

CG1S, a new Language for Data-Parallel GPU Programming

Nicolas Fritz*, Philipp Lucas*, Philipp Slusallek

Universität des Saarlandes, Germany

[cage|phlucas|slusallek]@cs.uni-sb.de

Abstract

In the last few years, GPUs have become new, promising targets for general purpose programming. Their inherent parallel architecture makes them particularly suited for scientific numerical computations with high arithmetical density.

There have been several proposals to exploit the computational power of GPUs for data-parallel algorithms. These approaches vary greatly in the abstraction level of the graphics processing unit exposed to the programmer.

Despite the emergence of GPU programming languages, there is still need for a single high-level programming language that naturally lends itself to compilation into efficient code, yet does not overburden the programmer with peculiarities of GPUs.

We present a novel unifying approach to facilitate the use of GPUs for data-parallel computation. We describe the language CG1S and the associated infrastructure that allows scientific programmers to express data-parallel computations efficiently on an appropriate level of abstraction.

CR Descriptors: ACM Subject Classification: C.1.4: Processor Architectures—Parallel Architectures; D.3.0: Programming Languages—General; I.3.1: Computer Graphics—Hardware Architecture

Keywords: GPUs, general purpose, data parallel, scientific computing, parallelization

1 Introduction

With the increasing programmability of graphics hardware, as exemplified by the NVIDIA NV40 architecture, new ways to access the computational power of GPUs have emerged. This power, which is growing at a faster rate than that of CPUs [1] to augment the realism in computer generated images, can be used outside of its original application

domain. Researchers are using GPUs to accelerate highly parallel algorithms on regular data structures [3].

GPUs can be (and often still are) programmed in assembly language, but in recent years higher-level languages have emerged. Still, the level of abstraction in these languages is rather low, because most of them are intended to merely provide means to code small *shaders* to realize visual effects. The implementors of data parallel algorithms would benefit from higher abstraction, to be able to concentrate on *algorithmic* optimization instead of, for example, register usage optimization.

There are two promising GPU programming languages, CG [17] and BROOK [2]. Although with both the graphics hardware is regarded as a streaming co-processor, the closeness of CG to the underlying hardware is apparent in its syntax and semantics. BROOK, albeit providing a higher level of abstraction, still requires the user to think in two different application domains: In the world of streaming computations as well as in the traditional CPU world.

With CG1S, *Computer Graphics in Scientific programming*, we propose a language that enables the user to write a program on a reasonably high level of abstraction as is customary from CPU languages. A scientific programmer willing to enhance the performance of his application should be able to describe his algorithm in a single high-level language, which is robust enough to persist the upcoming advances in graphics hardware development.

The main contributions of this paper are the following. We argue for the need of using a single language for data-parallel computations instead of a multitude of languages. We present CG1S as an exemplary language and argue for its suitability. Furthermore we show how a compiler framework for CG1S would look like.

We assume that the reader has background knowledge in graphics, and thus we will use the fa-

*Supported by DFG grant WI 576/10.

miliar notions of the graphics domain.¹

In Section 2, we describe properties of current GPUs and of available languages. Section 3 deals with our new language, CGIS, comparing it in detail with other languages. Section 4 concludes the paper and gives an outline to future work.

2 Background Information

2.1 Hardware: State of the Art

We assume basic familiarity with programmable GPUs on the readers' part. For a more detailed explanation of the capabilities of various generations, see [14]. The highlights of the latest programmable GPUs are advanced programmability of the vertex and the fragment processor, more powerful texture access, the precision of computation, and data-dependent control flow.

The NV40 architecture of NVIDIA [16] allows data-dependent loops in vertex and fragment programs, guarding of instructions via condition codes, and texture access in vertex programs. Input and output to the programs, as well as the internal computations, can be performed with single-precision floating point values. In general, architectural limitations on program size are disappearing, but APIs might still impose artificial restrictions [16].

The reader should be aware that ATI's GPUs still work on data with 24 Bit floating point precision and do not support as dynamic a control flow.

