LA-UR- 07-0114

Approved for public release; distribution is unlimited.

Title:	CELLFS: TAKING THE "DMA" OUT OF CELL PROGRAMMING
Author(s):	LATCHESAR IONKOV, ANDREY MIRTCHOVSKI, AKI NYRHINEN
Intended for:	USENIX 2007, SANTA CLARA, CA

Los Alamos
NATIONAL LABORATORY
EST. 1943

Form 836 (7/06)

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

January 9, 2007

Abstract

In this present we present a new programming model for the Cell BE architecture of scalar multiprocessors. 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 PPU and the SPU.

1 Introduction

The Cell Broadband Engine [7] 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 promising being RoadRunner, to be delivered in the middle of 2007 here at the Los Alamos National Laboratory.

*LANL publication: —-

The Cell provides a novel and interesting new architecture which challenges the programmers to find new ways for exploiting its full potential. The Cell architecture has been exploited at all levels with research being done in new programming models, execution models, compilers, library optimizations and operating system design.

The current trends in computer hardware is towards incresing the paralelization 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 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 yet simple interface which provides the programmer with familiar paradigms for communicating between separate parts of the executing program or the architecture.

The CellFS programming model provides a POSIX-like 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. CellFS also aims at replacing the most often used triple-buffering programming with a simple, carefully designed, coroutine model in which computational units are scheduled cooperatively to provide a deterministic execution with non-blocking, asynchronous access to main memory.

The following sections will describe briefly the parts of the Cell Architecture pertaining to our programming model, the programming model itself, as well as 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] [7] [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 computational units: the Power Processing Unit or Element (PPU) and the Synergistic Processing Units or Elements (SPU) [2], processor. The operating system and user programs run on the PPU, while the 8 SPUs are used for computation offload.

The PPU of the Cell is a fully-compliant 64-bit Power Architecture processor. It has conventional access to main memory, disks and other resources such as networks and is directly connected to an on-chip bus, serving as the interconnect to the SPUs.

Even though the PPU is a full-blown processor with L1 and L2 caches, it is quite underpowered by today's standards set by processors from other

architectures such as AMD's 64-bit Opteron. In order to speed up the operation of the cell computation must be offloaded to the SPUs.

The SPU [2] implements a separate instruction-set architecture optimized for performance on computationally intensive applications. The SPU has a very limited local store to which all memory accesses are performed. Currently the local store is 256 kilobytes. In order to populate this local store data is transferred between main memory and the SPU via asynchronous, coherent direct memory access (DMA) commands. There is support for up to 16 outstanding DMA requests on each SPU. DMA is programmed either one-by-one by instructions executing on the SPU, by preparing DMA lists, or by inserting commands in the DMA queue from another processor (usually the PPU). All instructions on the SPU are 128-bit single instruction multiple data (SIMD). The element width of the data can vary down to 1-bit.

Besides DMA, the Cell architecture implements several modes for notification of external events and synchronization between the PPU and the SPUs: Mailboxes and Signals. Communication occurs through channels. Channels are unidirectional message-passing interfaces, supporting 32-bit messages and commands. Each SPU has its own set of channels supporting two mailboxes for sending messages from the SPU to the PPU and one mailbox for sending messages from the PPU to the SPU. The channel interface also supports two signal-notification channels (signals) for inbound messages to the SPU. This architecture often forces the PPU to be used as an application controller, managing and distributing work to the SPUs.

3 Existing Cell Programming Models

The Cell architecture marks a significant step in advancing hardware design and differs from conventional microprocessors significantly. 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 SPUs 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 implementations of scientific codes to an extent where application programmers need to explicitly care about such things as scheduling direct memory access between the SPUs and the PPU.

To aleviate the problem of micromanaging one's memory footprint and I/O on the SPU 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 with its benefits and drawbacks. Below we discuss the major methods proposed by the Cell's designers [4].

