Unleashing the Power of the Playstation 3 to Boost Graphics Programming

André Maximo*, Guilherme Cox†, Cristiana Bentes† and Ricardo Farias*
*System Engineering and Computer Science Program-COPPE, Federal University of Rio de Janeiro
Cidade Universitária, Centro de Tecnologia, Bl. H – Rio de Janeiro, RJ, Brazil, 21945-970
Email: andmax@cos.ufrj.br, rfarias@cos.ufrj.br
†Department of System Engineering, State University of Rio de Janeiro
Rua São Francisco Xavier, 524, Bl. D, 5o floor – Rio de Janeiro, RJ, Brazil, 20550-900
Email: cris@eng.uerj.br, cox@eng.uerj.br

***Abstract*—This tutorial is intended for programmers who are interested in boosting their graphics application using a different architectural paradigm: the Cell Broadband Engine (Cell BE). Our main idea is to focus on performance issues that can be efficiently handled by the multicore and vector facilities of the Cell BE. We aim to offer to programmers an alternative way for high-performance graphics rather than the use of Graphics Processing Units (GPUs). The Cell BE processor is the first implementation of a chip multiprocessor with a significant number of general purpose programmable cores. It is a heterogeneous multicore chip capable of massive floating point processing optimized for computation-intensive workloads that opens up the possibility of implementing highly parallel graphics application on a single chip. Our goal in this tutorial is to introduce the Cell BE Architecture, show the main differences in its programming model, describe its development environment, and give some step-by-step examples of Cell BE programs. We also introduce the usage of a Playstation 3 (PS3) as a high-performance Cell platform.**

***Keywords*-Cell Broadband Engine; Multicore Architecture; Parallel Programming.**

I. Introduction

Computer Graphics techniques are applied on several fields of knowledge to generate images from numerical data helping scientists on their analysis. The data can come from a variety of sources like simulations, experiments and data acquisition, among others, and as their sizes grow toward Tera scale and beyond, computer graphics researchers are faced with a huge challenge: **performance**. An efficient graphics program requires a fine-tunned implementation, which involves thorough understanding of the computer language used, as well as about the aimed architecture.

The field of computer architecture, however, has never been more dynamic. Conventional processors, in which performance gains are obtained by increasing clock frequency, are being replaced by power-efficient multicore processors and Graphics Processing Units (GPUs). The idea of these new architectures is to take advantage of chip die area, traditionally devoted to super scalar out-of-order scheduling and issue logic of large on-chip centralized caches, in favor to implement SIMD execution units, large register sets, and large memory units distributed among different levels of hierarchy. An exciting example of this technology trend is the Cell Broadband Engine (Cell BE) architecture which offers massive SIMD processing power on multiple computational units interconnected via a high bandwidth internal bus.

The Cell BE is a breakthrough microprocessor with unique capabilities for high performance computation on graphics, imaging and visualization, and for a wide scope of data parallel applications. It has a heterogeneous multicore architecture that was developed in conjunction by IBM, Sony, and Toshiba [1]. The architecture is composed of one PowerPC processor and eight synergistic processing elements, named *SPEs*. It is used in the game console Playstation 3 (PS3), and was also the basis for the fastest supercomputer in the world, as announced in the last Top 500 (in 2009) [2]. This cluster of processors, called *Roadrunner*, was the first cluster to operate at the speed of over 1 Peta flops.

All the computational power of the Cell BE processor, however, comes at the cost of a different programming model. The new architectural features of the Cell BE present different challenges to application and system programmers [3]. These challenges are: (i) use of multithreading, (ii) code vectorization, (iii) and management of a different memory model. In the memory model provided by the Cell BE processor, there is no cache management hardware to move code and data to and from main memory, i.e., the application must explicitly manage the memory hierarchy. In other words, the overall performance of the application strongly depends on the effective usage of the Cell BE hardware which is largely left to the programmer.

Therefore, advanced programming techniques are required to unleash the power of this new processor. They are the key to attaining the high performance computation of which the architecture is capable of. In this tutorial, we will discuss these programming techniques. We give a brief overview of the main Cell BE architectural features, in order to discuss some basic concepts on Cell programming, like vectorization, SPE parallelization and intra-chip communication. We also give an overview on how to use the PS3 as a platform for high-performance programming, detailing installation and configuration of a PS3. Finally, we will

do code walk-throughs, with guidelines on implementing multithreading and SIMD-ized codes, and exploring local memory communication issues.

The remainder of this survey is organized as follows. In Section II we give an overview on the Cell BE Architecture. In Section III we explains the differences in programming for the Cell BE processor. In Section IV we discuss the Cell BE development environment, while in Section V the hardware environment setup is addressed. In Section VI we illustrate Cell BE programming by several examples. Finally, in Section VII we draw our conclusions.

II. The Cell BE Processor Architecture

The Cell BE is quite unique as a processor. Its project starts in 2000, with the IBM, SCEI/Sony and Toshiba Alliance being formed. In 2001 the STI Design Center was opened in Austin, Texas, and more than four years later the first technical disclosure was announced. The project took about half a billion dollars and 500 people. In 2006, the Alliance was extended for 5 more years.

The Cell BE is a heterogeneous multicore architecture that combines a traditional PowerPC core with multiple mini-cores, that have limited, but SIMD-optimized, set of instructions. A single chip contains a Power Processing Element (PPE) and eight Synergistic Processing Elements (SPEs). The designation synergistic for this processor was chosen carefully because there is a mutual dependence between the PPE and the SPEs. The SPEs depend on the PPE to run the operating system, and, in many cases, the top-level control thread of an application. The PPE depends on the SPEs to provide the bulk of the application performance.

Figure 1 shows the architecture of the Cell BE processor. The PPE, SPEs, DRAM controller, and I/O controllers are all connected via four on-chip data rings, called Element Interconnect Bus (EIB). Following in this section, we will explain the PPE, SPE, EIB and the differences between PPE and SPE in more details.

A. The Power Processing Element (PPE)

The PPE is a traditional 64-bit dual-thread PowerPC, with a Vector Multimedia Extension (VMX) unit and two levels of on-chip 32 KB cache L1 for instruction and another 32 KB for data, and 512 KB of L2 cache. The PPE has a simultaneous multithreading (SMT) processor – PowerPC Processing Unit (PPU) – that provides two independent execution units to the software layer. In practice, the execution resources are shared, but each thread has its own copy of the architectural state, such as general-purpose registers.

Despite the quite high clock frequency of the PPE, 3.2 GHz, its main purpose is to serve as a controller and supervisor of the other cores on the chip.

B. The Synergistic Processing Element (SPE)

The SPEs are the primary computing engines of the Cell processor. Each SPE is a special purpose RISC processor with 128-bit SIMD capability that runs additional Cell-specific set of instructions. It consists of a processing core called the Synergistic Processing Unit (SPU), a Memory Flow Controller (MFC), and a 256 KB Local Store memory area. The local storage is used to store both code and data being processed by the SPE, but it is not a cache. The SPEs cannot access main memory directly, so all the code and data being processed by the SPE must fit in this local memory. Any data needed by the SPE, that is stored in the main memory, must be loaded explicitly by software into the local storage, through a DMA operation. Data transferred between local storage and main memory must be 128-bit aligned. The size of each DMA transfer can be at most 16 KB. Once the data is in the local memory, the SPU can use it by explicitly loading it into one of its 128 general-purpose registers, each 128 bits wide. The SPU instruction set is different from the PPE instruction set and consists of 128-bit SIMD instructions. Two SIMD instructions can be executed per clock cycle in the SPE.

