A Streaming Virtual Machine for GPUs

Kenneth Mackenzie[†], Daniel P. Campbell[‡] and Peter Szilagyi[†]
kenmac@reservoir.com, dan.campbell@gtri.com, szilagyi@reservoir.com
[†]Reservoir Labs, Inc, New York, NY
[‡]Georgia Tech Research Institute, Georgia Institute of Technology, Atlanta, GA

Abstract

We describe a work in progress to develop a prototype toolchain for compiling HPC applications to Graphic Processing Units (GPUs). The toolchain is based on the DARPA Polymorphous Computing Architecture (PCA) program's Streaming Virtual Machine (SVM) interface [2][4] and uses Reservoir's R-Stream compiler [5] to produce SVM code.

The elements of the prototype, then, are a machine model input to R-Stream, an adapter from SVM code to nVidia's Cg compiler [7] and a runtime system.

Results to date are that the SVM interface appears apt for controlling GPUs with low overhead but that SVM's C-based description of kernels, while sufficient, would benefit from extensions to ease the analysis required for translation to Cg. We have not tackled the (deep) problem of feedback from Cg into R-Stream but expect to gain insight into the issue through experiments with the prototype.

Why GPUs Need Compilers

The same architectural techniques that give GPUs high performance make them a difficult, moving target to program.

First, GPU programmers must co-optimize across multiple constraints imposed by the architecture. GPUs, like media processors, network processors and a number of current research architectures demonstrate that it is possible to approach the performance of fixed-function hardware with a programmable chip, as long as the application's mapping to the hardware is efficient. However, the essential characteristics of the hardware that provide high performance -- high degrees of parallelism, small bounded local memories, need for explicit communications management -- introduce severe restrictions into the programming model. From an architecture perspective, these techniques allow more of the chip to be devoted to computation than in a conventional CPU. From a compilation or programmer's perspective, these features introduce dependencies between optimization phases that are separate in a conventional compiler or programmer workflow.

Second, the constraints change with time. Part of the technique GPU vendors use to maintain their impressively steep technology curve is to substantially alter hardware parameters (degree and type of parallelism, local memory sizes, feature sets) from generation to generation. The consequence is that an application carefully optimized to a particular generation of GPU is suboptimal for subsequent generations. Rapid evolution compounds the programming difficulty by making the architectures a moving target.

Streaming Virtual Machine

The PCA SVM interface models a class of architectures typified by high-performance accelerators attached to explicitly-managed local memories as in Figure 1.

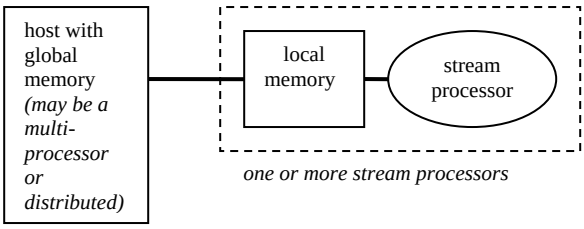

Figure 1: SVM machine model: host augmented with one or more stream processors, each operating from an explicitly-managed local memory.

The interface provides a control model for describing kernels that execute in the stream processor on blocks or streams of data placed in local memory. SVM uses C to describe the control code on the host and the kernels on the stream processor plus API calls for managing the local memories and the kernels. The intent is that SVM is generated automatically by an autopartitioning compiler (e.g. Reservoir's R-Stream) or some other tool.

Toolchain for GPUs

Current GPUs embody an implementation of a 3D graphics pipeline with embedded multiprocessors for several of the stages. General purpose computing on graphics processors uses these embedded multiprocessors for non-graphics purposes. Because of the special-purpose nature of the pipeline, these embedded multiprocessors are only partially accessible for such non-graphics purposes and come with a number of programming restrictions.

Specifically for our work, we model the fragment shader as a stream processor and the video RAM as a local memory. The fragment shader appears near the end of the graphics pipeline and applies a function to every fragment (potential pixel) in a polygon. The function may reference textures as inputs but can only write one output to its current coordinate. For general-purpose work, the textures are input arrays, the polygon is an iteration space (generally a rectangle) and the fragments are one output array.

The toolchain, then, must select kernels that fit the capabilities of the fragment shader, map kernels to GPU(s),

compile them to executable form and manage their execution and memory usage at runtime.

