3

Graphics Processor Units: New Prospects for Parallel Computing

Martin Rumpf[1] and Robert Strzodka[2]

[1] University of Bonn, Institute for Numerical Simulation, Wegelerstr. 6, 53115 Bonn, Germany
martin.rumpf@ins.uni-bonn.de
[2] caesar research center, Ludwig-Erhard-Allee 2, 53044 Bonn, Germany
strzodka@caesar.de

Summary. This chapter provides an introduction to the use of Graphics Processor Units (GPUs) as parallel computing devices. It describes the architecture, the available functionality and the programming model. Simple examples and references to freely available tools and resources motivate the reader to explore these new possibilities. An overview of the different applications of GPUs demonstrates their wide applicability, yet also highlights limitations of their use. Finally, a glimpse into the future of GPUs sketches the growing prospects of these inexpensive parallel computing devices.

3.1 Introduction

This introductory section motivates the use of Graphics Processor Units (GPUs) as parallel computing devices and explains the different computing and programming models. Section 3.1.3 reports on hands-on experience with this kind of processing. A comparison with shared memory machines clarifies the similarities and differences.

The rest of the chapter is organized as follows. Section 3.2 presents the most important aspects of graphics hardware related to scientific computing. For a wider context and specific GPU topics, the appendix (Section 3.5) is referenced in various places. Building on the experience from Section 3.1.3, Section 3.3 explains how to construct efficient linear equation solvers and presents partial differential equation (PDE) applications. It also contains a section with links to examples of code and other resources. In Section 3.4 we conclude with an outlook on future functionality and the use of multiple GPUs.

3.1.1 Motivation

Over the last decade, GPUs have developed rapidly from being primitive drawing devices to being major computing resources. The newest GPUs have as many as 220 million transistors, approximately twice as many as a typical Central Processor

Unit (CPU) in a PC. Moreover, the L2 cache consumes most of the transistors in a CPU, while GPUs use only small caches and devote the majority of transistors to computation. This large number of parallel processing elements (PEs) converts the GPU into a parallel computing system. Current devices have up to 16 parallel pipelines with one or two PEs each. A single processing element (PE) is capable of performing an addition or multiplication of four component vectors (4-vectors) of single-precision floating-point numbers in one clock cycle. This amounts to a total of 128 floating point operations per clock cycle. With a clock frequency of up to 500 MHz, peak performance of a single unit approaches 64 GFLOPS, and with the introduction of the PCI Express (PCIe) bus a motherboard will be able to accommodate several graphics cards.

We will concentrate on the exploitation of the high internal *parallelism* of a single GPU. Section 3.1.4 explains how the parallel PEs of a single GPU can be viewed in light of the familiar shared memory computing model, although all PEs reside on the same chip. The development of GPU clusters, where several graphics cards work in parallel, has recently been initiated and Section 3.4.2 provides some information on this quickly growing area of research.

High performance and many successful implementations of PDE solvers on GPUs have already caught the attention of the scientific computing community. Implemented PDE types include Navier-Stokes equations, Lattice Boltzmann equations, reaction-diffusion systems, non-linear diffusion processes, level-set equations, and Euler equations for variational functional minimization. The web site [10] offers an overview. In 2D, problem sizes go up to 4096^2 nodes, in 3D up to 256^3 nodes. The limiting factor is the size of the *video memory* on the graphics cards (current maximum 512Mb). If one is willing to accept a slower rate of data exchange by using the main memory, the problem size is not limited by the graphics card. Reported speedup factors, as compared to a modern single CPU solver, are often in the range 5-20.

Manufacturers of GPUs are now considering the potential of their devices for parallel computing, although the driving force of the development is still the computer game market. This influences the balance in some of the common antonyms:

- Performance - Accuracy
  For optimal performance GPUs offer different floating point formats of 16, 24 and 32 bit, but native support for a double precision format is unlikely in the near future. A hardware-assisted emulation could be feasible.
- Processing - Communication
  GPUs process large data sets quickly with many parallel processing elements (PEs), but direct communication between them does not exist.
- Generality - Specialty
  Fairly general high-level languages for GPU programming exist, but the setup of the execution environment for the programs and the data handling still requires some graphics-specific knowledge.

Luckily, more and more physical simulation is being used in computer games, which increases the demand for general computing capabilities. The fast development cycle

of GPUs reacts with high flexibility to the changing requirements and tends towards a general parallel computing device. At some stage, the growing demand for more scientifically -orientated GPUs could even make a separate production line worthwhile, including, for example, double precision arithmetic. Simultaneously, the recently emerging support for the utilization of multiple GPUs will increase. Current GPUs are not yet able to replace CPU-based parallel systems in scientific computations on a large scale. However, we want to familiarize the reader with the looming possibilities and demonstrate that many algorithms can already benefit from their being executed on GPUs.

3.1.2 Data-Stream-Based Architectures

Peak performance of computer systems is often in excess of actual application performance, due to the *memory gap* problem [32], the mismatch of memory and processor performance. In data-intensive applications, the processing elements (PEs) often spend most of the time waiting for data. GPUs have traditionally been optimized for high data throughput, with wide data buses (256 bit) and the latest memory technology (GDDR3). In contrast to instruction-stream-based (ISB) CPUs, they also subscribe to the data-stream-based (DSB) computing paradigm [13]. In DSB computing one exploits the situation in which the same operation is applied to many data items. Thus, the processing is not based on an instruction stream, but, rather, on a data stream. The PEs are first configured for the execution of the desired operation. Then, the data streams through the so configured pipeline of PEs undergoing the configured operations. The stream of data stops only when a new configuration must be applied. So, for the performance of DSB architectures, it is crucial that the configuration does not change frequently, but rather remains constant for a large data stream, e.g. for all components of a large vector.

The DSB model separates the two tasks of configuring the PEs and controlling the data-flow to and from the PEs. By contrast, an instruction prescribes both the operation to be executed and the required data. The separation of tasks deals much better with the memory gap problem, because the individual elements of the data streams can be assembled from memory before the actual processing. This allows the optimization of the memory access patterns, minimizing latencies and maximizing the sustained *bandwidth*. In ISB architectures only a limited prefetch of the input data can occur, as jumps are expected in the instruction stream. By contrast, it is inherent in the DSB model that no such jumps will occur for a long time. Thus, the resources can be concentrated on efficient data retrieval and parallel processing rather than jump predictions and speculative execution. Clearly, the advantage applies only to algorithms that exhibit this kind of regular behavior. Therefore, for some irregular algorithms, it is advantageous to increase the operation count in favor of more regular behavior, and thus faster execution, on DSB hardware.

The DSB architectures comprise reconfigurable logic, reconfigurable computing, processor-in-memory and stream architectures. GPUs may be seen as a restricted form of a stream processor. They are not the most powerful or efficient architecture, but offer an unrivaled price-performance ratio, which makes this advantageous

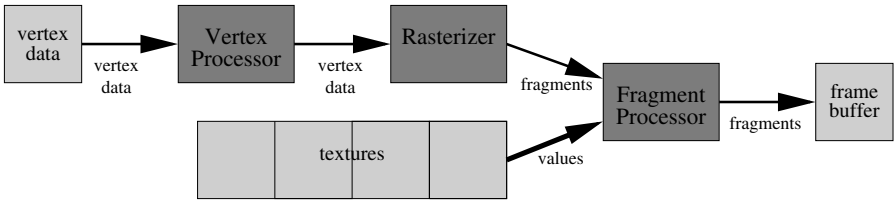

Fig. 3.1. A simplified diagram of the Direct X 9 graphics pipeline. Light gray represents data containers, dark gray processing units. The data containers are becoming fully interchangeable, i.e. a 2D data array can serve as an array of vertex data, a texture or a destination buffer within the frame-buffer. See Figure 3.5 for a more detailed diagram.

processing concept easily available on any PC, and not only on specially configured hardware systems. Moreover, GPUs have the great advantage that there exist widespread platform (Direct X) and operating system (OpenGL) independent Application Programming Interfaces (APIs) for access to their functionality, whereas other architectures require a proprietary environment. The API guarantees that despite the different hardware components of GPUs from different companies, the programmer can access a common set of operations through the same software interface, namely the API. Similarly to the situation with CPUs, the programming model for GPUs has evolved from assembly to high level languages, which now allow a clear and modular configuration of the graphics pipeline.

3.1.3 GPU Programming Model

Graphics Processor Units (GPUs) are, as the name suggests, designed to process graphics. Put simply, GPUs render geometric primitives such as points, lines, triangles or quads into a discrete representation of the $[-1,1]\times[-1,1]$ domain, called the *frame-buffer*. The geometric primitives are defined by *vertex* coordinates. The discrete elements in the frame-buffer are pixels. Because the primitives are continuous objects and the frame-buffer is a discrete representation, GPUs contain a so-called *rasterizer* that decomposes a primitive into fragments, which correspond to the set of affected pixels (see below why fragments and pixels are not the same). In the case of 2D primitives, one can choose whether to rasterize the contour or the interior, and we will always assume the latter.

The rasterizer divides the *graphics pipeline* into two parts where manipulation of data can take place. Prior to rasterization we have the Vertex Processor (VP), which operates on data associated with a vertex. Following rasterization we have the Fragment Processor (FP), which operates on data associated with a fragment (see Figure 3.1). Logically, the VP processes one vertex and the FP one fragment at a time, without any interaction with other vertices or fragments. Physically, there are several independent parallel pipelines, more for the FP than for the VP. See Section 3.5.2 for more details of the graphics pipeline and Section 3.2.3 for a discussion of the available parallelism.

Which data is associated with a vertex or fragment? In addition to the vertex coordinates, a vertex can also carry colors, a normal, and so-called texture coordinates (and a few more parameters). The VP can change all this data, including the vertex coordinates. The rasterizer interpolates the data between the vertices of a primitive when the fragments are generated. Therefore, each fragment has its own set of the above parameters. The FP combines the parameters to a final value, which is then assigned to the corresponding pixel in the frame-buffer. Currently, the frame-buffer position of a fragment generated by the rasterizer cannot be changed in the FP. Hence, there is a one-to-one correspondence between a fragment and the pixel to which the result of the FP will be written (unless it is discarded altogether). However, a fragment carries much more information than a pixel. For example, the texture coordinates associated with a fragment are typically used to retrieve values from textures, i.e. previously defined or computed 1D to 4D (typically 2D) data arrays (Figure 3.1). So, the FP reduces the information of a fragment to the single color value of a pixel. To be precise, a pixel may carry several values (see Section 3.5.2).

Both the VP and FP support a rich set of arithmetic, exponential and trigonometric functions on floating point numbers. They can be programmed by C-like high-level languages, which, to a certain extent extent, also support flow control instructions such as conditional statements, loops and function calls. The VP, and above all the FP, is decisive for accuracy and performance of computations on GPUs. Section 3.2.2 specifies the exact number formats and operations.

How can this setting be used for scientific computing? If we think of a square grid of dimension $N_x \times N_y$, then the node values form a vector of length $N_x \cdot N_y$ and can be represented by a 2D array, naturally preserving the neighbor relations. In GPUs we use a 2D texture for such a 2D array. Let us first describe the execution of a single operation; namely, the addition of two nodal vectors $\bar{A}$ and $\bar{B}$. Once the graphics environment is set up for this simple operation, it will be easy to add more functionality. For the addition of $\bar{A}$ and $\bar{B}$ on the GPU, we need to add the texels (elements) of the corresponding textures. For this to happen we must configure the graphics pipeline appropriately and then define the data streams to be processed. First we need a configuration, a so called *shader*, for the FP that executes an addition for a pair of vector components:

```
// shader FP_ADD2
float add2(float2 texCoord   : TEXCOORD0, // texture coords
     uniform sampler2D Tex_A : texunit0,  // texture A
     uniform sampler2D Tex_B : texunit1)  // texture B
  : COLOR                                 // color as output
{
  float valA= f1tex2D(Tex_A, texCoord);   // texel from A
  float valB= f1tex2D(Tex_B, texCoord);   // texel from B
  return valA+valB;                       // addition
}
```

The configuration of the VP and FP are nowadays usually written in a high-level graphics language. This is a listing in the language C for graphics (Cg). We will list all shaders in Cg, but the graphics languages are actually very similar and the reader

may prefer a different one (see Section 3.5.4). In addition to C we have the colon notation, which specifies the semantics of the input and output. Here, we use one set of coordinates (`texCoord`), two textures (`Tex_A, Tex_B`) and a float color value as output. The function call `f1tex2D(.,.)` reads one float from the given texture and coordinate. The actual addition happens in the last but one line. As noted earlier, *logically* the shader operates on one fragment at a time in a sequence, but *physically*, the parallel FP pipelines run this configuration simultaneously. The loop over the texture elements is implicit.

