

# **Sketching for Multi-core Programming**

## **A Summer Research Report**

Apollo Ellis

Intel Undergraduate Research Program, UC Berkeley

Advisor Rastislav Bodik

Mentor Chris Jones

The objective of my summer research with Professor Bodik was to gather data on parallel programming with the IBM Cell BE processor. The data gathered would then be used to port Ras' Programming by Sketching research to the Cell processor, a multi-core system. I began by going through the lectures, discussions, and assignments of an MIT IAP class given on programming on the Cell. The hardware used by the class was Sony's Playstation 3, a gaming console/computer using a Cell BE processor. The next task I would undertake would be to learn and extend of the MIT student's course projects from the IAP Playstation 3 programming course. I was also to be collecting buggy programs to help document what is difficult about programming the Cell in order for it to be abstracted away by Sketch Programming.

### **Cell Data**

My first area of concentration was on learning the Cell architecture, which is essential for multi-core programming on the processor. The Cell processor consists of a single Power PC unit (PPU), with a VMX 32 instruction set, that controls eight other synergistic processing units or SPUs. The SPUs are smaller processors optimized for data streaming and have vector SIMD instruction set. There is a main 512KB L2 Cache on the PPC and each SPU has a 256KB local store, which is similar to a large register file. Each SPU also has a 128x128 bit register file.

The most important things I learned about the architecture were in relation to the working of the chip's DMA (Direct Memory Access) transfers and the use of the C/C++ language intrinsics. There is a DMA engine on each SPU controlled by the SPU's MFC (Memory Flow Controller) unit. DMA is used to transfer data directly to and from the PPC's L2 Cache and the SPU's local store. The DMA transfers must be from 1 byte to 16KB by powers of 2 bytes. No other sized transfers or supported. 16 SPU to PPU outstanding DMA requests may exist at one time and the DMA bandwidth on the SPU's

is 128B/cycle. The SPU's MFC will also store 8 additional requests from the PPU to the SPU. The bandwidth of the Bus between the SPU's and the PPC processor is 96B/cycle. Finally the data transferred by DMA from SPU to SPU, PPU to SPU, or SPU to PPU must be transferred to and from the same offset with 16 bytes. The SPUs also have mailboxes used to send signals and small amounts of info among the SPUs, and to/from the PPU.

There were two main types of C/C++ language intrinsics that I came across. One type, included in the header file `spu_mfcio.h`, are MFC operations used for DMA. These include, among others, get operations for cache to local store transfers and put operations for local store to cache transfers. Other MFC operations are used for various things like tagging DMA transfers for checking later and waiting for DMA transfers to finish. The second type of intrinsics replace regular operations as vector operations such as vector add, vector multiply, vector shift etc. These appear in the form *arbitrary\_vector\_op(vec1, vec2...)*.

There is also a library for PPU programs referenced through `libspe.h`. This includes functions for initializing SPU threads (which are similar to `pthread`s) and various functions for checking on SPU status. There are also functions for writing to the mailboxes of the SPUs.

### **Programming Environment, A Little Language**

My mentor Chris Jones and I set out to begin programming the Cell. Chris had a head start working with a Cell simulator and was able to offer information on few bugs he had run into mostly concerning improper alignment and data size for DMA transfers. However, when I began to run programs on the PS3 we have in our office there were many issues to be resolved.

It turned out that software engineering tricks and knowledge of compiler flags proved to be an essential part of my research this summer. With my mentor Chris as the instructor about 50 percent of my time in the lab this summer was spent learning a new language, the language of the programming environment and the API (Application Programming Interface). We debugged a foreign SDK set up on Yellow Dog Linux. We installed Fedora Core Linux. We undertook developing in eclipse. I ran into more no-bug compilation errors and complaints than I have ever seen in my experience with programming. We migrated a graphics project from frame buffer rendering to rendering in a QT(a graphics API for Linux) window. More than anything I also had to work in

Linux for 6 hours a day every day. I finally learned what grep does, how to use which, chmod, su, find, and many other commands. I learned about yum package management, and how to kill a server with killall Xorg. It was a lot fun and I even now have a Fedora Core 6 Linux Vmware image on my laptop.

In a bit more detail the IBM SDK(Software Development Kit) was not set up in a standard way on the console running Yellow Dog Linux. We were able to fix the problems that arose piece by piece, but this became tedious and so we decided to install Fedora Core 6 reinstalling the SDK in a more standard way. I began running programs in eclipse and after a bit more unfortunate shuffling around of the include directories and experimenting we were able greatly simplify working in the programming environment.

