

Fig. 1. Instruction stream processing

This introductory paper aims to give an overview on the GPU from a data processing perspective. It presents an abstract view on the GPU accessible to anyone experienced with CPU programming. The focus is on the main concepts that allow us to determine if a given algorithm or computation can be efficiently performed on the GPU.

The remainder of this paper is structured as follows. Section 2 discusses conceptual aspects of GPU programming. Section 3 provides an algorithmic understanding of the GPU. Section 4 describes examples of GPU-based simulations, and Section 5 discusses developments to be expected in the near future and their impact on scientific computation.

2 Programming the GPU

This section presents a short discussion of GPU programming, focusing on the concept of data-stream programming (Section 2.1) and on programming languages (Section 2.2).

2.1 The Concept of Data Stream Programming on the GPU

Instruction stream programming is the traditional model used in CPU programming. Its data model is based on a von-Neumann architecture, where instructions and data are stored in the same memory. Instructions refer to the data needed for execution and potentially, to other instructions in the case of branching. During processing, the data required for an instruction's execution is loaded into the cache, if not already present. This model is very flexible, but has the disadvantage that the data-sequence is completely driven by the instruction sequence, yielding inefficient performance for uniform operations on large data blocks.

In data stream processing, on the other hand, the processor is first configured by the instructions that need to be performed and in the next step a data-stream is processed. The execution is performed efficiently by many depth-parallel units in the pipeline. Given sufficient memory bandwidth, and more than one processing-

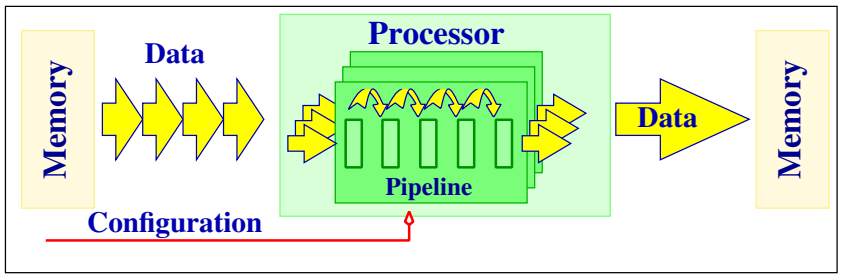

Fig. 2. Data stream processing

pipeline, the execution is also parallelized in breadth by distributing it among several pipelines. GPUs currently focus on breadth parallelism using Single Instruction Multiple Data (SIMD) processing units (4 component vectors), and pipeline arrangements similar to vector processors (16 pipelines). But they will eventually have to apply more depth parallelism (pipelining) to reduce the bandwidth requirements.

Data stream processing is advantageous when large data blocks undergo the same operation, because this allows the memory efficient streaming and parallel processing of the data. For a typical example, consider a matrix-matrix addition $C = A+B$ (assuming proper matrix dimensions). Listing 1 compares the different computing paradigms. The outer two loops over `i,j` drive the data access in the instruction processor, but do not have any effect on the final result. The alternative data-stream approach simply defines the input and output arrays, and the loop body in a *kernel*, which is the code used to obtain a single resulting data element. This model gives the processor the freedom to decide on the order of execution in the outer loops. This example is stereotypical for general purpose computations on GPUs and Section 3 further develops the understanding of leveraging GPUs as fast array processors.

2.2 Programming Languages

For programming the GPU one needs a graphics Application Programming Interface (API), which understands function calls similar to those in Listing 1, and a graphics language for the kernels that are passed to `loadKernel`.

```
// instruction stream                // data stream
for(i=0; i<NumRows; i++)             setInputArrays( A, B );
  for (j=0; j<NumCols; j++)          setOutputArrays( C );
    C[i][j]= A[i][j]+B[i][j];        loadKernel("return a+b;");
                                     execute( );
```

Listing 1: Two implementations of the matrix-matrix addition. The small letters a, b used in the kernel program refer to the current elements of the matrices A, B.