3.1 Function offload

In this model the application runs primarily on the PPU, while specific functions from standard libraries are optimized for execution on the SPU 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 PPU entirely, never requiring anything from the SPU).

While useful for fast deployment this model suffers significantly in its performance. In fact, running any computationally intensive code on the PPU should be frowned-upon as the PPU is significantly slower than other contemporary processors. While library functions do execute faster, the majority of science code requires that external support libraries such as BLAS [9] be ported and optimized for execution on the SPUs. 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 compututation by pipelining data either between multiple SPUs or by staging multiple buffers for DMA to and from the PPU, thus performing both computation and communication at the same time. Streaming usually involves passing the same chunk of data from one SPU to another, with each SPU performing a particular operation. Buffering models involve storing up to three chunks of data on the SPU, two of which are being transferred via DMA either to or from the PPU while the third is being worked on by the SPU itself.

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 SPU, 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 SPU and PPU units operate in a cache-coherent shared-memory programming model. Conventional memory loads from the SPU are transparently replaced with a DMA operation from shared memory to the SPU's local store. The shared-memory multiprocessor has the disadvantage that the SPUs and the PPU 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 SPU.

3.4 Computational acceleration

This programming model uses the SPU as an accelerator for all computationally-intensive code, with the PPU 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 here at LANL which are porting existing software for the Cell conclude that the most important and time-consuming effort goes into changing or manipulating the memory management model of the code to fit the ones 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 SPU by themselves. Having to drop to such 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, if not all programmers are familiar with.

We have identified that the SPU main memory access model is equivalent (in the sense that it utilizes DMA transfers) to the access model a traditional CPU has to hard disk. We believe that the file system abstraction is well suited for main memory access from the SPU. We also believe that this model is well understood by all programmers familiar with C and POSIX programming.

Representing operating system resources as files is a relatively old concept exploited to some extent in the original UNIX operating system, but it matured extensively with the development and release of the "Plan 9 from Bell-Labs" operating system [8].

"Plan 9 from Bell-Labs" uses a simple, yet very powerful communication protocol to facilitate communication between different parts of the system. The protocol, named "9P" [5] allows heterogeneous resource sharing by allowing servers to build a hierarchy of files corresponding to real or virtual system resources which then clients access via common (POSIX-like) file operations by sending and receiving 9P messages.

Our proposed programming model consists of two parts: a file server and client library. The file server code runs on the PPU and provides an interface to disk, main memory, pipes and files residing in main memory via several file systems. The file systems are as described in table 1.

Code running on the SPUs 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);
```

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 PPU. 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 SPUs 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 SPUs.
#l	Log file system used by lightweight library routines replacing `printf()`

Table 1: File systems served by the PPU

Due to the nature of DMA transactions, any single-threaded code that runs on the SPU will have to block after issuing a DMA request, thus wasting valuable cycles. Several methods for aleviating this problem have been developed, virtually all of them relying on scheduling DMA in advance in such a way that computation can continue on the SPU without waiting for DMA completion. This normally requires that more than one memory buffer is in the local store on the SPU. Typically our colleagues at LANL have decided on a triple-buffered model in which there are three buffers in memory: one being DMA'd out to main memory, one DMA'd in, and one on which the computation is being performed. The computational kernel then becomes more complicated and the flow in the computation is broken up by having to explicitly schedule DMA and check for their completion as soon as work on one of the buffers has completed. The multi-buffered solution explained above does have a drawback due to the fact that not all computations explicitly fit this model. Furthermore, the extra effort required to schedule the out-of-band communication causes code to be more error-prone than usual since it must carefully handle the boundary conditions between changing buffers.

We have solved the problem of blocking while DMA is in transit, by taking a carefully designed coroutine model allowing more than one computational code to execute in a single SPU. In the coroutine model we have adopted one or more functions run on the same SPU independantly. Whenever a function requests memory access to locations outside of the ones stored in the SPU's local store, this coroutine would block and another one will start executing while the DMA of the first completes. The model does not provide facilities for preempting a

coroutine, therefore one would continue execution until it is done processing the current memory chunk in the SPU's local store.

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 SPU code as long as they do not try sharing the memory areas for which DMA may be in progress.

Coroutines are created similarly to POSIX threads, namely, function pointer, parameter pointer and stack buffer pointers are passed to the library which handles the set-up stage. 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);
}
```