C. The Element Interconnection Bus (EIB)

The EIB can transmit 96 bytes per cycle, for a bandwidth of 204.8 Gigabytes/second, allowing more than 100 outstanding DMA requests. The EIB is build of four unidirectional rings, two in each direction.

D. PPE $\times$ SPE

The PPE is quicker at task switching. The SPEs are faster at compute-intensive tasks. Typically, the operating system runs on the PPE, while user-mode threads are executed on the SPEs. A significant difference between the PPE and SPEs lies on how they access memory:

- The SPEs instruction-fetches, and load and store instructions access the private Local Store (LS), rather than the main storage. The accesses to the main storage are done asynchronously by the MFC with explicit direct memory access (DMA) commands. This 3-level organization of storage (register file, local store, main storage), with asynchronous DMA transfers, helps hiding memory latency by overlapping computation and data transfers;
- The PPE accesses main storage directly, making use of L1 and L2 caches, with load and store instructions. The PPE can access each SPE's Local Store as well, working as a system manager in both fronts: memory and computational tasks.

Understanding of the programming model differences between PPE and SPEs is crucial to take advantage of this new architecture. It is important to remember that the Cell BE architecture is only powerful if properly tuned.

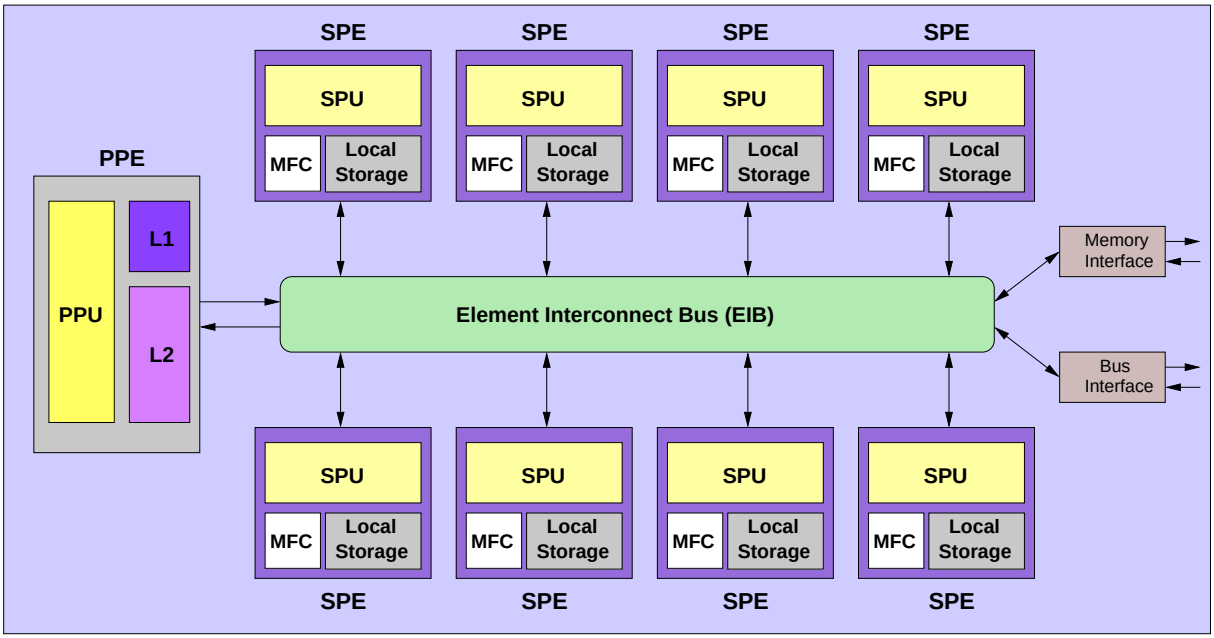

Figure 1. Architecture of the Cell BE processor.

III. Programming the Cell BE Processor

The Cell BE processor presents to the programmer a different programming model. The application performance depends on the effective use of the special features of the Cell BE processor. We emphasize here the aspects that distinguish the Cell BE programming from the programming of conventional processors.

The Cell BE processor can be programmed using standard C/C++ language, relying on libraries deployed by IBM Software Development Kit (SDK) [4]. The user application have to handle communication, synchronization, and SIMD computation. An interesting point is that an existing application would run on the Cell processor by a simple recompilation of the code using only the PPE core, with no effort, but also without advantages from a performance point-of-view.

There are several key differences in coding for the Cell processor compared to traditional multicore CPUs:

1) The power of the SPEs comes from SIMD vector operations. The SPEs are not optimized to run scalar code and handling unaligned data. High performance computation can only be achieved if the data is organized in a way that is suitable for SIMD calculations.
2) SPEs have no cache memory. The DMA engine is exposed. The explicit memory access programming poses an extra work when compared to normal cache-based memory hierarchy. All the code and variables must be allocated in the local store. Larger data structures in main memory can be accessed, via explicit DMA transfers. Furthermore, the DMA calls in the code have to be designed to use double buffering or similar tricks to avoid stalling due to latency.
3) SPEs lack branch prediction hardware. This feature allowed the engineers to pack more computation cores into the chip. However, a branch costs around 20 cycles, which implies that they should be avoided in performance-aware codes.

Nevertheless, the PPE and the SPE are not binary compatible. Summing up, the programming model is an important aspect which distinguishes the Cell processor from other multicore processors.

A. Exploring SIMD Facilities

SIMD processing exploits data-level parallelism, which means that the operations on all elements of vectors can be performed at the same time. That is, a single instruction can be applied to multiple data elements in parallel, as illustrated in Figure 2.

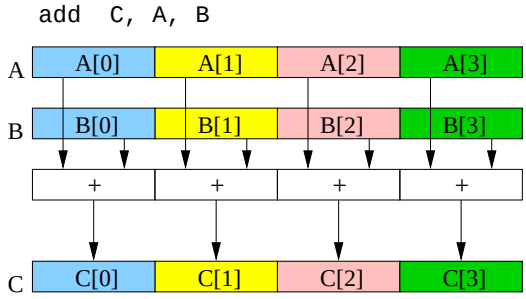

Figure 2. Four parallel add operations in each element of two vectors.

There is support for SIMD operations on both PPE and SPEs. In the PPE, they are supported by the Vector/SIMD Multimedia Extension (VMX) instruction set. In the SPEs, on the other hand, they are supported by the SPU Instruction Set Architecture (ISA). Nevertheless, the high-level functions providing access to these instructions are quite similar, for instance, the vector add function in the PPE is named `vec_add`, while in the SPE is `spu_add`. The process of preparing a program to run on a vector processor is called vectorization or SIMD-ization.

The SPEs represent the power behind the Cell BE processor, and they are inherently vector processors. They do not implement efficiently scalar (non-vector) operations.

B. SPE Parallelization

Programs running on the Cell BE typically partition the work among the eight available SPE, as each SPE is assigned with a different task and data to work on. From the programming point-of-view, managing the work among SPEs is similar to working with threads. The SDK contains a library, called *libspe* library (SPE runtime management library), that assists in managing the code running on the SPE and communicate with this code during execution. This library provides standardized low-level application programming interface (API) that enables applications to access the SPEs and run program threads on them.

It is not recommended, however, that the application allocates more SPE threads than the number of SPEs available. SPE context switching is expensive, since it requires to store most of the 256 KB of the local store in memory and reload it with the code and data of the new thread. That is why the operating system is not suitable to run on the SPE.