The two major graphics APIs are OpenGL (2005) and Microsoft's DirectX (2005). Implementations of these APIs are available for a variety of languages, e.g. C/C++, Java, Delphi, Fortran, Perl. But both, OpenGL and DirectX target graphics programming and thus one must encapsulate the native API functionality in a library to make the code look as nicely as in Listing 1.

The kernel programs can be written either in assembly or C-like high level languages, which are preferred. For OpenGL, the OpenGL Shading Language (GLSL), for DirectX9 the High Level Shading Language (HLSL) can be used. The third language Cg ("C for graphics") can be used with OpenGL and DirectX, but requires a compiler to translate the code to the required platform.

Current GPUs contain two programmable stages where kernels can be executed: the *vertex* and the *fragment* processor. Accordingly there are two types of kernel programs: *vertex programs* and *fragment programs* [1] . In array processing vertex programs typically control the index ranges of the input arrays and fragment programs contain the operations to be executed on the data. So fragment programs are usually the most important part of the algorithm as in Listing 1. In this example, no vertex program needs to be specified because the entire arrays are used.

Recapitulating, one needs several pieces of code to implement an array operation on the GPU: the data-flow specification written in a common high level language using the graphics API, the fragment program for the data computation in a graphics language, and possibly a vertex program for the setting of index ranges.

Several projects try to further simplify this procedure. Buck et al. (2004); McCool and Toit (2004); McCormick et al. (2004) describe stream programming languages which extend C to provide simple data-parallel constructs to allow using the GPU as a streaming coprocessor. They abstract and virtualize many aspects of the underlying graphics hardware so that the programmer does not need to understand how to use the underlying graphics API.

3 An Algorithmic GPU Model

This section discusses basic properties of GPUs which are decisive for the design of efficient algorithms on this architecture. It focuses on problems that can be solved by storing data in large arrays and operations that manipulate the array elements. The efficient implementation of data structures such as stacks, trees or hashes are much more difficult on GPUs and require a deep insight into the architecture. Consequently, only the similarities and differences of CPUs and GPUs as fast data array processors are discussed. The focus will lie on efficient manipulation of large arrays

[1] historically, these programs are often called *vertex shader* and *fragment* or *pixel shader*

Table 1
Algorithmic CPU-GPU comparison

Property	**CPU**	**GPU**
native memory layout	1D	2D
input arrays	native 1D; higher dimensions with offsets	native 1D, 2D or 3D; higher dimensions with offsets
output arrays	native 1D, higher dimensions with offsets	native 2D; other dimensions with offsets
overlap of input and output regions	allowed	not allowed
gathers	arbitrary	arbitrary
scatters	arbitrary	global, regular, or emulated
dynamic branching	supported through speculative execution, but still not desirable in loops	primitive or no support, should usually be resolved by subregion processing
highest precision number format	double (s52e11) or long double (s63e15)	float (s23e8)
ideal computational intensity for floats	≈ 8, (= 3200 MHz $\cdot$ 16 byte (SSE float4) / FSB800 / 64 bit)	$\approx$ 8 (= 16 pipelines $\cdot$ 500 MHz $\cdot$ 32 byte (2xfloat4) / $\underbrace{\text{DDR}}_{\times 2}$ 500 / 256 bit)

or large array regions which allow parallel processing of the array elements.

3.1 Native Memory Layout

The native memory address space for a CPU is 1D. Due to the caching mechanism, accessing a data element in direct 1D-neighborhood to the one recently used is fast. Accessing elements farther away, in sense of the 1D distance, is slower. Since higher dimensional arrays are realized by applying address-offsets, the access is not equally fast in all spacial directions, e.g. after reading $a[i][j]$ access to $a[i][j+1]$ or $a[i][j+2]$ is fast, while access to $a[i+1][j]$ is slow.

