Data Access Problem on the Cell BE Architecture

a master thesis by

Aleš Kozumplík

supervised by

René Rydhof Hansen

Department of Computer Science,
Aalborg University

May 2009

Věnováno rodičům.

Contents

Résumé

Cell BE is a novel processor architecture suitable for multimedia applications, video games and complex scientific computations. To overcome the technological problems that prevent the current architectures from achieving higher performance, Cell BE introduced several novel features in its design which cause it to be more difficult to develop software for. Eight of the nine cores on the chip are not allowed to read or write the main memory directly using its instruction set. Each of these cores has a 256 KB of local memory instead and can initiate DMA communication between this memory and the local memory. Processing data on these cores therefore has to be wrapped in calls to the DMA subsystem. Moreover, to avoid limiting the performance potential of the processor, the DMA communication should be interleaved with computation. This implies that advanced data fetch using double buffering and other techniques is often employed. Managing buffers, DMA communication and synchronization litters the source code and counts for a substantial number of lines in it.

In this work we present a semi-automatic approach to the data access problem on Cell BE. This is done by extending a traditional C-like imperative programming language with new syntax and semantics to form an experimental language called *Dali*. The two main extensions are accessor declaration and accessor application. In an accessor declaration the programmer specifies a strategy of main memory access. Then, using accessor application, the programmer applies the declared accessor to a variable. Within a scope of the application, the variable is accessed according to the declaration.

For instance in the most common scenario of traversing an array of items one by one the programmer can suggest a double buffering accessor to be applied on the array variable. This will cause the code managing the DMA communication and the two buffers to be generated automatically by the *Dali* compiler. The programmer can then focus on the problem itself more and less on writing the boiler plate code. Other access methods than double buffering are also allowed in *Dali*. Those include speculative methods that, based on the programmer's suggestion, prefetch data with high probability of being needed in close future. Caching methods are another possibility. Those expect high data locality and for each read element they also keep in cache other elements within the near proximity of the original location.

The main limitation of the approach is that the program correctness depends on the programmer's judgement when applying accessors. The programmer has to make sure that the suggested access strategy is in fact consistent with the way the memory is dynamically accessed during runtime. For instance, it would not make sense to apply a double buffering accessor on a part of the memory which is accessed in mostly a random pattern. Depending on the implementation, such solution would either slow down the system unnecessarily because most of the memory and DMA bandwidth would be wasted, or, in case no runtime checking was implemented, it would even yield incorrect program output.

To provide a complete overview of the problematics, other existing approaches to managing memory access are summarized and taken into account in the discussion of our solution. To evaluate the viability of the approach, an experimental *Dali* compiler was developed as a part of this work. It operates by parsing the *Dali* source code and emitting C++ code ready to be built by the Cell toolchain and executed on a Cell machine. Several simple experiments were performed with programs generated using our compiler. Both the implementation effort and the experimental results are also documented in this work.

Abbreviations

ALF
: Accelerated Library Framework, a programming environment for data and task parallel applications. Supplied with the Cell SDK.

AST
: Abstract Syntax Tree, a tree data structure generated by a parser, represents the source code in a simplified, easier to manipulate form.

Cell BE,
: Cell Broadband Engine Architecture.

CBEA
: see Cell BE.

CMOS
: Complementary metal-oxide-semiconductor, the class of integrated circuits used in the current microprocessors.

DSL
: Domain Specific Language, a high-level language tailored for a specific task.

EA
: Effective Address, an address pointing in the main storage, main memory-mapped local stores and memory-mapped IO registers.

EIB
: Element Interconnect Bus, is a communication bus internal to the Cell processor connecting various on-chip components.

GFLOP
: gigaflop, or a 10^9 single precision floating point operations.

GFLOPS
: gigaflop per second, a measure of a computer's performance.

ISA
: Instruction Set Architecture, a part of the computer architecture defining the programming interface.

LS
: Local Store, the local memory of an SPE.