2.2 Programming Languages

While the graphics pipeline was becoming more programmable, researchers began to implement various algorithms on GPUs, progressing on a way paved earlier for nonprogrammable GPUs [5, 9, 19]. Programming GPUs in assembler is tedious work and error prone as well. With the release of DirectX 8 compatible hardware, enough boundaries had fallen to bring out general purpose programming languages. Those languages differ in the level of abstraction, code generation and specification.

¹We note that in our experience, this is a stumbling block in talking to non-graphics people. We encourage papers and talks targeted at potential users without a firm background in computer graphics or graphics card technology to use denotations from the domain of traditional programming. Indeed, this is precisely one aspect that a language should allow: A programmer should not necessarily need to know about texture memories and register kinds to write code which runs efficiently on a GPU.

Currently, the most interesting representatives are NVIDIA's CG and BROOK [2, 11, 17]. The abstraction level of the OpenGL shading language is alike to that of CG, so we do not discuss it further [7]. [12] proposes an embedded language, SH, with which the user programs the GPU inside normal C++-code. However, there is still a clear distinction between GPU and CPU code in SH making it effectively consist of two languages. It remains to be seen whether SH will get as widespread a usage as CG or BROOK.

2.2.1 CG (HLSL)

The first real high-level GPU programming language is CG [11, 17]. It is a C-like language and follows its archetype not only in syntax but also in the philosophy to be a hardware-oriented language. In general it corresponds to Microsoft's High Level Shading Language (HLSL) and was co-developed by Microsoft and NVIDIA. For supporting different architectures (both different GPU targets and OpenGL/DirectX), CG uses *profiles*. Each profile defines a subset of the CG language that can be mapped to the corresponding architecture.

Although CG is lifting the abstraction level, the user's knowledge about intricacies of the hardware is crucial to the efficiency of the generated code. The programmer has to split code into different kernels, write different code segments for architectures of varying proficiency and has to take care of data distribution and data flow by hand on the very low level of single registers, pixels, or texels.

CG has been used in various general purpose applications, but authors found the quality of the generated code not always satisfying [20]. NVIDIA tries hard to position (their) GPUs with CG as a tool for general purpose programmability [8], so advances in support for this domain are likely.

2.2.2 BROOK

BROOK is a language developed at Stanford University in connection with the Merrimac supercomputing project [13]. The usage paradigm of BROOK for GPUs sees the GPU as a streaming co-processor scheduled by the application. The programmer writes *kernels* with their operations on single stream elements. These kernels either work in a classical streaming model, or perform reduction operations.

In contrast to CG, the target hardware is completely hidden to the programmer. In fact, it is possible to set a particular hardware target at the application's run time. Additional to GPUs, BROOK also works on CPUs, automatic multithreading included.

The auxiliary files of the BROOK system define various C++ templates that are used to define the input and output data. The user is responsible for assigning the data to these streams. Furthermore, control flow has to be specified explicitly. Classical approaches such as light-weight communication channels (especially occlusion queries) or stencil buffer masking are not supported but could be implemented by hand on top of the kernel codes, but this means that the user now has to program in *three* different languages: C++ for the main application, CG for the streaming kernels, and custom code for the graphics pipeline.

BROOK uses CG to generate GPU code for each kernel. Thus, BROOK's syntax as well as its performance directly depend on those of CG, and each kernel has to be mapped to a single pass.

3 CGIS

In this section we present the key aspects of CGIS, followed by a comparison with CG and BROOK.

3.1 Design Goals

In designing CGIS, we aimed at the following goals.

Single language: The user should be able to express a complete algorithm, including both the computation that is ultimately mapped to the GPU's processors and the control flow to be performed on the CPU, in the same language. This is particularly important, because it is a prerequisite for many of the remaining goals.

Portability: One of our main goals is to accommodate the steady stream of new GPU architectures starting from the existing ones. The language must be robust to the evolution of GPUs for enabling the user to get the most out of new hardware without rewriting his code.

A unified language is a prerequisite for portability: The programmer should not need to rewrite his implementation on an evolutionary step of hardware, e. g., from NV3x to NV4x, when a larger part

of control flow can be performed in single pass programs.