5 Implementation

Main targets 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 SPU. 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 from Bell-Labs" 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
- 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 Xepu [6] 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's requests by issuing a *T-message*, the server responses 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 contains a sequence of bytes representing the size of the message, the type, the tag (transaction id), control fields depending of 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 SPU take only 20K of local store memory on the SPU, thus leaving ample

space for user code. The maximum number of coroutines running on a single SPU 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 SPU.

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

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.

Coroutines are implemented through the library calls of the CellFS. library. If the user code calls a routine involving DMA to or from the PPU 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 waiting the library blocks until it receives a notification that a DMA is complete, or a signal notification for a request from the PPU.

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. We didn't have access to a blade with HugeTLB setup.

The first benchmark tests the memory bandwidth that can be achieved. The SPU program opens a 128MB file stored on the Unix file system, then creates a numer of coroutines, each of which reads 16777216 8K blocks from the files. As mentioned in the previous section, the regular Unix files are mmap-ed when accessed from the SPU, 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 SPUs and coroutines. We also implemented a simple SPU program that implements the conventional double buffering method of reading 16777216 (multiplied by two) 8K blocks via DMA.

Table 3 shows the results of running the memory bandwidth benchmark. The performance in both cases is comparable. The memory bandwith we achive is much lower than the theoretic 204.8 GB/s. We believe that the main reason for

that is that we don't use HugeTLB.

# SPU	Std.	# Coroutines						
		1	2	3	4	5	6	7
1		3.58	5.62	6.25	6.10	6.09	6.09	5.95
2		6.85	11.19	11.61	11.59	11.58	11.59	11.51
3		10.09	16.30	17.41	17.44	17.23	17.52	17.35
4		13.12	19.89	21.95	21.85	22.22	21.21	21.46
5		14.62	21.71	24.04	23.42	23.87	21.51	21.87
6		16.94	22.84	24.56	24.11	23.90	22.34	22.32
7		18.68	22.90	25.00	23.83	24.53	21.64	22.27
8		20.15	23.89	23.92	23.55	23.62	22.30	20.67

Table 3: Memory Bandwith (in GB/s) for various numbers of active SPUs and Coroutines

The second benchmark is a modification of the IBM optimized matrix multiplication workload. We left the optimized functions MatInit_MxM and MatMul_MxM intact, changing only the logic that reads and writes the blocks. Instead of using dma directly, our implementation uses two coroutines that each calculates half of the blocks assigned to the SPU. 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 comparision 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.

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

Table 4: Time to run 10000 multiplications of two 512x512 single floating point matrices

Table 4 shows the results of running the original and the modified applications. There is about 20% penalty of using our modified application.

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 aleviating the Cell programmer from explicitly

One unexpected benefit from our programming model is that the development and testing cycles 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 SPU 9P client libraries are ported or stubbed to an OS, our programming model is also OS-independent. The only thing a programmer needs to worry about is the requirement that code fits into the local store of the Cell SPU.

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 [6] cluster management suite, our libraries allow for starting a new program on a separate SPU (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 SPU. For example, if the SPU events mechanism is used by the program running on the SPU the stack needs to be at least 8Kbytes. Another drawback is that the client library requres 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 SPUs without the involvement of the PPU. 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 PPU and allow user code to be run there for tasks such as controlling and synchronization between the different SPUs. Note that this is still achievable with the current implementation, but requires that a coroutine on an SPU 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. Maeurer, 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] R. Minnich and A. Mirtchovski. Xcpu: a new, 9p-based, process management system for clusters and grids. In *Cluster 2006*, 2006.

[7] 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.

[8] 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.

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