To start the processing of the loop, we must first configure the pipeline with our shader and bind the textures $\bar{A}$ and $\bar{B}$ as input sources:

```
cgGLBindProgram(fpProg[FP_ADD2]);              // bind shader

glActiveTexture(GL_TEXTURE0);                  // texunit0
glBindTexture(GL_TEXTURE_2D, texID[TEX_A]);    // bind TEX_A

glActiveTexture(GL_TEXTURE1);                  // texunit1
glBindTexture(GL_TEXTURE_2D, texID[TEX_B]);    // bind TEX_B
```

This code is a part of a normal C/C++ file. The functions are defined by the OpenGL and Cg API. We assume that `fpProg[FP_ADD2]` is a handle to our shader configuration from above, and `texID` is a vector that contains the OpenGL IDs of our textures. Now everything is configured and we only need to specify the geometry to be rendered. The above C/C++ code continues with the appropriate calls of OpenGL functions:

```
// function drawTex2D()
glBegin(GL_QUADS);                      // render quad
  glMultiTexCoord2f(GL_TEXTURE0, 0,0);// texture bottom left
  glVertex2f(-1,-1);                    // vertex  bottom left
  glMultiTexCoord2f(GL_TEXTURE0, 0,1);
  glVertex2f(-1,1);
  glMultiTexCoord2f(GL_TEXTURE0, 1,1);
  glVertex2f(1,1);
  glMultiTexCoord2f(GL_TEXTURE0, 1,0);// texture bottom right
  glVertex2f(1,-1);                     // vertex  bottom right
glEnd();
```

With the function call `glEnd()` the processing of the data streams starts and we obtain the result $\bar{C} = \bar{A} + \bar{B}$ at the end of the pipeline in the *frame-buffer* (see Figure 3.1).

The reader may have noticed that the dimensions of our textures $N_x \times N_y$ do not show up anywhere. This is because graphics APIs work mostly with normalized coordinates. From the code above, we see that a *texture* is accessed via coordinates from $[0,1]^2$ and the *frame-buffer* with vertex coordinates from $[-1,1]^2$. Hence, the values N_x, N_y are only used in the definition of the textures and the viewport of the frame-buffer, and not in the rendering. As the VP can change all parameters, including the texture and vertex coordinates, we could also address the textures and

the frame-buffer by other number ranges $T \subset \mathbb{R}^2$ and $F \subset \mathbb{R}^2$, if we were to bind appropriate constant matrices to the VP that performs the mappings $T \to [0,1]^2$ and $F \to [-1,1]^2$. In this way, it is possible to use the integer indices of the texels in the textures and pixels in the frame-buffer for the addressing, but this requires that the constant matrices be changed and re-bound each time the texture size or viewport size of the frame-buffer changes.

So far, we have described the execution of a single operation; namely addition, on the vector components. Now, we could easily add many more operations to our FP shader. The entire data from up to 32 different textures can be involved in the computations. In particular, we can read different elements of the textures to compute discrete gradients or, in general, any filters. However, usually we cannot map the entire problem into one shader (see Section 3.3.4). So the question arises, how can we use the result from the frame-buffer in a subsequent computation? Logically, the most elegant way is to define a texture $\bar{C}$ before the operation, and then render the result directly into that texture, using it as a destination buffer. After the operation we would bind a different texture as destination, say $\bar{D}$, and $\bar{C}$ could be bound as a source. Section 3.2.1 explains the details and also other possibilities.

As it becomes apparent that there are even more issues, not discussed above, that must be addressed, it will also become apparent to the reader that the handling of this kind of processing is very demanding. In fact, the low-level setup of the graphics pipeline can sometimes frustrating, even to experts. Luckily, though, there exist several libraries that will do most of the tedious work and let the programmer concentrate on the algorithm. Section 3.3 presents examples at this higher level of abstraction. The end of Section 3.5.4 discusses the development of even more abstract data-stream-based (DSB) programming approaches. For someone new to GPUs this will still feel unfamiliar at first, but parallel programming with the Message Passing Interface (MPI) [16, Section 2.2] or OpenMP [16, Section 2.3] also needs some practice before it can be performed comfortably.

3.1.4 Comparison with Shared Memory Model

Because the pipelines of GPU operate independently of each other on a common memory, the graphics card is similar to a shared memory parallel computer. (See Figure 1.3 in [16].) This section describes the GPU architecture from this perspective.

On graphics cards, not entire processors, but relatively few PEs constitute a pipeline. By a graphics processing element (PE) we mean an element that can perform a general multiplication or addition on a 4-vector or a special function on a scalar (e.g. $\sqrt{x}, 1/x$) for the largest available number format in one clock cycle. By viewing each pipeline as an individual node of a parallel computer, the GPU can be regarded as a restricted form of a shared memory machine. The following restrictions apply:

- All pipelines read from the same memory.
  This functionality is very similar to that of a general shared memory machine. The pipelines interact with each other by reading common memory. There is no

direct communication between them. In detail, a FP pipeline cannot specify a general memory address for reading directly, but it can read from any position within the bound textures. In other words, it can *gather* data from textures without restriction. Sufficiently large textures will cover all of the available *video memory* space on the graphics card. Hence, practically all problem data can be accessed, but it must be grouped in textures. This is fairly natural, because data arrays also provide some kind of logical memory grouping to a CPU. The memory access behavior is also similar, with local data reads being cheaper than random accesses. Currently, up to 32 textures can be bound during the execution of a FP shader and the choice must be made before the processing. Both restrictions are expected to disappear almost completely with Windows Graphics Foundation (WGF), the next generation of graphics API. (See Section 3.5.3.)

- All pipelines operate on the same stream of data.
  This is different from most shared memory machines, which utilize the Multiple Instruction Multiple Data (MIMD) model with no restriction on the sources from of the multiple data. Older GPUs use the Single Instruction Multiple Data (SIMD) model exclusively. This is efficient in terms of transistor count, but if the execution requires different branches, performance loss ensues. For small branches, the solution is to use *predication*, in which both branches are evaluated and thereafter the appropriate changes to the registers are written. Long branches are usually inefficient in pure SIMD architecture.
  In the latest graphics hardware supporting VS3 Vertex Shader and PS3 Pixel Shader models (see Section 3.5.3) two different solution paths have been taken. The VP pipelines are fully MIMD capable and thus need dynamic load balancing, but since the pipelines work on the same data stream this can be done automatically with little overhead. The FP is basically still SIMD but can evaluate different branches consecutively by keeping track of the current state and invalidating the results of individual pipelines. This results in some loss of performance, but such loss is acceptable if the executed branch changes very infrequently in the data stream. In the future, the FP will probably become fully MIMD too, although there is an ongoing debate as to whether this is really desirable, because the additional logic could also be used for more SIMD *parallelism*, which benefits the common cases.
- All pipelines write to the same destination arrays (frame-buffer).
  Shared memory machines usually do not have this restriction, although it avoids synchronization problems. A GPU pipeline cannot decide to output its data to an arbitrary position in memory. The destination memory must be defined beforehand and it cannot be read during the processing (in the general case). Therefore, there are no write-read collisions and no problems occur with cache coherency. For the FP, the restrictions go even further. Currently, the FP pipeline cannot change the destination address of a fragment at all. In other words, it cannot *scatter* data. This avoids completely any write collisions and allows parallel out-of-order processing. However, because the VP can scatter within the frame-buffer, fragments are roughly sorted by the primitives from which they were created.

> Future FPs are likely to allow scatter at some stage, but the bound on chosen memory regions as destinations seems reasonable to avoid the general synchronization problems.

So, the two main restrictions are the lack of scattering in the FP and the poor ability to handle branching. With respect to lack of scattering, it is possible to turn to the gathers for help. The gathers are almost fully general and often exploited to alleviate other problems. In particular, scatters can be reformulated as gathers. Concerning branching, it is usual to try to move the branch condition away from the FP into the VP or even higher into the main program where one decides about the geometry to be rendered. A common practice is to divide the domain into tiles. A classification step determines which tiles need to take which branch of the code. Then each tile is streamed through a shader that contains the appropriate branch. This assumes that the branch condition will most likely evaluate to the same value for all pixels within one tile. Tiles that contain pixels scheduled for different branches must be processed twice (if both branches are non-empty), which is basically equivalent to predication. See [20, 30, 4] for application-specific implementations of this technique.

Since a GPU is so similar in certain respects to a shared memory machine, the reader may wonder why the programming model is so different (Section 3.1.3). A particular difference is that while OpenMP allows an incremental parallelization of an existing code [16, Section 2.3], the GPU forces us from the beginning into a new design with a distinction between shaders for the configuration of the VP and FP and the specification of the dataflow in the form of geometry to be rendered. Remember that this distinction is innate to DSB architectures (Section 3.1.2), which assume implicitly that changing data and non-changing instructions dominate the work load. This requires different data processing in the hardware and a different programming model. It also brings new opportunities and new restrictions. The massively parallel performance depends heavily on some of these restrictions and therefore a general incremental way to replace serial code with GPU parallelism is not feasible. The required generality would destroy the envisioned advantage. Future GPU programming will look more and more like CPU programming, and in the long run they might even use the same code basis. However, such code will have to respect the hardware characteristics of the GPUs, which often is not the case for current software. For efficient *parallelism* the programming model must support the hardware.

3.2 Theory

In Section 3.3 we extend the example from Section 3.1.3 to a linear equation system solver. For an exact derivation, more background is required on the data containers, control of global data-flow, the available operations and parallelism. However, the reader may choose to continue directly with Section 3.3 and look up the necessary information as needed.

3.2.1 Dataflow

The general dataflow in a GPU is prescribed by the *graphics pipeline* (Figure 3.1). The standard data path from the main memory and the textures to the *frame-buffer* always has been fast, but in iterative PDE solvers we need more than one pass and intermediate results must be reused for subsequent computations. This means that the content of the frame-buffer must be resent through the graphics pipeline repeatedly. The efficiency of this general data handling has improved significantly over the years, but a fully flexible solution is still in development. There are several possibilities for further processing of the results from the frame-buffer:

- Read-back (`glReadPixels`).
  We can read the selected content of the frame-buffer back to the main memory. This is a slow operation, because data transfer has always been optimized in the direction from main memory to the graphics card. With the PCI Express bus with a symmetric bandwidth in both directions, this has finally changed this year. However, even then, the available bandwidth on the card is much higher than over the bus, so transferring data to the main memory and back onto the card is inefficient. Data should be read back only if it requires analysis by the CPU.
- Copy-to-texture (`glCopyTexSubImage1D/2D/3D`).
  The frame-buffer content can be used to redefine parts of an existing texture, or to create a new one. This also requires copying data, but the high data bandwidth on the card makes this operation much faster than read-back.
- Copy-to-frame-buffer (`glCopyPixels`).
  It is possible to copy data from the frame-buffer onto itself. The per-fragment operations can be applied to the copied data, but not the programmable FP programs (see Section 3.5.2). Hence, this operation is mainly useful for copying data between different buffers in the frame-buffer and possibly combining the content of the source and destination with simple operations.
- Render-to-texture (`WGL_ARB_pbuffer, WGL_ARB_render_texture`).
  This is the current state of the art, but currently supported only under Windows. It is possible to allocate a *pbuffer*, i.e. a non-visible target buffer that serves as the destination for the output data stream. As soon as the pbuffer is not a render target any more, it can be used as a texture. Ultimately, this means that it is possible to render directly to a texture. Hence, we can continue to talk about textures, which now can be rendered to. The only problem with pbuffers is that they carry a lot of static information, which causes a performance penalty when binding a new pbuffer as the destination for the output data stream. The switch between the use of a texture as a data source and data destination is fast only for special configurations; see below.
- Architectural Review Board (ARB) superbuffers.
  Current graphics driver development addresses the problem of slow pbuffer switches by introducing a new, light-weight mechanism for using raw data arrays as source or destination at various points in the graphics pipeline. The idea

is to define a memory array together with properties that describe the intended usage. The graphics driver then decides where to allocate the memory (cacheable, Accelerated Graphics Port (AGP) or video memory), depending on these properties. To some extent, the functionality is already available with the Vertex Buffer Object (VBO) and the Pixel Buffer Object (PBO) extensions, but the OpenGL ARB superbuffer group works on a more general and fully flexible solution.

Apart from the incurred switch delay, pbuffers serve the purpose of flexible data handling on GPUs well. In actual code, the mechanism for binding the pbuffer as the source or destination is encapsulated in a class. When the superbuffers appear, a reimplementation of this class immediately yields the additional benefits without any further changes to the applications themselves. A great problem for the job sharing between the CPU and the GPU is the limited bus width between the chipset and the graphics card. Even the PCI Express bus, which promises a theoretical 4GB/s data transfer rate in each direction, cannot approach the excess of 30GB/s of on-board bandwidth. Systems with multiple GPUs must also respect this discrepancy; see Section 3.4.2.

The *frame-buffer* and pbuffers are actually collections (typically 1-6) of 2D data arrays (surfaces) of equal dimensions (Section 3.5.2). Current GPUs support Multiple Render Targets (MRTs), i.e. the shaders can output results to several of these surfaces simultaneously. No *scatter* is allowed here, i.e. exactly the same position in all surfaces is written to, but more than four float results can be output at once. This technique is compatible with the render-to-texture mechanism, i.e. each surface is a different texture and all of them can be written to in one pass. However, each write goes to exactly the same position in each of the textures.

Multi-surface pbuffers also help to avoid general pbuffer switches. Those surfaces that are not the destinations of the current render pass can be used as sources (textures). Swapping the roles of destination and source on the surfaces is far less expensive than a general pbuffer switch. Thus, iterations are usually performed on a multi-surface pbuffer in a ping-pong manner, i.e. for iteration 0 we have surface 0 as source and surface 1 as destination; for iteration 1 we have surface 1 as source and surface 0 as destination, etc. In addition, more surfaces and other textures can be sources and the MRT technique even allows the use of several of the surfaces as destinations simultaneously. During the ping-pong procedure the same pbuffer is read from and written to, but the source and destination memory is disjoint, so that no write-read collisions can occur. In comparison to the superbuffers, however, multi-surface pbuffers are still restricted, because GPUs offer only a few surfaces (up to 6) and they must have the same size and format. Moreover, pbuffers in general do not support all of the available texture formats.

