CellFS: Taking The "DMA" Out Of Cell Programming

Latchesar Ionkov
*Los Alamos National Laboratory**
lionkov@lanl.gov

Aki Nyrhinen
University of Helsinki
aki@helsinki.fi

Andrey Mirtchovski
Los Alamos National Laboratory
andrey@lanl.gov

April 18, 2007

Abstract

In this paper we present a new programming model for the Cell BE architecture. We call this programming model CellFS. CellFS aims at simplifying the task of managing I/O between the local store of the processing units and main memory. The CellFS support library provides the means for transferring data via simple file I/O operations between the PPE and the SPE.

1 Introduction

The Cell Broadband Engine [8] is a new architecture aimed at providing high-performance computational facilities for scientific and media applications. Even though the Cell BE was designed initially for the gaming industry and specifically for Sony's Playstation 3 game console, Cell computers have been embraced by the scientific community for their potential to deliver high-performance to certain applications. Several large clusters comprised of Cell processors are currently being built at various facilities, with the most high profile being RoadRunner, to be delivered in the middle of 2007 at Los Alamos National Laboratory.

*LANL publication: LA-UR-07-0016

The Cell provides a novel and interesting architecture which challenges programmers to explore new ways for exploiting its full potential. The Cell architecture has sparked interest at all levels of research with scientists looking at new programming and execution models, compilers, library optimizations and operating system design. The current trend in computer hardware is towards increasing the parallelization on a single chip by putting more processors, cores or co-processors on the same die. With Cell leading the front in multi-core, heterogeneous systems we hope that research in software support for such machines will have long-lasting effects and implications on the future of high-performance computing.

In this paper we describe a novel programming model for the Cell called CellFS. The goal that we set for developing CellFS is to present to the programmers a fast, simple interface which provides them with familiar paradigms for communicating between separate parts of the executing program.

The CellFS programming model provides a POSIX-like file I/O interface for accessing the Cell's main memory from the computational units. This model can be easily extended to provide communication channels between the different units directly and, more generally, can provide access to all resources available on the Cell hardware, such as networks or externally connected devices. CellFS also aims at replacing the commonly used double-buffering programing model with a simple coroutine model in which the computational units are scheduled in a way that avoids waiting for DMA completion.

The following sections will describe briefly the parts of the Cell Architecture pertaining to our programming model, the programming model itself, and the implementation we have completed. The paper also provides performance metrics for the current implementation of CellFS and discusses improvements and future work we have planned.

2 Cell Architecture

The Cell architecture has been discussed in detail elsewhere [4] [8] [3], however we will describe the most important aspects as they pertain to our new programming model, namely the memory architecture, the processor architecture and the messaging model used throughout.

The Cell achieves its performance by utilizing two different types of computational units: the Power Processing Unit or Element (PPE) and the Synergistic Processing Units or Elements (SPE) [2]. processor. The operating system and user programs run on the PPE, while the 8 SPEs are used for computation offload.

Figure 1 depicts the organization of the cell elements as well as the bus connection between them.

The PPE of the Cell is based in IBM's 64-bit PowerPC Architecture and is similar to the other 64-bit PowerPC processors. It is primarily responsible for running operating systems and managing the system's resources. In order to utilize the full performance potential of the Cell architecture, the most intensive

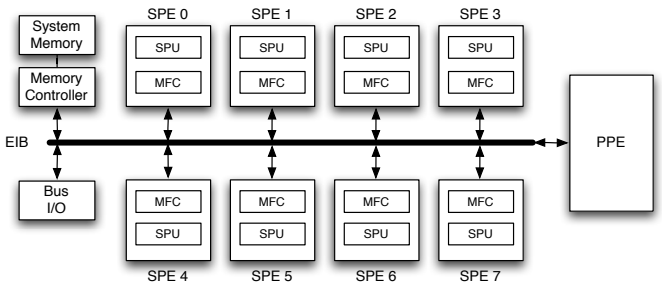

Figure 1: The cell architecture

parts of the computation must be offloaded to the SPEs.

