# The Development of the Data-Parallel GPU Programming Language CGiS

Philipp Lucas*, Nicolas Fritz*, and Reinhard Wilhelm

Compiler Design Lab, Saarland University, Saarbrücken, Germany
{phlucas, cage, wilhelm}@cs.uni-sb.de

**Abstract.** In this paper, we present the recent developments on the design and implementation of the data-parallel programming language CGiS. CGiS is devised to facilitate use of the data-parallel resources of current *graphics processing units (GPUs)* for scientific programming.

## 1 Introduction

The last few years have seen a rapid development in programmable graphics hardware. To exploit the vast computational power of these highly parallel architectures, scientists successfully ported algorithms to GPUs [5]. This is commonly known as *General Purpose Programming on GPUs (GPGPU)*.

While highly tailored solutions could be implemented by specialists, the common programmer without knowledge of the details of GPU programming was left out. For wider access, higher-level languages have emerged, such as Brook for GPUs [1], Sh [4], CGiS [2, 3] and the recent Accelerator [6].

CGiS is designed to improve GPU accessibility by further raising the abstraction level. Scientific programmers not accustomed to programming graphics hardware can transparently use performance enhancing features of the target. Generating efficient GPU code for a general purpose language program is a demanding task. The goal of the CGiS project is to explore the possibilities of high-level data-parallel programming.

This paper presents recent developments of CGiS, its compiler and its applicability with respect to [2]. By example, we also show that, even with the higher level of abstraction, parallel algorithms can be implemented efficiently on GPUs with CGiS.

The remainder of this paper is organised as follows. Section 2 describes CGiS' decisive features, and Section 3 shows a more detailed example. Section 4 concludes the paper with an outlook on future development and work in progress.

## 2 CGiS

In contrast to other GPU languages, CGiS offers the possibility to write a *whole algorithm* in one language. To achieve portability and performance optimisation possibilities, the compiler has full control over code and data distribution.

* Supported by DFG grant WI 576/10-3.

V.N. Alexandrov et al. (Eds.): ICCS 2006, Part IV, LNCS 3994, pp. 200–203, 2006.

Because the programmer never sees the actual GPU kernels, the compiler may reorder functions into kernels for improved code distribution. The compiler can thus use upcoming hardware generations, so that CGiS programs do not have to be rewritten.

### 2.1 CGiS Compiler Architecture

Figure 1 shows the usage of CGiS. The CGiS compiler takes a CGiS source file as an input and generates GPU and C++ code. Together with the user-provided code which consists of calls to data passing and initialisation routines, and with the runtime library, an executable can be linked. It uses the GPU transparently. The GPU is accessed through OpenGL, and the GPU code is written in standard assembly language. Thus, the compiler supports by design several back-ends. Back-ends for multi-media CPU instruction sets are also planned.

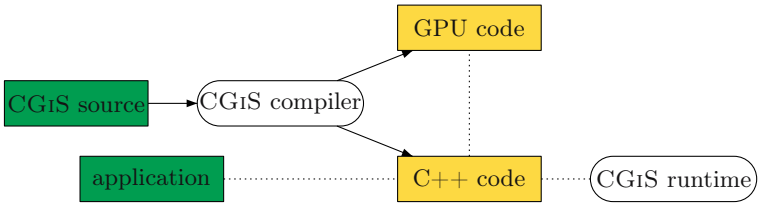


**Fig. 1.** Basic usage pattern of CGiS. Dotted lines denote linkage, solid arrows denote in- or output. Darker, green rectangles denote user-provided sources, the other rectangles are the output of the CGiS compiler. The ellipses stand for the CGiS base system components. There is no direct connection between the application and the GPU.

### 2.2 Features

CGiS is a data-parallel programming language, focused on GPUs. It allows parallel and independent computations on streams, as well as reductions, e. g. summing up the elements of a stream into a scalar. For further details, see [2].

The CGiS compiler takes care of optimisations to accommodate inexperienced GPU programmers, but advanced programmers are allowed to specify guidings and hints; for example, whether a certain conditional is to be translated with if-conversion or with native conditional instructions on architectures supporting such. The generated program drives the computation and all necessary rearrangements of data and the data interface to the main application. The GPU stays invisible to the programmer. Textures may be exposed to the outside for visualisation, or shown directly. Like other languages, CGiS excludes recursive functions and pointers because of the hardware's memory constraints.