3.2.2 Operations

First let us examine the available floating point number formats. Three different formats have been introduced with the development of GPUs that support Direct X 9 (Table 3.1). Soon, the standard IEEE s23e8 format (without denormalized numbers)

Table 3.1. Precision of floating point formats supported in graphics hardware. These formats were introduced with Direct X 9, which required the graphics hardware to have a format with at least the fp32 precision in the VP and fp24 in the FP. The unit roundoff, i.e. the upper bound on the relative error in approximating a real number with the corresponding format, is half the machine epsilon ε.

format	fp16	fp24	fp32
GPUs with FP precision	Wildcat Realizm, GeForceFX 5800/5900/6800	DeltaChrome S4/S8, Volari V8, Radeon 9700/9800/X800	Wildcat Realizm, GeForceFX 5800/5900/6800
GPUs with VP precision	-	-	all Direct X 9 chips, Wildcat Realizm (fp36)
setup	s10e5	s16e7	s23e8
ε	$9.8 \cdot 10^{-4}$	$1.5 \cdot 10^{-5}$	$1.2 \cdot 10^{-7}$

will be a common standard, because chips that support the PS3 model are required to have a corresponding PEs throughout the pipeline. Hence, the half-precision format will be mainly useful to save memory and bandwidth, and possibly for fragment blending, which to date has no full floating point support. The implementation of a double float format is unlikely in the near future, though a hardware emulation could be feasible.

Both the VP and FP support a rich set of operations. There is a difference between the functionality offered by the high-level languages and the assembly languages, as the latter more closely express which functional units really reside in the hardware. However, since the languages intentionally include more primitive functions with the expectation that they will receive hardware support in future GPU, we want to present the functionality at this language level. Unfortunately, there is, as yet, no unified shader model for the VP and the FP. The FP imposes some additional restrictions, although this is not caused by a lack of language constructs, but rather by their use. In the following we will use the Cg syntax, but Direct X High-Level Shading Language (HLSL) is almost identical and OpenGL Shading Language (GLSL) very similar (see Section 3.5.4).

- Floating-point types: `half, float, half2, float4, float4x4`.
  The `half` is a s10e5 and the `float` a s23e8 (or s16e7) floating-point format (see Table 3.1). For both scalar types there exist native vector types of up to 4 components and all matrix types up to 4×4. Components of the native vectors can be arbitrarily swizzled, i.e. they can be duplicated and their order can be changed, e.g.:

```
float4 a(0, 1, 2, 3);
float4 b= a.xyzw;  // b==float4(0, 1, 2, 3)
float4 c= a.wyxz;  // c==float4(3, 1, 0, 2)
float3 d= a.ywy;   // d==float3(1, 3, 1)
```

Most graphics PEs operate internally on 4-vectors, so using the native vector types can greatly reduce the number of required operations.

- Data types: `float[5], float[6][3], struct`.
  General vectors and arrays can be defined, but there is only a limited number of temporary registers (up to 32 float 4-vectors), so for practical purposes, the size is extremely limited. There are more constant registers (up to 256 float 4-vectors). Arrays are first-class types, i.e. they are copied when assigned, since there are no pointers or associated functionality. In the VP constant vectors/arrays can be indexed with variables. Only the newest PS3 model for the FP supports such indexing for the texture coordinates.
- Mathematical functions.

Arithmetic	`+, -, *, /, fmod`
Sign, Comparison	`abs, sign, min, max, clamp`
Integers	`ceil, floor, round, frac`
Exponential	`sqrt, exp, exp2, log, log2, pow`
Trigonometric	`sin, cos, tan, asin, ..., sinh, ...`
Interpolation	`step, smoothstep, lerp`
Vector	`dot, cross, length, normalize, distance`
Matrix	`mul, transpose, determinant`

  Almost all scalar functions can also operate component-wise on the native floating-point vector types.
- Data access: `tex1D, tex2D, tex3D, texRECT`.
  In the FP, one to three dimensional textures (data arrays) can be read from arbitrary positions, e.g.:

```
float4 coord= IN.texCoord; // current texture coordinates
float4 a= tex1D(Tex_A, coord.x);
float4 b= tex2D(Tex_B, coord.xy);
float4 c= tex3D(Tex_C, coord.xyz);
```

  Currently, normalized coordinates from $[0,1]^2$ are used for texture access and only special rectangular 2D textures (`texRECT`) are accessed by coordinates from $[0,w] \times [0,h]$, which depend on the width (w) and height (h) of the texture. The texture samplers `Tex_A, Tex_B, Tex_C` cannot be chosen dynamically. This is expected to change in the future. In the newest VS3 model the VP can also access textures.
- Conditions: `bool, bool4,` $\&\&, ||, !, <, >, ==, !=, ?:$.
  Conditions must evaluate to a Boolean type. The operations can work component-wise on Boolean vectors. In case of the operator ?: this allows an individual decision for each vector component, e.g.

```
bool4 cond(true, false, false, true);
float4 a= cond? float4(0,2,4,6) : float4(1,3,5,7);
// Now a==float4(0,3,5,6)
```

- Control flow: `int, int4, if/else, while, for`.
  The conditions must be scalar Booleans. In the VP dynamic branches are fully

supported, so there are no further restrictions on the constructs. In the newest PS3 model, there is restricted support for dynamic branching in the FP (see Section 3.1.4). Otherwise `if/else` is resolved with *predication*, i.e. both branches are evaluated and conditional writes update the registers with the correct results. Without PS3 loops are unrolled, which must be possible. The integer types are currently not supported in hardware and are internally represented as floats. They are basically supposed to be used as loop counters and in case of unrolled loops, for example, they do not show up at all in the end.

- Abstraction: `struct, typedef`, functions, function overloading, interfaces. The high level languages offer increasingly more of the abstract constructs known from C/C++ or Java, although some restrictions apply. As the abstraction is not dependent on the available processing elements (PEs) in the hardware, it is likely to evolve further.

Since the PS2 model (Direct X 9 GPUs) and the introduction of floating-point number formats, the desire for arithmetic functionality has been basically fulfilled. The limits are now set by the control flow and variable indexing of vectors/arrays. For some configurations, the number of available temporary registers may also be a restriction. Future GPUs will relax these constraints further.

3.2.3 Parallelism

Figure 3.1 visualizes the stream processor nature of GPUs. We see two types of *parallelism* there: (i) the parallelism in depth innate to the pipeline concept, and (ii) the parallelism in breadth given by the breadth (4-vectors) and number of parallel vertex (up to 6) and fragment pipelines (up to 16). Because there is no configurable routing in GPUs, unlike FPGAs for example, these numbers are fixed, which has several consequences.

The deep pipeline makes frequent invocations or reconfigurations inefficient, i.e. for each rendering call the same operations should be applied to at least several thousand data items; the more the better. This does not mean that we cannot treat primitives smaller than 32×32 efficiently, but small regions undergoing the same operations should store their geometry in a common VBO. Then, one invocation suffices to execute the configured operations on all defined regions. Unfortunately, in the case of points, even then performance is highly reduced, because GPUs are optimized for processing 2D regions. Therefore, it is currently difficult to implement algorithms that require updates of singular, spatially unrelated nodes.

Up to 128 floating point operations can be executed per clock cycle in the FP, but the 256 bit wide Double Data Rate (DDR) memory interface delivers only 64 bytes. This means that to avoid a memory *bandwidth* problem the *computational intensity* should be, on average, above 8, i.e. eight or more operations should be performed in the FP on each floating point value read from the memory (assuming four bytes per float). Because the memory on graphics cards clocks higher than the GPU, and because of small internal caches, in practice the computational intensity may be a bit lower, but the general rule remains. The significant overbalance of processing power

against bandwidth has arisen only with the recent generation of graphics hardware. This trend is likely to continue, because computer games now also use programs with higher computational intensity and the integration of additional PEs into the GPUs is cheaper than the corresponding bandwidth increases. Note that, despite less internal parallelism, the high clock frequencies of the CPUs, and less bandwidth from the main memory system require a similarly high or even higher computational intensity for the CPUs. However, the bandwidth efficient programming methodologies for CPUs that exploit the large and fast on-chip caches cannot be directly applied to GPUs, which have only small caches. GPUs reduce the bandwidth requirements best in the case of strong data locality, e.g. node neighbors in a grid. See Section 3.3.3 for a discussion of efficient matrix vector products.

3.3 Practice

Section 3.1.3 offers a glimpse of the programming of GPUs. Now, after getting to know the dataflow and processing functionality in more detail, we want to demonstrate how to build up an efficient solver for a linear equation system on a GPU. Then we will present some of the existing PDE applications and list links to resources for GPU programming.

3.3.1 Setup

So far, we have talked about rendering to a frame-buffer. However, what we see on the screen are individual windows controlled by a window manager. Window management, the allocation of pbuffers and initialization of extensions depend on the operating system. Luckily, there exist libraries that abstract dependencies in a common interface. We will use the GLUT library for the Graphics User Interface (GUI), the GLEW library for the extension initialization and the RenderTexture utility class for the handling of pbuffers. Links to all resources used in the code samples are given in Section 3.3.6.

With the libraries, the main function for the addition of two vectors $\bar{A}$ and $\bar{B}$ as discussed in Section 3.1.3 needs only few lines of code:

```
#include <GL/glew.h>          // extension initializer GLEW
#include <GL/glut.h>          // window manager GLUT
#include "WinGL.h"            // my GUI
#include "AppVecAdd.h"        // my GPU application

int main(int argc, char *argv[]) {
  glutInit(&argc, argv);      // init GLUT window manager
  glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);

  // simple example: addition C= A+B
  WinGL winAdd;               // my GUI based on GLUT
  glewInit();                 // init extensions with GLEW
```

```
    AppVecAdd add;                // my GPU application
    winAdd.attachApp(&add);       // attach App to GUI

    glutMainLoop();               // start main loop of GLUT
    return 0;
  }
```

The first lines in `main` initialize the GLUT manager and set the default display mode. Then we create a GUI. The GUI manages a window, keyboard strokes and a menu with GLUT. Via the resource section (Section 3.3.6) the reader may find many tutorials that demonstrate the ease with which a GUI may be created with GLUT. The GLEW library call initializes all available OpenGL extensions. The extensions are necessary to use pbuffers and the programmable pipeline, for example. Most current GPUs support them. Then, we create our application class and attach it to the GUI such that the user can call the application functions. Finally, we start the event loop of GLUT and wait for the invocation of these functions by user interaction.

It is in the application class that the more interesting things happen. The constructor uses the Cg API to compile the application-specific shaders and load them to the graphics card. For the vector addition example above, we need a trivial VP shader that passes the vertex data through unchanged, and the `fpProg[FP_ADD2]` shader for the addition from Section 3.1.3. However, for other application classes, more shaders are loaded and can be later accessed by the vectors `vpProg[]`, `fpProg[]`. The constructor also uses the RenderTexture utility class to allocate the textures and store them in a vector `texP[]`:

```
  // enum EnumTex { TEX_A, TEX_B, TEX_C, TEX_NUM };
  for(i= 0; i<TEX_NUM; i++) {         // allocate textures
    RenderTexture* tp= new RenderTexture("r=32f tex2D rtt");
    tp->Initialize(256, 256);         // texture size
    texP.push_back(tp);               // store in a vector
  }
```

The mode-string requests a 32 bit float 2D texture suitable for the render-to-texture (`rtt`) mechanism (see Section 3.2.1). Currently, only Windows supports the render-to-texture mechanism, so on other systems copy-to-texture (`ctt`) should be used instead. The RenderTexture class has a simple interface for using the textures as either a destination or source of a data stream, possibly emulating render-to-texture by copy-to-texture internally. To set the initial data in a texture we simply define the values in a float array (`floatData`) and then render it:

```
texP[TEX_A]->BeginCapture();           // TEX_A is destination
glDrawPixels(texP[TEX_A]->GetWidth(),
             texP[TEX_A]->GetHeight(),
             GL_RED, GL_FLOAT, floatData);
texP[TEX_A]->EndCapture();             // TEX_A is source
```

In this way, the application class contains vectors with the shaders `vpProg[]`, `fpProg[]` and a vector with the initialized textures `texP[]`. These are the main

steps during the initialization of a GPU application and are independent of which operations will be performed later.

After the above initialization, the entire function for the addition of the vectors $\bar{A}$ and $\bar{B}$, which gets called via the GUI, reads as follows:

```
void AppVecAdd::exec() {
  CGprogram curVp= vpProg[VP_IDENTITY];// vertex shader
  CGprogram curFp= fpProg[FP_ADD2];    // fragment shader

  texP[TEX_C]->BeginCapture();       // TEX_C is destination

  cgGLEnableProfile(cgGetProgramProfile(curVp)); // enable
  cgGLEnableProfile(cgGetProgramProfile(curFp)); // profiles

  cgGLBindProgram(curVpProg);          // bind
  cgGLBindProgram(curFpProg);          // shaders

  glActiveTexture(GL_TEXTURE0);        // texunit0
  texP[TEX_A]->Bind();                 // bind TEX_A
  glActiveTexture(GL_TEXTURE1);        // texunit1
  texP[TEX_B]->Bind();                 // bind TEX_B

  drawTex2D();                         // render-to-texture

  cgGLDisableProfile(cgGetProgramProfile(curVp)); // disable
  cgGLDisableProfile(cgGetProgramProfile(curFp)); // profiles

  texP[TEX_C]->EndCapture();           // TEX_C is source
}
```

The shader `fpProg[FP_ADD2]` and the function `drawTex2D()` are listed in Section 3.1.3. All other calls fall within the functionality of the Cg API or the RenderTexture class. Because we have two different program sources, namely this C++ code and the Cg shaders, the passing of arguments to the shaders relies on the numbering of the texture units: `GL_TEXTURE0` corresponds to `texunit0` in the shader. The numbering corresponds to the numbering of arguments passed to the multi-dimensional function realized by the shader. Above, we do not see the OpenGL texture IDs explicitly, as in Section 3.1.3, because they are taken care of automatically by the RenderTexture utility class. Alternatively to the numbering, Cg also allows the association of the OpenGL texture IDs with the sampler names of the textures in the shader.

