

Computation on GPUs: From a Programmable Pipeline to an Efficient Stream Processor

João Luiz Dihl Comba¹

Carlos A. Dietrich¹

Christian A. Pagot¹

Carlos E. Scheidegger¹

Resumo: O recente desenvolvimento de hardware gráfico apresenta uma mudança na implementação do pipeline gráfico, de um conjunto fixo de funções, para programas especiais desenvolvidos pelo usuário que são executados para cada vértice ou fragmento. Esta programabilidade permite implementações de diversos algoritmos diretamente no hardware gráfico.

Neste tutorial serão apresentados as principais técnicas relacionadas a implementação de algoritmos desta forma. Serão usados exemplos baseados em artigos recentemente publicados. Através da revisão e análise da contribuição dos mesmos, iremos explicar as estratégias por trás do desenvolvimento de algoritmos desta forma, formando uma base que permita ao leitor criar seus próprios algoritmos.

Palavras-chave: Hardware Gráfico Programável, GPU, Pipeline Gráfico

Abstract: The recent development of graphics hardware is presenting a change in the implementation of the graphics pipeline, from a fixed set of functions, to user-developed special programs to be executed on a per-vertex or per-fragment basis. This programmability allows the efficient implementation of different algorithms directly on the graphics hardware.

In this tutorial we will present the main techniques that are involved in implementing algorithms in this fashion. We use several test cases based on recently published papers. By reviewing and analyzing their contribution, we explain the reasoning behind the development of the algorithms, establishing a common ground that allow readers to create their own novel algorithms.

Keywords: Programmable Graphics Hardware, GPU, Graphics Pipeline

¹UFRGS, Instituto de Informática, Caixa Postal 15064, 91501-970 Porto Alegre/RS, Brasil

e-mail: {comba, cadietrich, capagot, carlossch}@inf.ufrgs.br

Este trabalho foi parcialmente financiado pela CAPES, CNPq e FAPERGS.

1 Introduction

Recent advances in graphics hardware technology have allowed a change in the implementation of the visualization pipeline used in graphics hardware. Instead of offering a fixed set of functions, current processors allow a large amount of programmability by having the user develop special programs to be executed on a per-vertex or per-fragment basis. This programmability created a very interesting side-effect: it is now possible to interpret the flow of graphical data inside the pipeline as it passes through a stream processor. This leads to the possibility of efficient implementation of a series of different algorithms directly on the graphics hardware.

In this tutorial we will present the main techniques that are involved in implementing algorithms in this fashion. The various examples that will be presented come from recently published papers, and by reviewing and analyzing their contribution, we will enable the attendee both to understand the reasoning behind the development of the algorithms and to create novel algorithms of his or her own. We do not assume knowledge of how fragment or vertex programs work, but we do assume knowledge of the structure of a graphics hardware pipeline. Some working knowledge of linear algebra is also assumed, but the examples presented will be thoroughly examined.

2 History of Graphics Hardware Development

Specialized hardware for computer graphics (referred here as “graphics hardware”) has advanced in an unprecedented rate in the last several years. Part of the evolution can be explained by industry pressure; real-time graphics has a great demand in military and medical applications, for example. The interactive entertainment industry has also pushed much of the development of graphics hardware. Computer games market were evaluated at around eleven billion dollars in 2003 in the United States [23], which is more than the movie industry evaluations.

Technical factors also pushed the evolution of graphics hardware. Many graphics algorithms in the generation of synthetic images are easily expressed in a parallel fashion: hardware implementation of these algorithms becomes very efficient. Moore’s Law also helps: more and more transistors are being packed into a single chip. It is interesting to note that graphics hardware has actually outpaced Moore’s law, because of the easy parallelization of those algorithms when compared with the complexity of scaling up SISD performance.

To give a feel of the evolution of the graphics hardware, we follow with a short presentation of the representative GPU’s of the last decade. The format and nomenclature follows [4].

Pre-GPU's Before the development of the so-called GPU's, specialized graphics hardware was only available in integrated hardware-software solutions provided by companies such as Silicon Graphics (SGI) and Evans and Sutherland (E&S). This hardware was wholly inaccessible to the general public because of the cost involved. Nevertheless, these companies provided the first hardware-based solutions for vertex transformation and texture mapping.

First-Generation GPU's The first generation of consumer-level graphics processors appeared around late 1998 and its main representatives are the NVIDIA TNT2, the ATI Rage and the 3DFX Voodoo3. These processors operated mostly in the rasterization portion of the pipeline, and required previously transformed vertices. Some of the chips supported "multi-texturing", which meant automatic blending of two or more textures in the rasterization step. There was a fixed set of blending functions available to the programmer. These processors had around 10 million transistors.

Second-Generation GPU's The GPU's of these generation were called, at the time (late 1999 and early 2000), the T&L (transformation and lighting) GPU's, because of their main improvement with regard to the previous generation. Here the CPU is freed from an additional burden: the vertices can be passed untransformed, and the GPU takes care of keeping the transformation matrices and making the transformation and lighting calculations. This represented a great improvement for interactive graphics, because these operations are typically performed many times during the rendering of a scene. There are more operations available for the programmer, but no real programmability is yet present. The typical processors of this generation are the NVIDIA GeForce256 and GeForce 2, the ATI Radeon 7500 and the S3 Savage3D. The transistor count for these GPU's was in the order of 25 million.

Third-Generation GPU's This generation represents the first significant change that will allow the interpretation of the graphics hardware pipeline as a stream processor. Now the configurability of the pipelines is enhanced with a programming model that performs vertex computations, and a very limited form of programming at the pixel level. Vertex programs have a limited number of instructions (128) and accept around 96 parameters (each a vector of four floating-points). At the pixel level, the programs are limited by the way they can access texture data, and by the format of the data available: dependent accesses are severely restricted, and only fixed-point numbers are available. Another limitation that appears here is the absence of program flow control: no branches are allowed. The maximum program sizes are also small: programs can have no more than a dozen or so instructions (with small variations depending on the specific GPU). Other features of these GPU's include native support for shadow maps and 3D textures. The main representatives of this generation are the NVIDIA GeForce3 and GeForce4, and the ATI Radeon 8500. The year was 2001 (and early 2002), and the GPU's used around 60 million transistors. Around this time the first steps

towards general computation techniques using stock GPU's were being taken [7].

Fourth-Generation GPU's This generation represents the current state of the art in graphics hardware. Here we see a significant generalization of the fragment and vertex programmability present in the previous generation. Pixel and vertex programs now can comprise thousands of instructions, and the limitation on the usage of texture data is mostly gone. Dependent texturing are now more flexible, which is the use of texture data as index for other lookups. This is the basic technique for the implementation of data structures later on. The GPU's now have floating-point units, and textures are no longer clamped to the $[0.0, 1.0]$ range. This enables their use as ordinary arrays, and is the other main contribution that enables general computation. The main representatives of this generation of GPU's currently are the NVIDIA GeForce FX series and the ATI Radeon 9700 and 9800, with more than 100 million transistors each.

In this course, we will focus on fourth-generation GPU's, because the difference in programmability between earlier and current processors are quite big. And while it is possible to perform certain computational tasks using earlier GPU's, there is a very limiting lack of precision in the data formats available and unneeded complexity involved [7], [8].