For GPUs the native memory address space is 2D. Therefore, 2D arrays are native objects for GPUs and data is commonly organized as a collection of 2D arrays [2]. Access times are optimal if the 2D distance to the current location is small, e.g. after $a[i][j]$ access to $a[i][j+1]$ and $a[i+1][j]$ is equally fast. Other dimensions can

[2] in graphics terminology, a native 1D, 2D or 3D array is a 1D, 2D or 3D *texture*

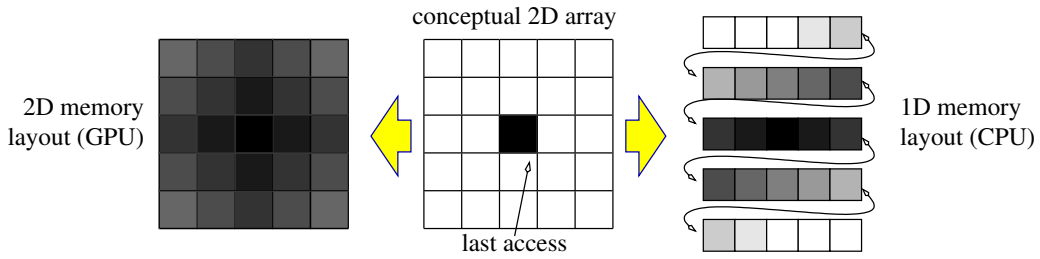

Fig. 3. Memory layout for the GPU (left), CPU (right); dark gray indicates fast access.

be represented with offsets as for CPUs, but this always includes the costs of offset computations. Lefohn et al. (2005) discuss such kind of mappings.

GPUs also support native 1D and 3D arrays to some extent, but restrictions concerning their usage apply. Additionally, 1D arrays lack the fast access to vertical neighbors in 2D, and for 3D the access to neighbors in the z-direction is not always as optimized as for x and y. Therefore, conceptually one can think of GPUs as fast 2D array processors.

3.2 Array Sizes

The sequential execution engine of a CPU can process any amount of data: large arrays, small arrays and single elements. Depending on the cache size ($0.5-4$ MiB [3]) a certain array size is ideal for performance, but below that size large sequential data collections are better than many distributed small ones. For the GPU a similar reasoning applies but with much higher factors. The limit size beyond which performance starts to degrade significantly is very high (> 64 MiB). But performance drops dramatically if only small amounts of data are processed sequentially.

Therefore, efficient processing always has to specify an entire array or a sufficiently large region thereof, say at least 1000 elements. For 2D arrays the regions are typically rectangles. In general, a polygon, horizontal and vertical line segments, or even a point cloud can also serve as a region, but this is in the given order less and less efficient. The performance of GPUs benefits significantly from large and spatially coherent regions. Therefore, one usually operates on rectangular regions, even if this leads to unnecessary processing of certain elements.

Currently each array dimension on all GPUs is restricted to 4096. Therefore, 1D arrays cannot hold many items. On the other hand a 4096×4096 float 2D array already consumes 64 MiB, such that it is fairly easy to utilize the entire memory of a graphics card (128-512 MiB) with a few 2D arrays.

[3] International standard IEC (2000): M$= 10^6$, Mi$= 2^{20}$ and similar for Ki, Gi.

3.3 Input and Output Regions

On the CPU it is possible to read and write to any array during an operation. There are no restrictions on the input and output addresses. This is convenient, but overlapping input and output regions causes severe cache synchronization problems and thus can degrade performance.

The computation on the GPU is organized in processing steps each involving at least thousands of elements. During a processing step GPUs maintain a clear separation between the input and output regions (with few special exceptions). This avoids data synchronization problems and allows massively parallel processing of the array elements, and long pipelines. The output arrays in GPU operations are always 2D, while the input arrays may be 1D, 2D or 3D.

There are current restrictions concerning the maximal number of input (32) and output (4) arrays which can be involved simultaneously in a processing step. But both numbers are likely to rise significantly in the near future.

3.4 Data Flow

Given several input and output arrays, the steps of an algorithm are mainly a combination of two different types of data flow:

Gather: After specifying the output regions, each element of the output region is computed by combining the information from various positions of the input arrays.

Scatter: Having specified an input region, each element of the input region determines both the position and the value of a corresponding output element. In general, the same output value may be written to several positions or to none.