MFC
Memory Flow Controller, an SPE component supporting DMA transfers.

MMIO
Memory-mapped I/O, EA mapping of registers allowing other devices connected to the EIB communication with the MFC of an SPE.

PPE
Power Processor Element, the processor on a Cell BE chip, generally suitable for an arbitrary program task; see SPE.

PPU
PowerPC Processor Unit, the execution component of PPE.

PS3
Playstation 3, a video game console produced by Sony Computer Entertainment, featuring Cell BE processor.

SIMD
Single Instruction, Multiple Data, an instruction set achieving data-level parallelism by performing the same operation over many data instances at the same time.

SDK
Software Development Kit, generally a set of compilers, libraries, tools and documentation that supports software development for a particular platform or framework; in this document, unless otherwise stated, SDK refers to the Cell Broadband Engine SDK available from IBM website.

SPE
Synergistic Processor Element, a component of the Cell BE chip suitable for compute-intensive tasks with a low amount of branching, also called accelerator; see PPE.

SPU
Synergistic Processor Unit, the execution component of the SPE.

STI
Sony-Toshiba-IBM, a consortium of companies standing behind the Cell Architecture.

TLB
Translation Lookaside Buffer, CPU cache mapping a virtual memory address to a physical address.

1 Introduction

Since the invention of the microprocessor in the 1970s its users have never ceased to be hungry for a better performance. In the recent years the trend has been in a great deal driven by scientific computations, multimedia applications and video games. During the 1990s an increase in the transistor on-chip density caused an increase in the clock frequency which sufficed for a stable growth of the microprocessor performance. For various technological reasons, most severe of which probably is the problem of power dissipation, different strategies had to be adopted by manufacturers recently to increase CPU performance, including multiplying the number of cores on the chip. Even while the Moore's law still holds true, its application in the last ten years has not been making microprocessors faster as steadily as it was before.

Cell Broadband Engine, or simply Cell BE, built on IBM's PowerPC microprocessor is also a novel architecture aiming to deliver a superb performance. Not only has the chip got nine cores on it, but to overcome the technological limits the processors design had to be in some important ways *simplified* when compared to previous generations. The processor specialization is one of them—only one fully-fledged processor is present. The remaining processors, also called accelerators, are optimized for performing high volumes of floating point operations but need to be programmed using an SIMD vector instruction set and lack features that the programmers have so much grown dependent on with other architectures, particularly the branch prediction and transparent main memory access.

On a modern Intel x86 processor, the hardware branch prediction looks ahead in the code to estimate what branches will be taken and to what locations. This allows the processor to fetch instructions into the pipeline and start processing them without waiting for the intermediate branches to be resolved. Not having this feature available on the accelerator of the Cell BE requires the programmer to limit all branching to minimum.

The transparent memory access in the context of this work means that a given processor can read and write data in the main memory using its load and store instructions. The programmer can then directly access any part of the memory at any time. For efficiency reasons load and store instructions with similar semantics are *not* supported by the Cell BE accelerators. When data should be processed there it first needs to be copied from the main memory to the accessor memory (called local store) using a DMA transfer and only then processed and optionally copied back to the original location.

Software development for Cell BE is thus plagued with several issues. The separate memory model and vectorization (transforming traditional problem to exploit vector instructions) were already mentioned. Further, to use the power of the processor the programmer has to parallelize his algorithm over all the available processor elements and implement the program in a way that ensures determinism and correctness. This process

Figure 1.1: The PlayStation 3 game console is sporting a Cell BE processor

is called parallelization and it is in fact the most important problem that needs to be dealt with when programming Cell BE, but also any other multiprocessor/multicore platform. The topic is a center of attention for contemporary computer scientists, who often state that it is the current programming languages and patterns that are aggravating the issue instead of helping to contain it [28, 31].

Similarly to the attempts of solving the parallelism problem by designing more suitable programming languages, the presented work seeks to solve the memory access problem by proposing modifications to the current procedural languages.