3 High-level Shading Languages

It is hardly argued that the most significant advance in consumer-level graphics hardware is the programmability of the graphics pipeline. This is achieved by replacing, as already explained, portions of the pipeline with specialized programs. A critical issue that arises is the need to describe these programs. Assembly languages are typically provided by vendors, but they have well-known portability issue and are cumbersome to program.

Each different GPU has a different instruction set. This is complicated by the fact that since the GPU's are different, not all programs that are compilable in one GPU are in the other. Additionally, this problem should be tackled in a vendor-independent way. Some solutions have been proposed, each with their own tradeoffs. In the Direct3D HLSL, for instance, a program is previously compiled to an intermediate GPU-independent assembly language, and from that to the target language. The main advantage is the simplicity of implementation. On the other hand, certain optimizations are not performed because the GPU instruction set is shadowed by the intermediate step. This leads to other two solutions proposed. One of them, used by NVIDIA in its Cg language, is to use different *language profiles*, mechanisms that provide a way to probe the processor capabilities and to configure the compiler to produce different outputs. This eliminates the need for the intermediate step, but requires potentially different programs for different profiles.

The last proposal, used by 3D Labs in its GLslang (GL Shading Language) [15], is to embed the programmability directly in OpenGL. The GLslang will be used directly through the OpenGL calls, and the compiler will be hid from the user by the runtime. This enables each driver to provide its implementation, which is the ideal amount of decentralization, as each vendor can implement the features in its own way, provided that the interfaces are respected (concerning errors and compilation, for example).

In this section we will take a quick look at some proposals for high-level shading languages. These languages work at different levels of abstraction, and have some interesting differences that are worth mentioning.

3.1 Stanford shading language

The Stanford Shading Language [17] was one of the first high-level shading language designed for real-time graphics. Its design used ideas from many languages, but the biggest influence was undoubtedly the RenderMan shading language [20]. The Stanford shading language revolves around different *computation frequencies*. Procedural shading elements can be associated with primitives, vertices or pixels, and the language takes care of providing a unified interface in which the programmer can mix operation from different frequencies.

One interesting aspect of the Stanford shading language, and one that separates it from the other languages presented here is that the language takes care of virtualizing the graphics resources. This means that if the graphics card is unable to perform a certain operation that is present in the shader, this operation is delegated to the host CPU. This convenience is important if the user wants to have the shader operation no matter what, even with a significant decrease in performance. This makes it somewhat unsuitable for our work here, since we want to exploit the performance advantages of the GPU as much as possible, and we are willing to develop a different (but equivalent) shader that will run on the GPU.

3.2 C for Graphics (Cg)

Cg is a high-level shading language designed for the programming of GPUs [1], developed by the NVIDIA Corporation. Cg attempts to preserve most of the C language semantics. In addition to freeing the developer from hardware details, Cg also has all of the usual advantages of a high level language such as easy code reuse, improved readability and the presence of an optimizing compiler.

The Cg language is primarily modeled after ANSI C, but adopts some ideas from modern languages such as C++ and Java, and from earlier shading languages such as RenderMan and the Stanford shading language. These enhancements and modifications that make it easy to write programs that compile to highly optimized GPU code, as well as increasing

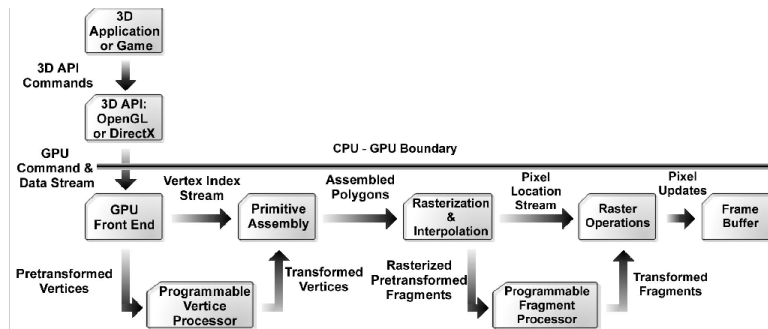


Figure 1. The Cg model of the GPU (figure from [4])

readability (through overloading, for example).

The Cg design considers some aspects of the GPU architecture, like the programmable multi-processor units (the vertex processor and the fragment processor, plus other non-programmable hardware units). The processors, the non-programmable parts of the graphics hardware, and the application are all linked through data flows. Figure 1 illustrates Cg's model of the GPU. The Cg language allows to write programs for both the vertex processor and the fragment processor. NVIDIA refers to these programs as vertex programs and fragment programs, respectively. (Fragment programs are also known as pixel programs or pixel shaders, and we use these terms interchangeably in this document).

The Cg API introduces the concept of profiles, to deal with the lack of generality in the vertex and fragment programming. A Cg profile defines a subset of the full Cg language that is supported on a particular hardware platform or API. Cg code can be compiled into GPU assembly code, either on demand at run time or beforehand. Cg makes it easy to combine a Cg fragment program with a handwritten vertex program, or even with the non-programmable OpenGL or DirectX vertex pipeline. Likewise, a Cg vertex program can be combined with a handwritten fragment program, or with the non-programmable OpenGL or DirectX fragment pipeline. In this tutorial, we'll use Cg as the language of choice to describe the programs that will be implemented.

3.3 Microsoft HLSL

One of the most empowering new components of DirectX 9.0 is the High Level Shading Language (HLSL). Shaders were first added to Direct3D in DirectX 8 [3]. At that time, several virtual shader machines were defined - each roughly corresponding to a particular

graphics processor produced by each of the top 3D graphics hardware vendors. For each of these virtual shader machines, an assembly language was designed. In DirectX 8.0 and DirectX 8.1, programs written to these shader models (named vs_1_1 and ps_1_1 through ps_1_4) were relatively short and were generally written by developers directly in the appropriate assembly language. In DirectX 9.0, the application passes an HLSL shader to D3DX via the `D3DXCompileShader()` API and gets back a binary representation of the compiled shader which is in turn passed to Direct3D.

In addition to the development of the HLSL compiler in D3DX, DirectX 9.0 also introduced additional assembly-level shader models to expose the functionality of the latest generation of 3D graphics hardware. Application developers can feel free to work directly in the assembly languages for these new models (vs_2_0, vs_3_0, ps_2_0 and ps_3_0).

In fact, Cg and HLSL are actually the same language, mainly due to its development by NVIDIA and Microsoft consortium. They have different names for branding purposes. HLSL is part of Microsoft's DirectX API and only compiles into DirectX code, while Cg can compile to DirectX and OpenGL.

3.4 OpenGL 2.0 shading language

The OpenGL 2.0 proposal, developed by 3DLabs and recently submitted for comment by the OpenGL ARB, includes a specification for a shading language [15]. The syntax of the shaders is very similar to the previous three languages described, but the really interesting aspect of this language is that, because of its inclusion in the OpenGL standard, it is probably the first shading language that will be adopted in a truly cross-platform way.

The main difference between GLSLang and the other languages is that the compiler will be more tightly integrated in the graphics pipeline, hopefully providing an easier to use interface. 3DLabs claims that to use GLSLang, it won't be necessary to link to additional libraries or use specific development tools. This can be an important advantage for the developer. Unfortunately, we haven't had access to complete implementations of GLSLang at the time of writing.