High abstraction: Although CGIS aims at scientific programming, and indeed its whole reason for existence is the presumed time efficiency of implementations of time sensitive algorithms, a higher abstraction level than that provided by CG and assembler would endorse widespread usage.² Abstraction is also a prerequisite for portability.

Usability: A programmer should not need to know details of all GPU characteristics. Of course, a general knowledge of what is possible at all may be required. After all, CPU programmers need to know some aspects of language implementation and hardware features to decide on efficient data structures or algorithms in high-level languages as well.

Efficiency: Scientific programmers willing to use their graphics hardware to perform their calculations likely do so to increase the performance of their application. High performance can only be gained with the generation of efficient code. We want CGIS to be as light-weight as possible, so that it is feasible to generate efficient code.

Familiarity: The language should be familiar to a programmer. We have designed CGIS with C and CG in mind to shorten the adaption process.

We stress one important feature relating to the goals of a unified language and portability: The concrete implementation of control flow should be left to the compiler, which then can generate adequate code for different kinds of architectures. On some architectures (e. g., Pixel Shader 2.0) it might be necessary to divide the control flow decisions between the host and the graphics card, since they cannot all be realized on the GPU. We are convinced that it is not helpful to condense this *implementation* artifact in the *language*, because upcoming architectures (e. g., Pixel Shader 3.0) will have less and less restrictions.

3.2 Language Features

3.2.1 Overall Structure

A CGIS source file is divided into three parts: **CODE**, **CONTROL** and **INTERFACE**.

²The authors of [20] report on having used CG for an algorithm, although it used 52 instructions instead of a hand-written assembly program with 19 instructions. They argue that even this decrease in runtime efficiency is offset by the increase in productivity.

The **CODE** section describes the portion of a parallel computation for a single data item. Herein, the code has access to the data constituting this item only.³ The **CONTROL** section has access to the complete *array* of items. As such, here is the place to specify the computations to be performed in parallel on *every* data element. Control flow may diverge only inside the **CODE** section. The **INTERFACE** section declares external data. These data can comprise user defined structures: It is not necessary for the application code to split structures into their constituents (see Section 3.6.2). The compiler generates data access code for the data structures and takes care of distributing and rearranging the data as well as possible for a given target.

We emphasize that the division into the various sections is fundamentally different from BROOK’s two-languages approach. We do not introduce three different languages: The separation into sections serves to structure the code and to point out the naturally distinct levels of the computation.

In our approach, a single compiler has complete control over the generated code, and, in particular, over the data layout. Contrary to BROOK, where optimization decisions by the C++ compiler cannot take the generated code from the CG compiler into account, and vice versa, the CGIS compiler has the overview over the complete algorithm.

3.2.2 Features of CGIS

The language comprises features to allow easy specification of data-parallel computations.

- *Control structures*: The targeted algorithms work on sets of homogeneous data that can be processed in parallel. CGIS supports this, for example, by a `forall` construct for parallel loops.
- *Data structures*: CGIS allows multi-dimensional arrays of structures and can cope with specialized access patterns (index transformations).
- *SIMD code generation*: It is possible to automatically exploit instruction level parallelism by generating SIMD code from scalar specifications [10]. This is a feature absent in other GPU languages, yet very valuable to general purpose programmers: Often it is more intuitive to specify a computation on scalars and

let the compiler figure out the details of efficient implementations on a SIMD architecture.

- *Specialized operators*: Because the targets of CGIS, GPUs, and SIMD CPUs, feature dedicated vector instructions such as dot products or cross products, these operations are lifted to language level through operators, as in [12], operating on CG-like data types (e. g., `float4`).
- *Hints*: Although a compiler can extract certain kinds of information from the code, sometimes the user has a greater knowledge of particularities of the algorithm or data structures. This information can be communicated to the compiler via *hints* to guide the optimization. For example, the compiler can be instructed that a particular function featuring a loop should best be implemented in one pass, possibly with unrolling for less capable architectures, whereas another loop should be carried out with help of the CPU.