If several values are written to the same output position, a new value may either replace the old one or all values may be accumulated.

In the first case the output region is clearly structured and the input elements maybe chaotically distributed, in the second case it is vice versa. Both types of data flow are often used simultaneously.

For CPUs there are no restrictions from which positions in the input arrays the data is gathered, and to which positions in the output arrays it is scattered. For GPUs there are also no restriction on the gathering, but scattering is highly restricted. There are four options for scattering:

Global static scatter, i.e. the same index offset is applied to all output positions.
Point cloud scatter, i.e. each output location is processed individually, including

possible discards and overwrites of output values. Processing of point clouds is significantly slower than the processing of rectangular regions.

Scatter reformulated as gather, i.e. a gather operation is performed which produces the same result as the scatter. This technique can easily deal with multiple output positions per output value, but for an efficient gather operation the scatter must have almost the same form for all input positions. Ideally the offsets of the output positions to the corresponding input position are the same for all input positions.

Sorted scatter, i.e. initially, data is not scattered but rather the output position is appended to the output value and the items are sorted according to the output position in a subsequent step. To reduce sorting requirements a small bound on the maximal offset of the output position to the corresponding input position is helpful.

Hence, gather operations are exploited widely on GPUs, while the above scatter emulations must be applied when complex scatters cannot be avoided in algorithms. On the other hand it is sometimes advantageous to use global static scatters instead of gathers (see Section 4.1).

3.5 Conditionals

Within the loop body the CPU may always execute element dependent if-else branches or even nested loops. This often deteriorates performance and whenever possible one should move the conditionals before the loops and possibly code different loop bodies for different subsets of the processed region.

The same strategy should be applied more aggressively for GPUs, because the resulting gains are higher. Formally GPUs also allow to use arbitrary conditional constructs in loop bodies, but high costs are associated with *dynamic branches* if they are not spatially coherent. Quickly oscillating branches in a loop prohibit the pure parallel SIMD processing of the array elements. Therefore, ideally the same operations should be applied to all elements of the input regions. The problem lies only with the dynamic branching, i.e. when different code is executed depending on a condition, e.g. conditional assignments with the ternary C-operator `?:` or unrollable loops, i.e. loops with a fixed number of iterations, are not a problem.

3.6 Number Formats and Operations

CPUs natively support integer and float operations of different precision. GPUs currently have no integer formats and offer fixed point and floating point numbers of different precision. Major representations are 8 bit fixed point numbers and s23e8 single float. Besides the scalar values, GPUs also have native vectors of up to 4

components, e.g. float2, float4. Basically all standard mathematical operations (e.g. +-*/, sin, atan, log, sqrt, pow) are available and most of them can be executed directly on these native vectors. Therefore, they should be used when the input data suggests such a grouping, but it is not necessary to formulate the entire code in float4 operations, e.g. GPUs can process a scalar and a float3 instead of a float4 operation (co-issue).

In scientific computations GPUs usually operate on floats which are four times larger than the previously used 8 bit fixed point numbers; thus the bandwidth requirement has quadrupled. This leads to a similar situation as on the CPU, that more operations can be executed in the chip than the memory bus can provide data for in one clock cycle. Thus current GPUs require a *computational intensity* of approximatively 8, i.e. 8 operations must be performed on each float read from memory to keep the processing power and the memory bandwidth in balance.

3.7 Summary

GPUs are fast 2D array processors. They concentrate on the parallel processing of the array elements and therefore require a relatively large region (e.g. 32x32 elements) in each processing step. During the processing step the corresponding input and output regions must be distinct. The data flow is primarily controlled by gathering arbitrary elements of other input arrays while the scattering of output values must be realized with other methods. Concerning operations all standard mathematical functions are available but conditionals should be avoided inside of loops. Table 1 also presents a summary of the main properties.

4 Scientific Computations on the GPU

Data in scientific computations is often represented in arrays and large data regions undergo the same operations. The GPU is optimized for such processing and can therefore speedup the solution of various problems. Two popular spatial discretization methods for differential equations are discussed, namely grids and particles.

