GPU Fluids in Production: A Compiler Approach to Parallelism

Dan Bailey   Ian Masters   Matt Warner
Double Negative*

Left to right: *Squirt simulations from Sorcerer's Apprentice © 2010 Disney, Inception © 2010 Warner Bros, Scott Pilgrim © 2010 Universal*

Abstract

Fluid effects in films require the utmost flexibility, from manipulating a small lick of flame to art-directing a huge tidal wave. While fluid solvers are increasingly making use of GPU hardware, one of the biggest challenges is taking advantage of this technology without compromising on either adaptability or performance. We developed the Jet toolset comprised of a high-level language and compiler for structured grids and migrated the grid solver from our proprietary fluid solver, Squirt[1] achieving significant acceleration.

1 Introduction

Structured grids are at the heart of our Navier-Stokes fluid solver, however providing a highly efficient GPU implementation for them requires frequent code reuse, complex layers of meta-programming and a fundamentally different approach to that of the CPU. In addition, performing sophisticated adjustments to simulation data often requires exporting the data out to a 3D package such as Houdini, as the internals of the solver are tricky to change rapidly in line with the demands of production.

Our approach to solving this is to use the Jet language for expressing the logic for our structured grids and the Jet compiler to determine the optimum indexing strategy and low-level memory management for a target architecture. Separating the application logic from the low-level implementation allows for more seamless development in a highly production-driven environment as artists can assemble complex workflows while developers independently refine the compiler framework.

2 The Language

Our Jet language is domain-specific and non-Turing complete. Unlike much of the active research into automatic parallelisation, a bottom-up approach is used, taking the smallest unit of computation and scaling it up to a larger system. Programming using this language means it is easy to write code that scales well in parallel and hard to write code that doesn't scale well.

Fundamental new constructs are introduced, such as the colon-bracket 'focus' operator for handling grid offsets relative to the current computation voxel. This simplifies common stencil computation and coupled with an otherwise familiar syntax makes the code easier to understand for both developers and artists.

*e-mail:{drb,iim,mw}@dneg.com

[1]originally co-authored by Marcus Nordenstam and Robert Bridson

3 The Compiler

Our Jet compiler is built upon the LLVM Compiler Infrastructure [Adve 2003], and uses their intermediate representation (IR) and optimisation pass framework as the foundation for applying parallel transformations. A Bison-based parser and code generation phase produces valid 'parallel-aware' LLVM IR, which the translation passes can adapt for sequential or parallel execution. The resulting IR is then lowered using the X86 backend for the CPU, or the PTX backend for NVidia GPUs. Using LLVM, the Jet compiler generates NVidia PTX instructions directly, obviating the need to program intricate grid logic in low-level parallel languages such as CUDA or OpenCL.

The indexing strategy for parallel computation is as shown in [Bailey 2010] and relies on the compiler to optimise desired grid access patterns for memory bandwidth through careful use of shared memory. Working in a highly atomic, modular way, the compiler is free to perform late-stage and inter-kernel optimisations that would just not be feasible with a more fixed compilation cycle.

50^3 Smoke Simulation	Arch	Time	Speedup
CPU (GCC 4.1.2)	X5570	29m34s	1.0x
CPU (Jet 1.3)	X5570	14m00s	2.11x
GPU (Jet 1.3)	FX4800	07m19s	4.04x
GPU (Jet 1.3)	C2050	01m07s	26.7x

The single-threaded X5570 CPU solver is twice as fast when using the Jet compiler over the original codebase and demonstrates a 26.7x speedup when targeting the NVidia C2050 Tesla GPU.

4 Conclusion

The Jet language and compiler provides a simple, efficient way of tackling problems involving stencil computations for structured grids. Target-specific optimisations are moved upstream from complex layers of templates and macros to form clean, modular passes. These passes are applied in turn to transform each unit of computation to take advantage of features of the requested architecture.

References

ADVE, V., AND LATTNER, C. 2003. LLVA: A Low-level Virtual Instruction Set Architecture. In *Proceedings of the 36th annual ACM/IEEE international symposium on Microarchitecture (MICRO-36)*.

BAILEY, D., AND MASTERS, I., 2010. GPU fluids in production: Accelerating the pressure projection, July 25. SIGGRAPH Talk.

Copyright is held by the author / owner(s).
SIGGRAPH 2011, Vancouver, British Columbia, Canada, August 7 – 11, 2011.
ISBN 978-1-4503-0921-9/11/0008