Tests have shown that for naturally parallel algorithms, CGiS programs can offer performance benefits with respect to CPU code [3], although the quality is less than that of hand-written GPU code.

## 3 Example: Refraction

In this section we show how CGiS programs differ from programs in other GPU languages. This is exemplified by a program computing wave propagation and refraction on watery surfaces.

CGiS programs consist of three sections: An `INTERFACE` section declares scalar and stream variables, a `CODE` section defines functions working on single elements of streams and a `CONTROL` section defines how these functions work together. Because of the aforementioned hardware constraints and a desired closeness to C, the `CODE` section is similar to other languages. In the following, we will concentrate on the other two sections.

In the `INTERFACE` section, the programmer may provide ids to the compiler for a detailed specification of packing streams into textures. If not, the compiler

```
PROGRAM viswave;

INTERFACE

extern inout float LAST<_,_> : texture (1) A; // _ is a size wildcard.
extern in float CURRENT<_,_> : texture (2) A; // Flipped on each step.
extern in float RINDEX, DAMP, WID, HEI; // Pass as program parameters.
intern float X<_,_> : texture (4) R; // These two streams shall reside
intern float Y<_,_> : texture (4) G; //    in the same texture (id=4).
extern in float3 TEXTURE<_,_>: texture (3) RGB; // Use RGB components
extern out float3 IMAGE<_,_> : texture (5) RGB; //  for visualisation.

CODE
... // Declare kernels called from this section and from CONTROL.

CONTROL

// Single step wave propagation:
forall (float last in LAST; float current in CURRENT){
  propagate (last, current, indexX(last), indexY(last), DAMP, WID, HEI);
}
// Compute refractions in X- and Y-dimension:
forall (float x in X; float y in Y; float height in LAST){
  refractionX (RINDEX, x, height, indexX(height), WID);
  refractionY (RINDEX, y, height, indexY(height), HEI);
}
// Compute refracted image:
forall (float3 pixel in IMAGE; float height in LAST;
        float x in X; float y in Y){
  render (TEXTURE, pixel, height, x, y);
}
// Display image on screen:
show(IMAGE);
```

**Fig. 2.** Part of a CGiS program for calculating refractions

will try to minimise the texture accesses. Another use for the ids is data passing between separately compiled programs, which is implemented by shared textures. For display on the screen, the image should reside in specific colour components, which also is specified. Scalar values are always passed as fragment program parameters.

The `INTERFACE` section gives rise to C++ functions, which the application invokes to pass and retrieve the data. All data transfer is handled by the generated code, and the GPU is invisible.

The `CONTROL` section specifies the calls of kernels which operate on streams, passing index values, stream elements or whole arrays of data. The compiler generates code to upload the kernels, hook necessary textures, run the kernels and copy the data back to textures. Again, the GPU remains invisible. This example also features a `show` statement for interactively displaying the computation results.

## 4 Conclusion and Future Work

We have described the CGiS language and shown that it is feasible for GPU applications. In special domains, GPU implementations of CGiS programs offer good performance, even at the current development stage of the compiler.

With the design of CGiS finished, we focus our future efforts on the compiler framework. What remains to be done is to implement more program analyses and more optimisations to the code generator, thus removing some of the overhead for more complex programs. The compiler also has to be retargeted to the most recent generation of GPUs and to SIMD-CPUs. When the compiler framework is ready, it will be released as Open Source Software under the BSD license.

## References

1. I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream computing on graphics hardware. In *SIGGRAPH*, 2004.
2. N. Fritz, P. Lucas, and P. Slusallek. CGiS, a new language for data-parallel GPU programming. In *"Vision, Modeling, and Visualization" Workshop*, 2004.
3. P. Lucas, N. Fritz, and R. Wilhelm. The CGiS compiler—a tool demonstration. In A. Mycroft and A. Zeller, editors, *Proceedings of the 15th International Conference on Compiler Construction*, LNCS. Springer-Verlag, 2006.
4. M. D. McCool, Z. Qin, and T. S. Popu. Shader metaprogramming. In *Eurographics Workshop on Graphics Hardware*, pages 57–68, 2002. (Revised).
5. J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. In *Eurographics 2005*, pages 21–51, 2005.
6. D. Tarditi, S. Puri, and J. Oglesby. Accelerator: Simplified programming of graphics processing units for general-purpose uses via data-parallelism. Technical Report MSR-TR-2005-184, Microsoft Research, December 2005.