In line with the current hardware capabilities, CGIS does not allow (mutually) recursive procedures. The programmer has to explicitly simulate the (parameter) stack in CGIS. With regard to efficiency, this is no substantial restriction, because it is unlikely that GPUs will gain the power to allow true recursive procedures (supporting stack frames and the like) in the foreseeable future.

3.3 Example

As an example, we have chosen an implementation of ray-triangle intersection with *early kill*, as displayed in Figure 1. All rays (specified by the `extern input` line in the **INTERFACE** section) are intersected with a dynamic list of triangles (also in **INTERFACE**). Both are arrays of arbitrary size, denoted by `[,]` (two-dimensional) or `[]` (one-dimensional), respectively. As soon as a hitpoint for a particular ray is found, the computations for that ray are stopped.

Here, the **CONTROL** section is pretty short: It tells the compiler that the computation should be done in parallel on the input rays. There are several ways to access the external data. It is possible to use explicit indexing or sequentially access each element of a given array in standard C order as denoted by `:2D` here.

The **CODE** section includes two functions for which hints support the compiler’s optimization,

³Special syntax can be used to lift this requirement in a limited way, to implement convolution operators for example.

```

PROGRAM EarlyKillRayTriangleIntersection;

INTERFACE

typedef struct {
    float3 a, b, c;
    int id;
} tri_t;

typedef struct {
    float3 origin, direction;
    float near, far;
} raytype_t;

extern input raytype_t[,] raydata;
extern output int[,] rayhits;
extern input tri_t[,] t_list;

CONTROL

forall (raytype_t r in raydata:2D;
        int tid in rayhits:2D) do {
    earlyKillIntersect (r, tid, t_list);
}

CODE

function earlyKillIntersect
    (input raytype_t ray, output int tid,
     input tri_t[] triangleList)
#HINT(GPU: pure) {
    foreach (tri_t t in triangleList) {
        intersection_triangle (tid, ray, t);
        if (tid != -1) break;
    }
}

function intersection_triangle
    (output int tid, input raytype_t ray,
     input tri_t t)
#HINT(GPU: pure, singlepass) {
    float3 edge1 = t.b-t.a;
    float3 edge2 = t.c-t.a;
    // &: cross product, |: dot product
    float3 pvec = ray.direction & edge2;

    float det = edge1 | pvec;
    if ((det >= -0.005) && (det <= 0.005))
        {tid = -1; return;}

    float3 tvec = ray.origin - t.a;
    float lambda = tvec | pvec / det;
    if (lambda < 0.0 || lambda > 1.0)
        {tid = -1; return;}

    float3 qvec = tvec & edge1;
    float mue = ray.direction | qvec / det;
    if((mue < 0.0) || ((lambda + mue) > 1.0))
        {tid = -1; return;}

    float td = edge2 | qvec / det;
    if ((td < ray.near) || (td >= ray.far))
        {tid = -1; return;}

    tid = t.id;
}

```

Figure 1: A Möller-Trumbore ray-triangle intersection [15] algorithm with early-kill in CGIS. The interface section describes the external input, whereas the control section specifies a parallel iteration over a set of rays. The code section describes the computations to be done for each single ray.

classified by a tag (here: GPU) for specific targets. The pure hint reassures that this function has no access to data elements of neighboring elements in a parallel computation. `singlepass` means that the function `intersection_triangle` shall be implemented in one pass, if possible. Its absence in `earlyKillIntersect` allows the compiler to use the CPU for the iteration: The CPU would then feed the elements of the triangle list to the GPU one by one, possibly in the vertex parameters.

3.4 Compiler Infrastructure

Figure 2 shows the infrastructure of the compiler and the runtime system. The compiler takes source code in CGIS (see Figure 1 for an example) as input and creates both auxiliary C++-code and code for the target architecture. Currently we plan to support fragment shader and SSE output, although other formats may be included in the future.

Focusing now on the GPU output, there are two possibilities for a compiler of a higher-level language: Either it can use CG for the GPU program code and rely on the CG compiler, or it can directly compile down to assembly language code.⁴ Should CG develop slower than expected, the compiler might generate GPU assembly code right away.