4.1 Grids

Grids discretize a continuous domain by introducing a number of discrete control nodes in the domain. In these locations the nodes typically represent the value of the continuous function (Finite Differences), or the integral of this function in the surrounding Voronoi cell (Finite Volumes), or the integral of this function weighted

by a basis function of this node (Finite Elements). In any case the collection of the nodal values forms a nodal vector $\bar{V} = (\bar{V}_i)_{i \in I}$, where I is some enumeration of the nodes. This vector is the discrete representation of the continuous function in the domain.

On the GPU the nodal vector $\bar{V}$ is stored in the native 2D or 3D arrays. Given a certain node one often requires access to its spatial neighbor nodes. For unstructured grids this is cumbersome on GPUs and the reader is referred to Bolz et al. (2003); Krueger and Westermann (2003) for possible arrangements. For dynamic adaptive grids see Lefohn et al. (2005). But for tensor grids the situation is very convenient. If the nodal vector $\bar{V}$ of a d-dimensional tensor grid is stored in a d-dimensional array, then the spatial neighborhood relations are preserved in the array. For example, the nodal vector $\bar{V}$ of a 257×257 grid discretizing $[0, 1]^2$ can be stored in $v[257][257]$, such that the node value (i, j) is stored in $v[i][j]$. In most GPU applications the tensor grid is even fixed with node positions $(x = j/w, y = i/h)$. Note that irrespective of how the vectors are stored, they are always treated as one dimensional structures in the following linear algebra operations.

When numerically solving a differential equation, the discretization procedures often result in explicit (e.g. $\bar{X}^{n+1} = A\bar{X}^n + \bar{F}^n$) or implicit schemes (e.g. $A\bar{X}^{n+1} = \bar{X}^n + \bar{F}^n$) with some matrix $A = (A_{i,j})_{i,j \in I}$. Because the matrix usually also depends on some data (e.g. $A(\bar{X}^n, \bar{F}^n)$) there are two main computing tasks: the assembly of the matrix and the matrix vector product. The assembly of the matrix often involves differential or even integral quantities, and non-linear mappings. For the implementation on the GPU the required operations are not so important since all of the standard mathematical functions are available. The main question is which types of data flow (gathers, scatters) are involved. The matrix vector product is used to demonstrate the different data flow types.

A matrix vector product can be formulated as a series of gather or a series of scatter operations. Considering the gathers first, the matrix vector product is a series of gathers in form of inner products between the matrix rows and the vector:

$$A\bar{V} = \left(\bar{A}_{i,\cdot} \cdot \bar{V}\right)_{i \in I} = \left(\sum_{j \in I} A_{i,j}\bar{V}_j\right)_{i \in I}, \qquad \bar{A}_{i,\cdot} := (A_{i,j})_{j \in I} \quad \text{(matrix row)}.$$

The fragment program implements the inner product $\sum_{j \in I} A_{i,j}\bar{V}_j$. Recall that the outer loop over i is implicit, see Section 2.1. This executes fast on the GPU if the node values required to compute the new element $(A\bar{V})_i$ are spatially (in the grid) close to $\bar{V}_i$. Commonly, band matrices are encountered which fulfill this property. A matrix band is defined as the vector [4]

$$\tilde{A}^k := (A_{i-k,i})_{i \in I}, \quad k \in I.$$

[4] illegal matrix entries, e.g. $A_{-1,0}$, are set to zero

For example in 2D, if for all $i \in I$ the computation of the element $(A\bar{V})_i$ requires $\bar{V}_i$ and its 8 spatial neighbors, then the matrix has 9 bands. The gathers in the matrix vector product can be executed in parallel very efficiently as they only access 2D neighbor values of the current input value, which is very fast (see Section 3.1).

Similar to the other vectors, bands are stored as d-dimensional arrays on the GPU. When the matrix is represented in this form the assembly of the matrix is also easy. One defines the arrays as output arrays and executes a kernel which assembles the bands of the matrix in the problem specific manner.