Even without the preparatory work of the initialization, the `exec()` function above seems a lot of code for the simple addition of two vectors $\bar{C} = \bar{A} + \bar{B}$. In practice, such operations are always encapsulated into a single function call. One option is to derive a class from RenderTexture and add functionality to it, such that the destination texture manages the operation:

```
texP[TEX_C]->execOp(fpProg[FP_ADD2],texP[TEX_A],texP[TEX_B]);
```

For convenience, one could even define an `operator+` function in this way, but this would be of little use since practical shaders do not represent elementary functions. Another option is to have a class for each shader and then simply write something like

```
fpProg[FP_ADD2].exec(texP[TEX_C], texP[TEX_A], texP[TEX_B]);
```

For a particularly short notation it is possible to have a function within our application class that only takes the indices:

```
execOp(TEX_C, FP_ADD2, TEX_A, TEX_B);  // C= A+B
```

We will use this notation in what follows. Clearly, the graphics setup described here is an example of how to get started fairly quickly. For large projects, more abstraction is recommended.

3.3.2 Vector Operations

Once the graphics specific-parts are abstracted, it is easy to realize further operations on vectors. We simply need to write a new FP shader, e.g. `FP_ATAN2` and then call it:

```
execOp(TEX_C, FP_ATAN2, TEX_A, TEX_B);     // C= atan(A/B)
```

Remember that the high-level languages support most of the standard mathematical functions directly (Section 3.2.2). So to write the `FP_ATAN2` we only need to exchange the `return` value in `FP_ADD2` (Cg listing in Section 3.1.3) for

```
return atan2(valA,valB);
```

For the vectors $\bar{A}, \bar{B}$ and $\bar{C}$ represented by the textures `TEX_A, TEX_B, TEX_C` this would correspond to

$$\bar{C}_\alpha = \text{atan}(\bar{A}_\alpha / \bar{B}_\alpha)\,.$$

With a standard set of such shaders it is easy to evaluate formulae, e.g. linear interpolation

```
execOp(TEX_C, FP_SUB2, TEX_B, TEX_A);      // C= B-A
execOp(TEX_D, FP_MUL2, TEX_C, TEX_M);      // D= C*M
execOp(TEX_R, FP_ADD2, TEX_A, TEX_D);      // R= A+D= A+M(B-A)
```

However, it is much more efficient to have one shader that does exactly the same in one pass. This avoids the costly pbuffer switches (Section 3.2.1) and increases the computational intensity (Section 3.2.3). So, in general, the shaders of an application should execute as much as possible in one go. The application class `AppVecAdd` that contains the addition as the only shader is, in this respect, an unrealistic example.

Unfortunately, the instruction to pack everything into one shader whenever possible easily results in the generation of a multitude of shaders. Consider, for example, the task of applying $h(f_i(\bar{V}_\alpha), g_j(\bar{V}_\alpha)), 1 \le i, j \le N$ to the components

of a vector $\bar{V}$, where the choice of i, j depends on some computed entity. For optimal performance we would have to write N^2 shaders S_{ij}. Duplication of code could be minimized by using include files that implement the $2N + 1$ functions $h(.,.), f_i(.), g_i(.)$, $1 \leq i \leq N$, but even so, N^2 short, different files would have to be generated. The remedy in such cases is to generate the few changing lines of code at run time and use run time compilation. If only a few run time compilations are necessary the performance does not degrade, especially if the GPU does not have to wait for the compiler but can be occupied with some other task during the software compilation process.

3.3.3 Matrix Vector Product

Matrix vector products are ubiquitous in scientific computing. The application of a matrix can be seen as a series of *gather* operations on the vector components. The matrix rows $\bar{A}_{\alpha,.}$ define what to gather and the weights. The gathers are inner products between the rows and the vector:

$$A\bar{V} = \left(\bar{A}_{\alpha,.} \cdot \bar{V}\right)_\alpha , \qquad \bar{A}_{\alpha,.} := (A_{\alpha,\beta})_\beta .$$

For a 2D problem we store the nodal vector $\bar{V}$ of the corresponding 2D grid as a 2D texture. Then α and β must be seen as 2-dimensional multi-indices as above. This means that without renumbering of indices, a full matrix for a 2D problem is a 4D structure. Due to the fact that GPUs restrict each dimension of a texture to 4096 or less, we usually cannot store a full matrix in a 1D or 2D texture. The best option is to use a 3D texture (4D textures are rarely supported), where in each 2D slice of the texture we pack several 2D row vectors $\bar{A}_{\alpha,.}$. Depending on the current pixel position, the FP shader retrieves the correct weights and performs the multiplications and additions. The result is obtained in one pass.

Because of the packing, a certain amount of address translation must be performed to retrieve the correct values. The Vertex Processor (VP) is usually not the bottleneck in scientific computing and is, therefore, used for the task of precomputing the offsets to the packed rows. The offset texture coordinates are passed to the FP. Another way of packing is to define a 4-valued texture and thus quadruple the number of values that are stored per texel. The point of optimizing operations for an execution on 4-vectors is discussed at the end of this section. From the point of view of memory, the packing of floats into 4-vectors is a disadvantage, because the large amount of data that has to be retrieved with a single command can lead to a local *memory gap* problem. Reading the four floats individually gives the compiler more freedom to place some computation between the reads and thus hide memory *latency* and insufficient *bandwidth*.

In general, full matrices can be handled only for small dimensions. A full-float matrix for a 128x128 grid requires 1Gb of memory, which exceeds the present *video memory* of graphics cards. Future GPU will offer hardware virtualization, such that a texture can also reside (partly) in main memory. However, the necessary data transfer to the graphics card is a strong bound on the performance in this case. Luckily, in

practice most matrices are sparse. Typical examples are band matrices. Each band can be stored in a texture of the same dimension as the grid. The bands can be packed together in one texture, which will reduce the lines of code necessary for the texture binding. This also avoids the problem that a maximum of 32 different textures can be bound to a FP shader. The VP can perform the offset computation, and the matrix vector product can be obtained in one pass again.

In the case of Finite Element codes and the discretizations of local operators, it is also possible to store the matrix in the form of elemental matrices (see (3.2)). Then, for each element, the components of the elemental matrices must be stored in separate textures or a packed arrangement. This is a special case of the general idea of partial matrix assembly that is presented below. It is particularly advantageous if the elemental matrices possess some common structure, such as symmetry or parameterization by few variables, since this greatly reduces the storage requirements. In the case of parameterization, one would only store the few values from which the elemental matrices can be built up (see (3.3)), and thus favorably increase the computational intensity of the matrix vector product. One problem arises, however, for the output of the local application of an elemental matrix. The GPU cannot scatter data, which means that only one node within the element can be updated from the result of the FP. The updating of the other nodes in the same pass would require the recomputation of the matrix vector product on the element for each node. One remedy for this is to output the complete result vector on each element, and from this gather the information onto the nodes in a second pass. This is a typical strategy on GPUs for reformulating a regular *scatter* operation in terms of a *gather*.

For irregular sparse matrices, two strategies are available: (i) cover the non-zero entries efficiently with some regular structures, i.e. rows, columns, sub-diagonals, and encode this structure statically into the shader, or one (ii) use a level of indirection in the processing, such that, e.g. the matrix entries contain not only the value but also the address (or offset) needed to access the corresponding component in the vector. The result can be computed in one pass. However, the irregularity of the entries can result in a serious performance problem if the number of entries per row differs significantly. Therefore, different vector components may require very different numbers of multiplications and additions. The PS2 model for the FP cannot stop the computation dynamically, i.e. all gather operations take the same time within the same shader. In the worst case one full row in the matrix suffices to make the matrix vector product as expensive as one with a full matrix. The newer PS3 model can make the distinction, but in terms of performance it is only beneficial if all spatially coherent vector components require approximately the same number of operations (see Section 3.1.4). Otherwise, the longest case dominates the execution time again.

Recapitulating, we can say that within the noted restrictions, matrix vector products can be computed for arbitrary matrices. Usually the matrices are not constant and have to be assembled first (see (3.1)). In many cases it is best not to assemble matrices explicitely, or at least not fully. Recall from Section 3.2.3 that current GPUs require a *computational intensity* of approximately 8 to avoid bandwidth shortage (in the case of floats). However, in a matrix vector product, we read both the matrix entry and the vector component and perform just one assembly operation: a multiply

and add (MAD). In other words we exploit only 6.25% of the available processing power. Consider three flavors of the matrix vector product for improvement:

- On-the-fly product: compute entries of A for each $A\bar{V}$ application.
  At first, it may seem foolish to compute the entries of A over and over again. However, this can still be faster than simply reading the precomputed data, because the comparably slow reading will stall the computation. Clearly, the advantage can only be gained for simple entries that can be computed quickly from little data. This technique has the lowest memory requirement and thus may also be applied in cases when the entire A would not fit into memory.
- Partial *assembly*: apply A on-the-fly with some precomputed results.
  This is a flexible technique which allows the computation and bandwidth resources to be balanced. On-the-fly products are infeasible if many computations are required to build up the matrix entries. In this case, a few intermediate results that already encompass most of the required operations should be generated. Then, during the matrix vector product, these few intermediate results are retrieved and the matrix finishes the entry computation on-the-fly. This reduces the bandwidth requirement and targets an optimal computational intensity. The few intermediate results have also the advantage of modest memory consumption.
- Full *assembly*: precompute all entries of A, use these in $A\bar{V}$.
  This makes sense if additional operations of high computational intensity hide the bandwidth problem of the pure matrix vector product. To achieve this, it even makes sense to execute operations unrelated to the current matrix vector product in the same shader. The Multiple Render Target (MRT) technique (Section 3.2.1) allows the unrelated results to be output into separate textures. If the bandwidth shortage can be hidden (though this is hard to achieve), full assembly is the fastest option for the matrix vector product, but also the one with the highest memory requirements.

The above discussion is not specific to GPUs, because the same considerations apply to CPUs. Yet there is a relevant and important difference between GPUs and CPUs. While block matrix techniques exploit the large caches on typical CPUs, this is not possible in the case of GPUs, because they have only small caches and rely strongly on the right balance of operations and *bandwidth* capacity. This is a crucial factor and should be checked carefully in the case of poor performance on the GPU. Pure matrix matrix multiplications, for example, are not faster on GPUs than on current CPUs [6].

Following the considerations about bandwidth, it is also possible to opt for low-level optimizations of the matrix vector product [2], related to the fact that the processing elements (PEs) operate, in general, on 4-vectors. However, newer GPUs can split the 4-component PEs into a 3:1 or even 2:2 processing mode, evaluating two different commands on smaller vectors simultaneously. The high-level language compilers optimize for this feature by automatically reordering commands whenever possible. The resource section offers links to tools that analyze the efficiency of shaders for a given GPU.

3.3.4 Solvers of Linear Equation Systems

We have repeatedly encouraged the reader to put as many operations as possible into one shader. Large shaders avoid pbuffer switches (Section 3.2.1) and help to hide bandwidth shortage (Section 3.2.3, Section 3.3.3). Why, then, should the entire problem not be solved in one shader? If this can be done without unnecessary additional operations or data access, it is the right choice. However, the implementation of a separable filter in one pass is a waste of resources. The same applies to the iterative solution of a linear equation system $A\bar{X} = \bar{R}$,

$$\bar{X}^0 \;=\; \text{initial guess}, \qquad \bar{X}^{l+1} \;=\; F(\bar{X}^l),$$

where $F(.)$ is the update method, e.g. conjugate gradient. The implementation of several iterations in one shader is unwise, because it multiplies the number of operations and, in particular, data accesses.

Which solvers are suitable for GPUs? They must allow parallel independent processing of vector components, and do so without direct write-read cycles. The first is important to exploit the parallel pipelines, while the second is a restriction of the FP which, in general, has no access to the destination buffer during the processing. An alternative is to process the vector in blocks with several passes, such that during the processing of a block the previously computed blocks can be accessed.

The conjugate gradient solver and its variants (preconditioned, asymmetric) rely on operations of the following forms:

$$F(\bar{X}^l) = \bar{X}^l + \frac{\bar{r}^l \cdot \bar{p}^l}{A\bar{p}^l \cdot \bar{p}^l}\bar{p}^l, \qquad \bar{p}^l = \bar{r}^l + \frac{\bar{r}^l \cdot \bar{r}^l}{\bar{r}^{l-1} \cdot \bar{r}^{l-1}}\bar{p}^{l-1}, \quad \bar{r}^l = \bar{R} - A\bar{X}^l\,.$$

The main ingredients are the matrix vector product, which was discussed in the previous section, and the inner product, which is a *reduction* operation.