The user does not interact with the graphics card or the generated GPU code directly. Instead, he calls the provided data transfer functions in the auxiliary code and orders the computation to be started. The various runtime systems and outputs just have to be linked with the application code. The runtime system for the GPU branch uses OpenGL for platform compatibility.

Note that the programmer is both freed from the tedious details of the hardware and is not responsible for control flow and data layout code, because the glue code takes care of these issues.

Let us now briefly consider SSE output. In recent years, mainstream CPUs were improved with multimedia extensions. Special instructions perform arithmetical operations on small tuples of data, usually 4-tuples [6]. Because the SIMD parallelism on CPUs strongly resembles the SIMD parallelism on GPUs, support for CPUs seems reasonable.

The CPU output is especially important for debugging. The user does not have to change from

⁴Note that even the assembly code is subject to optimization such as reordering of instructions inside the assembler in the device driver.

his familiar CPU debugging environment, but can use the advanced features of CPU debugging tools. Currently, debugging tools for GPU code are still less powerful than their CPU counterparts.

3.5 Evaluation

In the following we evaluate the design of CG1S with respect to the design goals in Section 3.1.

Single language: CG1S enables the user to specify complete algorithms in a single language. The user needs to call C++ functions for data transfer and to start the computation, but this is decoupled from the algorithm.

Portability: The fact that the code generation as well as the data layout are handled by the compiler, makes CG1S more robust with respect to future changes in quantitative parameters of the underlying architecture.

High abstraction: The level of abstraction is adequate for the expression of scientific algorithms. Object-oriented features could be realized, but our functional approach seems well suited to the hardware.

Usability: We abstract the hardware so that detailed knowledge about the intricacies of GPUs are not necessary. Additional information can be communicated to the compiler via hints.

Efficiency: Clearly, as we do not have a complete compiler yet, we cannot make claims about the efficiency of CG1S implementations. Our provisional results convince us, though, that an efficient implementation is possible.

Familiarity: The CG1S syntax bears strong resemblance to C and CG. The main omission with respect to C is the lack of pointers. In CG1S, the programmer can make indexed lookups into arrays. There is no need for general pointer structures on the GPU level, as these could not be implemented efficiently anyway. See [11] for a detailed discussion of the absence of pointers in CG.

3.6 Comparison with other Languages

3.6.1 CG

CG1S strongly differs from CG. Among the goals of CG was an easy and incremental change from traditional hand-written shaders to CG shaders. Therefore, CG targets only one particular aspect of GPU programming: Coding a particular shader [11].

Optimization deficiencies of the current compiler aside, when writing an application using CG kernels, the main difficulties of data layout and distribution, communication, and global control flow lie still in the hands of the programmer.

3.6.2 BROOK for GPUs

The prominent feature of BROOK is the abstraction of the peculiarities of the target architecture by presenting it as a streaming processor. It is truly invisible to the *user* that a program written in BROOK runs on a GPU instead of the CPU. But still, the *programmer* has to be well aware of the target, because of the data access and control flow restrictions.

The streaming model of BROOK is adequate for tasks easily subjugated under the streaming template. But it is not suited for a large set of algorithms that a scientific programmer might want to implement profitably on GPUs. The benefit of this abstraction notwithstanding, it omits the usage of graphics hardware features such as occlusion queries, which can be efficiently applied to communicate between GPU and host. In our example in Section 3.3, BROOK would not support data transfer through vertex parameters, which is the best way to pass the triangle data to the GPU.

The streaming model of BROOK and the use of CG as the effective language for the kernels add up to a certain limitation of applicability. Inherent in the BROOK model is the absolute distinction of CPU computations and GPU computations. The CG1S triangle loop in `earlyKillIntersect` (Figure 1) can be compiled to work on GPU alone or on GPU with help of the CPU, because CG1S is a unified language. This is not possible with BROOK.

We stress that the approaches of BROOK and CG1S are fundamentally different. BROOK abstracts the GPU as a streaming co-processor and provides means to program this co-processor. CG1S, on the other hand, gives the possibility to express an *algorithm*, letting the compiler choose how to implement it using whatever mechanisms on the GPU are suitable.