1.1 The Problem

Clearly, processors in computers are turning multicore and, if we do not want to witness sharp decrease in programming productivity, better tools and languages need to be devised to target the new architectures. Concretely, the developers should be given programming languages with abstractions allowing them to exploit every feature the modern architectures offer, and compilers that transform these abstract descriptions into the most effective machine code, performing as much work as possible automatically along the way.

Currently, the main approach of managing the DMA transfers is leaving it up to the accelerator programmer to manually call routines performing DMA transfers whenever data in the main memory needs to be read or written. Because an evident drawback of this approach is the reduced programmer efficiency and programming comfort, IBM is developing a C compiler that, based on static analysis of the code, classifies how the memory is accessed on the accelerator and automatically generates code handling the transfers. Unfortunately, when the target generated this way suffers from degraded performance due to incorrect memory hierarchy management, the programmer has no means of redeeming the problem.

The method of dealing with the data access problem we are about to propose does not critically depend on the compiler's ability to analyze the source code. Instead, a language will be designed and implemented that lets the programmer make a high-level decision about the data access strategy. She can for example say that the given array should be double buffered, or that elements of the given linked list should be prefetched six steps ahead in the local store. Once the compiler is informed about the preferred access strategy, the programmer uses the main memory variables in an entirely transparent way, traversing arrays and dereferencing pointers at will. The compiler then makes sure that the data is treated as was requested. Since the programmer guarantees that she will access the data in certain pattern by requesting the specific access method, the compiler can better optimize the generated code for the DMA transfers and the buffer management.

Such an arrangement does not take out the programmer's responsibility for correctness and efficiency. Accessing elements without respecting the conditions of the used accessor will result in invalid local store accesses or corrupted data, just like it would if the same access methods were coded manually. The hope is that many of the cases of incorrect or suboptimal application of access strategies could be in the future reported by the compiler.

1.2 Structure of the Thesis

The thesis is organized as follows. Chapter 2 comprehensively describes the Cell BE architecture, the ideas that drove its design, its heterogeneous organization and especially the implications the novel features have to the software development on it. Besides the memory hierarchy and the parallelism problem, it is mainly the need for vectorization of programs running on the SPEs.

Chapter 3 focuses closely on the memory access problem. Current approaches are presented and compared with each other in terms of programming comfort, performance and susceptibility to introducing bugs. New language extensions are proposed and evaluated next. The language, called *Dali* for Data Access Language Interface, embodies the abovementioned method of splitting the responsibility for memory transfers between the programmer and the compiler. Supported access methods and possible optimizations are outlined.

Next the Chapter 4 describes how the first compiler for the language was designed and implemented. Measurements of its performance are documented in Chapter 5 along with an evaluation. To give the reader a broader perspective we compare the performance not only to other Cell BE implementations, but also to x86 implementations in C and Python. The thesis concludes in Chapter 6 with the summary of the work done. Opportunities for continuing the research in the area are outlined there as well.

Acknowledgment

I would like to thank my supervisor René Hansen for the help, comments and encouragement I received from him during the work on this thesis. Further thanks to Alexandre David who provided code for the x86 implementation of the experiment in Section 5.2.

2

Cell Broadband Engine

This chapter overviews the Cell Broadband Engine, first from the architectural and then from the programmer's point of view. Stressed are those software development concepts that diverge from the x86 architecture.

2.1 Background

For some time now it has been a known fact that increasing the CPU performance can no longer be achieved by increasing its clock frequency like the manufacturers were doing until the late 90s. Let us mention the three main reasons for this, labeled as memory wall, power limitation wall and frequency limitation wall [25].

The *memory wall* problem occurs because higher processor clock frequencies are not met by decreased dynamic random memory access (DRAM) latencies and this gap increases with every new generation of processors and memories. The memory latencies are in the magnitude of hundreds to thousands of cycles for a multi-GHz processor. Hence the memory becomes a bottleneck of the system and a (theoretical) two-fold increase in frequency does not cause an increase in the performance anywhere near the double of the original.