Reductions are not directly supported in hardware on GPUs. An easy solution is to write a shader with a loop that runs over all texels and performs the summation. By rendering a single pixel with this shader, the result is obtained; but this does not utilize the parallel pipelines. At least 16 pixels with subtotals should be rendered before a final summation is performed. A second possibility is to perform consecutive additions of neighboring texels and thus reduce the dimensions of the texture by 2 in each pass. In the end, the result is also a 1x1 texture. Which option is better depends strongly on how data is arranged in textures: linearly or hierarchically. Traditionally, GPUs are optimized for the second option of fast access to neighbor texels. With the first, there may be general problems with the maximal instruction number in the FP (see Section 3.5.3). The result of the reduction can be either read back to the CPU or left in a 1×1 texture for further use. In an interactive approximation, the read-back is necessary at some stage to retrieve the norm of the residual and decide whether the iterations should be stopped. However, the asynchronous read-back mechanism does not stop the computation.

We see that all ingredients necessary for a solver of a linear equation system can be implemented on a GPU. The initialization of the graphics pipeline requires some

effort (Section 3.3.1), but once the work is done or prepared by some library, it is possible to concentrate on the algorithm. Concerning the matrix vector product (Section 3.3.3), attention should be paid to the high ratio of processing elements (PEs) against the bandwidth in GPUs. The use of a fully-assembled matrix consumes a lot of bandwidth and is only appropriate if this disadvantage can be hidden with accompanying computations. Finally, solvers of linear equation systems must allow for parallel processing of the vector components. Reduction operations are not native to GPUs, but can be resolved efficiently. Several researchers have solved PDE problems along these lines. The next section discusses some applications.

3.3.5 PDE Applications

We consider the discretization of partial differential equations on GPUs. In the field of continuum mechanics, various physical processes have been simulated in graphics hardware [18, 17, 21, 19, 12]. Beyond physical simulation, GPU-accelerated PDE methods are also very popular in geometric modeling and image processing [20, 3, 11, 29, 14]. The GPU Gems book series also contains an increasing number of GPU-accelerated PDE solvers [7, 25] and the site [10] offers an extensive overview of GPU-based computations. The processing of triangular grids, shading and texturing of highly resolved meshes, and the processing of images (usually regarded as surface textures), are the original applications for which graphics cards have been designed and optimized. Before we provide an overview of a number of applications in this field, we outline the basic setup for the treatment of PDEs on GPUs.

Consider a general differential operator A that acts on functions u defined on a domain Ω and ask for a solution of the differential equation

$$A[u] = f$$

for a given right-hand side f. In addition, require certain boundary condition to be fulfilled on $\partial\Omega$. In the case of variational problems, we ask for minimizers of energies E over functions u, such that a differential equation appears as the Euler Lagrange equation, with $A[u] = \operatorname{grad} E[u]$ and $f = 0$. If we take into account some time-dependent propagation, relaxation or diffusion process, we frequently obtain a differential equation of the form

$$\partial_t u + A[u] = f\,.$$

Now, we ask for solutions u that depend on the time t and the position x on Ω. In the case of a second-order diffusion we usually deal with $A[u] = -\operatorname{div}(a[u]\nabla u)$, where $a[u]$ is a diffusion coefficient or tensor that possibly depends on the unknown solution u. In the case of Hamilton Jacobi equations that describe, for instance, the propagation of interfaces, we deal with $A[u] = H(\nabla u)$. E. g. $H(\nabla u) = v(t,x)\,\|\nabla u(t,x)\|$ corresponds to the propagation of the level-sets of the function u at time t and position x with a speed $v(t,x)$ in the direction of the normaly. In many cases, v itself depends non-linearly on u.

Now consider the discretization of these differential equations based on Finite Elements. Obviously, other ways to discretize PDEs such as Finite Volume or Finite Difference approaches lead to fairly similar computational requirements. We consider a simplicial or rectangular mesh $\mathcal{M}_h$ on Ω with grid size h and a Finite Element space V_h with a $N = \#I$-dimensional basis $\{\Phi_\alpha\}_{\alpha\in I}$ consisting of basis functions Φ_α with local support on the domain. Now, we ask for a discrete solution

$$U(x) = \sum_{\alpha\in I} \bar{U}_\alpha \, \Phi_\alpha(x)$$

of the stationary problem, such that U approximates the continuous solution u, or we compute space and time discrete solutions $U^k(x) = \sum_{\alpha\in I} \bar{U}^k_\alpha \, \Phi_\alpha(x)$, with $u(t_k, x) \approx U^k(x)$, for $t_k = k\,\tau$ and some time-step τ.

Usually, Finite Element algorithms consists of two main ingredients; namely, the *assembly* of certain discrete vectors in $\mathbb{R}^N$ or matrices in $\mathbb{R}^{N^2}$ and the *discrete solution update*, with an iterative linear equation system solver, an explicit update scheme in time, or a combination of both in case of an iterative scheme with an inner linear system of equations to be solved:

- Assembly.
  In an explicit gradient descend algorithm, we usually compute the discrete gradient

  $$\left(\mathrm{grad}_{V_h} E[U]\right)_\alpha = \langle E'[U], \Phi_\alpha \rangle$$

  via a traversal over the grid $\mathcal{M}_h$. Locally on each element we collect contributions to the integral $\langle E'[U], \Phi_\alpha \rangle$ for all Φ_α such that its support intersects the current element. Similarly, the assembly of a Finite Element matrix, e. g. the stiffness matrix in the above-mentioned diffusion process

  $$L_{\alpha,\beta} = \int_\Omega a[U] \nabla\Phi_\alpha \cdot \nabla\Phi_\beta \, \mathrm{d}x \tag{3.1}$$

  starts by initializing $L = 0$, followed by a traversal of all elements. On each element E a corresponding local *elemental matrix*

  $$l_{\alpha,\beta}(E) = \int_E a[U] \nabla\Phi_\alpha \cdot \nabla\Phi_\beta \, \mathrm{d}x \tag{3.2}$$

  is computed first, corresponding to all pairings of local basis functions relevant on this element. Then, we can either store the collection of elemental matrices or assemble them into the global matrix L (see Section 3.3.3).
  All these operations match the data-stream-based (DSB) computing paradigm perfectly. The instruction set is always the same. Only the data to be processed changes and this data can be gathered by simple texture reads. In the case of a linear Finite Element space, the relation between the texels in the textures and the degrees of freedom is particularly simple. For example, if we process an image,

the values at the image pixels correspond directly to the degrees of freedom in the Finite Element space, and thus a coordinate vector in the Finite Element space is again an image. Similarly, we can treat each row in an irregular sparse matrix as a floating point texture and the corresponding index field, which contains the global position of the entries, as an integer texture [3]. The indirect access is less efficient because it partly breaks the paradigm of reading all data in streams. However, GPUs have also internal mechanisms to reduce the incurred performance penalty in such cases. The same problem cannot appear for the output because GPUs do not support the scattering of data.
For vector-valued functions u, e.g. positions in space, deformations, gradients or 2D Jacobians, the data can be kept in 4-valued textures. Then, it is also easy to take advantage of the processing elements (PEs) operating on 4-vectors (see Section 3.2.2). However, for larger texels (4 floats = 16B) it is more difficult to hide the incurred memory latency, so storage of the individual components is often the better choice (see Section 3.3.3). After realization of the correct functionality, the optimal option can be determined by a profiling tool.

- Discrete solution update.
  In the case of a simple update scheme, it is preferable to interleave the assembly with the update. That is, for a time-step of a discrete gradient descent

$$U^{k+1} = U^k + \tau \mathrm{grad}_{V_h} E[U^k] \,,$$

we immediately add the element-wise components of the update to the old discrete solution. When an iterative solver for a linear equation system is involved, i.e. the matrix is required in more than one matrix vector product, there are three possibilities: on-the-fly products, a partial or a full *assembly*. These possibilities were discussed in Section 3.3.3.
For a regular grid, standard linear stiffness matrices or mass matrices can be applied efficiently on-the-fly, because the matrix entries are locally the same for all elements, and can be stored in constants in the shaders. This changes if we consider non-linear stiffness matrices as defined above for the diffusion problem. For example, if $a[u]$ is a diffusion tensor and we use the midpoint integration rule for $a[u]$ in (3.2), we precompute the midpoint values $a[u]_E^{i,j}$ in a texture and store the constants $C_{\alpha,\beta}^{i,j} = \int_E \partial_i \Phi_\alpha \cdot \partial_j \Phi_\beta \, \mathrm{d}x$ in the shader. For isometric elements, the constants are few because they depend only on the difference $\alpha - \beta$. Then, the elemental matrices are parameterized by $a[u]_E^{i,j}$ and can be quickly reconstructed on-the-fly:

$$l_{\alpha,\beta}(E) = \sum_{i,j} a[u]_E^{i,j} C_{\alpha,\beta}^{i,j} \,. \tag{3.3}$$

The advantages are higher *computational intensity* in the matrix vector product and reduced memory requirements. Recall that for very large triangular meshes or images, the full assembly of a matrix still conflicts with the limited video memory size of graphics cards.

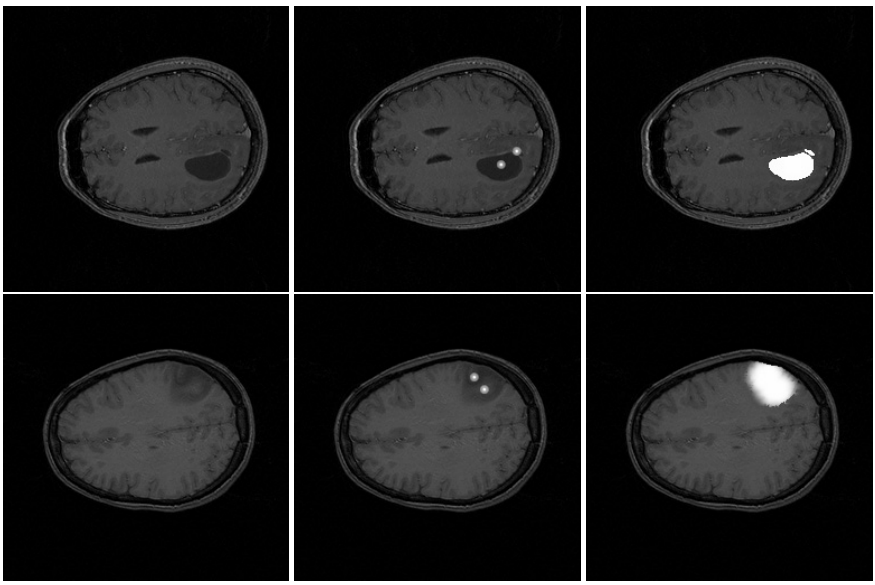

Fig. 3.2. Segmentation of tumors computed in Direct X 7 graphics hardware. The expansion of the level-set depends on the image values, its gradient and the placement of the seeds.

Now consider the processing of images as a concrete application field. Classical tasks in image processing are

- segmentation,
- feature-preserving image denoising,
- image registration.

Images are perfectly matched to rectangular textures and can be regarded as functions in a piecewise bilinear Finite Element space. Furthermore, if it comes to real-time processing and visualization, the results of our PDE algorithm reside already on the graphics boards, where they are needed for display. This underlines the conceptual benefits of PDE-based image processing directly on the GPU. In what follows we provide a brief sketch of some methods:

- Segmentation.
  Consider a region-growing algorithm for the segmentation of image regions whose boundaries are indicated by steep gradients. Therefore, a segment domain is represented by a level-set of a function u and sets $v(t,x) = \eta(\|\nabla I\|)$, where I is the image. Here, $\eta(\cdot)$ is some non-negative edge-indicating function, which is zero for $\|\nabla I\|$ larger than a certain threshold. Now, we ask for a family of level-set functions and corresponding evolving segment domains, such that

$$\partial_t u + v(t,x)\,\|\nabla u\| = 0\,.$$

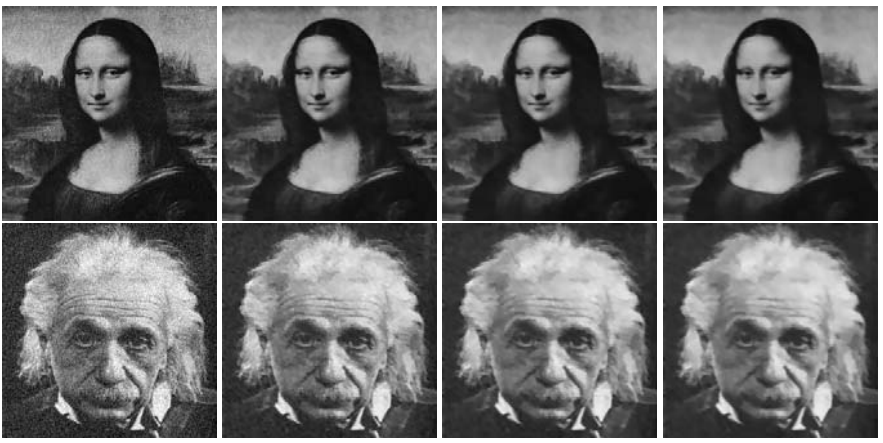

Fig. 3.3. Anisotropic diffusion for image denoising computed in Direct X 8 graphics hardware. The anisotropy allows the smoothing of noisy edges without blurring them completely.

The initial data $u(0,\cdot)$ is supposed to represent a user-defined mark on the image [26] (see Figure 3.2).