4 Case studies

General-purpose computation using graphics hardware is a rapidly growing field. In this section, we hope to present some of the most important works to date. We will start with very specific examples, which illustrate basic techniques, and work our way to more general ones. During the presentation, the main methods to translate a problem to a GPU-based one will be highlighted. This will enable the reader to apply the techniques in other problem domains.

The first cases use the GPU to implement ray tracing. Even though this is a computer graphics algorithm, it is important to see which parts of the algorithm are given to the GPU and the CPU, and to understand the fundamental differences in the approaches. The following cases are iterative solvers, which introduce the concept of *feedback*. By reusing the results of the computations and input, we are able to express many more problems using the tools available to the GPU's. The remaining cases concern general linear algebra solvers, and the main concept used is that of indirect texture access. This technique is used in the implementation of data structures using textures, and allows much more complex algorithms to be cast in a GPU fashion.

4.1 GPU-based Ray Tracing

Until recently, limitations in computing power have forced real-time synthetic image generation to methods that explore image and object-space coherency. Typically, scan-based rasterization with local illumination models is used in these situations. Ray tracing offers a completely different approach. Ray tracing allows more sophisticated models by solving the radiance equation at a single point under reasonably realistic assumptions. For example, it can be used to realistically simulate shadows, reflections and refractions in a general way while approximate and specific solutions are needed in rasterization based algorithms, such as shadow mapping, shadow volumes and environment mapping.

Ray tracing can easily produce images with higher quality, mostly because complex interactions between the light and the scene can be simulated. On the other hand, ray tracing has a reputation for being hopelessly inefficient in real-time applications. In this section we will describe techniques that take advantage of the programmability of current graphics hardware, allowing a substantial speed-up in ray tracing applications that brings ray-tracing closer to real-time applicability.

The implementation of the rasterization pipeline in hardware allowed developers to employ more sophisticated rasterization-based techniques such as the ones described above, while keeping the frame rate well above real-time constraints. As we have seen, advances in VLSI processes and demands for higher flexibility in the pipeline caused the abandon of the fixed-function structure in favor of programmable vertex and pixel units. This change has inadvertently allowed the implementation of entirely different synthetic image generation algorithms. We will examine ray tracing here, but other techniques have also been implemented, such as radiosity [2] and photon mapping [19].

We will examine two proposals for hardware-based ray-tracing. The first one [3] relies directly on pixel programs for ray-triangle intersection tests, somewhat limiting the use of the GPU to what it does most efficiently. The latter [18] base their implementation in simulators that incorporate features currently unavailable on GPU's and is discussed mostly

for illustrative purposes of the expected direction in development of the GPU's.

4.1.1 The Ray Engine

Defining the Role of the CPU and the GPU The ray tracing algorithm can be divided in four distinct phases: ray generation, ray intersection and point shading. Ray intersection assessment is classically the most expensive step in computational terms, and, as such, is usually preceded by an acceleration structure traversal of some kind, where as many ray intersection tests as possible are avoided.

We have seen that current graphics hardware can be interpreted as a stream processor. As such, it lacks some typical features of regular CPU's. The critical missing feature here is the control stack. This imposes some severe constraints on a direct ray tracing implementation on a CPU, because ray tracing is very easily formulated as a recursive procedure. Path tracing uses ray tracing and is not necessarily recursive, but we will deal exclusively with classic ray tracing.

This first example uses the GPU exclusively for ray-triangle intersection tests. This means that the ray tracer implemented is not very general: regular ray tracers can be implemented to support arbitrary geometric structures, defined by many different shapes, including implicitly defined surfaces and cubic spline patches [16]. The restriction of polygonal meshes is not stringent, though. Most ray-tracers designed with strict performance requirements deal only with triangles. The remaining steps of the ray tracing algorithms are left to the CPU. Having decided the role of the GPU, we must now turn to the next question: how do we go about implementing these tests efficiently?

Choosing and Implementing a Ray-Triangle Intersection Computation The pipeline of a modern programmable GPU is customizable in two different stages of the pipeline. The vertex programs replace the view volume normalization stage just before the primitive assembly, and the fragment programs replace the lighting and shading calculations of the (typically) Gouraud-interpolated, phong-shaded polygons.

It is possible, in principle, to implement the test in either stage. Because of its intrinsically parallel nature, fragment programs are able to process more than ten times as many fragments as the vertex programs can process vertices. The authors chose to do it in the fragment program because of the precision, performance difference and the possibility of cheap access to texture data. The vertex program would have the advantage of allowing more data to be output (the fragment program currently can only produce $\langle R, G, B, A \rangle$ values and a depth component). After considering different intersection tests, a modified version of the Möller test [12] was chosen.

The Möller test returns the parameter t of the intersection between the ray and triangle plane. It can be calculated as:

1. $ao = o - a$
2. $bod = bo \times d$
3. $bo = o - b$
4. $u = ac.aod$
5. $v = -ab.aod$
6. $aod = ao \times d$
7. $w = cb.bod$
8. $t = -(n.aod/n.d)$

where

- a, b, c — the vertices of the triangle
- n — front facing triangle normal
- o, d — ray origin and direction

Additionally, some values (ab , ac , cb) are constant per triangle and are evaluated of-line for efficiency purposes. The result of the intersection is stored in the framebuffer, and will be sent back later to the CPU for actual shading of the fragment or to respawn secondary rays. Since each triangle may have different material properties, and hence will interact differently with the ray, there must be a way to identify the closest triangle intersected by each ray. This is done by the standard technique of storing a unique identification number as a color that will be stored in the framebuffer. We have the following parameters to be passed to the GPU: a, b, c, ID, o, d . For additional performance, ab , ac , cb and n are computed inside the vertex shader and passed into the fragment shader.

The output data should be packed in a suitable and compact form. Special care had to be taken when defining this packing format, mainly because the output data should be sent back to the CPU, and would consume AGP bandwidth. Considering that AGP is an asymmetric port, presenting a narrower bandwidth on the GPU to CPU direction, it can significantly slow down the entire process if the amount of data to be sent through it is large. At the same time, the output data must have the minimal information ID, o, d , necessary to enable the

CPU to compute the shading calculations from it. The data elected to be returned by the GPU, for each ray tested, are: the color of the first triangle the ray intersects (the triangle's ID), the result of intersection in the alpha channel and the t-value []. The question now is: how to send and retrieve those parameters to and from the GPU?

Passing and Retrieving Parameters from the GPU The format in which the input parameters should be packed to be sent to the hardware should be defined. The authors decided for packing ray's data into texture units and triangle's data as properties of the four vertices of a screen filling quadrilateral (quad).

Ray data was packed into two screen resolution textures. The first texture stores the coordinates of the origin of each ray as RGB values. The second texture stores the direction of each ray as RGB values. Textures can be directly accessed at the fragment program level by texture samplers that wrap the GPU texture unit functionally.

Triangle data are stored as properties of the four vertices of a quad that covers all screen area. As the process of rasterization progresses, each pixel of the screen is evaluated, and the quad vertices' properties are properly interpolated. As the property values of those four vertices are equal, the linear interpolation process just replicate those values across each pixel of the screen.

The quadrilateral texture coordinates related to texture 0 (zero) have a special meaning. Those coordinates are different at the four vertices of the quad (they are set to (0, 0), (0, 1), (1, 0), (1, 1)) and, when interpolated across the quad, they enable the pixel shader to address the two textures units containing ray data.

