Improving program productivity, performance and portability through a high level language for graphics and game development

James R. Geraci Erek R. Speed
Square Enix Research Center*

1 Introduction

Our work focuses on the area of using a high level language to improve program productivity, performance and portability. In general, this has been an area of intense research. There are a number of previous efforts including ZPL [Chamberlain and et al 2004], X10/Fortress/Chapel from IBM/SUN/Cray [Weiland 2007], Intel's CT/RapidMind [McCool 2006] and parallel VSIPL++ [Lebak and et al 2005] to name a few. However, while these languages do great things in simplifying parallel implementation of code, extensions beyond that are limited. The primary exception to this is VSIPL++ which implements several high level functions useful to the signal processing community. While most of these languages can be used to implement graphics or game related algorithms if necessary, none of them attempt to provide a platform that makes such development particularly easy. On the other hand, high level engines such as Renderman and Unreal provide the wanted abstractions but with little or no guarantees about extensibility, portability, or parallel performance. Our research focuses on adapting the parallel VSIPL++ API from the signal processing community to the graphics and game development environment.

The choice of parallel VSIPL++ as the starting point bears some discussion. Our goal was to have the base language provide as much of the needed parallel framework as possible while being easy to extend into our goal domain. VISPL++'s map construct fills the first requirement, though it is unclear if it does it best. For instance, Intel's CT/Rapidmind abstracts away even this map construct by assigning data to computing units dynamically with an efficient runtime system. However, for a game development environment, such an add-on would be a large step away from current programming paradigms and possibly a hindrance to the various closed systems in the gaming world. Moreover, because VSIPL++ is an open API with capable extensible implementations, something none of the other options can provide, it is a ready target for language research and development. VSIPL++'s implementation of a similar level of functionality in its primary domain provides an excellent example of what one might expect while building a domain specific extension of VSPIL++.

2 Contributions

We contribute three main results: we extended the VSIPL++ API with a ray/triangle intersection function, made data maps distribute across an architecture's compute units instead of just its processors as is presently done, and demonstrate the performance, productivity and portability of the API by implementing a Monte Carlo path tracer that can run on any one of 4 different platforms without any code modification.

First, we extended the VSIPL++ API with a ray/triangle intersection function. This is significant as it is the basis function for many algorithms in rendering and conceptually similar to the object/object intersections used in physics. Thus, by showing we can implement a high level cross platform API for rendering, we have also shown that it is theoretically possible to do the same for physics. Our function runs on parallel x86 processors (or PowerPC processors), the Cell Broadband Engine (PS3) across multiple Synergistic Processor Elements, or on a GPU using the streaming/CUDA units of the GPU.

Next, we modified how parallel VSIPL++ maps data. Presently, parallel VSIPL++ maps data across system processors. This hides system coprocessors from developers. We modify VSIPL++ maps to run across a set of homogeneous computational units. These computational units could be from any number of different computational architecture families. For example, they could be x86 processors, the streaming units on a Graphics Processing Unit, or the Synergistic Processor Elements on a Cell Broadband Engine.

We demonstrate the usefulness of our API by implementing a Monte Carlo path tracer in VSIPL++ for games and run it on 4 extremely different hardware architectures without changing any code. We present performance results for each platform and discuss implementation difficulties we encountered when writing at such a high level.

3 Conclusions

We conclude our presentation with a discussion of future work. We're primarily left with one question: What is the optimal set of high level functions for the domain? Each added function is an opportunity for both usefulness and bloat that harms usability. We discuss which functions we think should be included and solicit feedback.

References

CHAMBERLAIN, B. L., AND ET AL. 2004. The high-level parallel language ZPL improves productivity and performance. Proceedings of the IEEE International Workshop on Productivity and Performance in High-End Computing.

LEBAK, J., AND ET AL. 2005. Parallel VSIPL++: An open standard software library for high-performance parallel signal processing. *Proceedings of the IEEE 93*, 2 (February), 313–330.

MCCOOL, M. D. 2006. Data-parallel programming on the Cell BE and the GPU using the RapidMind development platform. GSPx Multicore Applications Conference, Santa Clara.

WEILAND, M. 2007. Chapel, Fortress and X10: novel languages for HPC. Tech. rep., EPCC, The University of Edinburgh, October.

*e-mail:geraci,speed@square-enix.com

SIGGRAPH 2010, Los Angeles, California, July 25 – 29, 2010.
ISBN 978-1-4503-0210-4/10/0007