If the matrix vector product requires slow accesses to values far away from the current input value, a reformulation in terms of global static scatters (see Section 3.4) is advantageous:

$$A\bar{V} = \sum_k T_k \left(\tilde{A}^k \bullet \bar{V} \right), \quad \left(\bar{X} \bullet \bar{Y} \right)_i := \bar{X}_i \bullet \bar{Y}_i, \quad T_k(\bar{X}) := \left(\bar{X}_{i+k} \right)_{i \in I} .$$

Here, $\bullet$ denotes the component-wise multiplication, and the global static scatter is represented by the index translation T_k. This version needs two processing steps. The first step computes the component-wise multiplication for all matrix bands in a fragment program. The second step executes the sum over the results from the first step in a new fragment program, and a vertex program performs the index translations. So the slow access to distant values necessary in the former gather formulation, is replaced by a series of cheap index translations. But because the translations are global this scatter reformulation is only applicable to band matrices or at least matrices with local band structure. See Bolz et al. (2003); Krueger and Westermann (2003) for totally unstructured sparse matrices.

Unfortunately, there is one more complicating factor concerning linear algebra operations on GPUs. Because of the very low computational intensity, the matrix vector product cannot exploit the high parallel processing power of GPUs (see Section 3.6) and thus performs very bad. The solution to this problem is the intermingling of the assembly of the matrix with the matrix vector product. The matrix should almost *never* be assembled completely on the GPU. Only few expensive intermediate results should be generated. Then, instead of executing the totally inefficient pure matrix vector product, it is combined with the assembly step which finishes the generation of the matrix on-the-fly before each product. This increases the operation count but the few intermediate results decrease the bandwidth requirements and thus dramatically gain performance. Rumpf and Strzodka (2005) give more details on this technique.

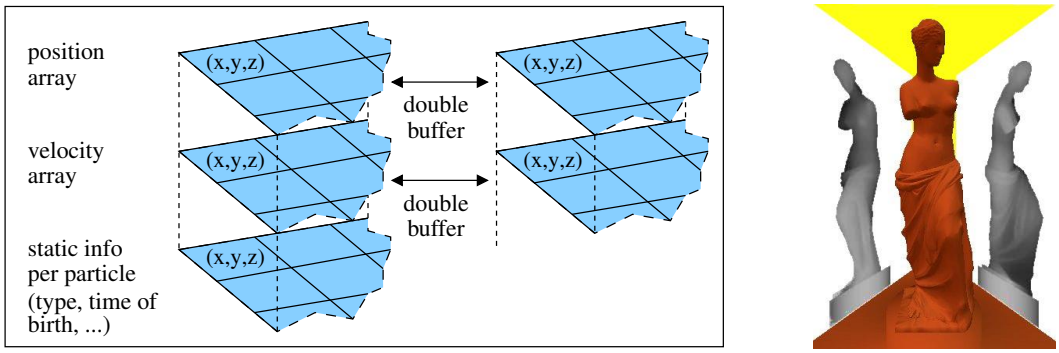

Fig. 4. Data storage concept for particle systems (left) and two depth-maps as parts of a complex boundary representation of a statue (right).

4.2 Particles

Besides grid-based approaches, particles are often used in simulations, in particular in Computational Fluid Dynamics (CFD). In general, the particles in a particle system can be uncoupled, statically or dynamically coupled. Starting with uncoupled particles, only the particle motion needs to be computed.

Since the GPU's native memory layout is 2D (see Section 3.1), the data for a single particle is naturally stored at a unique 2D array position in several arrays (see Figure 4, left). Given the position $\vec{x}^n$, the velocity $\vec{v}^n$, some force $\vec{F}^n$ and the time-step width τ in time-step n a particle with mass m is traced

$$\vec{x}^{n+1} := \vec{x}^n + \tau\vec{v}^n\,, \qquad \vec{v}^{n+1} := \vec{v}^n + \tau\vec{F}^n/m\,.$$

Because input and output regions are distinct on GPUs (see Section 3.3) two arrays for the data of two consecutive time-steps $n, n+1$ are needed, and a flip-flop algorithm which exchanges the role of input and output. The above formula describes first order motion. Applying higher precision integration, possibly more than two data arrays have to be used in a ring-buffer like manner. More details can be found in Kipfer et al. (2004) and Kolb et al. (2004).