The output data is stored in the framebuffer. For each tested ray, there are three values that must be returned: color of the triangle (triangle ID), result of the intersection computation and the t-value. All those values were packed, for each ray, as an unique vector of the form $\langle R, G, B, A, depth \rangle$.

Putting Everything Together It was defined, in previous sections, the rule of the CPU and the GPU in this particular ray tracing implementation based on their properties. It was shown how an efficient ray-triangle intersection computation can be successfully implemented into the GPU, and how to send and retrieve parameters from it.

Now it is time to an overview of the entire system. It is obvious that just accelerating the ray-triangle intersection through the use of the GPU is not sufficient to really improve the performance of the ray tracing, and additional techniques can be used. The reduction of ray-intersection tests to be computed and use of coherency are good points to be observed.

The ray engine uses an octree to keep the geometry coherence and a 5-D ray tree to exploit ray coherence. A ray cache, composed by small buckets of rays, is maintained by the

CPU to better utilize the texture cache of the GPU. Each bucket contains 256 rays, organized as two textures 64×4 , that store the values of origin and direction of those rays. Those small textures are better handled by the GPU, and are kept inside the texture cache during its use, accelerating the intersection computations.

As they are created, new rays are stored inside those buckets. Each time a bucket is filled, and a new ray is added, this bucket is split into four new buckets. Once the ray cache becomes filled with rays, or the GPU finishes previous calculations, the filled buckets are sent to the GPU. During the intersection computations on the GPU, the CPU can calculate the shading of already returned intersected points, add new rays to the buckets, etc.

4.1.2 Ray Tracing on PGH A different approach was chosen by [18][19], where they decided for implementing the entire ray tracing algorithm into the GPU. As GPUs had serious limitations concerning floating point precision and control flow, extended GPU simulators were used to test and generate the results.

In this method, the ray-tracing process was broken into four main kernels: ray generation, structure traversal, ray intersection and point shading. The ray generation kernel is responsible for the generation of rays based on the camera description and for the initial intersection test between these new rays and the bounding box of the scene. The structure traversal kernel is responsible for searching triangles inside the scene structure for future intersection tests. The intersection kernel receives (one ray, n triangles) pairs and computes the intersection of those rays against their respective triangles. The shading kernel is responsible for computing the shade of an intersected point or for the respawn of new rays, that will be shoot against the scene again. Each one of the four kernels was implemented into the GPU through a separate pixel shader program.

As ray tracing involves significant looping, two different extended GPU architectures were simulated for the tests: a multipass and a branching one. The branching architecture was allowed to loop and to branch, while the multipass architecture wasn't. One can simulate looping in the multipass architecture through the use of multipass rendering, whereas the flow control is simulated using the stencil buffer.

To improve performance, a structure to keep geometry coherence should be used. The authors decided for using a 3D uniform grid (implemented by a 3D texture) to divide the scene into voxels, for two main reasons: first, uniform grids can be more easily handled by actual hardware (that does not support hierarchical structures), and second, due to efficiency gains.

Each voxel of the scene may contain scene geometry. If this is the case, the respective texel will contain a pointer to the beginning of a triangle list. If it's not, its content will be null. Triangle lists are arrays of pointers to triangle data, and they are all stored in a one-

dimensional texture. Each triangle list is separated by a null element in the 1D texture. The vertices of each triangle are spread over three 1D textures.

Purcell et al. demonstrated that various ray tracing techniques could be successfully implemented through the use of those simulators, including: shadow cast, ray casting, ray tracing and path tracing. As the solution presented required a lot of inexistent extensions in the GPU, its practical implementation is not possible at the moment, but the article can be viewed as an interesting study about GPU structure, and presents a series of interesting extensions that could be found in future releases of those processors.

4.2 Iterative Solvers

In this section, the two papers that will be presented use the same main technique to achieve their results. By carefully using *feedback*, that is, by interpreting the rendered results as inputs to another step of the simulation, a much larger set of problems can be tackled. This feedback is achieved by copying the framebuffer to a texture or by directly rendering to textures. This later feature is only available in OpenGL through the use of extensions, but Direct3D offers it natively. One straightforward use of the feedback technique is to iteratively solve differential equations, and the time integration is done by using the result of the last time step as input to the next one. As we'll see, this is close to what happens in these examples.

4.2.1 CML's This work [7] demonstrates the use of graphics hardware to implement simulations of physical phenomena in a simple but visually accurate way. The simulation is achieved using *coupled map lattices* [10], which are a simple extension of cellular automata [22] where the cell states are real values, and the next-state computation is an arbitrary arithmetical expression. CML's have been used to simulate many different physical phenomena, and the ones we'll discuss here are the ones used in the paper, namely boiling and reaction-diffusion.

This work is particularly interesting because the operations that are to be performed during a given simulation using CML's have a very close correspondence with the kind of fragment-level programming that is used in most of the examples we'll see. At the same time, CML's don't involve any particularly difficult mathematical modelling that may obscure the programming techniques employed. For example, the representation of the state of a given CML in hardware is simply a texture with appropriately valued channels, not requiring any specific data structure or manipulation.

The authors divide a single step of the CML computation in four stages that are simple and broadly applicable techniques for GPU computation, and so we will follow their presentation here almost exactly. The main difference is that the authors divide the stages of neighbor computation and new state computation, because of the hardware features at the

```
glViewport(0, 0,  
           lattice->width(),  
           lattice->height());  
  
glOrtho(0, lattice->width(),  
        0, lattice->height(),  
        -1, 1);
```

Figure 2. Setting up the projection for next-state computation

time the paper was written. Since we are not interested in this historical perspective, and currently it is much easier to interpret and implement these steps as a single one, we will treat them as such.

Next-state computations For now, we'll suppose that we have the current state of the CML in a texture, and will consider the problem of obtaining the value of the next-state. Since we are interpreting the GPU as a stream processor and the CML simulation involves continued iteration of functions, we must eventually find a way to feed the results of a computation back to the beginning of the stream. This is a simple operation, and will be described later.

The authors use the frame buffer as a temporary placeholder for the next state of the CML. This means that we'll render the CML directly to the screen, and use fragment-level programming to do the computation. This also means that each texel in the texture must be mapped exactly to a pixel in the frame buffer. A simple orthographic projection will take care of that, as shown in figure 2.

Neighbor Sampling The first stage described is the one in which we determine the value of the cells in the local neighborhood of the cell we are computing. This is done by offsetting the texture coordinates in the direction of the neighbors. Some simulations require a weighted average of the neighbors and the local value, and this can be done without additional computation, using the interpolation capabilities of the texturing hardware. This blurs the distinction between the stages of sampling and computation, but the technique is highly useful, since computing the average comes at no cost at all.

Computation on Neighbors and New State Computation After having determined the values of the state in the neighbors, we have to compute the next state of the local cell using fragment-level programming. In the original paper, this is not entirely trivial because of the need to use texture shaders and texture combiners, two GeForce3-specific techniques at the time. Since in this tutorial we'll be using Cg, the two stages can be fused into one, and the clarity of the operation is much improved. A simple fragment program that computes

the diffusion step of the boiling simulation using only simple arithmetic is available in the additional material provided, as an illustrative example.

