The CGiS Compiler—A Tool Demonstration

Philipp Lucas*, Nicolas Fritz*, and Reinhard Wilhelm

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

Abstract. The CGiS programming language is designed to open up the parallel performance possibilities of graphics processing units (*GPUs*) to general purpose programmers. This tool demonstration paper sums up the ideas behind CGiS and the compiler framework and shows its usage.

1 Introduction

Graphics processing units (*GPUs*), the processors used by standard graphics hardware in PCs, underwent a fast and incessant development in the past few years. Designed to execute small programs determining the pixel colours in an image, they make use of parallel execution units. Such programs can also be used for non-graphics related general purpose programming on GPUs (*GPGPU*) [6].

Scientists have developed various parallel algorithms on GPUs and experienced performance gains for several kinds of algorithms with high algebraic density. But nearly all of such applications were implemented using the GPU's assembly language or languages with a very low level of hardware abstraction, or the programmer had to interact with the graphics API to program the GPU. Recently more general purpose programming languages for GPUs have emerged (Brook for GPUs, Sh, CGiS [1, 8, 4]). In this tool demonstration paper we introduce GPGPU and the compiler for our language, CGiS.

In Section 2, we briefly describe GPUs as targets for general purpose programming. In Section 3, we describe the programming language and the usage of CGiS. Section 4 gives an outlook into future development.

2 GPUs

We give a very short introduction without using the terminology of the graphics world. For further information on features of current GPUs, the reader should consult the documentation of APIs [9, 12] or homepages of vendors (ATI, NVIDIA, 3Dlabs).

Because of their legacy, GPUs have a number of features distinguishing them from usual CPUs. They are built around a pipeline model of graphics operations, eventually transforming geometry data into screen pixels. The latter part of the pipeline, which works on single pixels, is implemented with parallel execution

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

A. Mycroft and A. Zeller (Eds.): CC 2006, LNCS 3923, pp. 105–108, 2006.

pipelines, which have become programmable in recent years. The processor provides the usual arithmetical instructions on single-precision floating-point four-vectors and some special computer graphics instructions. It is this part with programmable, relatively simple and slow but parallel processing units[1] that is used in general purpose programming. A wide variety of applications have been ported to GPUs, from image synthesis [10] and linear algebra [7] to database operations [5] and cryptography [2].[2]

The main restrictions of GPUs lie in the memory model and the support for control flow. Sections of memory are used either for reading or for writing during the execution of a program. This can be switched by the controlling application after a GPU program (hereafter: *kernel*) has completed its execution, or the data have to be copied from write-memory into read-memory. Thus, only a streaming kind of execution is possible. Newer GPUs offer restricted dynamic control flow, whereas only straight-line control flow was available before. The restriction concerns the nesting level of conditionals and the maximal iteration count of loops, which are bounded.[3] In general, only naturally parallel algorithms without complex control flow can benefit from GPUs; but those which can take advantage of the massive raw floating point power can outperform current CPUs.

The restrictions of this memory model and the number of outputs pose the main difficulties to a compiler writer. Only in the newest generation of NVIDIA's GPUs, each of the programs running in parallel can output more than one four-vector (upto four such vectors). Functions have to be split at appropriate points, such that only few values need to be passed between the different kernels [3, 11]. Also, the severe limits on the control flow (if supported it at all) pose a difficulty to compilers. But it is exactly this combination of restricted features with powerful capabilities which makes high-level GPU languages desirable.

3 CGiS

A CGiS program [4] describes the computation as a sequence of parallel executions of functions over streams of data, where each function operates on a single element of each stream. Figure 1 gives an example of the general layout of a CGiS program. More elaborate examples and a detailed explanation can be found in [4].

The usage of CGiS is illustrated in Figure 2. The programmer writes the code to be executed on the GPU in CGiS. The compiler generates the kernels and directing C++ code for the platform independent graphics API OpenGL [12], as well as all necessary code to switch between kernels, to realign the streams and to transfer data. The user interfaces with the generated code by giving pointers to input data, starting the computation and receiving the output data.

[1] For example, the current NVIDIA chip *GeForce 7800* features 24 lanes at about 400 MHz and a memory bandwidth of more than 38 GB/s to 256 MB RAM.

[2] See [6] for pointers to other applications.