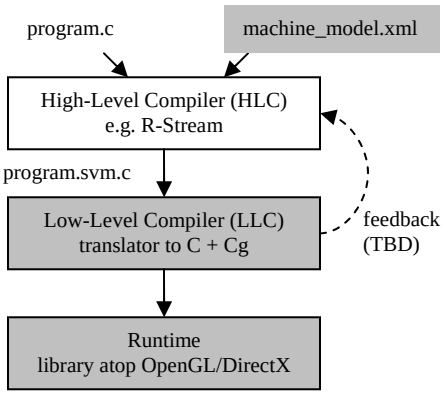

Figure 2: Toolchain work flow. The High-Level Compiler selects, maps and schedules kernels, emitting code for both the host processor and the GPU(s) in SVM format. This project is developing the grayed items: the machine model, the Low-Level Compiler and the runtime.

The toolchain consists of three components (Figure 2):

1. A high-level compiler (or programmer framework) that identifies, maps and schedules kernels. In the case of R-Stream, the input is an annotated C program along with a machine model describing the architecture in abstract terms (processors, memories, interconnect) and the output is an SVM file with separate kernels, DMA invocations and control code.
2. The low-level compiler is target specific. Our prototype low-level compiler for GPUs translates SVM kernels into fragment shader programs and translates kernel invocations to match.
3. The runtime system is also target specific. Our prototype runtime interprets SVM operations to manage GPU texture memory and to invoke fragment shader programs.

Progress and Results

The three goals of this exercise are to determine feasibility, to evaluate performance and to provide a platform for experimenting with feedback from the low-level compiler to the high-level compiler.

First, we hypothesize that SVM contains sufficient information to convert the SVM kernels and invocations into Cg programs and invocations. The crux here is moving the explicit loops out of the kernel into the implicit loops over polygons of Cg's invocation mechanism. Our conclusion is that there is enough information but it is not extractable in general without dataflow analysis and pointer disambiguation. We find ourselves relying on R-Stream idioms. We expect we will have to annotate SVM kernels in some way to ease this problem.

Second, we predict that SVM's kernel description and invocation mechanism mechanisms will add negligible overhead to a purpose-built interface such as Brook [1] or PUG [3]. At the time of writing we have no measurements to substantiate this claim but we have encountered no obvious overheads in implementation.

Third, we expect to gain insight into feedback required from Cg to R-Stream. This is a deep issue exacerbated by the two-level compiler approach. The issue is that GPUs (and stream processors in general) come with exotic constraints and performance models. An HLC has to do its job of selecting and scheduling kernels using a necessarily abstract model of the hardware. The constraints can be part of the abstract model ("feed-forward") or the HLC can make test calls to the LLC and receive results ("feedback") as part of a search. The two-level compiler approach makes this problem abject because this feedback information must be exposed as part of the interface between the compilers, but even a monolithic tool would have this problem.

Related Work

There are at least three other interfaces to GPUs useful for general-purpose computation.

Brook-GPU [1] combines a streaming language (Brook) with a GPU runtime. The Brook language provides explicit kernels connected by streams as extensions to C. Brook-GPU manages the kernels and memory on the GPU. In our toolchain, R-Stream selects the kernels from loops in the annotated C source. The SVM interface is similar to but more primitive than Brook in that SVM kernels are explicitly scheduled. SVM also supports multiple stream processors.

PUG [3] provides a minimal interface to map arrays and iteration spaces to fragment shaders in Cg and to bind fragment program formal parameters to invocation arguments.

Sh [6] is embeds shader programs as sequences of API calls in a C++ program which gives the two domains the same namespace.

References

[1] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston and Pat Hanrahan, “Brook for GPUs: Stream Computing on Graphics Hardware,” Proceedings of SIGGRAPH 2004, 2004.

[2] Daniel P. Campbell, Dennis M. Cottel, Randall R. Judd and Mark A. Richards, "Introduction to Morphware", http://www.morphware.org/PCA101/

[3] Mark Harris, “Mapping Computational Concepts to GPUs”, chapter 31 in GPU Gems 2, Addison-Wesley, 2005.

[4] Francois Labonte, Peter Mattson, Ian Buck, Christos Kozyrakis and Mark Horowitz, “The Stream Virtual Machine,” in International Conference and Parallel Architectures and Compilation Techniques, 2004.

[5] Richard Lethin, “R-Stream 3.0: Technologies for High Level Embedded Application Mapping,” HPEC, 2004.

[6] Michael D. McCool, Zheng Qin and Tiberiu S. Popa, “Shader Metaprogramming,” in Graphics Hardware, 2002.

[7] nVidia Corporation, “Cg Toolkit User’s Manual”, Release 1.1, February, 2003.