In case we need to use more complex functions that are either not available on the GPU or too expensive, we can employ a very easy and very powerful technique. We trade a potentially expensive set of operations for a simple texture lookup, at the expense of a texture sampler. Previously to the iteration cycles, we compute the complex function across the portion of the domain that we'll use and store it in a texture. In the GPU code, the complex function is then traded for a texture look-up. In a sense, the earlier stage of sampling the neighbors can be interpreted as looking up a potentially complex function, possibly in multiple dimensions. This will be exactly the case in the other case studies, where we will turn the GPU into general numerical PDE solvers. The more general technique of using texture values as indices for other textures is the *dependent texturing* technique described earlier, and will be put to extraordinary use in the case studies. The buoyancy operator described in the original paper uses this very technique through the `DEPENDENT_GB_TEXTURE_2D_NV` GeForce3-specific (at the time) texture shader, and a fragment program that computes it in Cg is also available in the additional material.

State Update After the fragment programs are executed throughout the entire texture, the frame buffer will contain the next state of the CML computation. We must now find a way to feed these results back to the original texture. There is a simple OpenGL call that does just this: `glCopyTexSubImage2D`. This is a very efficient operation in the case where there is no transfer of data across the graphics bus or port, which is what happens here, copying from the frame buffer to a texture.

4.2.2 Cloth Simulation This example will show an implementation of a cloth simulation using only graphics hardware, which includes some interaction with scene geometry [13]. It will serve as an illustration of the methods described above — specially the feedback from the frame buffer — and, although it is not portrayed in this way, it should be clear to the reader that the CML model discussed earlier serves as a very suitable model for this simulation.

A cloth patch can be represented as a bidimensional array of 3D points. These points represent a discretization of the real piece of cloth, and will be subject to a series of constraints that simulate the real world, such as collision between objects and distance constraints between neighboring points of the mesh, in addition to being subject to gravity.

The gravity-induced acceleration will be integrated using Verlet integration [21]. This method was chosen because it eliminates the need for explicit velocity storage. Verlet integration is more stable than explicit Euler integration, but there are better integrators in terms of stability (RK4 is the usual integrator used). An additional problem with Verlet integration is the subtracting of two terms of approximately the same magnitude, which can result in se-

```
void Integrate(inout float3 x, float3 oldx,  
              float3 a, float timestep2, float damping)  
{  
    x = x + damping*(x - oldx) + a*timestep2;  
}
```

Figure 3. Verlet time integration in Cg

vere loss of precision in floating-point operations. On the other side, Verlet integration makes the treatment of constraints rather easier. Using Verlet integration, velocity is estimated from the current and previous positions only. Since there is no explicit velocity, when a collision is detected only a recomputation of the correct position is needed. This is important here, as we will simulate the cloth interaction with the environment using fragment programs, and as we have shown earlier, fragment programs have no way to write more than a single value in one execution. So, storing the velocity would require an additional pass and fragment program, and would make the program clumsier. The implementation of the simulator follows closely [9].

Representation The state of the simulation will be represented using two floating-point textures. The first one will store the actual positions of the 3D points, and the second one will store the normals that will be used for the lighting model. The normals are only needed for the visualization, and will not take part in the simulation itself.

Time integration Time integration is done by a fragment program that will update the position according to the forces being applied at the points of the cloth patch and the previous and current positions. In this case, only gravity is considered, so the total acceleration on a body is simply the acceleration vector. The integration step is shown in figure 3.

In the code, `a` represents the acceleration vector, `timestep2` represents the square of the timestep, `x` and `oldx` are respectively the current and previous position of the vertex, and `damping` is a damping coefficient to prevent instability.

Constraints In this simulation, we have three constraints that must be met. Two of them concern interaction of the cloth with the environment, and the last one concerns the maintenance of the cloth integrity. The first two will be implemented simply checking whether the points are invading the solid volumes and updating them accordingly. The floor constraint is trivial (Figure 4).

The sphere constraint is also straightforward: the point is inside the sphere if the distance between the point and the center of the sphere is smaller than its radius. To enforce

```
void FloorConstraint(inout float3 x, float level)
{
    if (x.y < level) {
        x.y = level;
    }
}
```

Figure 4. Floor constraint in Cg

the constraint, the point must be displaced outward from the center a distance equal to the portion of the sphere it has penetrated (Figure 5).

```
void SphereConstraint(inout float3 x,
                    float3 center, float r)
{
    float3 delta = x - center;
    float dist = length(delta);
    if (dist < r) {
        x = center + delta*(r / dist);
    }
}
```

Figure 5. Sphere constraint in Cg

The final constraint is the trickiest. Since we are simulating a piece of cloth, each point must be kept at the initial rest distance of each other. This constraint is approximated by summing the displacements created by each constraint — the left, right, up and down neighbors. The sum is then scaled down by a stiffness value, that serves to stop big displacements, as these can throw the Verlet integration out of stability. In the following code, `x` is the central point, and `x2` will have the values of each of the neighbors. `restlength` is the expected rest length of the joint, and `stiffness` is self-explaining (Figure 6).

The constraints have to be applied in a proper order, so that the user doesn't see the cloth temporarily penetrating the floor or the sphere. We will have two passes, then: the first will integrate the time, and the second will satisfy the constraints, as follows.

Normal computation The first texture is now being updated properly. On the GPU side, what is left now determining the normal of the cloth pieces. This is done by computing the normalized cross product between the vectors formed from each the current point to the vertical and horizontal points.

```
float3 DistanceConstraint(float3 x, float3 x2,  
                        float restlength, float stiffness)  
{  
    float3 delta = x2 - x;  
    float deltalength = length(delta);  
    float diff = (deltalength - restlength) / deltalength;  
    return delta*stiffness*diff;  
}
```

Figure 6. Distance constraint in Cg

Visualization We have now an interesting problem. The textures describe the cloth patch appropriately, and we are able to simulate the behavior of the cloth and its interaction with the environment as time passes, but a fundamental piece of the puzzle is missing: how do we visualize the results? A naive solution would be to read the textures back to the CPU memory using a direct `glReadPixels` call. However, this would largely defeat the original purpose, that is to alleviate the CPU from the entire process. Additionally, we know that this memory transfer can be problematic if stringent real-time constraints are present.

The solution proposed involves a currently proprietary OpenGL extension. Fortunately, this extension is expected to be included in the ARB specs [6]. The procedure is similar to the initial idea. Since we were going to use the CPU simply as a way to interpret that textures as series of vertices and normal values, there should be way to do just that, without resorting to copying data through the graphics port. The NVIDIA *pixel data range* extension [14] allows just that. By specifying a vertex array range using that extension, we can do a `glReadPixels` directly from the texture to the vertex array, avoiding a round trip to and from the CPU.

This final part is probably the most interesting part of this example. Now, along with being able to perform iterated computation on textures, we can interpret them as arbitrary geometry, which means that, for example, the CML techniques can be used in different problem domains. One of the possibility this creates is the one of implementing particle systems directly on the GPU.