A PPE module starts an SPE module by creating a thread on the SPE, using the `spe_context_create`, `spe_program_load`, and `spe_context_run` library calls. The `spe_context_create` call creates a context for the SPE thread which contains the persistent information about a logical SPE. This information should not be accessed directly by the application. Before being able to run an SPE context, an SPE program has to be loaded into the context using the `spe_program_load` call. An SPE context is executed on a physical SPE by calling the `spe_context_run` function. This function causes the current PPE thread to transition to a SPE thread by passing its execution control from the PPE to the SPE whose context is scheduled to run on. Since the PPE only resumes execution when the SPE stops, separated threads must be created on the PPE for each SPE in order to achieve multiple threads of execution.

A thread can poll or sleep, waiting for SPE threads, using the `spe_get_even` or `spe_wait` calls. SPE Runtime Management library document [5], [6] contains a detailed description of the API for managing the SPE threads.

C. DMA Transfers

The Cell BE processor has a unique memory architecture and understanding this architecture is one of the key issues for Cell programming. A SPE program references its own LS using a Local Store Address (LSA). The LS of each SPE is also assigned a Real Address (RA) range within the system's memory map. This allows privileged software to map LS areas into the effective address (EA) space, where the PPE, other SPEs, and other devices that generate EAs can access the LS.

Memory Flow Controller (MFC) is the hardware component that implements most of the Cell BE's inter-processor communication mechanism including the most significant means to initiate data transfer – DMA data transfers. While located in each of the SPEs, the MFCs interfaces may be accessed by both program running on a SPE, or a program running on the PPE.

The MFC supports naturally aligned transfer sizes of 1, 2, 4, or 8 bytes, and multiplication of 16-bytes, with a maximum transfer size of 16 KB. Peak performance can be achieved for transfers when both the EA and LSA are 128-byte aligned and the size of the transfer is a multiple of 128 bytes.

Software running on a SPE may access the MFC facilities through the channel interface, while software running on a PPE may access the MFC facilities through the *Memory-Mapped I/O* (MMIO) interface. MFC functions are a set of convenient functions, each one performs a single DMA command. The basic operations for requesting and sending data from/to the main memory are `mfc_get` and `mfc_put`, respectively. These functions are non-blocking, so the software will continue its execution after issuing those commands. These functions will block only if the command queue is full.

After DMA command was initiated, the software may wait for the completion of a DMA transaction. The three main functions for this end are `mfc_write_tag_mask`, `mfc_read_tag_status_any`, and `mfc_read_tag_status_all`. The function `mfc_write_tag_mask` writes a tag mask which determines to which tag IDs a completion notification is needed. The function `mfc_read_tag_status_any` waits until any of the specified tagged DMA commands is completed, and the function `mfc_read_tag_status_all` waits until all of the specified tagged DMA commands are completed. MFC also supports a set of synchronization and atomic commands that can be used to control the order in which DMA storage accesses are performed.

For a more detailed description see Programming Support for MFC Input and Output chapter in C/C++ Language Extensions for Cell BE Architecture documentation [5], [6].

D. Efficient DMA Transfer and Computation Overlapping

The DMA asynchronous data transfers avoids stalls in the SPE, by allowing the overlap of DMA transfers and computation. One usual parallel programming technique that can be used to explore this overlapping is a well-know technique in computer graphics called *double buffering*, illustrated in Figure 3. A non-overlapping sequential data transfer forces the SPE to wait for the data before it can be computed (see Section VI for examples).

The double buffering technique uses two buffers, B_0 and B_1, in the following way: while the SPE processes the current data in B_0, an asynchronous data transfer is in course, generating the next data for B_1. In the ideal case, both the transfer and the computation take the exact same amount of time, as shown in the example of Figure 3. Therefore, all the data transfers (except for the first) are hidden behind computation, with no stall in the SPE.

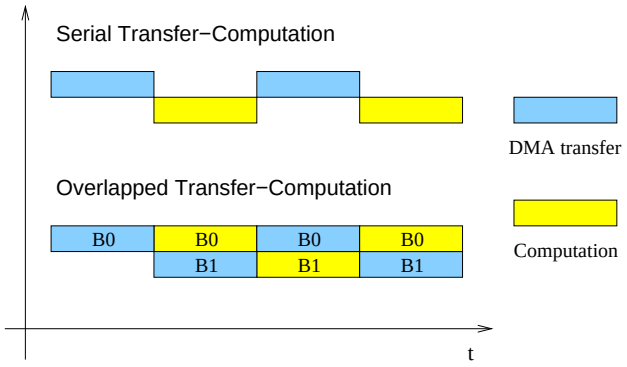

Figure 3. Double buffering technique example.

E. Branch Elimination

The SPE architecture does not include dynamic branch prediction. So, as branches are relatively expensive in the Cell BE processor, sometimes it is important to remove performance-critical conditions in the code. The secret to eliminate branches is to exploit the select bits instruction. That is, an *if-then-else* statement can be eliminated by processing both clauses and using select bits to choose the result as a conditional function. Listing 1 shows an example of removing a branch in the SPU. The first piece of code uses normal branching, while the second avoids the branch.

```
// Normal branching code
if (value < 0) {
  temp = p0;
  p0 = p1;
  p1 = temp;
}
// Avoiding branching
test = !spu_cmpgt(value, zero);
temp = p0;
p0 = spu_sel(temp, p1, test);
p1 = spu_sel(temp, p1, spu_andc(all,test));
```

Listing 1. Eliminating branching in the SPU.

All variables are vectors. The command `spu_cmpgt` sets the variable `test` for one if `value < 0`; and zero otherwise. After that, the `spu_sel` functions select the first or second parameter, depending on the value of the test, in order to assign to the corresponding variable on the left side of the equal sign.

F. PPE Programming

Applications running on the Cell BE processor normally have at least two source codes. One for the PPE and another one for the SPE, since the SPEs and PPE are designed to run different types of code. The SPE core uses a different instruction and register set from the PPE core, so the programs written for the PPE and SPEs must be compiled by different compilers. Table I shows a summary of the differences between the PPE and the SPEs.

Feature	PPE	SPE
# SIMD Registers	32 (128 bits)	128 (128 bits)
Register files	separate fixed, float and vector	unified
Load latency	variable(cache)	fixed
Addressability	2^{64}bytes	256K bytes (local) 2^{64}bytes (DMA)
Instruction set	PowerPC	optimized for single-precision float
Single-precision	IEEE 754-1985	extended range
Doubleword	none for SIMD	double-precision float SIMD

Table I
PPE VERSUS SPE ARCHITECTURAL DIFFERENCES [5].

The 128-bit VXU operates concurrently with the PPU's fixed-point integer unit (FXU) and floating-point execution unit (FPU). Like PowerPC instructions, the VXU instructions are 4 Bytes long and word-aligned, they have three or four 128-bit vector operands. The VXU instructions support simultaneous execution on multiple elements that make up the 128-bit vector operands. These instructions have been chosen for their utility in digital signal processing algorithms, including 3D graphics. They include the following types: Vector Integer, Vector Floating-Point, Vector Load and Store, Vector Permutation and Formatting, Processor Control, and Memory Control Instructions. The vector elements may be byte, halfword, or word.

A set of C-language extensions are available for PPE Vector/SIMD programming [7]. These extensions include additional vector data types and a large set of scalar and vector commands, called *intrinsics*. The VMX intrinsics and predicates use the prefix, `vec_` in front of an assembly-language or operation mnemonic. In Listing 2, we illustrate an example of a very simple program that illustrates the ease way in which vector instructions can be incorporated into a PPE program.

```c
#include <stdio.h>
typedef union {
  int iVals[4];
  vector signed int myVec;
} vecVar;
int main() {
  vecVar v1, v2, v3; // Define variables
  // Load two vectors values into v1 and v2
  v1.myVec = (vector signed int){2, 2, 2, 2};
  v2.myVec = (vector signed int){10, 20, 30, 40};
  // Add vectors using intrinsic function
  v3.myVec = vec_add( v1.myVec, v2.myVec );
  printf( "Sum: %d, %d, %d, %d",
          v3.iVals[0], v3.iVals[1],
          v3.iVals[2], v3.iVals[3] );
  return 0;
}
/// Output result:
/// Sum: 12, 22, 32, 42
```

Listing 2. Example of PPE's VXU intrinsic function usage.