The SPE [2] implements a special SIMD instruction set that is optimized for computationally intensive applications. It has 128 128-bit registers and 256 kilobytes of local store. The SPE's local store is the only memory accessible directly by code running on the SPE. Furthermore, the size of the local store is further limited by the fact that it also needs to store the code of the currently executing program. This is the main limitation of the architecture: every access to global memory on the PPE needs to be explicitly scheduled, and programmers need to do a balancing act between how much code can be put on the SPE and how much memory this code will have to work with.

The SPE uses asynchronous, coherent direct memory access transfers (DMA) to access the main storage. The memory address translation is controlled by the PPE. There is support for up to 16 outstanding DMA requests on each SPE. DMA is programmed either one-by-one by instructions executing on the SPE, by preparing DMA lists, or by inserting commands in the DMA queue from another processor (usually the PPE). Instructions on the SPE work with 128-bit wide data. Although the element width of the data can vary down to one byte, the single-instruction multiple-data (SIMD) capabilities of the SPE mean that it is most efficient when the full 128-bit length of the registers is used.

Besides DMA, the Cell architecture implements two modes for notification of external events and synchronization between the PPE and the SPEs: Mailboxes and Signals. Communication occurs through channels. Channels are unidirectional message-passing interfaces, supporting 32-bit messages and commands. Each SPE has its own set of channels supporting two mailboxes for sending messages from the SPE to the PPE and one mailbox for exchanging messages between the PPE and the SPEs. The channel interface also supports two signal-notification channels (signals) for inbound messages to the SPE.

3 Existing Cell Programming Models

The Cell architecture differs significantly from the conventional microprocessors in several important aspects, making it hard to use conventional programming

models for it. In order to run efficiently, the software needs to be redesigned before deployment. While there are significant improvements in computational power and memory bandwidth of the Cell processors, they can easily be lost if software is unable to utilize fully the resources provided by the hardware. While the Cell SPEs are fast, the small size of local memory and the communication restrictions imposed on them forces programmers to be very careful in structuring their computational code when porting it to the Cell. This complicates already non-trivial scientific codes to an extent where application programmers need to explicitly care about such things as scheduling direct memory access between the SPEs and the PPE.

Several different programming models have been proposed [1], which hide the single-instruction-multiple-data (SIMD) nature of the dataflow behind a set of libraries or methods. Each of these methods has its benefits and drawbacks. Below we discuss the main ideas of those methods as proposed by Cell architecture's designers [4].

3.1 Function offload

In this model the application runs primarily on the PPE, while specific functions from standard libraries are optimized for execution on the SPE and offloaded to them for faster computation. This model is the simplest from an application programmer's point of view and offers the benefit of minimal change requirements for existing code (in its most reduced form the application runs on the PPE entirely, never requiring anything from the SPE).

While useful for fast deployment this model suffers significantly in its performance. In fact, running any computationally intensive code on the PPE is undesirable as the PPE is significantly slower than the SPE. While library functions do execute faster, the majority of science code requires that external support libraries such as BLAS [10] be ported and optimized for execution on the SPEs. There is however a significant benefit for game programmers, which rely heavily on graphics libraries.

3.2 Streaming and buffering

Streaming and the triple-buffering programming models provide a method for optimizing I/O or computation by pipelining data either between multiple SPEs or by staging multiple buffers for DMA to and from the PPE, thus performing both computation and communication at the same time. When triple-buffering model is used, the program issues DMAs transfer the results from the last step from one buffer, performs computations on a second buffer, and transfers data from the main storage to a third one. Streaming usually involves passing the same chunk of data from one SPE to another, with each SPE performing a particular operation.