4.3 General Linear Algebra Solvers

The next two examples will raise the abstraction level we are dealing with. Instead of coding each problem into a different algorithm, the following two papers describe the implementation of general linear algebra methods that are applicable in a wide range of simulations. The two papers specifically illustrate their data structures by implementing solvers

```
main(...)  
{  
    // ...  
    // sample x1, x2, x3, and x4, the point neighbors  
  
    float3 dx;  
    dx = DistanceConstraint(x, x1, constraintDist, stiffness);  
    dx = dx + DistanceConstraint(x, x2, constraintDist, stiffness);  
    dx = dx + DistanceConstraint(x, x3, constraintDist, stiffness);  
    dx = dx + DistanceConstraint(x, x4, constraintDist, stiffness);  
  
    x = x + dx;  
  
    SphereConstraint(x, center, r);  
    FloorConstraint(x, level);  
}
```

Figure 7. Putting the constraints together

for finite-difference equations. These solvers require three different classes of operations:

- Vector-vector operations, such as element-wise sum, difference and multiplication
- Sparse matrix-vector operations: Multiplication of a sparse matrix by a vector is the typical operation. Dense matrices are seldom used, because the typical matrix that appears in the discretization of differential equations is by nature sparse.
- Vector reductions: Vector reductions are a class of operations which reduce a vector into a scalar, by applying a binary associative operator to all elements of the vector. This is used, for example, to compute the inner product between two vectors, where first we compute the element-wise multiplication between the vectors and then reduce the result using a sum.

The first example we'll examine [11] explores a particular layout present in most sparse matrices that arise in physics-based numerical simulation, that of being banded or nearly-banded. The second example [1] employs a more general technique of rows of sparse matrices into matrices and executing different fragment shaders for rows with a different number of elements. In showing these two data structures, we'll explore a range of techniques for mapping CPU constructs to the GPU.

Linear Algebra Operators for GPU Implementation of Numerical Algorithms We'll start describing the implementation of the data structures involved. We basically must describe vectors and sparse matrices, and since vectors are simpler structures, we'll begin there.

Intuitively, one could think about implementing vectors directly as one-dimensional textures. This straightforward solution has the advantage of extreme simplicity, but is limited in some ways. First of all, we should know that the size of a texture is limited by the maximum width and height allowed in the hardware implementation. The maximum size of a 1D texture is then bound linearly by the maximum width representable in the graphics board. 2D textures, on the other hand, have their maximum size bound by the square of the width. This allows a much larger vector size. Secondly, square textures are rendered faster in graphics hardware. This way, we'll represent vectors as 2D textures and convert the coordinates in the fragment programs that will manipulate the structure.

The design of the sparse matrix data structures involves an observation concerning the typical layout of non-zero entries in the matrix that are to be used in the implementations. Typically, matrices that are created from the discretization of differential equations are banded, typically with a low bandwidth. This means that most of the non-zero values are very near the diagonal of the matrix. The authors exploit this fact by storing the matrix not as a set of columns or set of lines, but as a set of diagonal vectors. Each of these vectors is stored as described in the previous paragraph, but instead of laying them out as columns, the authors move the diagonals under the main diagonal to the right of the matrix, achieving a "skewed" layout like that of figure 8:

This way, if one of the diagonals (which, apart from the main diagonal, actually now represent *two* matrix diagonals) has no non-zero values, it doesn't have to be stored at all. Since we know the matrices are likely to be banded, this represents a potentially big gain in space. A matrix with bandwidth 2 needs only three vectors for its representation, for example.

The case of more general sparse matrices needs to be handled in a different way, because we no longer have the guarantee of the diagonal coherence. The solution suggested by the authors is to abandon the diagonal layout and turn to a more traditional column-based layout. Instead of arranging each column as a vector represented as a texture in the way described earlier, the columns will be represented as individual vertex arrays that will be rendered as points. Vertex arrays are a feature present in all modern graphics cards which enable the programmer to store geometry in the GPU memory [NVIDIA or OpenGL]. The vertex position will be the position of the cell in the matrix, and its color will be the value of the matrix cell. We will also store here texture coordinates, that will be used in a way explained later.

Now that we know the representation of the structures that will be used, let us turn to how the operations themselves are implemented, starting with the simpler vector-vector

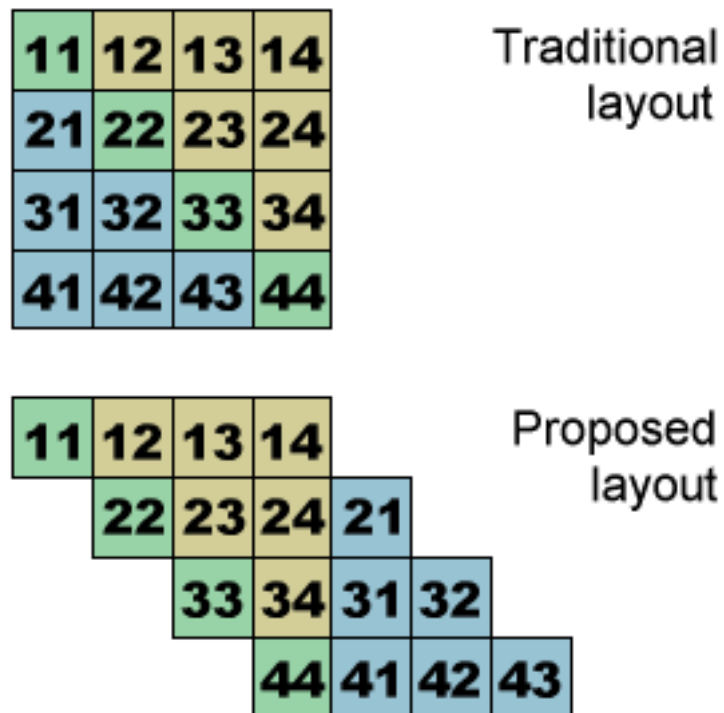


Figure 8. Moving the sub-diagonals, creating a skewed “diagonal matrix”

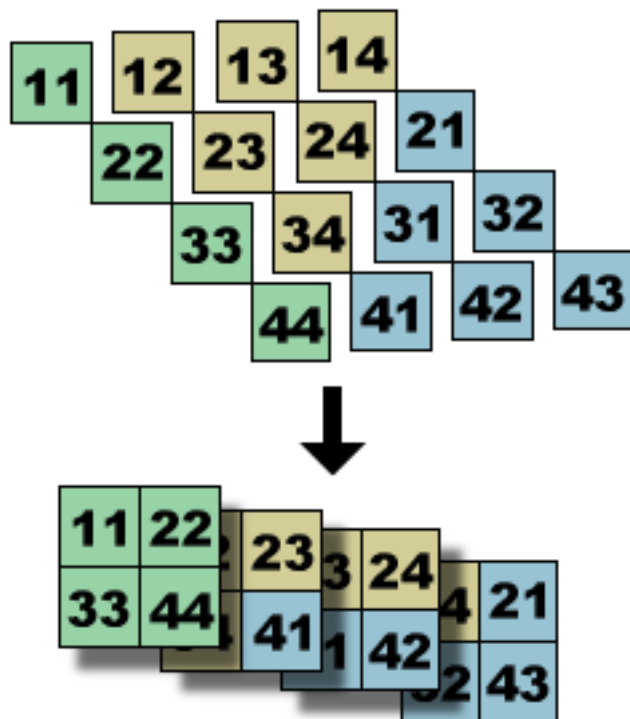


Figure 9. The matrix is represented as a set of diagonals, and the diagonals are themselves represented as vectors, in the way described above.

operations. Vector-vector operations use very simple fragment programs that operate over each fragment performing the desired operation. The operations that the authors describe can scale the input vectors arbitrarily before applying the operations. This is also very simple. The result of the renderization is the new vector. No fancy texture coordinate calculation is needed, because both vectors share the same layout, so that the right elements are already aligned.