[3] To enable the programer to write general loops in CGiS, the language allows to annotate loops with a guaranteed *maximal* number of iterations.

```
PROGRAM vector_add;
INTERFACE // Declare streams.
extern inout float4 in_out_data<_>;
extern in float4 in_data<_>;
CODE // Declare element functions.
function add(in float4 a, in float4 b, out float4 c){
    c = a + b;
}
CONTROL // Perform parallel computation on streams.
forall(float4 io in in_out_data; float4 i in in_data){
    add(io,i,io);
}
```

Fig. 1. A small CGiS program, computing the sum of two vectors of unspecified length

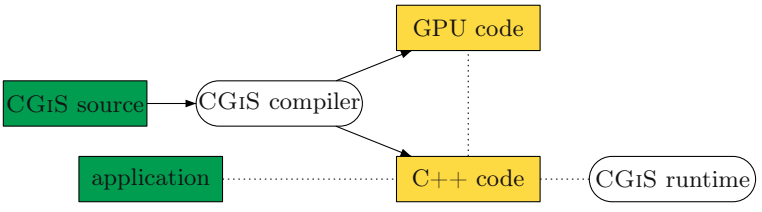

Fig. 2. Basic usage pattern of CGiS. Dotted lines denote linkage, solid arrows denote in- or output. The left rectangles denote user-provided sources, the other rectangles are the output of the CGiS compiler. The ellipses stand for the CGiS base system components. There is no direct connection between the application and the GPU.

The GPU is invisible to the programmer and to the end user. The programmer interacts only with the CGiS runtime system. For the user, the use of the GPU is invisible, because the program computes in off-screen memory space.

With each GPU generation, new features become available. The compiler generates code using features of a desired generation. Thus, the GPU code needs a GPU of the chosen kind or a newer model.[4] Currently, the main focus of the CGiS compiler is on the NV30 generation of GPUs. We have begun upgrading our compiler to the newer, more powerful NV40 generation.

The generated C++ code is independent of operating system or windowing system. All such differences are either abstracted away in the runtime library of CGiS (such as the procedure of creating an invisible window and an OpenGL context) or are part of OpenGL proper and thus in itself platform independent. Thus, the generated code can be compiled and run on any system with the runtime and an appropriately powerful GPU. Currently, the CGiS runtime is available for Windows and Linux (i386), though we expect it to be adaptable trivially to other systems with the Windows or X window model and OpenGL support. The main prerequisite for porting is the availability of current OpenGL

[4] The programmer may generate code for various GPU architectures, the best fitting of which is to be used at run-time of the application.

drivers. NVIDIA drivers are developed also for Solaris and FreeBSD on i386 and for other processors with Windows, Linux or MacOS.

4 Future Work

When the adaption to the NV40 architecture is completed, we will focus on the development of a general analysis and optimisation framework in the compiler. At present very few optimisations are implemented. Then we will develop other back-ends to support newer generations of GPUs. We also plan to create a general library for linear algebra functions and a visualisation framework.

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 *Proceedings of SIGGRAPH*, 2004.
2. D. L. Cook, J. Ioannidis, A. D. Keromytis, and J. Luck. CryptoGraphics: Secret key cryptography using graphics cards. In *Proceedings of the RSA Conference*, pages 334–350, February 2005.
3. T. Foley, M. Houston, and P. Hanrahan. Efficient partitioning of fragment shaders for multiple-output hardware. In *Proceedings of Graphics Hardware*, August 2004.
4. N. Fritz, P. Lucas, and P. Slusallek. CGiS, a new language for data-parallel GPU programming. In B. Girod, H.-P. Seidel, and M. Magnor, editors, *Proceedings of "Vision, Modeling, and Visualization"*, pages 241–248, 2004.
5. N. K. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha. Fast computation of database operations using graphics processors. In *SIGMOD*, 2004.
6. General-purpose computation using graphics hardware. `http://www.gpgpu.org`, 2005.
7. J. Krüger and R. Westermann. Linear algebra operators for GPU implementation of numerical algorithms. In *Proceedings of SIGGRAPH*, 2003.
8. M. D. McCool, Z. Qin, and T. S. Popu. Shader metaprogramming. In *Eurographics Workshop on Graphics Hardware*. ACM, 2002. Revised version.
9. Microsoft. DirectX 9.0 C++ reference. `http://msdn.microsoft.com/library/en-us/directx9_c/directx/graphics/reference/reference.asp`, August 2005.
10. T. J. Purcell, C. Donner, M. Cammarano, H. W. Jensen, and P. Hanrahan. Photon mapping on programmable graphics hardware. In *Proceedings of the SIGGRAPH/Eurographics Workshop on Graphics Hardware*, 2003.
11. A. Riffel, A. E. Lefohn, K. Vidimce, M. Leone, and J. D. Owens. Mio: Fast multipass partitioning via priority-based instruction scheduling. In *Proceedings of Graphics Hardware*, August 2004.
12. M. Segal and K. Akeley. *The OpenGL Graphics System: A Specification (Version 2.0)*, 2004.