Streaming and buffering both offer significant performance improvements, however they also have drawbacks. Streaming may not well suited for some computational models such as some where data may flow through different paths

depending on decisions made earlier in the pipeline. Buffering also suffers from that issue, but has the added drawback that the local storage in the Cell SPE, at 256 kilobytes, is minimal by today's standards.

3.3 Shared-memory multiprocessor

The Cell can be programmed as a shared-memory multiprocessor with the help of both the operating system and compilers and libraries. In a shared-memory multiprocessor environment the SPE and PPE units operate in a cache-coherent shared-memory programming model. Conventional memory loads from the SPE are transparently replaced with a DMA operation from shared memory to the SPE's local store. The shared-memory multiprocessor has the disadvantage that the SPEs and the PPE have different instruction sets. Also, the cost of constant DMA for random memory accesses should force the system programmers to think about implementing a cache system for the local store of the SPE.

3.4 Computational acceleration

This programming model uses the SPE as an accelerator for all computationally-intensive code, with the PPE used as an I/O arbiter. We describe Computational Acceleration fully in section 4, as our proposed method is a variation of this model.

4 Our proposed programming model

Observations of programmers who are porting existing software for the Cell show that the most important and time-consuming effort goes into changing or manipulating the memory management model of the code to fit the one available for the new architecture. Due to the specifics and memory constraints of the Cell architecture, nearly all porting efforts we have identified end up having to manage DMA to and from the SPE by themselves. Having to drop to such a low-level abstraction is not trivial and requires a lot of effort to "get right" on the part of the programmer. Our desire is to provide a higher-level abstraction that hides the particularities of each DMA access behind a model which most programmers are familiar with.

4.1 Memory access

The Cell architecture provides a single location for storing information on its accelerated SPEs and that is 256KB of local store. The local store must also hold the application executing, which further restricts the amount of memory available for use by the application. Furthermore, the only communication channels that the SPE has to the outside world are mailboxes, signals and DMA to/from main memory. When a program running on an SPE needs to access disks, network or any other device it must do so through an intermediary

running on the PPE which much receive, decipher and respond to the SPE's request. In all cases, both the PPE and the SPE need to agree on a set of messages, or a communication protocol, which allows such access to external devices to occur.

We observe that the model of access by the SPE to the main memory on the PPE, which holds all data for the code currently executing, is very similar to accessing disk in a conventional system. Namely, the SPE sends an asynchronous request specifying the memory address and the size of the requested segment. Eventually, given a small time delay, the request is fulfilled and data is transferred to the local store of the SPE. This resembles the interface that early operating systems provided to disk and tape storage, in which user programs had to keep track and manage disk blocks. On modern operating systems such low details of the the communication channels with storage devices are hidden in the OS kernel under an abstraction layer, which exposes a higher-level interface to user programs. In most cases the data abstracted is in the form of hierarchical tree structure comprised of directories which hold files. This interface is very familiar to programmers and users since it is the most pervasive one in existence. For example, on modern file systems a user program reads data on disk assuming a contiguous allocation (starting at byte zero, until the end of the file is reached). The kernel file system interface and associated drivers translate file offsets into block locations on physical media.

The directory and file hierarchy interface is abstract enough to allow other devices to be represented that way too, examples of which include UNIX terminals, user processes, devices. "Plan 9 from Bell-Labs" [9], a descendant of UNIX, extended the concept even further by representing network interfaces and other system components such as DNS or even pipes as files mountable in a user's namespace. Unfortunately, the local memory restrictions will not allow us to run a full OS on the SPE. Implementing a complex interface to external devices is expensive, yet we would like to provide access from the SPE not only to the resources it directly "sees" but also to the disks, networks and memory located on the PPE. We found that a simple interface to a protocol that allows communication with external resources as files is not only feasible, but fits the SPE/PPE duality rather nicely. Thus, we have implemented a library that allows the communication with external resources from the SPE to occur via a directory and file abstraction model.

Depending on the underlying storage we define five filesystems (listed in table 1) that are served by the PPE. Most commonly used ones are #r and #U.