Vector reduction operations work on a single vector, and must accumulate all the vector values into a scalar using some operation. In general the reduction operation need not be commutative, but this implementation imposes a commutative restriction on the operation. The way this implementation work is to repeatedly turn a $N \times N$ texture representing a vector into a $N/2 \times N/2$ texture, where each new element corresponds to four elements accumulated using the operation supplied. This new texture is again reduced using the same technique, and this is done iteratively until all values are accumulated into a 1×1 texture. The value of the single pixel will correspond to the final value of the reduction operation. The whole reduction takes $\log n$ passes, and not more than $4/3 * n^2$ pixel operations, being reasonably efficient. The way this operation is implemented illustrates a key missing feature in current graphics hardware: there is no accumulator register present that can be used concurrently by the different fragment program executions. Were this the case, the implementation would work in a single pass. This illustrates a bottleneck that is created when an algorithm is translated to streaming style: the data-parallel operations become more efficient, but if reduction operations are a significant part of the algorithm, the translation might not be efficient. Occlusion queries [14] can be used as described in [5] to simulate the determination of the L_{inf} norm in a single pass. Similar techniques may be used in restricted cases.

Finally, we have to implement the sparse matrix vector multiplication. When a sparse matrix is represented by sets of diagonal vectors, a matrix-vector multiplication can be seen as the vector sum of a series of vector-vector element-wise multiplications with an appropriate shifting of indices. This is much clearer to understand by looking at the figure 11.

The other case to be treated is that of a general sparse matrix, which, as we've seen, is represented using vertex arrays. These arrays are stored in a column-based layout. The matrix multiplication is performed in a similar way, but are simpler in one important way: instead of having to shift indices around to achieve the appropriate skewing of the matrix, the authors use the an additional parameter for each vertex, passed as a texture coordinate that was mentioned above. This texture coordinate is used to directly access the corresponding vector element for element-wise multiplication, so that the fragment program does not need to compute the appropriate indices. Since the sparse matrix is computed offline, this is possible and incurs little overhead.

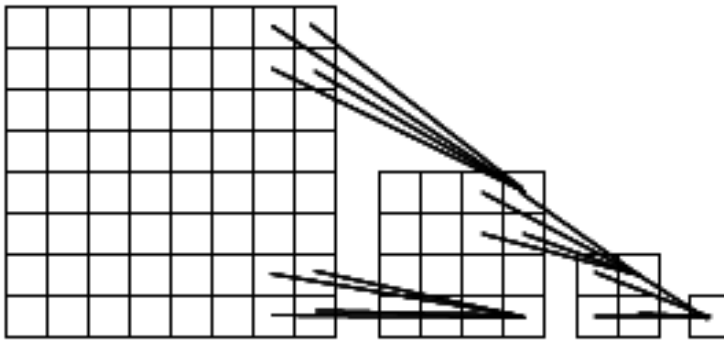


Figure 10. Blocked reduction: every 4 elements are “joined” using a reduction operation, and the procedure works iteratively until we’re left with an 1x1 texture.

Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid The other paper we will describe differs mainly in the implementation of the sparse matrix data structure. Instead of arranging the non-zero elements in the diagonals as vertex arrays, or skewing the matrix so that it is represented in terms of sets of diagonals, here the authors will make extensive use of indirect access through dependent texturing. First, the diagonal of the matrix is stored in a texture by itself, which is the same approach mentioned above. Then, all non-zero off-diagonal entries of each row the matrix are packed in a single texture, and additional textures that work as pointers both to this packed texture and to the diagonal texture are created to allow the multiplication to be performed efficiently. We will now turn to the explanation of these indirection textures.

The technique used by the authors involves interpreting the texture values not as scalars or intensities or even vectors, as is usual in many algorithms. Here, the values of indirection textures will be interpreted as texture coordinates to be used in looking up other values, acting effectively as pointers to the textures containing the matrix values. This is an example of the dependent texturing technique described earlier, but here it is used to create data structures, and not directly other functions. We will follow the authors convention of superscripting each texture with a letter. Two different textures sharing the same superscript also share the same layout: texture coordinates that can be used to access one element of one texture can be used to access a natural correspondent element of the other. A sparse matrix will consist, then, of four separate textures:

- A texture A_i^x , which will store all the diagonal elements of the matrix;

$$\begin{bmatrix} 11 & 12 & 13 \\ 21 & 22 & 23 \\ 31 & 32 & 33 \end{bmatrix} \times \begin{bmatrix} v1 & v2 & v3 \end{bmatrix}$$

$$= \left[\begin{array}{c|c|c} \begin{array}{c} 11 \ v1 \\ + \\ 12 \ v2 \\ + \\ 13 \ v3 \end{array} & \begin{array}{c} 21 \ v1 \\ + \\ 22 \ v2 \\ + \\ 23 \ v3 \end{array} & \begin{array}{c} 31 \ v1 \\ + \\ 32 \ v2 \\ + \\ 33 \ v3 \end{array} \end{array} \right]$$

Figure 11. Column-based matrix-multiplication as the sum of dot products of elements

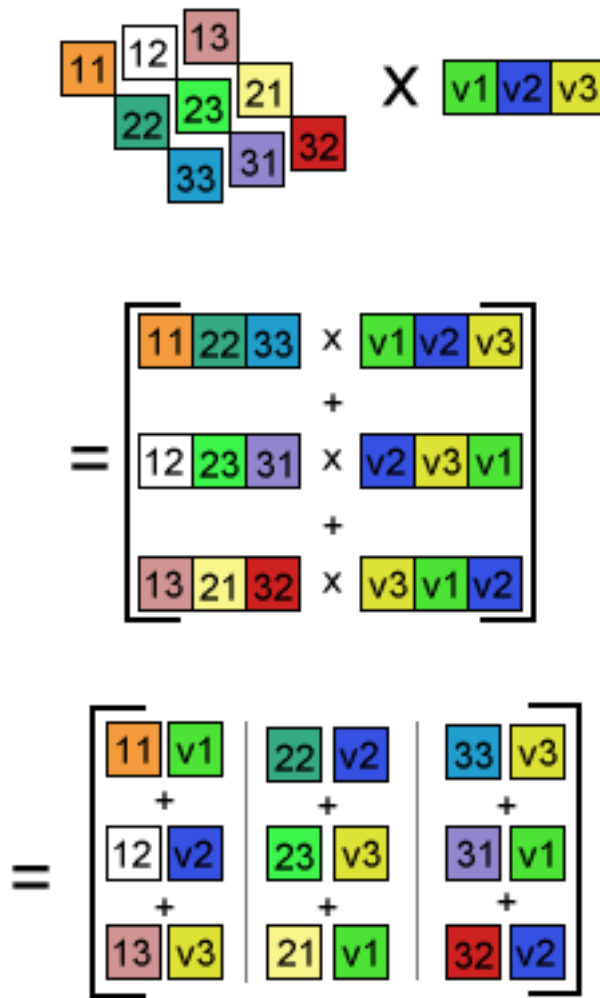


Figure 12. Equivalence of column-based matrix-vector multiplication with diagonal-based matrix-vector multiplication. In the diagonal based-one, the indices used the access the vector have to be appropriately shifted by the distance of the diagonal vector to the main diagonal. Notice that the result is (necessarily) the same as the column-based matrix multiplication, modulo reordering of the sum terms.