Put another way, the languages approach the task from two different directions. BROOK is an abstraction of streaming hardware, which can be used to implement data-parallel algorithms; CG1S is an abstraction of data-parallel hardware.

Currently, BROOK does not support streams of structured data, which is uncomfortable to a pro-

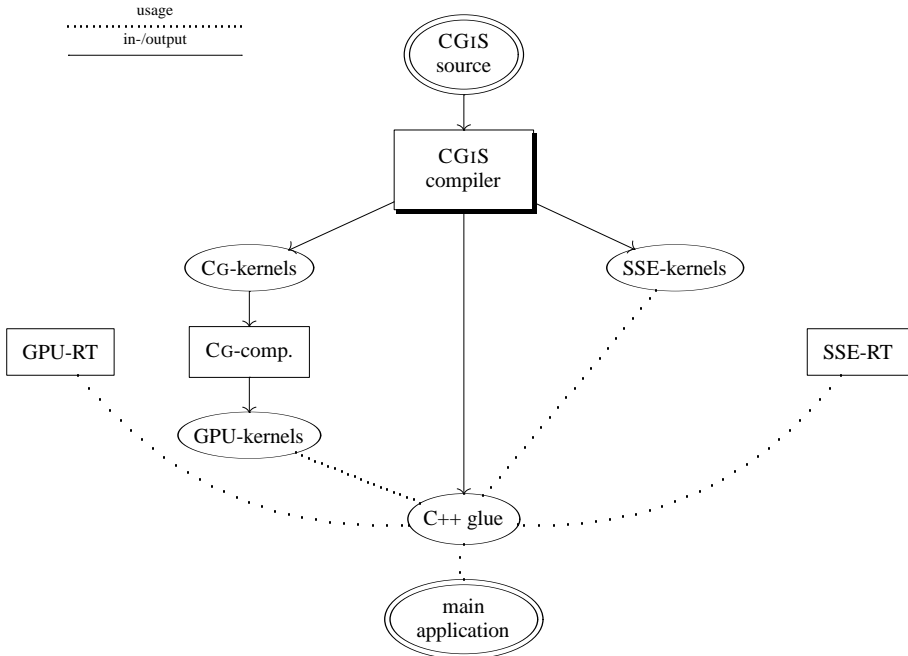


Figure 2: Overview of the CG1S compiler infrastructure. Dotted line denotes usage. Round nodes are documents, angular nodes are programs. Double framed nodes denote user supplied code.

grammer. As noted in Section 3.2, CG1S also plans to support arrays of `structs`.

3.6.3 Data-Parallel Languages for CPUs

Existing CPU languages provide several means to express parallel computations (cf. Fortran or OpenMP [18]). However, the architectures targeted by those languages are significantly different to GPUs. Therefore, we would have to adapt these languages as well to fit to GPUs.

A prime distinction of GPUs and CPUs lies in the different memory architecture and the way in which the execution units can be scheduled. In contrast to the CPU world, the parallel execution units on GPUs are indistinguishable. Furthermore, although they share a common memory, the texture memory with its read-/write limitations is strongly distinct from the random access CPU memory. Therefore, many provisions in these languages cannot readily be applied to GPU programs.

4 Conclusion and Future Work

We have discussed the need of a new language for data-parallel computations of GPUs even in the presence of BROOK and CG. The main idea here is to provide a single, unifying language to express complete algorithms on homogeneous data. The key aspects are increased optimization potential and robustness. Global optimizations are possible only with knowledge of the whole source. Furthermore, only if all aspects of data layout, control flow, distribution into kernels and usage of the GPU's facilities are under the control of the compiler, the generation of efficient code without the programmer's explicit intervention is possible. These subaspects are also prerequisites for robustness with respect to GPU development. If the *user* is in charge of, for example, data layout, it is the *user* who has to adapt it with each new hardware generation. Thus, we believe that a unified language is the only approach likely to give widespread usage to GPUs for scientific programming.

When designing CG1S, we took care in giving

programmers the opportunity to use high-level control and data structures that can be mapped on current as well as future hardware. By keeping the language simple, we will be able to quickly adapt the compiler to new hardware features. The absence of strictly GPU-related features makes retargetability to SSE code feasible.