- Feature-preserving image denoising.
  Multiscale methods in image denoising are fairly common nowadays. The desired result is a family of images that exhibit different levels of detail, fine scale to coarse scale, and which are successively coarser representations of the initial fine-scale image. The aim of denoising is to filter out fine-scale noise on coarser scales while preserving important edges in the image. Such a scale of images can be generated solving a non-linear diffusion problem of the type

$$\begin{aligned} \partial_t u - \operatorname{div}(a[u]\nabla u) &= 0\,, \\ a[u] &= g(\|\nabla(G^\sigma * u)\|)\,. \end{aligned}$$

  The diffusivity $g(s) = (1 + \frac{s^2}{\lambda^2})^{-1}$ is large away from the edges and small in the vicinity of the edges, as indicated by large image gradients. To ensure robustness a prefiltering of the image by some Gaussian filter G^σ of filter width σ is invoked here. One can further improve the results, allowing, in addition, for a smoothing along the tangential direction on the edge [27] (see Figure 3.3).
- Image registration.
  Matching of a template image T with a reference image R via a non-rigid deformation ϕ - often called registration - can be formulated naturally as a variational problem. The aim is to achieve a good correlation of the template image T and the deformed reference image R:

$$T \circ \phi \approx R\,.$$

  In the simplest case of unimodal registration we can ask for a deformation ϕ given on image domain Ω, such that the energy

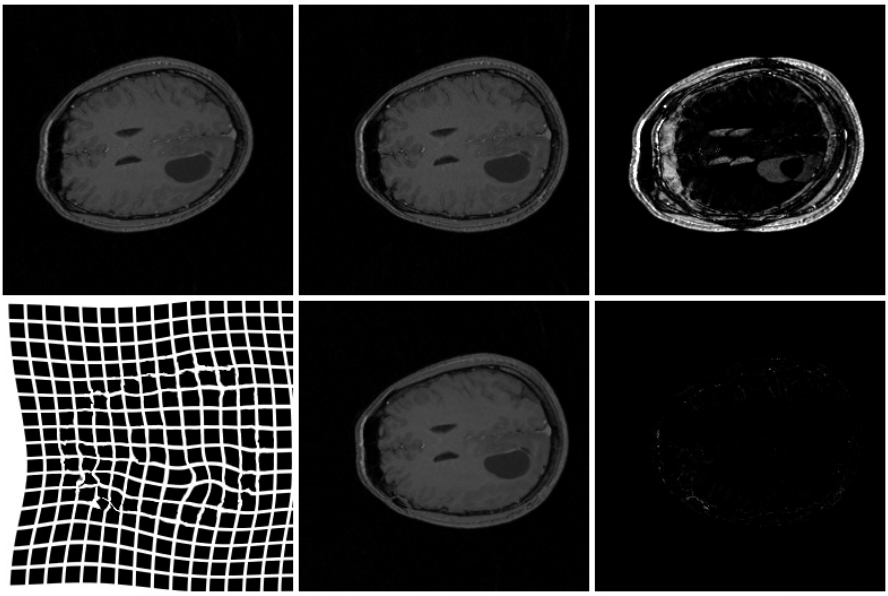

Fig. 3.4. Registration of medical images with a possible acquisition artefact computed in Direct X 9 graphics hardware. The six tiles are arranged in the following way: on the upper left we see the template that should be deformed to fit the reference image to the right of it; on the lower left we see the computed deformation applied to a uniform grid and to the right the registration result, i.e. the template after the deformation. The rightmost column shows the scaled quadratic difference between the template and the reference image before (upper row) and after (lower row) the registration.

$$E[\phi] = \int_\Omega |T \circ \phi - R|^2 \, \mathrm{d}x$$

is minimal in a class of suitable deformations. This problem turns out to be ill-posed and requires a regularization, by, for example, adding an elastic energy $\int_\Omega W(D\phi)\,\mathrm{d}x$ that measures the quality of the deformation itself and not only the quality of the match. Alternatively, a regularized gradient flow, which ensures smoothness of the resulting deformation, can be applied. After a discretization, the result is a global, highly non-linear optimization problem. Thus, the procedure is to consider a scale of matching problems ranging from coarse to fine. First, match on the coarse scale is found and then successively finer scales are treated [29] (see Figure 3.4).

The next section lists websites that point to many other PDE applications realized on GPUs including demos and code examples.

3.3.6 Resources

Up-to-date links to the sites below and the code samples discussed in this chapter are available online at the Springer site associated with this book.

The low-level programming of GPUs can be very tedious. Therefore, one usually uses libraries that facilitate the programming and abstract the details. The code examples in this chapter are based on the following resources:

- Graphics API: OpenGL
  `www.opengl.org`
- Shader language and API: Cg
  `developer.nvidia.com/page/cg_main.html`
- Window manager: GLUT
  `www.opengl.org/resources/libraries/glut.html`
- Extension initializer: GLEW
  `glew.sourceforge.net`
- Pbuffer handler: RenderTexture
  `gpgpu.sourceforge.net`

The choices are fairly common, apart from the last one where many still use self-made *pbuffer* handlers. However, we encourage the reader to explore the links below and discover other possibilities that might suit them better. To all of the above there are good alternatives and the authors themselves have used different tools, depending on the project requirements. The different combinations of graphics APIs and shader languages are discussed in more detail in Section 3.5.4. The rest of this section is a collection of useful links related to GPU programming.

- Scientific Computing on GPUs
  - GPGPU - General Purpose Computation on GPUs
    `www.gpgpu.org`
    This site addresses specifically general purpose computations, while other resources have usually a stronger focus on graphics applications. Related news, papers, code and links to resources are given and a forum for discussion is maintained. The site also features two full-day tutorials from the SIGGRAPH 2004 and Visualization 2004 conferences on scientific use of GPUs.
  - ShaderTech - Real-Time Shaders
    `www.shadertech.com`
    Here, shaders in general are discussed, and scientific examples are included. The site features news, articles, forums, source code and links to tools and other resources.
- Major Development Sites
  From time to time, one encounters technical GPU problems that have been solved already. The following development sites contain a huge store of examples, libraries, white papers, presentations, demonstrations, and documentation for GPUs. In particular, they offer well-assembled Software Development Kits (SDKs) that demonstrate various graphics techniques.

  - OpenGL Resources
    `www.opengl.org`
  - DirectX Resources
    `msdn.microsoft.com/directx`
  - ATI Developer Site
    `www.ati.com/developer`
  - NVIDIA Developer Site
    `developer.nvidia.com`
- Developer Tools
  These sites are good starting points for exploring the numerous freely available tools for GPUs. They include advanced Integrated Development Environments (IDEs) for shader development, debugging and performance analysis.
  - ShaderTech Tool Archive
    `www.shadertech.com/tools`
  - OpenGL Coding Resources
    `www.opengl.org/resources/index.html`
  - Microsoft Direct X Tools
    `www.msdn.microsoft.com/library/default.asp?url=/library/en-us/directx9_c/directx/graphics/Tools/Tools.asp`
  - ATI Tools
    `www.ati.com/developer/tools.html`
  - NVIDIA Tools
    `www.developer.nvidia.com/page/tools.html`
  - Babelshader - Pixel to Fragment Shader Translator (D. Horn)
    `www.graphics.stanford.edu/~danielrh/babelshader.html`
  - Imdebug - The Image Debugger (B. Baxter)
    `www.cs.unc.edu/~baxter/projects/imdebug/`
  - Shadesmith - Shader Debugger (T. Purcell, P. Sen)
    `www.graphics.stanford.edu/projects/shadesmith`

3.4 Prospects

The development of GPUs is rapid. Performance doubles approximately every nine months. Many new features are introduced with each generation and they are quickly picked up by implementations. This fast pace is likely to continue for at least several more years. What should we expect in the future?

3.4.1 Future GPUs

Throughout the chapter we have pointed to expected developments of GPUs. The information is based mainly on the features of the Windows Graphics Foundation

(WGF) announced by Microsoft for the new Windows generation (Longhorn [22]). Let us summarize the points.

- Parallelism/Bandwidth
  The parallelism will continue to grow rapidly, as well as the bandwidth. However, since increasing the first is cheaper than the second, programming will have to focus on *computational intensity* even more strongly than at present.
- Shaders
  Unlimited instruction counts and a unified shader model will be introduced. Much of the fixed pipeline functionality will be replaced by the use of programmable shaders. Without the fixed functionality, GPUs will basically be a collection of parallel PEs. This will introduce scheduling tasks that are likely to be hidden from the programmer. Memory will be virtualized to operate on data that would otherwise not fit the *video memory*. Further improvements will include indexing of textures and greater temporary storage for intermediate results in shaders.
- Dataflow
  We have emphasized the view that textures, pbuffers, *vertex* data and the *framebuffer* can all be seen as interchangeable collections of 2D data arrays, although full flexibility is not yet available. Future GPUs will fully incorporate this view and shaders will decide on their own how they want to interpret the data. The *graphics pipeline* will also offer several exit points for the data streams and not only the one at the end of the pipeline. As a result it will, for example, be possible to manipulate a mesh iteratively with the VP.
- Precision
  The latest VS3, PS3 model prescribes 32 bit float precision throughout the pipeline. Several GPUs offer this already and many more will soon follow. The support for double floats is unlikely in the near future, although there are, in principle, no barriers. The problem of development lies, rather, in the difficulties of creating a demand: a strong demand for double precision GPUs would make production feasible, yet at present, such demand is unlikely from the scientific community, because GPUs receive little attention from that quarter precisely because they do not have double precision. Further, a demand is unlikely to come from the graphics or computer game community where GPU vendors earn their money.
- Dynamic branching/MIMD in the FP
  Currently, GPUs with PS3 support are only efficient at infrequent dynamic branching. The problems with MIMD are additional transistors and scheduling problems, but the development of the processing elements (PEs) points clearly towards MIMD in the near future. The unification of the shader model does not necessarily mean that the PEs become the same, however, common PEs would allow better resource utilization in view of changing loads on vertices and fragments.

- Scatter
  The read-only, write-only regions avoid many synchronization problems. One-sided communication models are also known for their efficiency from CPU-based parallel computers; see [16, Section 2.2.3, page 17] Writing to arbitrary memory addresses would destroy too many of these advantages. However, scattering within a specified region does not have a negative effect on synchronization. Current GPUs can already scatter data by rendering it as a set of points. WGF will allow the generation of new primitives so that data duplication of individual items will be possible too. However, current GPU are not efficient at point processing; and this, it will be difficult to change.

Many of the expected features are already available, to some extent, through different extensions (Section 3.5.1). Respecting the current limitations on resources and performance, this already allows current development to be directed towards the new hardware. In other words, it is worth exploring the GPU as a general parallel processor, even if some restrictions still apply.

3.4.2 GPU Cluster

A single GPU already offers a lot of parallelism, but similar to CPUs, demand for higher performance suggests the use of multiple GPUs to work on a common task. The integration hierarchy is developing similarly to that of CPU-based parallel computers. One node, represented by a mainboard with several PCI Express slots, can accommodate several graphics cards. Clusters of multiple nodes are connected with the usual fast interconnects. However, both developments are in their infancy. NVIDIA offers a technology to couple two of their newest GPUs [23], ATI is expected to present a similar technology for their products, and Alienware announced a solution for all PCI Express graphics cards [1]. The solutions claim full transparency, so that the programmer only has to consider a number of general rules that will minimize the implicit synchronization between the cards. In addition, extensions to more than two boards seem feasible.

Initial academic work on the utilization of GPU clusters for parallel visualization [28, 15, 9] and computing [8, 5] also exists. Clearly, these approaches carry with them the same complexity as do CPU clusters. In particular, the considerations on partitioning and dynamic load balancing in [31] apply. The communication is even more complicated, because the cluster interconnects transport the data to the main memory and there is another stage of indirection in exchanging this data with the video memory of the graphics cards. In addition, once we are willing to pay the price of the comparably slow data transport between the graphics card and the main memory, it makes sense to involve the CPU in the processing too. We see the cluster as eventually being a pool of heterogenous processors with different computing paradigms and interconnects between them. While future graphics APIs will address the topics of job sharing and multiple GPUs and research on heterogeneous computer systems in general is ongoing, the efficient utilization of all available resources in GPU clusters is likely to remain a challenge for a long time.

3.5 Appendix: GPUs In-Depth

Graphics hardware has undergone a rapid development over the last 10 years. Starting as a primitive drawing device, it is now a major computing resource. We here outline the technological development, the logic layout of the graphics pipeline, a rough classification of the different hardware generations, and the high-level programming languages.

3.5.1 Development

Up to the early 1990s, standard graphics cards were fairly unimpressive devices from a computational point of view, although having 16 colors in a 640x350 display (EGA) as opposed to four colors in a 320x200 display (CGA) did make a big difference. Initially, the cards were only responsible for the display of a pixel array prepared by the CPU. The first available effects included the fast changing of color tables, which enabled color animations and the apparent blending of images. Then the cards started to be able to process 2D drawing commands and some offered additional features, such as video frame grabbing or multi-display support.

The revolutionary performance increase of graphics cards started in the mid 1990s, with the availability of graphics accelerators for 3D geometry processing. The already well-established game market welcomed this additional processing power with open arms and soon no graphics card would sell without 3D acceleration features. Since then, the GPU has taken over more and more computational tasks from the CPU. The performance of GPUs has grown much faster than that of CPUs, doubling performance approximately every nine months, which is the equivalent of a 'Moore's Law squared'.

During the late 1990s the number of GPU manufacturers decreased radically, at least for PC graphics cards. Although other companies are trying to gain or regain ground in the market, NVIDIA and ATI have clearly been dominant, both in performance and market shares, for several years now. Hence, the following discussions we cite primarily their products. Concerning the market, we should mention that actually Intel is the largest producer of graphics chips, in the form of integrated chip-sets. However, these are inexpensive products and rank low on the performance scale, so we will deal only with stand-alone GPUs on graphics cards.