Correct treatment of boundaries is very important for particle motion. One can represent the boundary using simple primitives parameterized by a set of static variables. Alternatively, more complex shaped boundaries can be described using several 2D distance-maps from different perspectives (see Figure 4 right and Kolb et al. (2004)). Each distance map represents a specific portion of the boundary as z-distance values w.r.t. a 2D-plane in an appropriate local coordinate system. Of course, this approach is restricted in the sense, that local concavities may cause an erroneous boundary representation.

In uncoupled systems the forces $\vec{F}^n$ may be dynamically computed, time-varying and attracting or repelling point-forces, or static and global 3D-force-fields stored in 3D arrays. In coupled systems the forces $\vec{F}^n$ depend on the position of other a-

priori specified particles (static coupling), or all particles in a certain neighborhood (dynamic coupling).

The above data concept can be easily extended to statically coupled systems using the gathering technique. The demonstration NVIDIA (2005) shows the principal approach using simple linear springs.

In dynamically coupled systems usually the n-nearest neighbor problem has to be solved. Parallel sorting algorithms can be utilized to realize a sorted scatter approach (see Section 3.4). Algorithmically, parallel sorting is not optimal due to its runtime complexity, but only these kind of algorithms can be used on GPUs. Kipfer et al. (2004) use the bitonic merge sort for global particle sorting combined with a space subdivision technique to approximatively detect particle-particle collisions [5] .

Applying this technique to more complex coupling schemes is rather difficult, e.g. *Smoothed Particle Hydrodynamics (SPH)* models fluids based on particle motions and applies forces to ensure the Navier-Stokes equations (see Gingold and Monaghan (1977)). Here, the neighborhood may include several hundred particles and approximative neighborhood detection causes problems, i.e. discontinuous forces over time. Müller et al. (2003) discusses a cache optimized CPU implementation allowing interactive SPH simulations up to a few thousand particles using a space subdivision technique. Kolb (2005) gives a concept of implementing dynamic coupled particle systems on the GPU without sorting. The key idea is to accumulate force contributions of single particles described by the SPH-equations in 3D arrays, which represent a spacial discretization. The resulting forces are modeled as accumulations of 3D "foot-prints" of the particles.

Using GPUs for particle simulation performs well for uncoupled particle systems. Up to 1 Mi particles can be interactively simulated and rendered. The high computational costs for keeping track of the spatial neighborhood for dynamic coupling reduces the interactivity significantly.

5 Future Developments

The recent trends in increasing processing power are expected to continue as GPUs continue to drive ever improving graphics for games. A large part of this progress has been improvements in control flow for the vertex and fragment programs in the GPUs, including features such as predication, loops and jumps. A recent change in the graphics pipeline that improves program flexibility is the *unified shader architecture* used in Microsoft's XBox360 GPU. This architecture unifies what was separate physical GPU resources in the vertex and fragment programs and creates

[5] not all neighboring particles are detected

a single large resource for both types of operations. For simulation this can result in potentially better performance as most computations are performed in the fragment program and the vertex program is often idle. Another feature of the XBox360 GPU is a scatter write. This instruction allows the program to write to several dynamically computed addresses in the video memory without the restrictions seen in current GPUs. Similar functionality is to be expected in future GPUs for PCs.

As kernel programs have become more common place in GPU applications the length of these programs has grown. At the same time the number of outputs that can be written to memory in parallel has not increased as rapidly. Future games will tend to have more instructions per fragment and current and future GPUs will take advantage of this increased arithmetic intensity by increasing the number of instructions that can be executed in parallel on the GPU per clock cycle. Recent PC graphics chips have 16 pipelines while the XBox360 GPU has 48.