Code running on the SPEs accesses this file system via library calls corresponding to normal POSIX file operations. To open a file named `test` in `/tmp` on the main file system of the Cell and write to it one would write:

```
fd = spc_open("#U/tmp/test");
spc_write(fd, data, num);
spc_close(fd);
```

Our model is easily extensible to provide access to more resources if necessary. Our tests show that we can get reasonably good performance for access to

Name and type	Description
#r	File server allowing operation on files existing in ramdisk on the main memory of the Cell
#U	File server allowing operation on files existing on the UNIX file system accessible by the PPE. Files served by #U are mmap()-ed to main memory to increase I/O bandwidth
#R	Similar to #U, but changes to the files are not propagated to the disk. This is equivalent to a read-only file system, however it allows the SPEs to communicate data between each other as the computation progresses
#p	Clients can use this file system to create a named pipe which can be used to communicate between clients running on different SPEs.
#l	Log file system used by lightweight library routines replacing `printf()`

Table 1: File systems served by the PPE

main memory of the Cell. This approach combines simplicity, very important when memory is limited, with extensibility.

4.2 Asynchronous execution

In order to use the SPE effectively, data transfer must not be blocking, i.e., the execution of code and I/O must overlap. There are two approaches for interface design that achieve the requirement: asynchronous I/O or multiple executions threads. The asynchronous I/O model has been experimented with in the double- and triple-buffered programming modes, but for a complete solution an implementation of an interface similar to POSIX Asynchronous I/O [6] must exist. We decided to implement a simple coroutine model that allows more than one function to run independently on the SPE since we believe that we have found a model simpler to implement and understand than asynchronous I/O. Our model does not require locking of information and provides completely deterministic execution even in the presence of multiple threads with overlapping I/O.

In our threaded model there exists only one function that is using the processor at any give time. Whenever the running function performs an I/O operation, the library initiates the transfer to or from the PPE and gives the control to a different coroutine running on the same SPE. In order to keep both the interface and implementation simple, the switch between the coroutines can occur only when a function from the I/O interface is called, providing a fully deterministic stream of operations. The fact that the switching points are known in advance requires no locking of data and greatly simplifies programming for the

developers.

Coroutines are created similarly to POSIX threads. The `mkcor()` function receives a function pointer, parameter pointer and stack buffer pointers and handles the set-up stage for the new coroutine. A new coroutine is not immediately scheduled, but is put in a FIFO waiting queue. The following code creates a coroutine with a stack size of 4096 bytes which prints a greeting. Note that this code avoids using the `printf()` library call, instead using the much more lightweight `spc_log()` provided by our library.

```
char stack[4096];
void
cor(void *arg)
{
    spc_log("hello world\n");
}

void
cormain()
{
    mkcor(cor, NULL, stack, sizeof stack);
}
```

Figure 2 provides an example of the coroutine scheduling that may occur on an SPE with two coroutines.

The benefits of the coroutine model are several. Unlike threads or normally-scheduled OS processes, coroutines execute completely deterministically, thus removing the need for locking or mutual exclusion. The coroutines can share all variables defined in the SPE code as long as they do not try sharing the memory areas for which DMA may be in progress.

A disadvantage of the coroutine model is that it requires separate state to be kept and stored on the SPE for each coroutine.

5 Implementation

The main implementation goals for our library are to provide for optimized I/O operations to and from main memory and to have a very low memory footprint on the SPE. We were very careful in choosing a protocol for our system. Our requirements called for something that is both lightweight, yet provides support for all file operations that we required. We chose the 9P [5] protocol as its used in the "Plan 9" operating system. There are several reasons for this:

- Simplicity: the protocol has only a handful of messages which encompass all major file operations, yet it can be implemented (including the coroutine code explained above) in around 2000 lines of C code