Together with the reduction of GPU designers, the number of different APIs to access their functionality has also decreased. The OpenGL API and the Direct X API are the survivors. The API guarantees that despite the different hardware internals of GPUs from different companies, the programmer can access a common set of operations through the same software interface, namely the API. The proprietary graphics driver is responsible for translating the API calls into the proprietary commands understood by the specific GPU. In this respect, the API is similar to an operating system, which also abstracts the underlying hardware for the programmer and offers standardized access to its functionality, although an operating system does more than that.

If the hardware offers new features and is downward compatible, an old API still functions, but it lacks the new functionality. However, the use of new features in a new API results in an incompatibility with older hardware. Therefore, programmers are reluctant to use new features as long as they expect a significant demand for their applications on older hardware. The hardware vendor can promote the use of the new API by emulating the new hardware features in software on older systems, but this may turn out very demanding or impractical if the software emulation is too slow. So, in practice, programmers opt to assume very low requirements for the hardware and ignore incompatibility issues. Only the time-critical parts of the code are sometimes implemented for each hardware standard separately and chosen dynamically upon identification of the hardware. The above applies both to programs for different versions of an operating system and programs (mainly games) for different versions of graphics APIs. However, graphics hardware has evolved much quicker and game performance is often a critical factor, such that the changes of API versions and the lowest common requirements are moving faster than in the CPU market.

OpenGL and Direct X have been incorporating the quickly evolving feature set of GPUs differently. OpenGL uses a very flexible extension system. Each vendor can expose the whole functionality of its hardware product by proprietary extensions to the API. The OpenGL ARB [24], which includes the main players in the graphics field, helps in the standardization of these extensions to prevent the undermining of the common interface idea through too many incompatible proprietary extensions. In practice, the proprietary extensions appear first and then the standard access points evolve over time. The different versions of Direct X on the other hand, are prescribed by Microsoft and thus simply define a fixed set of requirements. Naturally, these requirements are discussed with the GPU designers beforehand. If the hardware supersedes them quantitatively, then Direct X often allows the use of these additional resources, but qualitatively new features have to wait for the next generation of APIs. So, we may say that the Direct X API changes more or less step in step with the new graphics hardware generations, while OpenGL evolves continuously, first on proprietary and subsequently on ARB paths. Currently, OpenGL is undergoing its first major revision since 1992, from the 1.x versions to version 2.0 [24] in an attempt to include many of the already well-established and new extensions into the core and prepare the API for future developments.

3.5.2 Graphics Pipeline

The Graphics Processor Unit (GPU), the central computational chip on a graphics card, may be seen as a restricted form of a stream processor (see Section 3.1.2). Via a set of commands, a particular state of the *graphics pipeline* in the GPU is configured and then data streams are sent through that pipeline. The output stream is visualized on the screen or resent through the pipeline after a possible reconfiguration. Although graphics cards have not, in the past, been seen in this context, current developments show a clear tendency towards the production of a general parallel computing device.

A schematic view of the graphics pipeline is presented in Figure 3.5. The abstraction omits some details but offers a clear perspective on the available functionality.

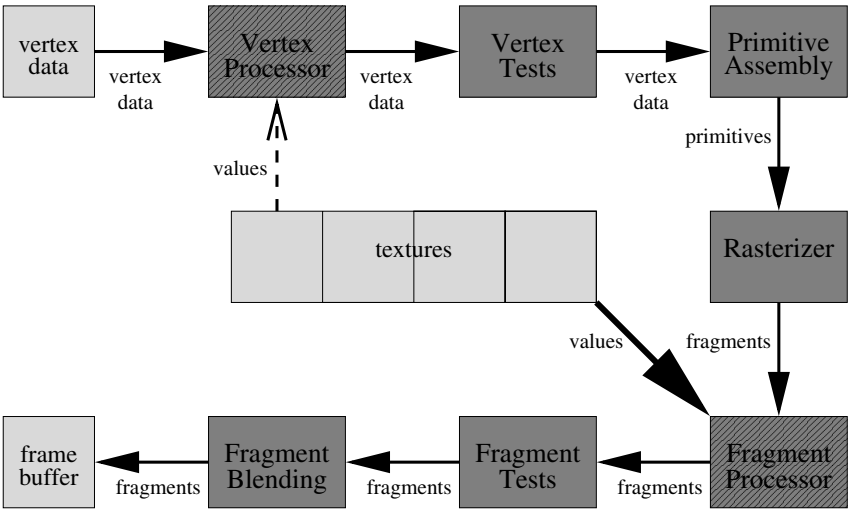

Fig. 3.5. A diagram of the graphics pipeline. Light gray represents data containers, dark gray processing units. The emphasized VP and FP are the units that evolved most in the graphics pipeline over the years, up to the stage where they accept freely programmable shader programs as configurations. Actually, the names VP and FP refer only to the new programmable pipeline stages, but the older functionality was located in the same place.
The thick arrow from the textures to the FP represents the largest data streams in the pipeline. Accordingly, the FP consumes the majority of resources in a GPU. The access to textures from the VP is a recent feature, as is the upcoming full interchangeability of the data containers in the pipeline, which allows a 2D data array to serve as an array of vertex data, a texture, or a destination buffer within the frame-buffer.

The logical pipeline has remained basically the same during the evolution of graphics hardware and changes can be identified by the increased flexibility and functionality of the individual components. Let us describe the operational tasks of the individual components:

- *Vertex* data
  We need an array that defines the geometry of the objects to be rendered. Beside the vertex coordinates, the vertex data may also contain color, normal and texture coordinate information (and a few more parameters). Although the data may be specified with one to four components, both coordinates (XYZW) and colors (RGBA) are internally always processed as 4-vectors. During the evolution of graphics hardware, it was principally the choices for the places where the vertex data can be stored (cacheable, AGP or video memory) and the efficiency of handling that data that increased. Modern VBOs allow us to specify the intended use and let the graphics driver decide which type of memory is ideally suited for the given purpose.
- Vertex Processor (VP)
  The VP manipulates the data associated with each vertex individually. Over the

years, the number of possible operations has increased dramatically. In the beginning, only multiplications with predefined matrices could be performed. Nowadays, the VP runs shader programs on the vertex data and the new generation has a restricted texture access from the VP. However, each vertex is still processed individually without any implicit knowledge about the preceding or succeeding vertices.

- Vertex tests

  Vertex tests determine the further processing of geometric primitives on the vertex level. They include mainly back-face culling, which eliminates polygons facing backwards (if the object is opaque one cannot see its back) and clipping, which determines the visible 3D space with an intersection of several 3D half spaces, defined by clipping planes. The vertex tests are still controlled by parameters and there have been only quantitative improvements in the number of clipping planes over time.
- Primitive assembly, *rasterizer*

  The geometric primitives that can be rendered are points, line segments, triangles, quads and polygons. Each vertex is processed individually and the clipping of primitives may introduce new vertices such that primitives have to be reassembled before rasterization. In addition, for simplicity, the rasterizer in many graphics architectures operates exclusively on triangles, so other primitives must be converted into a set of triangles before processing. Given a triangle and the vertex data associated with each of its vertices, the rasterizer interpolates the data for all the pixels inside the triangle. The resulting data associated with a pixel position is called a *fragment*. The rasterization could be controlled with parameters, for example defining patterns for lines or the interior of objects.
- Textures

  Textures are user-defined 1D to 4D (typically 2D) data arrangements stored in the *video memory* of the graphics card. Their elements, which can have up to four components (RGBA), are called texels. In general, the dimensions of all textures had to be powers of 2, but now there exists a general extension for textures with other dimensions.

  Input images of a problem are usually represented as textures on the graphics card and their values are processed by the FP and fragment blending. Over the years, quantitative improvements of textures have included their maximal number, their maximal size and the precision of the used fixed-point number format. Qualitative improvements are the support of various dimensionalities, the different access modes, the floating-point number format, and flexibility in the creation and reuse of texture data in different contexts. From the modern point of view, textures represent just a special use of data arrays that can serve as input to the FP (texture mode), as the destination for the output stream of the graphics pipeline (output mode), or even as an array defining vertex data (vertex mode).
- Fragment Processor (FP)

  The FP manipulates the individual fragments. Similarly to the way in which vertices are processed, each fragment is processed independently of the others in the

same data stream. With the interpolated texture coordinates, the FP can access additional data from textures. The functionality of the FP has improved enormously over the years. In a qualitative sense, the range of available access modes of texture data and operations on these values in the FP has grown rapidly, culminating in a FP controlled by assembly or high-level code with access to arbitrary texture positions and a rich set of mathematical and control operations. In a quantitative sense, the number of accessible textures and the number of admissible fragment operations has increased significantly.

- *Frame-buffer*

  The frame-buffer is the 2D destination of the output data stream. It contains different buffers of the same dimensions for the color, depth and stencil (and accumulation) values. Not all buffers need to be present at once. In addition, while each buffer allows certain data formats, some combinations may not be available. There exists at least one color buffer, but typically there is a front buffer, which contains the scene displayed on the screen, and a back buffer, where the scene is built up. Over the years, it has mainly been the maximal size, the number and the precision of the buffers that has increased. A recent development, already sketched in the discussion of textures, regards the frame-buffer as an abstract frame for a collection of equally-sized 2D data arrays. After rendering, the same 2D data arrays may be used as textures or vertex data.
- Fragment tests

  Equivalent to the vertex tests for vertices, the fragment tests determine whether the current fragment should be processed further or discarded. However, the fragment tests are more numerous and powerful than the vertex tests and some of them allow a comparison against the values stored at the associated pixel position of the fragment in the depth or stencil buffer, and also a restricted manipulation of these values, depending on the outcome of the tests. Because they access the frame-buffer directly their functionality cannot be realized in one pass, even in the newest FP.
- Fragment blending

  Before the FP became a powerful computational resource, computations were mainly performed by different blending modes. The blending operation combines the color value of the fragment with the color value in the color buffer, controlled by weighting factors and the blending mode. For instance, the blending operation can be a convex combination of the values using a certain weight. Blending has become less popular in recent years, because on most GPUs it has not supported the higher precision number formats, while the much more powerful FP does. However, currently, support for higher precision blending is increasing again. The advantage of blending is the direct access to the destination value in the frame-buffer, which is not supported by the FP on most GPUs.

  The blending modes are continuous functions of the input values. In addition, logical operations can be performed at the end of the pipeline, but these are

seldom used, because they have received no hardware support from the manufacturers of GPU.

As outlined in Section 3.1.3, for general purpose computations the FP is the most relevant part of the pipeline. The VP can be often used to reduce the workload of the FP by precomputing data that depends bilinearly on the vertex data across the domain, e.g. positions of node neighbors in a regular grid. The vertex and fragment tests are useful for masking out certain regions of the computational domain for special treatment and fragment blending can be used for a fast and simple combination of the output value with the destination value, e.g. accumulation.

3.5.3 Classification

Because of the almost synchronous evolution of the Direct X API and the generations of graphics hardware in recent years, it is easiest to classify GPUs according to the highest version of Direc tX that they support. In fact, it is only the Direct3D API that concerns us, but Microsoft releases the different APIs in a bundle, so it is usually the version of the whole release that is referred to. From Direct X 8 on, it is possible to differentiate the versions further by the functionality of the Vertex Shaders (VSs), which configure the VP, and the Pixel Shaders (PSs), which configure the FP. This Direct X (DX), VS, PS classification is useful, even if the OpenGL API is used for the implementation, because in contrast to Direct X, OpenGL evolves continuously with the introduction of individual extensions. In what follows, we provide an overview of the recent graphics hardware generations and list some typical representatives. The paragraphs point out the main functionality associated with the VS1 to VS3 and PS1 to PS3 shader models.

- Direct X 8 (VS1, PS1) GPUs, 2001-2002,
  e.g. 3DLabs Wildcat VP, Matrox Parhelia 512 (VS2, PS1), NVIDIA GeForce 3/4, ATI Radeon 8500.
  These GPUs introduced programmability to the graphics pipeline, i.e. assembly programs for the VP and highly restricted programs for the FP. However, the number formats were still restricted to low-precision fixed-point number systems.
- Direct X 9 (VS2, PS2) GPUs, 2002-2004,
  e.g. S3 DeltaChrome S8, XGI Volari Duo V8, NVIDIA GeForceFX 5800/5900, ATI Radeon 9700/9800.
  Direct X 9 is the current standard. With these GPUs, floating-point number formats appear. The programmability of the VP gains function calls, dynamic branching and looping. The PS2 model finally allows freely programmable code for the FP. High-level languages (HLSL, GLSL, Cg) facilitate the programming of the VP and FP.
- Direct X 9+ (VS2-VS3, PS2-PS3) GPUs, 2004,
  e.g. 3DLabs Wildcat Realizm (VS2, PS3), NVIDIA GeForce 6800 (VS3, PS3), ATI Radeon X800 (VS2, PS2).
  In the VS3 model, the VP gains additional functionality in the form of restricted

Table 3.2. The number of supported instructions in the VP and FP for the different *shader* models.

VS1	VS2+loops	VS3+loops	PS1	PS2	PS3	WGF
128	256	512-32768	8-14	96-512	512-32768	unlimited

texture access and more functionality for register indexing. The PS3 FP now also supports the features of function calls and restricted forms of dynamic branching, looping and variable indexing of texture coordinates.