In practice increasing frequency of a CMOS processor is only possible together with increasing its input power. Dissipation causes some of this power to leave the circuit as heat, requiring the system designers to use sophisticated cooling mechanisms. There are first-order limits on systems, for example the space occupied by a cooling device, its maximum temperature at any time or amount of air leaving it. Clearly, there are limits on the amount of processor's input power and so on its frequency—a barrier known as the *power limitation wall*. Also, with current desktop processors consuming around 100W, improved power efficiency is desirable for economical and ecological reasons.

Finally, the *frequency wall* is a barrier existing due to the observation that a point of diminishing returns for frequency increase has been reached. This is because with increased frequency, the processor's pipeline must be deepened too. With deeper pipelines the number of latches increases thus increasing the instruction latencies. Increasing the frequency only makes sense while the gain from doing so exceeds the penalties incurred by the higher instruction latencies.

With these observations in mind, the processor designers began to focus on other ways to spend silicon and power budgets. The era of multicore processors started in 2001 when IBM's POWER4 appeared, the world's first dual-core processor, targeted for server applications. Few years later Intel and AMD followed with moderately priced dual-core processors aimed for desktops and laptops.

In 2000 discussions between Sony and IBM about the processor for the next Playstation started, with the objectives of achieving $100\times$ the performance of Playstation 2. A third company of the consortium, Toshiba, was brought in as a development and manufacturing partner [40]. The high-level concept including 64-bit Power Architecture and "synergistic" accelerators was settled by the end of 2000. In March 2001, The STI Design Center representing a joint investment of the three companies was opened in Austin, Texas and in 2005 the first units of the new Cell processor were confirmed to appear on the consumer market in the forthcoming Playstation 3 game console.

2.2 Architecture of Cell BE

Before more details about the architecture are revealed, it should be remarked that any concrete information about the chip mentioned in this thesis, like the number of cores, memory sizes or benchmark results, references its version found within PS3. Section 2.5 discusses other applications of the processor.

The design of Cell had to resolve the mentioned struggle between higher performance and better power efficiency. Generally speaking, the approach taken is *decreased circuit complexity* for reduced power dissipation and *increased specialization of the chip components* for boosted performance. The resulting chip has nine processors (or, *processor elements*), connected to each other and to external devices by high-bandwidth, memory-coherent bus [22]. The processor elements are specialized in one of two kinds of tasks:

Control-plane code is code with much branching and conditional execution, typically found in an operating system or control applications, is suited to run on the *Power Processor Element* (PPE).

Data-plane code is data-processing, computation-intensive code, like for example computation kernels. The *Synergistic Processor Element* (SPE) is optimized for executing this kind of task.

The Cell BE has one PPE and eight SPEs. All components of the processor are connected by on-chip *Element Interconnect Bus* (EIB). See Fig. 2.1 for a high-level overview of the Cell processor.

The Two Elements

This section will attempt to describe the two processor specialization found in Cell, highlighting their differences.

PPE, a 64-bit PowerPC core, is the main processor, fully compliant with the 64-bit PowerPC Architecture, able to run 32-bit and 64-bit applications. It runs the operating system, manages the system resources and in most cases it also runs the main control thread of the application. Two instruction sets are supported: the PowerPC instruction set and Vector/SIMD multimedia extension set. The PPE hardware provides support for two simultaneous threads of execution, which is viewed by software as two independent processing units. The following register files are present per each thread:

- 32 by 64-bit general-purpose register file, used for fixed-point operations,

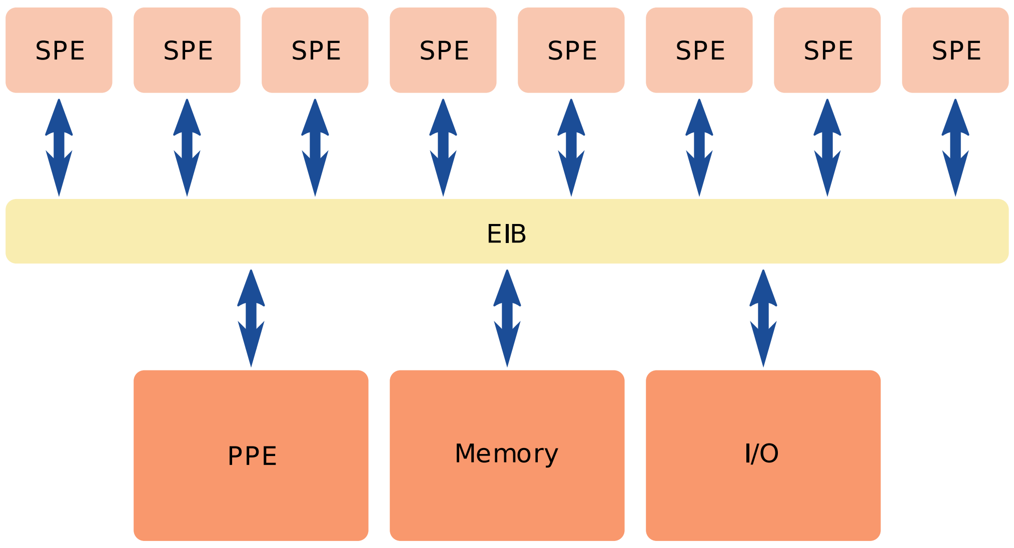

Figure 2.1: The architecture of Cell BE

- 32 by 64-bit register file for floating-point operations,
- 32 by 128-bit vector register file

Using its load and store instructions, the PPE can directly move data between its registers and the virtual memory subsystem.

The design of the PPE was simplified compared to the recent out-of-order microprocessors. The PPE design that does not reorder instructions at runtime ("in-order" issuing) allowed reducing pipeline depth to only 23 stages. There is only 32 KB of first level cache, 512 KB of second level cache on the processor, considerably less than on other modern processors[1].

The eight SPEs provide the bulk of the computing power of Cell BE (Figure 2.2). Each of them containing a 128-bit RISC core (called a Synergistic Processor Unit, SPU), they are optimized for compute-intensive, data-rich tasks. The idea is that the PPE runs the main control thread of an application and offloads the main computation to available SPEs which run their parts in parallel with the PPE and with each other.

There is only one instruction set supported on the SPE (called SPU Instruction Set Architecture [24, 23]), specifically designed for the class of problems that the Cell BE is supposed to excel in: multimedia applications and games. SPE uses *single instruction multiple data (SIMD)* organization and only vector operations (operations over multiple data instances) are supported. The SPE has one 128 by 128-bit register file with each register capable of storing multiple elements of varying width:

- sixteen bytes (8 bit)
- eight halfwords (16 bit)