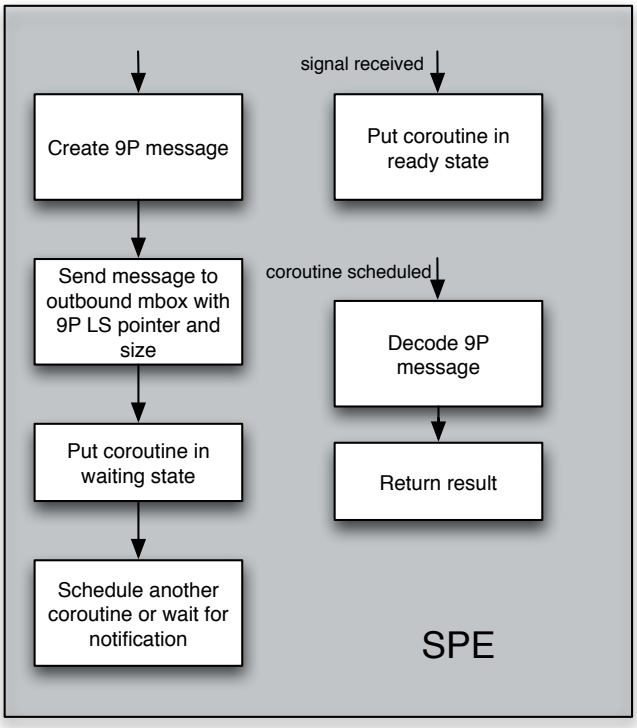

Figure 2: SPE coroutine scheduling

- Robustness: 9P has been in use in the Plan 9 operating system for over 15 years
- Architecture independence: 9P has been ported and used on all major computer architectures
- Scalability: our Xcpu [7] suite uses 9P to control and execute programs on several thousand nodes at the same time

Since the 9P protocol is not directly accessible by the user, all implementation details below do not concern end programs.

A 9P *session* between a server and its clients consists of requests by the clients to navigate the server's file and directory hierarchy and responses from the server to those requests. The client initiates a request by issuing a *T-message*, the server responds with an *R-messages*. A 9P *transaction* is the combined act of transmitting a request of particular type by the client and receiving a reply from the server. There may be more than one request outstanding, however each request requires a response to complete a transaction. There is no limit on the number of transactions in progress for a single session.

Each 9P message, listed in table 2, contains a sequence of bytes representing the size of the message, the type, the tag (transaction id), control fields depending on the message type and a UTF-8 encoded payload. Most T-messages contain a 32-bit unsigned integer called *Fid*, used by the client to identify the "current file" on the server, i.e. the last file accessed by the client. Each file in the file system served by our library has an associated element called *Qid* used to uniquely identify it in the file system.

9P message type	Description
version	identifies the version of the protocol and indicates the maximum message size the system is prepared to handle
auth	exchanges auth messages to establish an authentication fid used by the attach message
error	indicates that a request (T-message) failed and specifies the reason for the failure
flush	aborts all outstanding requests
attach	initiates a connection to the server
walk	causes the server to change the current file associated with a fid
open	opens a file
create	creates a new file
read	reads from a file
write	writes to a file
clunk	frees a fid that is no longer needed
remove	deletes a file
stat	retrieves information about a file
wstat	modifies information about the file

Table 2: Message types in the 9P protocol

The current implementation of the coroutine library and 9P client code for the SPE take only 20K of local store memory on the SPE, thus leaving ample space for user code. The maximum number of coroutines running on a single SPE has been limited to 8 to allow for a larger stack and thus more user-code variables. The size of the stack used by coroutines is user-defined and can vary between coroutines running on the same SPE.

In the proposed programming model the server runs on the PPE and clients run on the SPE. Clients construct 9P messages in the local store and send notifications to the PPE via the mailbox communication channel. The PPE retrieves the messages via DMA, performs the operation and sends back the responses to the same local store buffer of the originating request. The PPE notifies the SPE via a signal.

Figure 3 provides an overview of the tasks the PPE completes while servicing a DMA request.