G. SPE Programming

The SPE supports both single-precision and double-precision floating-point operations. Single-precision instructions are performed in a 4-way SIMD fashion, fully pipelined, whereas double-precision instructions are partially pipelined. The data formats for single-precision and double-precision instructions are those defined by *IEEE Standard 754*, but the results calculated by single-precision instructions are not fully compliant with this standard.

The SPE's Local Store (LS) can be regarded as a software-controlled cache that can be filled and emptied by DMA transfers. It holds both instructions and data. When there is competition to access the LS, the SPU arbitrates access to the LS according to the following priorities: (1st) DMA reads and writes by the PPE or an I/O device; (2nd) SPU loads and stores; and (3rd) Instruction prefetch.

For the communication with the PPE, the architecture offers three main mechanisms:

- **DMA:** To transfer data between main storage and LS.
- **Mailbox:** To control communications between a SPE and the PPE or other devices. Mailboxes holds 32-bit messages. Each SPE has two mailboxes for sending messages and one mailbox for receiving messages.
- **Signal Notification:** To control communications from PPE or other devices. Signal notification (also known as signaling) uses 32-bit registers that can be configured for one-sender-to-one-receiver signaling or many-senders-to-one-receiver signaling.

There are 204 instructions in the SPU Instruction Set Architecture (ISA), and they are grouped into 11 classes according to their functionality [8]. The classes are: Memory Load and Store, Constant Formation, Integer and Logical Operations, Shift and Rotate, Compare, Branch and Halt, Hint-for-Branch, Floating-Point, Control, SPU Channel, SPU Interrupt Facility, Synchronization and Ordering.

As SPE is specialized in operating with 128-bit vectors, it is useful to group the data in SIMD vectors. In a 3D application, for example, each vertex (v_0, v_1, v_2) of a triangle could be stored as homogeneous coordinates (x, y, z, w) in a SIMD vector [9]. In this case, the fourth component of each vertex is not used and for each triangle the memory space for one vertex is wasted.

For scalar operations, the SPE uses a scheme named *preferred slot* in the 128-bit vector registers, as shown in Figure 4. Using the preferred slot scheme requires extra operations of shifting the element to the preferred slot and then shifting it back to its original location, which explains why the SPEs are not suitable for scalar operations.

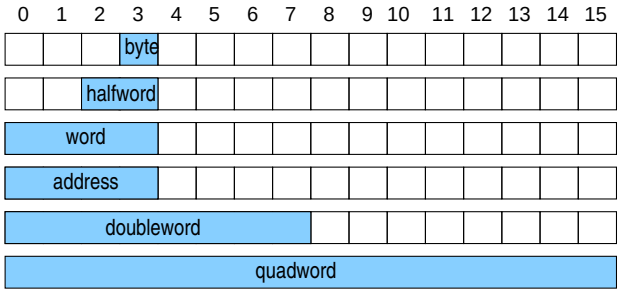

Figure 4. Register layout of data types and preferred slot.

A set of C-language extensions, also called intrinsics, are available for the SPE programming as well. They provide to programmers explicit control of the SPE SIMD instructions without directly managing registers, i.e. without using assembly instructions. For example, the SPU compiler provides the intrinsic function `t = spu_add(a, b)` to replace the assembly instruction `fa rt, ra, rb`. The intrinsic `spu_add` is replaced directly to the `fa` instruction when the SPE code is compiled. In Listing 3, we show the same example program illustrated for PPE programming, coded to run on the SPE.

```c
#include <stdio.h>
typedef union {
  int iVals[4];
  vector signed int myVec;
} vecVar;
int main() {
  vecVar v1, v2, v3; // Define variables
  // Load two vectors values into v1 and v2
  v1.myVec = (vector signed int){2, 4, 6, 8};
  v2.myVec = (vector signed int){1, 2, 3, 4};
  // Add vectors using intrinsic function
  v3.myVec = spu_add( v1.myVec, v2.myVec );
  printf( "Sum: %d, %d, %d, %d",
          v3.iVals[0], v3.iVals[1],
          v3.iVals[2], v3.iVals[3] );
  return 0;
}
/// Output result:
/// Sum: 3, 6, 9, 12
```

Listing 3. Example of SPE's SIMD intrinsic function usage.

H. Basic Guidelines on Code Development

Here we summarize several basic guidelines for code development in the Cell BE:

- Execute main code on the PPE;
- Parallelize code for the SPEs – maintain the memory access sequential and do not use vectorization yet;
- Vectorize SPE code;
- Properly prepare your data for processing on the SPE – data alignment;
- Include overlapping in DMA transfers;
- Avoid branches on the critical paths.

I. Implementation Tips

Besides the important programming practices presented so far, there are some extra programming tips presented by Brokenshire [10], that can be useful to programmers in order to improve the performance of their Cell-based applications:

1) **Offload as much work onto the SPEs as possible** – SPEs represent the computation power of the Cell BE, so use the PPE as the control processor, to command the SPE execution.
2) **Choose a partitioning and work allocation strategy that minimizes atomic operations and synchronization events** – This is a common practice in parallel programming, for the Cell BE one possible strategy is to let the SPEs algorithmically partition the work. For example, consider an image-processing application in which the scan lines are processed by n SPEs. Each SPE can algorithmically compute its work partition by dividing the scan lines to be computed by n.
3) **Accommodate potential data type differences** – The PPE and SPEs may have data types with different sizes. The PPE can be either ILP32 or LP64 (integers are 32 bits, longs and pointers are 64 bits), while the SPE is always ILP32. Therefore, 64-bit applications should be careful when dealing with data structures to be shared mong the PPE and the SPEs.
4) **Exploit multithreading on the PPE** – The PPE is two-way multithreaded in that two threads of architectural state are maintained. So, multithreading is strongly encouraged when the threads experience significant L1 and L2 cache misses, or the threads code have dependencies in the floating-point operations.
5) **Design data structures for efficient access** – To achieve efficient SPE data accesses, programmers should consider data alignment, access patterns, and location.
6) **Initiate DMAs from the SPE** – Instead of the PPE pushing data to the SPE, let the SPE pull the data using MFC to initiate the DMAs. This is done to avoid bottlenecks in the PPE, and because the number of cycles to initiate a transfer from the SPE is smaller than the number of cycles to initiate the same transfer from the PPE.
7) **Stay on-chip** – If your application allows, try to organize it in such a way to use as much of the LS as possible, and to share data with others SPE, avoiding communication with the global memory.
8) **Avoid external scalars on the SPE** – Dealing with scalars requires several dependent instructions, once they are usually rotated to the preferred vector element.
9) **Unroll and pipeline loops** – It should be removed all the loops in which the number of iterations are know to be constant. Possessing a large register file, the SPEs are able to unroll loops of considerable size.
10) **Avoid integer multiplies** – It must be avoided the execution of 32-bit integer multiplication, since the SPE contains only a 16x16 bit multiplier.
11) **Consider computing versus using pre-computed results** – A common strategy many programmers use to improve application performance, called tables of precomputed values, are not efficient on SPE programs, since it does not SIMD-ize well and consume valuable LS space. If possible, it is preferable to compute those values again rather than reading from memory.
12) **Design for limited local store** – The SPE local store is limited to 256 KB, available for program instructions, function stack, local data structures and DMA buffers.