To run Cell programs in eclipse we first installed the Cell C/C++ plug-in. To create projects I prefer the managed make. For the PPU code I select Cell PPU executable and Cell SPU executable for the SPU code. Under project properties I set the build configuration to ppu-xl64-debug. I then for either project (PPU or SPU) I set any libraries or include paths I am using. I set project dependencies of the PPU project to be the SPU projects it uses. To run I also set the inputs on the PPU project to be the binaries built from the SPU make.

### **Blue Steel Ray Tracing**

With the encouragement of my Advisor Ras I chose to look at the Blue Steel Ray Tracer, one of the projects of the MIT student's in the Cell course. The ray tracer was heavily hand optimized for performance on the Cell. The Blue Steel project supports ray tracing on objects in a 3D environment including triangles, spheres, and cylinders, with arbitrary materials such as phongs and checkerboards. It also supports reflection and transparency. Currently the objects are created and transferred to the SPU's local store. The SPUs go into a loop checking for messages in their respective mailboxes. When a message is received it has an "Op Code" and the proper handler is called on the SPU. The render scene opcode informs the SPU it should start ray tracing. Based on info sent to the SPU during initialization the SPUs know what part of the screen they are responsible for. Each SPU ray traces every sixth row of pixels. They ray trace each of their respective areas writing back to an image buffer on the PPU. The image was then rendered onto the screen via the frame buffer.

## QT

When Ras had a look at the rough rendering to the frame buffer. He suggested we seek out another technique for drawing to the screen. Qt was our choice. This API was fairly easy to use. However at first I was resorting to a somewhat cheap hack for rendering, which turned out to be convoluted and inefficient. After trying many different things and involving my mentor on numerous occasions it was pretty much decided that a custom widget must be implemented for our purposes. My mentor, plagued by thoughts of the inefficiency of our rendering, took to studying the API in more detail sure there was a fast way to do what we needed. Much to my appreciation he was able to create custom widget for raw rendering, which could take a pixel buffer and paint it directly in a window. After turning a few loops inside out and upside down. I was able to alter the image buffering process of Blue Steel to suit the custom widget and together we had implemented and very clean QT based windowing renderer for the Blue Steel Ray Tracer.

## Scaling

I moved on to extending the project with collision detection as my starting point. I built a scene with four spheres on a jagged plateau with walls and lighting. Then a problem occurred. The spheres, which were rolling around, began to roll very slowly. After adding only 8 walls or 16 vertical triangles the speed was at least half of what it had been. Ray tracing has never been good for real time rendering, however on the 6-spu enabled Playstation 3 it should scale a bit better. The first thing my mentor asked was whether Blue Steel had an exponential algorithm or not. The next clear thing to do was to examine the rendering/ray tracing code for slow or inefficient algorithms and bottlenecks and then do something about them. At this point I informed my advisor Ras about the new problem and he was excited that we had come to this point. In undertaking such a task as revamping the core of the project many Cell programming techniques and difficulties would no doubt expose themselves providing valuable data for future research.

## Out of Time... Fall Research Directions

The summer has now ended, but there is much work ahead. The first task will of course be to change/rewrite the blue steel internal ray tracing code, eliminating bottlenecks in the program, maybe implementing a better and faster ray tracing

algorithm, and quite possibly doing both. I plan to do some sort of space partitioning to make the distribution of work among the SPUs more efficient. I will look for data structures and searches that may be inefficient. I am going to check for things like proper memoization and finally consider changing the linear algebra or implementing a faster algorithm. My overall plan is to start by profiling performance on non-reflective materials and see how it scales, then move on to complicated reflective scenes and finally onto abstract object rendering.

The second, third and fourth tasks, will respectively be implementing collision detection, physics of some sort, and then AI. The implementation should be more straightforward than changing the ray tracing code, but there will be the issue of how to properly use parallelism to implement these aspects of what will essentially be a game engine. I expect to learn a great deal more about Cell multi core programming.

### **Sketching the Future**

Through extensive work with the Cell in implementing the above I will attempt to gather a surplus of information on multi-core programming on the Cell. When I have compiled and refined the data I will present it to Ras and his students for use in porting the Sketch research to the multi-core Cell processor. My mentor and I both are hoping that in some cases when we present a suggestion for new Cell specific features in Sketching we will be given the opportunity to implement them ourselves. I hope to extend my research in these directions. It will take a lot of work to learn the Sketching system and even more discover how to properly extend it onto the Cell. These are however exciting challenges and I look forward working more with Ras, Chris, and others sketching the future.