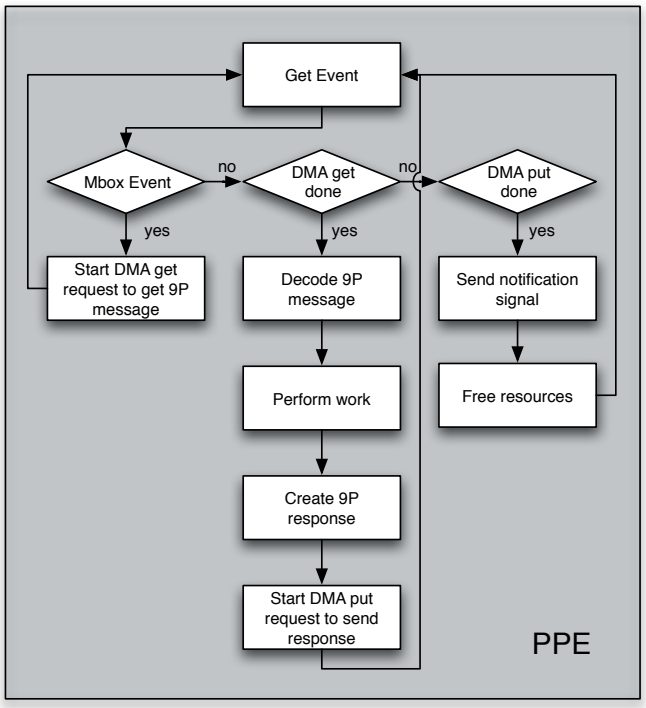

Figure 3: PPE workflow

A simple optimization is used to improve performance of the special case when the file is in main memory: in this case the Qid of the file to which the operation refers is a pointer to a file description structure. This structure contains information about the file size and its memory location (the files are contiguously stored in memory). This improves performance by allowing more information to be transferred in a single DMA than the maximum allowed by the 9P protocol implementation. Read and write operations to 16-byte-aligned buffers of files marked as in-memory schedule a DMA directly, bypassing the 9P protocol, thus further improving performance.

Table 3 illustrates how messages are broken down into the 9P protocol core types for non-optimized and optimized communication respectively.

Coroutines are implemented in the CellFS library. If the user code calls a routine involving DMA to or from the PPE the library schedules the DMA, saves the registers of the callee coroutine, restores the registers of the next coroutine and jumps to its PC. If there is no coroutine ready to be scheduled, the library blocks until it receives a notification that a DMA is complete, or a signal notification for a request completion from the PPE.

Programmer Code	Non-optimized 9P	Optimized 9P
`char buf[8192];`	← Tversion "9P2000" 1024 → Rversion "9P2000" 1024 ← Tattach ... → Rattach	← Tversion "9P2000" 1024 → Rversion "9P2000" 1024 ← Tattach ... → Rattach
`int maincor() {` `  int fd;`		
`  fd=open("#u/test",` `    Oread);`	← Twalk fid 0 newfid 1 "#u" "test" → Rwalk (...) ← Topen fid 1 mode 1 → Ropen (...)	← Twalk fid 0 newfid 1 "#u" "test" → Rwalk (...) (Dmem fhptr) ← Topen fid 1 mode 1 → Ropen (Dmem fhptr) ← DMA get ptr fhptr count 32 → DMA wait
`  n=read(fd,` `    buf, ...);`	← Tread fid 1 offset 0 count 1000 → Rread count 1000 data (...) ← Tread fid 1 offset 1000 count 1000 → Rread count 1000 data (...) ← Tread fid 1 offset 2000 count 1000 → Rread count 56 data (...)	← DMA get ptr fhptr count 8192 → DMA wait
`  close(fd);`	← Tclunk fid 1 → Rclunk	← Tclunk fid 1 → Rclunk

Table 3: Breakdown of I/O into 9P protocol primitives for non-optimized and optimized implementations

6 Performance

In order to evaluate the performance of the proposed programming model, we created two benchmarks. We run them on a Cell blade system that contained two 3.2GHz Cell processors.