IV. Development Environment

To enable the programmer to take advantage of the Cell BE Architecture, IBM has deployed a Software Development Kit (SDK) [4] that provides libraries, tools, and resources to develop and tune applications for the technology. It includes not only a set of development tools, but also a simulated environment capable of running the latest Linux kernel with Cell BE architecture extensions and runtime support. Among the many components available in the SDK [5], [6], we highlight:

- GNU extended tools including C/C++ compilers, (based on *gcc*), linkers, debuggers (based on *gdb*), assemblers and binary utilities for both processor units on the PPE and SPE;
- IBM specialized compilers, namely *xlc* for C/C++ and *xfl* for Fortran, for both PPU and SPU;
- Standardized SIMD math libraries for the PPU's Vector/SIMD Multimedia Extension and the SPU Instruction Set Architecture;
- A set of analyzing and performance measure tools, such as *oprofile*, *CellPerfCount*, *FDPR-Pro*, *CodeAnalyzer* and *spu_timing*;
- SPE Runtime Management Library providing a standardized, low-level application programming interface for special accesses to the SPEs;
- An Integrated Development Environment (IDE) for Eclipse, an open development platform;

- The IBM Full-System Simulator, a software application that emulates the behavior of a full system that contains a Cell BE processor;
- Source codes containing programming examples, sample libraries, benchmarks and demos.

V. Environment Setup

In this survey, we use the Playstation 3 (PS3) as the target architecture to develop on the Cell BE, showed in Figure 5. In order to be able to compile and run Cell BE codes in a PS3, we have to setup a programming environment on the game console. The first step is to install the *OtherOS bootloader*, which allows a Linux operating system installation on the PS3. Currently, the IBM SDK [4] can be only used on a Linux OS. To both bootloader and Linux kernel work properly, the PS3 *Firmware* must be up-to-date – under the PS3's main menu select the *System Update* feature. More information about PS3 system configuration can be found in the PS3's User Manual [11].

Installation of the Linux operating system on a PS3 system varies depending on the Linux distribution, for this reason we restrict this instruction to the Fedora Core distribution [12]. Fedora has been supporting PS3 hardware since Fedora 5, while IBM SDK has been made available for this distribution since Fedora 7.

To accomplish the first step, we need a storage media, such as an USB flash drive with FAT file system or a CD/DVD. The media is used to store the bootloader in a specific path (*PS3/otheros/otheros.bld*). The second step is to install the bootloader: turn on the PS3 with the storage media containing the bootloader inserted; select *Settings / System Settings / Install Other OS* under PS3's main menu; proceed with the boot installation. After installed, select *Other OS* as the *Default System* under *System Settings* menu. The system will start using the installed bootloader.

The final installation step is to insert a bootable Fedora Core disc in the PS3, and install it. Once the installation is finished, the PS3 is ready to use as a regular PC with a Cell BE processor instead of a CPU. The only configuration step remaining is the IBM SDK installation. At the present time, IBM supports Fedora Core 9 developer package installation, which we have successfully tested and used in the examples provided in Section VI.

Figure 5. Cell BE Architecture used: Playstation 3.

VI. Programming Examples

In this section, we show several examples of complete programs that run on the Cell BE processor and explore both PPE and SPE programming. Our idea is to illustrate the creation of threads, DMA transfer instructions, SIMD commands, and the parallelism between DMA transfer and computation with the double buffering technique.

A. Parallel Hello World

Our first example is a simple multithreaded *Hello World* that uses all the available SPEs to print a *Hello World*. This example shows the structures needed for thread creation, and uses different *Hello World* strings for each SPE in order to illustrate the use of the DMA transfer functions. We show the complete PPE and SPE code in Listing 4 and 5, respectively.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <libspe2.h>
#include <pthread.h>
#include "hello.h"

// Maximum number of SPEs to be used
// Regrettably, PS3 has only 6 available SPEs
#define MAX_SPE_THREADS      8

// Typedef inside hello.h (used by PPE/SPE code)
typedef struct _ctrl_block {
  // Address of the input and the SPE id
  unsigned long long in_addr;
  unsigned int       id;
} ctrl_block_t;

// Data structure used by the pthread function
typedef struct _ppu_thread_data {
  // SPE context and control block pointer
  spe_context_ptr_t spe_context;
  void*             ctrl_block_ptr;
} ppu_thread_data_t;

// The SPE program
extern spe_program_handle_t hello_spu;

// The Hello World strings
char szHelloWorld[MAX_SPE_THREADS][16]
     __attribute__ ((aligned (128)));

// PPE pthread function that starts the SPE thread
void *ppu_pthread_function( void *argp ) {
  ppu_thread_data_t *thread_data;
  spe_context_ptr_t ctx;
  unsigned int entry;

  thread_data = (ppu_thread_data_t *)argp;
  ctx = thread_data->spe_context;
  entry = SPE_DEFAULT_ENTRY;

  // Start the SPE thread
  spe_context_run( ctx, &entry, 0,
                   thread_data->ctrl_block_ptr,
                   NULL, NULL );

  // Kill the thread when the SPE returns
  pthread_exit(NULL);
}
```

```c
// The PPE main function
int main( void ) {
  int i, spus;
  spe_context_ptr_t ctxs[MAX_SPE_THREADS];
  pthread_t threads[MAX_SPE_THREADS];
  ctrl_block_t *ctrl_blocks[MAX_SPE_THREADS];
  ppu_thread_data_t ppu_thread[MAX_SPE_THREADS];

  // The Hello World strings
  strcpy( szHelloWorld[0], "Ola Mundo!"     );
  strcpy( szHelloWorld[1], "Hello World!"   );
  strcpy( szHelloWorld[2], "Hei Verden!"    );
  strcpy( szHelloWorld[3], "Hola Mundo!"    );
  strcpy( szHelloWorld[4], "Ciao Mondo!"    );
  strcpy( szHelloWorld[5], "Hallo Welt!"    );
  strcpy( szHelloWorld[6], "Hallo Wereld!" );
  strcpy( szHelloWorld[7], "Hei Maailma!"   );

  // Determine the number of usable SPEs
  spus = spe_cpu_info_get( SPE_COUNT_USABLE_SPES,
                           -1 );

  // Create and configure each SPE thread
  for (i = 0; i < spus; i++) {
    // Create SPE context
    ctxs[i] = spe_context_create( 0, NULL );

    // Load program into context
    spe_program_load( ctxs[i], &hello_spu );

    // Allocate control block with memory aligned
    posix_memalign( (void**)(&ctrl_blocks[i]),
                    128, sizeof(ctrl_block_t) );

    // Initialize the control block
    ctrl_blocks[i]->in_addr = szHelloWorld[i];
    ctrl_blocks[i]->id      = i;

    // Load ppu_thread_data
    ppu_thread[i].spe_context    = ctxs[i];
    ppu_thread[i].ctrl_block_ptr = ctrl_blocks[i];

    // Create thread for each SPE context
    pthread_create( &threads[i], NULL,
                    &ppu_pthread_function,
                    &ppu_thread[i] );
  }

  // All the SPEs are running in threads,
  // wait each one of them to finish
  for (i = 0; i < spus; i++) {
    pthread_join( threads[i], NULL );

    // Destroy context
    spe_context_destroy( ctxs[i] );

    free( ctrl_blocks[i] );
  }

  return 0;
}
```

Listing 4. PPE *Hello World* example.

The PPE code uses the library *libspe2* and creates the SPE threads using `spe_context_create`, `spe_program_load`, and `spe_context_run`. For each SPE thread, the PPE must create its own thread that will start the SPE thread and stay blocked until the corresponding SPE thread finishes.