- A texture A_j^a , which will store all the off-diagonal elements of the matrix, packed and in row order,
- A texture R^x , an indirection texture storing pointers to the beginning of each series of off-diagonal entries in A_j^a , and
- A texture C^a , that stores pointers to the vector that will be multiplied against the matrix. Note that this texture has the same layout as A_j^a . As such, the texture coordinates that access a matrix value in A_j^a access in C^a the value of the texture coordinates that will be used to fetch the right vector value for the multiplication of each row against the vector.

A vector will be stored as a single X^x texture. Notice that this means that the vector will have the same layout as the diagonal elements texture of the matrix. To compute the sparse matrix vector product, the authors use a more standard technique when compared to the previous description. Here, a matrix vector multiplication is interpreted as a series of dot products of matrix rows and the vector. So, the result of a matrix vector multiplication will be another vector, Y^x . Each of the elements of this vector will be calculated with a dot product.

A dot product is defined as the sum of element-wise multiplication between two vectors, and this is what we need to compute for each of the matrix rows. A pointer to the beginning of each compressed row is stored in R^x . Using this pointer, we traverse C^a and A_j^a concurrently. The values in A_j^a are the matrix values themselves, but the A_j^a is an indirection texture pointing back at the vector X^x . Because of that, we need to use the value at C^a as the address in which to fetch the right vector value at X^x . We then multiply these two values together, “increment the pointers” to C^a and A_j^a , and sum all the values of the row together. This value is the dot product between the row and the vector, and will be stored at the right place in Y^x (remember that they share the same layout).

The cautious reader will have noticed a problem in this description. We have, previously, repeatedly stressed the fact that a program fragment currently has no branch instructions or control flow of any kind. At the same time, we know the authors store rows with different sizes in the C^a and A_j^a texture. These different sizes mean that a program would have to stop as soon as it had reached the right row size, or it would invade the data belonging to a different row. The insight needed to solve this is that *for all rows of the same size, the same amount of computation needs to be performed*. The authors used this and developed a fragment program for each row size, and each of these programs is then used in the appropriate rows. This is another important principle to be kept in mind: the ability to identify constant amounts of computation and isolate them. This effectively works around the control flow restriction, and, as such, is a broadly applicable technique.

There is a final observation to be made. With many different programs that must be used in a single matrix-vector multiplication, we notice that most of the computation of the

program that adds rows of size n would be wasted if we used that program over the entire texture, because the sparse matrix could potentially have many different row sizes. What we need to do, then, is to generate fragments only where they'll be useful. In other words, we must find a way to send data to the fragment program so that it always corresponds to valid row sizes. Since the main primitive used here is the rectangle, we need to create rectangles that contain entries with the same row size. This is a packing problem that can be solved offline, as the size of the rows will not change during the computation. We refer the reader to the original paper [1] for a thorough discussion of the model.

5 Conclusion

In this paper we discussed the recent advances on graphics hardware from the perspective of understanding the possibilities that open up with such hardware. This approach is different from most tutorials on graphics hardware, which focus on the details necessary to be able to implement programs that use such technology.

Here, we took a different approach, by looking at classes of algorithms that can benefit from such hardware. For this, we chose significant papers in the literature that illustrate techniques that we judged to be important, such as ray-tracing calculations, simulation of physical phenomena using iterative solvers, and numerical computations involved in the solution of linear systems. This diverse set of problems illustrates the potential of such graphics hardware, and the possibilities of its use to other graphics (and non-graphics) problems.

We hope the summary can be used as common ground to readers so that they can start working on their own algorithms. Obviously that every new feature added to graphics hardware (and more are coming) will open more possibilities, and we encourage the reader to continue following recent papers on the area and the specifications of the graphics boards.

References

- [1] Bolz, Jeff et al. Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid. ACM Transactions on Graphics 2003 (Proceedings of SIGGRAPH 2003).
- [2] Carr, Nathan. Jesse. D. Hart, John. C. GPU Algorithms for Radiosity and Subsurface Scattering. Proceedings of the Eurographics/SIGGRAPH Graphics Hardware Workshop, 2003.
- [3] Carr, Nathan. Hall, Jesse. D. Hart, John. C. The Ray Engine. Proceedings of the Eurographics/SIGGRAPH Graphics Hardware Workshop, 2002.

- [4] Fernando, Randima. Kilgard, Mark L. The Cg Tutorial: The Definitive Guide To Programmable Real-Time Graphics. Addison-Wesley, 2003.
- [5] Goodnight, Nolan. et al. A Multigrid Solver for Boundary Value Problems Using Programmable Graphics Hardware. Proceedings of the Eurographics/SIGGRAPH Graphics Hardware Workshop, 2003.
- [6] Green, Simon. Stupid OpenGL Shader Tricks. Advanced OpenGL Game Programming Course, Game Developers Conference, 2003.
- [7] Harris, Mark, et al. Physically-based Visual Simulation on Graphics Hardware. Proceedings of the Eurographics/SIGGRAPH Graphics Hardware Workshop, 2002.
- [8] Harris, Mark. Analysis of Error in a CML Diffusion Operation. Technical Report, UNC, 2002.
- [9] Jakobsen, Thomas. Advanced Character Physics. Game Developers Conference, 2001.
- [10] Kaneko, K. (Ed.). Theory and Applications of Coupled Map Lattices. New York, Wiley, 1993.
- [11] Krüger, Jens. Westermann, Rüdiger. Linear Algebra Operators for GPU Implementation of Numerical Algorithms. ACM Transactions on Graphics, 2003 (Proceedings of SIGGRAPH 2003).
- [12] Möller, Tomas. Trumbore, Ben. Fast, Minimum-Storage Ray-Triangle Intersection. Journal of Graphics Tools, 1987.
- [13] NVIDIA. Cloth Simulation Demo, 2003.
http://developer.nvidia.com/view.asp?IO=demo_cloth_simulation
- [14] NVIDIA. OpenGL Extension Specifications, 2003.
http://developer.nvidia.com/view.asp?IO=nvidia_opengl_specs
- [15] OpenGL Shading Language Specification. Available for download at
<http://www.3dlabs.com/support/developer/ogl2/specs/index.htm>
- [16] POV-Ray. The Persistence of Vision Raytracer. Available at
<http://www.povray.org>
- [17] Proudfoot, Kekoa et al. A Real-Time Procedural Shading System for Programmable Graphics Hardware. Proceedings of SIGGRAPH 2001.
- [18] Purcell, Timothy et al. Ray Tracing on Programmable Graphics Hardware. ACM Transactions on Graphics, 2002. (Proceedings of SIGGRAPH 2002)

- [19] Purcell, Timothy et al. Photon Mapping on Programmable Graphics Hardware. Proceedings of the Eurographics/SIGGRAPH Graphics Hardware Workshop 2003.
- [20] The RenderMan Interface Specification. Available for download at <https://renderman.pixar.com/products/rispec/>
- [21] Verlet, L., Physical Review 159, 98 (1967).
- [22] Wolfram, Stephen. Cellular Automata. Los Alamos Science, Autumn, 1983.
- [23] http://www.usatoday.com/life/movies/news/2003-06-23-online-games_x.htm