- WGF 2, 2006?
  The next Windows generation (Longhorn [22]) will contain a new Windows specific graphics interface labeled WGF. The main expected features are a unified shader model, resource virtualization, better handling of state changes, and a general IO model for data streams. Future GPU generations will probably support all these features in hardware. See Section 3.4.1 for a more detailed discussion.

The number of supported instructions in the VP and FP for the different *shader* models is given in Table 3.2.
This classification shows a clear tendency of GPUs to be developing in the direction of a general parallel computing device. Clearly, the WGF functionality will offer more flexibility than the current APIs, but this should not deter the reader from working with the current standard Direct X 9 (VS2, PS2), because it is expected to be the baseline functionality for many years to come.

3.5.4 Programming Languages

With the advent of a fully programmable pipeline in Direct X 9, three high-level languages for the programming of shaders, i.e. VP and FP configurations, appeared. The differences between them are fairly small and stem from the underlying graphics Application Programming Interface (API).

- Direct X - HLSL
  `msdn.microsoft.com/library/default.asp?url=/library/en-us/directx9_c/directx/graphics/ProgrammingGuide/ProgrammablePipeline/HLSL/ProgrammableHLSLShaders.asp`
  The HLSL is used to define shaders for the VP and FP under Direct X. Usually, the shaders are configured directly with the high-level code, but the compiler can also be instructed to output the generated assembly code as vertex or pixel shaders. If desired, the assembly code can be changed or written from scratch, but this option will probably disappear in the future.
- OpenGL - GLSL
  `www.opengl.org/documentation/oglsl.html`
  In GLSL, shaders are defined for the VP and FP under OpenGL. Different extensions also allow the use of assembly code for the configuration. There exist ARB

extensions, which cover the common set of functionality among the different GPUs, and proprietary extensions, which expose additional features. However, the direct use of assembly code has become uncommon, because the GLSL compiler embedded in the graphics driver offers automatic optimization towards the specific GPU in use.

- Direct X, OpenGL - Cg
  `developer.nvidia.com/page/cg_main.html`
  Cg follows the same idea as HLSL and GLSL; namely, to allow high-level configuration of the VP and FP. However, Cg as such is independent of the particular graphics API used. The compiler can generate code for different hardware profiles. The profiles comprise different versions of the vertex and pixel shaders under Direct X and different versions of the vertex and fragment shaders under OpenGL. So, plugging the generated assembly code into the appropriate API slots establishes the desired configuration. To hide this intermediate layer, the Cg Toolkit also provides an API that accepts Cg code directly. To the programmer, it looks as though Direct X and OpenGL have a native Cg interface, just as they have a HLSL or GLSL one, respectively.

As the languages are very similar, the reader may wonder why there is any difference between them at all. They differ because the languages were not released at the same time and, more importantly, a smoother integration into the existing APIs was desired. Clearly, a common interface would have been nicer, since even slightly different syntax disturbs the work flow. It is to be hoped that there will be more standardization in the future. In the meantime, we encourage the reader to be pragmatic about the choice of languages and other resources (see Section 3.3.6).

In Section 3.1.3 we saw that the coding of the shaders is only one part of the work. In addition, the pipeline with the shaders and textures must be configured, and the geometry to be rendered must be defined. Hence, more information is needed to obtain the same result from the same shader. The Direct X FX and the Cg FX for Direct X and OpenGL formats allow this additional information to be stored. Appropriate API calls set up the entire environment to implement the desired functionality. These formats can also include alternative implementations for the same operation, e.g. to account for different hardware functionality. The only problem with these convenient tools is that in a foolproof solution, unnecessarily many state calls may be provided, even though the required changes from one operation to another are minimal.

A more general approach to GPU programming is to use stream languages that are not targeted directly at the VP and FP but, rather, at the underlying data-stream-based (DSB) processing concept. A compiler generates machine-independent intermediate code from the stream program. Then, back-ends for different hardware platforms map this code to the available functionality and interfaces. This generality is very attractive, but it does not mean that the stream program can be written without any consideration of the chosen hardware platform. Some language features might be difficult to realize on certain hardware and would lead to a significant performance loss. In these cases, less optimal solutions that avoid these features must be chosen.

Using code that is more hardware-specific clearly delivers better performance, but nobody opts for coding everything on the lowest level. Hence, offering a trade-off between performance and abstraction to the programmer makes sense. We sketch two prominent stream languages with a focus on GPUs.

- Sh - University of Waterloo
  `libsh.org`
  Sh uses the C++ language for the meta-programming of stream code. This has the advantage that the C++ language features are immediately available and the compiler does the necessary analysis and code processing. In addition, it addresses the above-mentioned problem of the specification of an appropriate accompanying graphics environment to the shaders. Sh allows a direct interaction of shader definition with texture and state configuration. With a fast compilation process, dynamic manipulation of the stream code is feasible. Another advantage of working within the familiar C++ environment is the potential for incremental introduction of GPU usage into suitable existing software. We say suitable, because the processing of many of the general methods of organizing data, such as trees, lists or even stacks, are difficult to accelerate on GPUs. Current back-ends support GPUs under OpenGL and different CPUs.
- Brook - Stanford University
  `www.graphics.stanford.edu/projects/brookgpu`
  The Brook language is based on the concepts of streams and kernels. It is an abstraction of the data streams and shaders of GPUs. This abstraction frees us from the entire consideration of texture handling and geometry processing. In particular, it breaks the emphasis on rendering passes. The focus is on the actual data and its processing. Hardware virtualization also overcomes the limits that the graphics API places on the number of bound textures, their sizes and types of manipulation. However, for the generation of efficient code, programmers must be aware of which features map well to the hardware and which require costly workarounds. The richer feature set is well-suited for the development and simulation of programs that assume additional hardware functionality in future GPUs. Current back-ends support GPUs under Direct X and OpenGL, and different CPUs.

The code examples in this chapter use the OpenGL API and the Cg language. For someone new to graphics programming, the use of the stream languages, which already include a lot of abstraction, would make the examples look simpler. We have chosen a medium level of abstraction to illustrate how efficient programming depends on the hardware characteristics of GPUs. This understanding is equally important for the more abstract approaches to GPU programming, because the abstraction does not free the programmer from considering the hardware characteristics during the implementation. Similarly, the abstraction offered by MPI for parallel computers presumes implicit knowledge about the architecture and its functionality. Nevertheless, the higher abstraction is very attractive, because the stream languages preserve the main performance characteristics of GPUs by construction. In practice, the type

of problem still determines whether it really is possible to obtain an efficient implementation on the high level. However, the stream languages are under active development and are extending their 'domain of efficiency' continuously. We recommend that the reader follow the links provided above for the download of the language libraries and detailed documentation.

Acronyms

AGP Accelerated Graphics Port
API Application Programming Interface
ARB Architectural Review Board
Cg C for graphics (high-level language)
CPU Central Processor Unit
DDR Double Data Rate (memory)
DSB data-stream-based
DX Direct X
FP Fragment Processor
GLSL OpenGL Shading Language
GPU Graphics Processor Unit
GUI Graphics User Interface
HLSL Direct X High-Level Shading Language
IDE Integrated Development Environment
ISB instruction-stream-based
MIMD Multiple Instruction Multiple Data
MPI Message Passing Interface
MRT Multiple Render Target
PBO Pixel Buffer Object
PCI Peripheral Component Interconnect
PCIe PCI Express
PDE partial differential equation
PE processing element
PS Pixel Shader
SDK Software Development Kit
SIMD Single Instruction Multiple Data
VBO Vertex Buffer Object
VP Vertex Processor
VS Vertex Shader
WGF Windows Graphics Foundation

References

1. Alienware. Alienware's Video Array. `http://www.alienware.com/alx_pages/main_content.aspx`.
2. C. Bajaj, I. Ihm, J. Min, and J. Oh. SIMD optimization of linear expressions for programmable graphics hardware. *Computer Graphics Forum*, 23(4), Dec 2004.
3. J. Bolz, I. Farmer, E. Grinspun, and P. Schröder. Sparse matrix solvers on the GPU: Conjugate gradients and multigrid. In *Proceedings of SIGGRAPH 2003*, 2003.
4. G. Coombe, M. J. Harris, and A. Lastra. Radiosity on graphics hardware. In *Proceedings Graphics Interface 2004*, 2004.
5. Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover. GPU cluster for high performance computing. In *Proceedings of the ACM/IEEE SuperComputing 2004 (SC'04)*, Nov 2004.
6. K. Fatahalian, J. Sugerman, and P. Hanrahan. Understanding the efficiency of GPU algorithms for matrix-matrix multiplication. In *Graphics Hardware 2004*, 2004.
7. R. Fernando, editor. *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics*. Addison-Wesley Professional, 2004.
8. J. Fung and S. Mann. Using multiple graphics cards as a general purpose parallel computer : Applications to computer vision. In *Proceedings of the 17th International Conference on Pattern Recognition (ICPR 2004)*, volume 1, pages 805–808, 2004.
9. N. K. Govindaraju, A. Sud, S.-E. Yoon, and D. Manocha. Interactive visibility culling in complex environments using occlusion-switches. In *ACM SIGGRAPH Symposium on Interactive 3D Graphics*, 2003.
10. GPGPU - general purpose computation using graphics hardware. `http://www.gpgpu.org/`.
11. M. Harris. *Real-Time Cloud Simulation and Rendering*. PhD thesis, UNC Chapel Hill, Sep. 2003.
12. M. J. Harris, G. Coombe, T. Scheuermann, and A. Lastra. Physically-based visual simulation on graphics hardware. In *Proceedings of Graphics Hardware 2002*, pages 109–118, 2002.
13. R. Hartenstein. Data-stream-based computing: Models and architectural resources. In *International Conference on Microelectronics, Devices and Materials (MIDEM 2003)*, Ptuj, Slovenia, Oct. 2003.
14. R. Hill, J. Fung, and S. Mann. Reality window manager: A user interface for mediated reality. In *Proceedings of the 2004 IEEE International Conference on Image Processing (ICIP 2004)*, 2004.
15. G. Humphreys, M. Houston, R. Ng, R. Frank, S. Ahern, P. D. Kirchner, and J. T. Klosowski. Chromium: a stream-processing framework for interactive rendering on clusters. In *SIGGRAPH'02*, pages 693–702, 2002.
16. R. A. Kendall, M. Sosonkina, W. D. Gropp, R. W. Numrich, and T. Sterling. Parallel programming models applicable to cluster computing and beyond. In A. M. Bruaset and A. Tveito, editors, *Numerical Solution of Partial Differential Equations on Parallel Computers*, volume 51 of *Lecture Notes in Computational Science and Engineering*, pages 3–54. Springer-Verlag, 2005.
17. T. Kim and M. Lin. Visual simulation of ice crystal growth. In *Proc. ACM SIGGRAPH / Eurographics Symposium on Computer Animation*, 2003.
18. P. Kipfer, M. Segal, and R. Westermann. UberFlow: A GPU-based particle engine. In *Graphics Hardware 2004*, 2004.
19. J. Krueger and R. Westermann. Linear algebra operators for GPU implementation of numerical algorithms. *ACM Transactions on Graphics (TOG)*, 22(3):908–916, 2003.

20. A. Lefohn, J. Kniss, C. Handen, and R. Whitaker. Interactive visualization and deformation of level set surfaces using graphics hardware. In *Proc. Visualization*, pages 73–82. IEEE CS Press, 2003.
21. W. Li, X. Wei, and A. Kaufman. Implementing Lattice Boltzmann computation on graphics hardware. *The Visual Computer*, 2003.
22. Microsoft. Longhorn Developer Center. `http://msdn.microsoft.com/longhorn`.
23. NVIDIA. NVIDIA scalable link interface (SLI). `http://www.nvidia.com/page/sli.html`.
24. OpenGL Architectural Review Board (ARB). *OpenGL: graphics application programming interface*. `http://www.opengl.org/`.
25. M. Pharr and R. Fernando, editors. *GPU Gems 2 : Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley Professional, 2005.
26. M. Rumpf and R. Strzodka. Level set segmentation in graphics hardware. In *Proceedings ICIP'01*, volume 3, pages 1103–1106, 2001.
27. M. Rumpf and R. Strzodka. Using graphics cards for quantized FEM computations. In *Proceedings VIIP'01*, pages 193–202, 2001.
28. R. Samanta, T. Funkhouser, K. Li, and J. P. Singh. Hybrid sort-first and sort-last parallel rendering with a cluster of PCs. In *Proceedings of SIGGRAPH/Eurographics Workshop on Graphics Hardware 2000*, pages 97–108, 2000.
29. R. Strzodka, M. Droske, and M. Rumpf. Image registration by a regularized gradient flow - a streaming implementation in DX9 graphics hardware. *Computing*, 2004. to appear.
30. R. Strzodka and A. Telea. Generalized distance transforms and skeletons in graphics hardware. In *Proceedings of EG/IEEE TCVG Symposium on Visualization VisSym '04*, 2004.
31. J. D. Teresco, K. D. Devine, and J. E. Flaherty. Partitioning and dynamic load balancing for the numerical solution of partial differential equations. In A. M. Bruaset and A. Tveito, editors, *Numerical Solution of Partial Differential Equations on Parallel Computers*, volume 51 of *Lecture Notes in Computational Science and Engineering*, pages 55–88. Springer-Verlag, 2005.
32. M. Wilkes. The memory gap (keynote). In *Solving the Memory Wall Problem Workshop*, 2000. `http://www.ece.neu.edu/conf/wall2k/wilkes1.pdf`.