```c
#include <stdio.h>
#include <stdlib.h>
#include <spu_mfcio.h>
#include "hello.h"

// Allocate 128-bits aligned variables in LS
char str[16] __attribute__((aligned (128)));
ctrl_block_t ctrlb __attribute__((aligned(128)));

// The SPE main function
int main( unsigned long long speid,
          unsigned long long argp,
          unsigned long long envp ) {
  unsigned int tag;
  unsigned long long addr;

  // Reserve MFC tag
  tag = mfc_tag_reserve();

  // Issue a DMA get command to MFC to retrieve
  // the control block structure from main memory
  mfc_get( &ctrlb, argp, sizeof(ctrl_block_t),
           tag, 0, 0 );

  // Wait for the DMA to complete
  mfc_write_tag_mask( 1 << tag );
  mfc_read_tag_status_all();

  // Set addr to the effective address
  addr = ctrlb.in_addr;

  // DMA get command to retrieve string
  mfc_get( &str, addr, 16, tag, 0, 0 );

  // Wait for the DMA to complete
  mfc_write_tag_mask( 1 << tag );
  mfc_read_tag_status_all();

  // print string
  printf( "[spe: %d] %s\n", ctrlb.id, str );

  return 0;
}
/// Output result:
/// [spe: 2] Hei Verden!
/// [spe: 4] Ciao Mondo!
/// [spe: 5] Hallo Welt!
/// [spe: 3] Hola Mundo!
/// [spe: 1] Hello World!
/// [spe: 0] Ola Mundo!
```

Listing 5. SPE *Hello World* example.

The SPE code starts by reserving an identifier tag in the MFC for the DMA commands, i.e. each DMA command is tagged with a 5-bit Tag Group ID. This identifier is used to check or wait on the completion of all queued commands in one or more tag groups. After that, a `mfc_get` is executed in order to bring the `ctrl_block_t` structure, defined in the shader file `hello.h`, from the main memory. This is an useful structure that allows the PPE to pass information to the SPEs. In this example, the information is the address of the *Hello World* string in the main memory and the SPE id (specified by the program).

The function `mfc_read_tag_status_all` makes the SPE program stall, waiting for the arrival of the control block. Once it arrives, the program requests the correspond-

ing *Hello World* string via DMA, with another `mfc_get` command using the address received by the control block. Note that, this address is the effective address (EA) of the string in main memory. After the string is received, it is printed on the screen together with the SPE id.

B. Using DMA

In our second example, we go deeper on the DMA transfer commands. The second program computes the sum of two vectors, $C = A + B$, in parallel. Each SPE is responsible for summing one block of A and B, generating a block of C. To accomplish this, the program must first transfer the blocks of A and B from main memory to the respective LS of each SPE, and, after the sum is completed, the program must send the blocks of C back to the main memory.

We show the complete PPE and SPE code in Listing 6 and 7, respectively.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <libspe2.h>
#include <pthread.h>
#include "defs.h"

#define MAX_SPE_THREADS      8

// Number of elements in a block
#define NE          (4096*1024)

// The following four typedefs are inside
// defs.h (used by PPE/SPE code)
typedef unsigned long long int ulli;
typedef unsigned int uint;

// Unsigned int 4D Vector type
typedef struct _ui4 {
  uint x, y, z, w;
} ui4;

typedef struct _ctrl_block {
  ulli in_addrA, in_addrB, out_addr;
  uint num_elem, id;
} ctrl_block_t;

typedef struct _ppu_thread_data {
  spe_context_ptr_t spe_context;
  void*             ctrl_block_ptr;
} ppu_thread_data_t;

extern spe_program_handle_t simpledma_spu;

void *ppu_pthread_function(void *argp) {
  ppu_thread_data_t *thread_data;
  spe_context_ptr_t ctx;
  uint entry;

  thread_data = (ppu_thread_data_t *)argp;
  ctx = thread_data->spe_context;
  entry = SPE_DEFAULT_ENTRY;

  spe_context_run( ctx, &entry, 0,
                   thread_data->ctrl_block_ptr,
                   NULL, NULL );
  pthread_exit( NULL );
}
```

```
// The PPE main function
int main( void ) {
  uint i, spus, offset;
  spe_context_ptr_t ctxs[MAX_SPE_THREADS];
  pthread_t threads[MAX_SPE_THREADS];
  ctrl_block_t *ctrl_blocks[MAX_SPE_THREADS];
  ppu_thread_data_t ppu_thread[MAX_SPE_THREADS];
  ui4 *bA, *bB, *bC; // data blocks

  spus = spe_cpu_info_get( SPE_COUNT_USABLE_SPES,
                           -1 );

  // Create and initialize data blocks
  posix_memalign((void*)&bA,128,NE*sizeof(ui4));
  posix_memalign((void*)&bB,128,NE*sizeof(ui4));
  posix_memalign((void*)&bC,128,NE*sizeof(ui4));

  // Initialize input data blocks
  for (i = 0; i < NE; i++) {
    bA[i].x = i+1; bA[i].y = i+3;
    bA[i].z = i+6; bA[i].w = i+9;
    bB[i].x = 2;   bB[i].y = 4;
    bB[i].z = 8;   bB[i].w = 16;
  }

  // Initialize result data block with zeroes
  memset( bC, 0, NE * sizeof(ui4) );

  // Split work across SPEs using address offset
  offset = NE / spus;
  for (i = 0; i < spus; i++) {
    ctxs[i] = spe_context_create( 0, NULL );

    spe_program_load( ctxs[i], &simpledma_spu );

    posix_memalign( (void**)(&ctrl_blocks[i]),
                    128, sizeof(ctrl_block_t) );

    ctrl_blocks[i]->in_addrA = &bA[i*offset];
    ctrl_blocks[i]->in_addrB = &bB[i*offset];
    ctrl_blocks[i]->out_addr = &bC[i*offset];
    ctrl_blocks[i]->num_elem = offset;
    ctrl_blocks[i]->id       = i;

    // Last SPE computes the extra elements
    if( i == spus - 1 )
      ctrl_blocks[i]->num_elem += NE % spus;

    ppu_thread[i].spe_context    = ctxs[i];
    ppu_thread[i].ctrl_block_ptr = ctrl_blocks[i];

    pthread_create( &threads[i], NULL,
                    &ppu_pthread_function,
                    &ppu_thread[i] );
  }

  for (i = 0; i < spus; i++) {
    pthread_join( threads[i], NULL );
    spe_context_destroy( ctxs[i] );
    free( ctrl_blocks[i] );
  }

  printf( "First vector: bC[0] = %d, %d, %d, %d",
          bC[0].x, bC[0].y, bC[0].z, bC[0].w );

  return 0;
}
/// Output result:
/// First vector: bC[0] = 3, 7, 14, 25
```

Listing 6. PPE DMA example.