The first benchmark tests the maximum memory bandwidth that can be achieved. The SPE program opens a 128MB file stored on the Unix file system, then creates a number of coroutines, each of which reads and writes 16777216 8K blocks from the files. As mentioned in the previous section, the regular Unix files are mmap-ed when accessed from the SPE, so after the initial page faults that bring the file content to the memory, reading from the file is equivalent to accessing the main memory. We tested the program running on different number of SPEs and coroutines.

Table 4 shows the results of running the memory bandwidth benchmark. The memory bandwidth we achieve is much lower than the theoretical maximum. We believe that the main reason for that is that we don't use HugeTLB.

The second benchmark is a modification of the IBM optimized matrix multiplication workload (`src/workloads/matrix_mul` directory in Cell SDK 1.1). We left the optimized functions MatInit_MxM and MatMul_MxM intact, chang-

# SPE	DBL	# Coroutines						
		1	2	3	4	5	6	7
1	8.19	5.68	10.63	10.71	10.75	10.81	10.78	10.79
2	15.59	11.03	20.48	20.74	20.79	20.88	20.89	20.96
3	20.05	14.75	23.20	23.26	23.27	23.28	23.29	23.28
4	20.48	17.84	23.28	23.20	23.32	21.84	23.32	23.32
5	21.63	20.71	23.26	23.27	23.28	23.28	23.28	23.29
6	21.82	21.89	23.26	23.25	23.26	23.26	23.26	23.26
7	23.05	21.05	23.24	23.23	22.05	23.23	23.24	23.23
8	22.34	21.43	23.22	23.20	23.21	23.21	23.21	23.21

Table 4: Memory Bandwidth (in GB/s) for various numbers of active SPEs and Coroutines, as well as the benchmark double-buffered code

ing only the logic that reads and writes the blocks. Instead of implementing double-buffering model by issuing DMA transfers directly, our implementation uses two coroutines that each calculates half of the blocks assigned to the SPE. The initial data of the A and B matrices is stored in regular Unix files and the result of the multiplication is also stored in a regular file. In order to have fair comparison we modified the original matrix multiplication program to read the content of A and B from the same files instead of generating it on the fly.

The main function opens the files containing the matrices data, then creates two coroutines that do the actual multiplication:

```
static void
mmulcor(void *a)
{
    int i, j, k, blkid;
    char *ename;
    Ctx *ctx;

    ctx = a;
    for(blkid = ctx->blkfirst; blkid < ctx->blklast; blkid++) {
        i = (blkid >> shift) & mask;
        j = (blkid) & mask;

        block_read(ctx, i, j, 0);
        MatInit_MxM(ctx->c, ctx->a, ctx->b);
        for(k = 1; k < (N/M); k++) {
            block_read(ctx, i, j, k);
            MatMult_MxM(ctx->c, ctx->a, ctx->b);
        }

        block_write(ctx, i, j);
    }
```

```
}

static int
block_read(Ctx *ctx, unsigned int by, unsigned int bx, unsigned int idx)
{
    int sz;
    u64 offa, offb;

    offa = 4*(by*M*N + idx*M*M);
    offb = 4*(bx*M*M + idx*M*N);
    sz = sizeof(float) * M * M;

    spc_pread(afd, (u8 *) ctx->a, sz, offa);
    spc_pread(bfd, (u8 *) ctx->b, sz, offb);
    return 0;
}

static int
block_write(Ctx *ctx, unsigned int by, unsigned int bx)
{
    int sz;
    u64 offc;

    sz = sizeof(float) * M * M;
    offc = 4 * (by*M*N + bx*M*M);
    spc_pwrite(cfd, (u8 *) ctx->c, sz, offc);
    return 0;
}
```

Table 5 shows the results of running the original and the modified applications. There is about 20% penalty of using our modified application.