We believe that the majority of algorithms that are at all efficiently implementable on GPUs can be expressed easily in CG1S. After the implementation has been finished, it will be evaluated with various successfully on GPUs implemented algorithms, such as ray tracing [19], linear algebra [9] or simulations [4].

Acknowledgments:

Thanks to Reinhard Wilhelm, Jörg Schmittler, and Sven Woop for many fruitful discussions.

References

- [1] M. BRETERNIZ JR., H. HUM, AND S. KUMAR, *Compilation, architectural support, and evaluation of SIMD graphics pipeline programs on a general-purpose CPU*, in 12th International Conference on Parallel Architecture and Compilation Techniques (PACT'03), 2003.
- [2] I. BUCK, T. FOLEY, D. HORN, J. SUGERMAN, K. FATAHLIAN, M. HOUSTON, AND P. HANRAHAN, *Brook for GPUs: Stream computing on graphics hardware*, in SIGGRAPH, 2004.
- [3] *General-purpose computation using graphics hardware*. <http://www.gpgpu.org>, 2004.
- [4] M. J. HARRIS, G. COOMBE, T. SCHEUERMANN, AND A. LASTRA, *Physically-based visual simulation on graphics hardware*, in Proceedings of the Eurographics Workshop on Graphics Hardware, 2002, pp. 109–118.
- [5] K. E. HOFF III, T. CULVER, J. KEYSER, M. LIN, AND D. MANOCHA, *Fast computation of generalized Voronoi diagrams using graphics hardware*, in SIGGRAPH, 1999, pp. 277–286.
- [6] INTEL, *Programming with the streaming SIMD extensions (SSE)*, in Intel Architecture Software Developer's Manual, Vol. 1: Basic Architecture, 1999.
- [7] J. KESSENICH, D. BALDWIN, AND R. ROST, *The OpenGL Shading Language, V. 1.10*, April 2004.
- [8] D. KIRK, *Technology directions*. http://developer.nvidia.com/docs/10/4106/Technology_Directions.pdf, 2003. Slides of a presentation at NVIDIA Analyst's Day.
- [9] J. KRÜGER AND R. WESTERMANN, *Linear algebra operators for GPU implementations of numerical algorithms*, in SIGGRAPH, 2003.
- [10] S. LARSEN AND S. AMARASINGHE, *Exploiting superword level parallelism with multimedia instruction sets*, Tech. Rep. LCS-TM-601, MIT Laboratory for Computer Science, November 1999.
- [11] W. R. MARK, R. S. GLANVILLE, K. AKELEY, AND M. J. KILGARD, *Cg: A system for programming graphics hardware in a C-like language*, in SIGGRAPH, 2003.
- [12] M. D. MCCOOL, Z. QIN, AND T. S. POPU, *Shader metaprogramming*, in Proceedings of the Eurographics Workshop on Graphics Hardware 2002, ACM, 2002, pp. 57–68. Revised version.
- [13] MERRIMAC, *Project homepage*, 2004.
- [14] MICROSOFT, *DirectX reference manual*. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/directx9_c/directx/graphics/reference/shaders/shaders.asp, 2004.
- [15] T. MÖLLER AND B. TRUMBORE, *Fast, minimum storage ray-triangle intersection*, Journal of Graphics Tools, 2 (1997), pp. 21–28.
- [16] *Cinefx 3.0*, tech. rep., NVIDIA, March 2004.
- [17] NVIDIA, *Cg Toolkit User's Manual, Release 1.2*, January 2004.
- [18] OPENMP ARCHITECTURE REVIEW BOARD, *OpenMP C and C++ application program interface*, March 2002.
- [19] T. J. PURCELL, I. BUCK, W. R. MARK, AND P. HANRAHAN, *Ray tracing on programmable graphics hardware*, in SIGGRAPH, 2002.
- [20] T. J. PURCELL, C. DONNER, M. CAMMARANO, H. W. JENSEN, AND P. HANRAHAN, *Photon mapping on programmable graphics hardware*, in Proceedings of the Eurographics Workshop on Graphics Hardware, 2003.