Each vector element is a struct called `ui4Data`, containing four integers: x, y, z, w. The PPE code starts each SPE thread in the same way done in the *Hello World* example. The PPE creates its own threads that stays blocked until the corresponding SPE thread finishes. The only difference in this code is that the control block structure is initialized, for each SPE thread, with the adress of the respective block of the vectors A, B, and C.

```
#include <stdio.h>
#include <stdlib.h>
#include <spu_mfcio.h>
#include "defs.h"

// Number of elements in each SPE iteration
#define CHUNK  (512)

// Allocate 128-bits aligned local buffers
ui4 lA[CHUNK] __attribute__((aligned (128)));
ui4 lB[CHUNK] __attribute__((aligned (128)));
ui4 lC[CHUNK] __attribute__((aligned (128)));

ctrl_block_t ctrlb __attribute__((aligned (128)));

// The SPE main function
int main( ulli speid __attribute__((unused)),
          ulli argp,
          ulli envp  __attribute__((unused)) ) {
  uint tag, offset, i, j, num_chunks, ne_chunk;
  ulli in_addrA, in_addrB, out_addr;

  tag = mfc_tag_reserve();

  mfc_get( &ctrlb, argp, sizeof(ctrl_block_t),
           tag, 0, 0 );

  mfc_write_tag_mask( 1 << tag );
  mfc_read_tag_status_all();

  in_addrA = ctrlb.in_addrA;
  in_addrB = ctrlb.in_addrB;
  out_addr = ctrlb.out_addr;

  // Compute the number of chunks
  num_chunks = ctrlb.num_elem / CHUNK;
  // Add 1 chunk if there are extra elements
  if( ctrlb.num_elem % CHUNK )
    num_chunks++;

  // Add vectors by chunks, the SPE has a limited
  // memory space (256 KB) in its LS
  offset = 0;
  for (i = 0; i < num_chunks; i++) {
    // Compute number of elements in this chunk
    if( CHUNK > ctrlb.num_elem - offset )
      ne_chunk = ctrlb.num_elem - offset;
    else
      ne_chunk = CHUNK;

    // Issue a DMA get command to retrieve
    // block A and B from main memory
    mfc_get( &lA, in_addrA + offset * sizeof(ui4),
             ne_chunk * sizeof(ui4), tag, 0, 0 );
    mfc_get( &lB, in_addrB + offset * sizeof(ui4),
             ne_chunk * sizeof(ui4), tag, 0, 0 );

    // Wait for the DMA to complete
    mfc_write_tag_mask( 1 << tag );
    mfc_read_tag_status_all();

    // Add elements in this chunk
    for (j = 0; j < ne_chunk; j++) {
      lC[j].x = lA[j].x + lB[j].x;
      lC[j].y = lA[j].y + lB[j].y;
      lC[j].z = lA[j].z + lB[j].z;
      lC[j].w = lA[j].w + lB[j].w;
    }

    // Issue a DMA put command to store block
    mfc_put( &lC, out_addr + offset * sizeof(ui4),
             ne_chunk * sizeof(ui4), tag, 0, 0 );

    // Wait for the DMA to complete
    mfc_write_tag_mask( 1 << tag );
    mfc_read_tag_status_all();

    offset += CHUNK;
  }

  return 0;
}
```

Listing 7. SPE DMA example.

The SPE code also starts in the same way done in the *Hello World* example. First, it reserves an identifier tag, and brings the `ctrl_block_t` structure from the main memory. After that, the program establishes chunks of vectors to be transferred from main memory. It is not possible to bring all the data from vectors A and B from main memory in one DMA transfer, due to the limited capacity of the LS. For each chunk, the program requests the data via a DMA `mfc_get` command, with the address received by the control block. Once the chunk arrived, the sum is processed and the result is put in the main memory through a DMA `mfc_put` command. When all the chunks were computed, the thread finishes.

C. Using SIMD

The third example illustrates the use of SIMD operations, in the code presented in Section VI-B. We used the same vector sum example, but the actual sum was SIMD-ized.

We show in Listing 8 only the code for the SPE, since the PPE code is exactly the same as the one presented in Section VI-B.

```
#include <stdio.h>
#include <stdlib.h>
#include <spu_mfcio.h>
#include "defs.h"

#define CHUNK  (512)

ui4 lA[CHUNK] __attribute__((aligned (128)));
ui4 lB[CHUNK] __attribute__((aligned (128)));
ui4 lC[CHUNK] __attribute__((aligned (128)));

ctrl_block_t ctrlb __attribute__((aligned (128)));

// The SPE main function
int main( ulli speid __attribute__((unused)),
          ulli argp,
          ulli envp __attribute__((unused)) ) {
  uint tag, offset, i, j, num_chunks, ne_chunk;
  ulli in_addrA, in_addrB, out_addr;
```

```
  tag = mfc_tag_reserve();

  mfc_get( &ctrlb, argp, sizeof(ctrl_block_t),
           tag, 0, 0 );

  mfc_write_tag_mask( 1 << tag );
  mfc_read_tag_status_all();

  // Input/output addresses for the current chunk
  in_addrA = ctrlb.in_addrA;
  in_addrB = ctrlb.in_addrB;
  out_addr = ctrlb.out_addr;

  num_chunks = ctrlb.num_elem / CHUNK;
  if( ctrlb.num_elem % CHUNK )
    num_chunks++;

  offset = 0;
  for (i = 0; i < num_chunks; i++) {
    if( CHUNK > ctrlb.num_elem - offset )
      ne_chunk = ctrlb.num_elem - offset;
    else
      ne_chunk = CHUNK;

    mfc_get( &lA, in_addrA + offset * sizeof(ui4),
             ne_chunk * sizeof(ui4), tag, 0, 0 );
    mfc_get( &lB, in_addrB + offset * sizeof(ui4),
             ne_chunk * sizeof(ui4), tag, 0, 0 );

    mfc_write_tag_mask( 1 << tag );
    mfc_read_tag_status_all();

    // Add elements using SPE intrinsic
    for (j = 0; j < ne_chunk; j++) {
      *((vector uint*)&lC[j]) =
        spu_add( *((vector uint*)&lA[j]),
                 *((vector uint*)&lB[j]) );
    }

    mfc_put( &lC, out_addr + offset * sizeof(ui4),
             ne_chunk * sizeof(ui4), tag, 0, 0 );

    mfc_write_tag_mask( 1 << tag );
    mfc_read_tag_status_all();

    offset += CHUNK;
  }

  return 0;
}
```

Listing 8. SPE SIMD example.

The only difference for this SPE code to the one presented for the DMA example lies on the sum of the chunks. In this code the four scalar sum operations were replaced by one SPE SIMD intrinsic function: spu_add. This function allows the parallel sum of the four integers (x, y, z, w) embedded in one vector element.

D. Double Buffering

The fourth and last example is a double buffering code. Double buffering essentially means starting a DMA transfer before the data is actually necessary. The data needed for the next computing step is requested while the current step is computed, as described in Figure 3. This technique is used to hide the memory latency as mentioned on Section III-C. Double buffering is well-known for its use in graphical systems where, by keeping the next frame to display in memory, it is possible to reduce flickering.

The ability to overlap memory transfers with computation can provide several advantages over mainstream cache-based architectures [13]. These advantages, however, can only be effective if there is a balance between the transfer time and the amount of computation done by the SPE. Ideally, the SPE would never be stalled waiting for data. The example shown here, however, is just a simple framework on double buffering. It is not intended to balance the amount of SPE computation with the memory transfer time.