Future APIs will look towards an increasing focus on standardized floating point behavior by using the IEEE standard for 32 bit floats. This will ensure that simulation results from GPUs closer match to those computed by CPUs. During recent years different generations of GPUs have had varying program sizes and the rapid change in these variations has made it difficult for users to know the underlying capabilities of any particular hardware. This has lead to API's moving towards requiring the hardware to provide virtualization of hardware resources and allowing the API to present the programmer with a programming model that has unlimited resources in terms of instructions and registers.

Modern games are using a form of lighting know as High Dynamic Range (HDR), which typically uses floating point precision to represent light in a scene. As this becomes common place in most games, GPUs will start to change the previous focus on low precision number formats such as 8 bit per channel and increasingly start to optimize for higher precision such as 32 bit floating point. This change in focus will also improve performance for simulation and computation using GPUs.

Acknowledgments

The authors thank Nehal Desai, Patrick McCormick and Wolfgang Wiechert for very helpful comments on the paper.

References

Bolz, J., Farmer, I., Grinspun, E., Schröder, P., 2003. Sparse matrix solvers on the GPU: Conjugate gradients and multigrid. In: ACM Proc. SIGGRAPH.

Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., Hanrahan, P., 2004.

Brook for GPUs: Stream computing on graphics hardware. In: ACM Proc. SIGGRAPH. Vol. 23. pp. 777–786.

DirectX, 2005. DirectX: multimedia application programming interfaces. Microsoft, http://www.microsoft.com/windows/directx/default.aspx.

Gingold, R., Monaghan, J., 1977. Smoothed particle hydrodynamics: theory and application to non-spherical stars. Monthly Notices of the Royal Astronomical Society 181, 375–389.

GPGPU, 2005. General-purpose computation using graphics hardware. www.gpgpu.org.

IEC, Nov. 2000. Letter symbols to be used in electrical technology - Part 2: Telecommunications and electronics. 2nd Edition.

Kipfer, P., Segal, M., Westermann, R., 2004. Uberflow: A GPU-based particle engine. In: Proc. Graphics Hardware. ACM/Eurographics, pp. 115–122.

Kolb, A., 2005. Dynamic particle coupling for GPU-based fluid simulation. In: 18th Symposium on Simulation Technique. To appear.

Kolb, A., Latta, L., Rezk-Salama, C., 2004. Hardware-based simulation and collision detection for large particle systems. In: Proc. Graphics Hardware. ACM/Eurographics, pp. 123–131.

Krueger, J., Westermann, R., 2003. Linear algebra operators for GPU implementation of numerical algorithms. ACM Transactions on Graphics (TOG) 22 (3), 908–916.

Lefohn, A., Kniss, J., Strzodka, R., Sengupta, S., Owens, J. D., Oct. 2005. Glift: An abstraction for generic, efficient GPU data structures. ACM Trans. on GraphicsTo appear.

McCool, M., Toit, S. D., 2004. Metaprogramming GPUs with Sh. AK Peters, Ltd.

McCormick, P. S., Inman, J., Ahrens, J. P., Hansen, C., Roth, G., 2004. Scout: A hardware-accelerated system for quantitatively driven visualization and analysis. In: Proc. IEEE Visualization. pp. 171–178.

Müller, M., Charypar, D., Gross, M., 2003. Particle-based fluid simulation for interactive applications. In: Sym. on Comp. Animation. pp. 154–159.

NVIDIA, 2005. Cloth simulation. http://developer.nvidia.com/object/demo_cloth_simulation.html.

OpenGL, 2005. OpenGL: graphics application programming interface. http://www.opengl.org/.

Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A. E., Purcell, T., Sep. 2005. A survey of general-purpose computation on graphics hardware. In: Eurographics, State of the Art Reports. pp. 21–51.

Rumpf, M., Strzodka, R., 2005. Graphics processor units: New prospects for parallel computing. In: Numerical Solution of Partial Differential Equations on Parallel Computers. Springer, to appear.