# SPEs	CellFS	Std.
1	151.10	124.23
2	75.51	62.18
4	37.86	31.13
8	19.11	15.60

Table 5: Time to run 10000 multiplications of two 512x512 single floating point matrices

7 Conclusions and future work

We have described a minimalistic programming model for the Cell BE computer architecture and the libraries supporting it. Our model aims at alleviating the Cell programmer from explicitly handling memory transfers to and from the SPE by presenting a file system abstraction layer which hides the particularities of the DMA transfers between different parts of the hardware.

One unexpected benefit from our programming model is that the development and testing can be completed on any computer since the server and client libraries hide the specifics of the Cell architecture. Also, as long as the coroutine and SPE 9P client libraries are ported or stubbed to an OS, our programming model is also OS-independent. The programmer still needs to worry about keeping the code and data within the SPE local store size limitations.

Our programming model can be extended further by providing file servers for other resources such as networks or specialized hardware. As part of the XCPU [7] cluster management suite, our libraries allow for starting a new program on a separate SPE (not necessarily located on the same Cell), thus providing extensibility and workflow management in a heterogeneous environment.

Unfortunately our model is not without drawbacks. One of those is the that the stack of the coroutine can be quite large, limiting the maximum number of coroutines (and thus bandwidth attainable) on a single SPE. For example, if the SPE events mechanism is used by the program running on the SPE the stack needs to be at least 8Kbytes. Another drawback is that the client library requires non-zero time for switching between coroutines and for constructing and deconstructing 9P messages. There is definite room for optimization in that area.

Future works improving this programming model is allowing for direct communication between SPEs without the involvement of the PPE. Something that the Cell architecture allows, but is currently not exploited by our libraries. We also plan to port part of the client library to the PPE and allow user code to be run there for tasks such as controlling and synchronization between the different SPEs. Note that this is still achievable with the current implementation, but requires that a coroutine on an SPE is dedicated to that task.

We also want to integrate more fully with the Xcpu Cluster Management Suite, which also utilizes the same file-based approach to sharing resources, so we can have a complete top-to-bottom stack for the new heterogeneous computers such as RoadRunner currently being built at LANL.

References

[1] P. Bellens, J. M. Perez, R. M. Badia, and J. Labarta. Cellss: a programming model for the cell be architecture. In *SC'06*, 2006.

[2] B. Flachs, S. Asano, S. H. Dhong, H. P Hofstee, G. Gervais, R. Kim, T. Le, P. Liu, J. Leenstra, J. Liberty, B. Michael, H.-J. Oh, S. M. Mueller, O. Takahashi, A. Hatakeyama, Y. Watanabe, and N. Yano. The microarchitecture

of the streaming processor for a cell processor. In *IEEE International Solid-State Circuits Conference*, pages 184–185, 2 2005.

[3] IBM. Cell broadband engine programming handbook, 2006.

[4] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Mauerer, and D. Shippy. Introduction to the cell multiprocessor. *IBM J. Res. & Dev.*, 49:589–604, 2005.

[5] AT&T Bell Laboratories. Introduction to the 9p protocol. *Plan 9 Programmer's Manual*, 3, 2000.

[6] The POSIX Asynchronous I/O Library Manual. At http://www.opengroup.org/onlinepubs/009695399/toc.htm.

[7] R. Minnich and A. Mirtchovski. Xcpu: a new, 9p-based, process management system for clusters and grids. In *Cluster 2006*, 2006.

[8] D. Pham, S. Asano, M. Bolliger, M. N. Day, H. P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, , and K. Yazawa. The design and implementation of a first-generation cell processor. In *Custom Integrated Circuits Conference*, 2005.

[9] Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, and Phil Winterbottom. Plan 9 from Bell Labs. *Computing Systems*, 8(3):221–254, Summer 1995.

[10] BLAS (Basic Linear Algebra Subprograms). At www.netlib.org/blas.