The double buffering program is also grounded on the vector sum code employed in the last two examples. We used the same vector structures, but the DMA transfers of vector chunks are done in a double buffering fashion. We show in Listing 9 only the code for the SPE, since the PPE code is exactly the same as the one presented in Section VI-B.

```
#include <stdio.h>
#include <stdlib.h>
#include <spu_mfcio.h>
#include "defs.h"

#define CHUNK  (512)

// Now we need two local buffers
ui4 lA[2][CHUNK] __attribute__((aligned (128)));
ui4 lB[2][CHUNK] __attribute__((aligned (128)));
ui4 lC[2][CHUNK] __attribute__((aligned (128)));

ctrl_block_t ctrlb __attribute__((aligned (128)));

int main( ulli speid __attribute__((unused)),
          ulli argp,
          ulli envp __attribute__((unused)) ) {
  uint tags[2], i, j, num_chunks, ne_chunk[2];
  ulli in_addrA, in_addrB, out_addr;
  // Current and next chunk ids
  uint curr, next;

  // Reserve two MFC tags
  tags[0] = mfc_tag_reserve();
  tags[1] = mfc_tag_reserve();

  mfc_get( &ctrlb, argp, sizeof(ctrl_block_t),
           tags[0], 0, 0 );

  mfc_write_tag_mask( 1 << tags[0] );
  mfc_read_tag_status_all();

  in_addrA = ctrlb.in_addrA;
  in_addrB = ctrlb.in_addrB;
  out_addr = ctrlb.out_addr;

  num_chunks = ctrlb.num_elem / CHUNK;
  if( ctrlb.num_elem % CHUNK )
    num_chunks++;

  // Compute the number of elements for
  // the first chunk
  curr = 0;
  if( CHUNK > ctrlb.num_elem )
    ne_chunk[curr] = ctrlb.num_elem;
  else
    ne_chunk[curr] = CHUNK;
```

```
// Issue a DMA get command to retrieve first
// chunk without waiting (asynchronous DMA)
mfc_get( &lA[curr], in_addrA, ne_chunk[curr]
         * sizeof(ui4), tags[curr], 0, 0 );
mfc_get( &lB[curr], in_addrB, ne_chunk[curr]
         * sizeof(ui4), tags[curr], 0, 0 );

// Update input address
in_addrA += ne_chunk[curr] * sizeof(ui4);
in_addrB += ne_chunk[curr] * sizeof(ui4);

for (i = 1; i < num_chunks; i++) {
  // Next and current id can only be 0 or 1
  next = curr ^ 1;

  // Compute the number of elements for
  // the next chunk
  if( CHUNK > ctrlb.num_elem - i*CHUNK )
    ne_chunk[next] = ctrlb.num_elem - i*CHUNK;
  else
    ne_chunk[next] = CHUNK;

  // Issue a DMA get command to retrieve next
  // chunk and store it in each local buffer
  mfc_get( &lA[next], in_addrA, ne_chunk[next]
           * sizeof(ui4), tags[next], 0, 0 );
  mfc_get( &lB[next], in_addrB, ne_chunk[next]
           * sizeof(ui4), tags[next], 0, 0 );

  // Wait current buffer DMA command to complete
  mfc_write_tag_mask( 1 << tags[curr] );
  mfc_read_tag_status_all();

  // Update input addresses
  in_addrA += ne_chunk[next] * sizeof(ui4);
  in_addrB += ne_chunk[next] * sizeof(ui4);

  // Add current chunk elements
  for (j = 0; j < ne_chunk[curr]; j++) {
    *((vector uint*)&lC[curr][j]) =
      spu_add( *((vector uint*)&lA[curr][j]),
               *((vector uint*)&lB[curr][j]) );
  }

  // Issue a DMA put command to store current
  // chunk reading it from local buffer C
  mfc_put( &lC[curr], out_addr, ne_chunk[curr]
           * sizeof(ui4), tags[curr], 0, 0 );

  // Update the next output address
  out_addr += ne_chunk[curr] * sizeof(ui4);

  // Walk to the next chunk
  curr = next;
}

// Last chunk was not yet computed
// Wait the last DMA get command
mfc_write_tag_mask( 1 << tags[curr] );
mfc_read_tag_status_all();

// Compute last sum
for (j = 0; j < ne_chunk[curr]; j++) {
  *((vector uint*)&lC[curr][j]) =
    spu_add( *((vector uint*)&lA[curr][j]),
             *((vector uint*)&lB[curr][j]) );
}

// Issue a DMA put command to store last chunk
mfc_put( &lC[curr], out_addr, ne_chunk[curr]
         * sizeof(ui4), tags[curr], 0, 0 );
// Wait for the last chunk writing
mfc_write_tag_mask( 1 << tags[curr] );
mfc_read_tag_status_all();

return 0;
}
```

Listing 9. SPE Double Buffering example.

In this SPE code, the `mfc_get` for the first chunk is started before the iterations begin. For each iteration i, the program starts the transfer for chunk $i+1$ and waits for the arrival of chunk i. After that, the elements of chunk i are summed, using the SIMD instruction `spu_add`. At the end of the iteration, a `mfc_put` DMA command transfers the computed results to the main memory. When the loop ends, the SPE program has still to compute the sum of the last chunk and transfer it to the main memory.

VII. Conclusions

The Cell BE processor offers an innovative architectural approach that has a radically different design than those offered by other multicore processors. This new architecture has a great potential for improving speed, performance, and energy efficiency of many high-performance applications, including graphics applications.

The most distinguishing feature of the Cell BE processor is that the multicore design is heterogeneous and cache hierarchies are replaced by a three level software-controlled memory architecture, which completely decouples main memory load/store from computation. Moreover, the architecture provides vector SIMD capabilities and a massive register file.

Therefore, high-performance programming for the Cell BE is challenging in many ways. Programs that run on Cell BE require the use of multithreading, with the orchestration of computation and memory transfers, and the effective use of SIMD capabilities.

The inherent challenge of programming the Cell BE is also observed in others architectures as well. The fast evolution of GPUs and multicore CPUs frame the end of sequential single-processor programming, in favor for massively parallel programming models. Indications of this future path can be noticed in the nVidia's Compute Unified Device Architecture – *CUDA*, and a recent announcement of a new Intel's architecture – *Larrabee* – converging CPU and GPU pipelines and designing strategies.

In this survey, we address the difficulties of programming the Cell by providing a brief introduction on the Cell BE architecture and by unleashing the potential of this architecture for high-performance graphics programming. We give an overview on the differences in the Cell programming model, with some implementation tips, and provide some complete examples for Cell starters that deals with control of the memory architecture and taking advantage of SIMD capabilities. We hope that this introductory survey will help making the jump on the Cell BE world slightly less arduous.

ACKNOWLEDGMENTS

We would like to thank Prof. Thadeu Penna and the Complex System Group of the Institute of Physics at Fluminense Federal University for providing us full access to the PS3 used in our experiments. We also acknowledge the grant of the first author provided by Brazilian agency CNPq (National Council of Technological and Scientific Development).

REFERENCES

[1] 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.*, vol. 49, no. 4/5, pp. 589–604, 2005.

[2] "Top500 site. `http://www.top500.org`." [Online]. Available: http://www.top500.org

[3] M. Scarpino, *Programming the Cell Processor: For Games, Graphics, and Computation*. Prentice Hall, 2008.

[4] *IBM SDK for Multicore Acceleration for Fedora 9*, IBM, 2008.

[5] "IBM SDK Resources." [Online]. Available: http://www.ibm.com/developerworks/power/cell

[6] "Cell Documentation." [Online]. Available: http://cell.scei.co.jp/e_download.html

[7] *C/C++ Language Extensions for Cell Broadband Engine Architecture*, IBM, 2008.

[8] *Synergistic Processor Unit Instruction Set Architecture - version 1.2*, IBM, 2007.

[9] G. Cox, A. Maximo, C. Bentes, and R. Farias, "Irregular grid raycasting implementation on the cell broadband engine," in *21st International Symposium on Computer Architecture and High Performance Computing*, accepted for publication 2009.

[10] D. A. Brokenshire, *Maximizing the power of the Cell Broadband Engine processor: 25 tips to optimal application performance*, IBM, Jun 2006.

[11] "PS3's User Manual." [Online]. Available: http://www.us.playstation.com/Support/Manuals/PS3

[12] "Fedora for PS3." [Online]. Available: http://fedoraproject.org/wiki/PlayStation

[13] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick, "The potential of the cell processor for scientific computing," in *CF '06: Proceedings of the 3rd conference on Computing frontiers*, 2006, pp. 9–20.