[1]compare to Pentium Dual-Core E5200 featuring 2048 KB of L2 cache

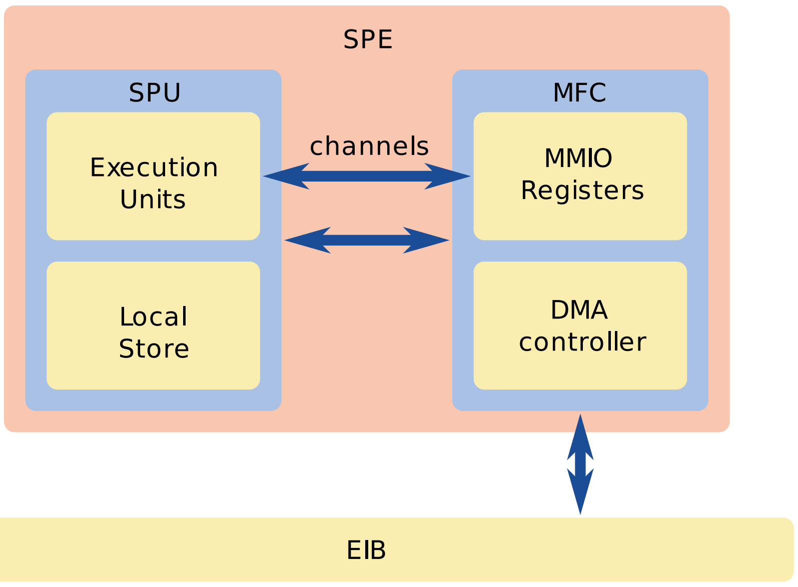

Figure 2.2: The SPE processor

- four words (32 bit)
- two doublewords (64 bit)
- one quadword (128 bit)
- four single-precision floats (32 bits)
- two double-precision floats (64 bits)

Every SPE also includes 256 KB local store (LS). A local store serves as a unified memory: an SPU fetches its code from it and it loads and stores data there. This brings us to an important point, that is *an SPU can not access the main memory directly*. Instead, it needs to move the data from the main storage to the local store by issuing a DMA request and later store the results back to the main storage using another DMA request.

Memory Flow Controller

The Memory Flow Controller (MFC) is an SPE component the main function of which is providing support for DMA transfers on the SPE. It may also be thought of as an interface to the EIB, connecting the SPE with other devices of the chip.

MFC operates autonomously and asynchronously with respect to the SPU. The SPU issues a DMA command and it gets queued on the MFC. The MFC processes this queue while the SPU continues computation. Both simple DMA commands and DMA lists are supported by the MFC.

Additionally, the MFC also provides facilities for SPE synchronization and control: mailboxes and signal notifications. Mailboxes are queues for exchanging 32-bit messages.

Two mailboxes are provided for sending messages from an SPE to the PPE and one for the opposite direction. Similarly, signals are also used for communication between processors, in this case only in the PPE to SPE direction.

The SPU communicates with the MFC through *SPU channels*. The PPE and other devices (including SPEs) communicate with an MFC through *memory-mapped I/O* (MMIO) registers associated with the SPU's channels.

Besides using DMA over MFC, there is no way for the SPU to access the main storage. For code running in privileged mode it is possible to have an SPE local store mapped to the effective-address space of the main storage. Access to the LS through an EA pointer is however generally not cache coherent and this possibility is only mentioned here for the sake of completeness. More information on this aspect can be found in [19, pp. 126]

Element Interconnect Bus

EIB is an on-chip bus that allows different components to communicate with each other. It connects the memory controller, I/O interfaces and, most importantly, the PPE and SPE processors. The EIB is a 4-ring structure for data and a tree structure for commands. The EIB's bandwidth is 96 bytes per cycle, and more than 100 outstanding DMA requests between a LS and the main storage are supported.

2.3 Tackling the Walls

Now that the architecture of Cell BE has been presented, let us take a look at how it deals with the design limitations on the current generation of microprocessors as discussed at the start of this chapter.

The power limitation wall, implying that it is only possible to increase the processor clock frequency if its power efficiency is improved, has been overcome by limiting the chip complexity.

Poor power efficiency of the current CPUs is caused by their high power dissipation, which is in turn caused by high complexity of their circuits. Cell decreases this complexity by making its processors as simple as possible, yet highly specialized for their specific tasks:

- PPE, optimized for execution of control-intensive code
- SPE, the computation accelerator, optimized for execution of computation-intensive code

The memory wall is caused by an order-of-magnitude discrepancy between CPU speed and RAM memory access speed, causing high memory latencies for a cache miss. Instead of transparent caching, Cell BE introduces another level of memory hierarchy, the local store, at each SPE accelerator. Because this memory is on-chip, access to it is extremely fast.

The frequency limitation wall occurs because deepening instruction pipelines are required to achieve higher operating frequencies. Here again, the specialization of PPE and SPE processors and so their simpler design helps to achieve higher frequencies without excessive overhead. The good performance of the PPE is given by its ability to run two threads simultaneously, rather than by deep-pipeline speculation. SPEs include quite large vector register files and support parallel instructions, removing the need for techniques such as register renaming and out-of-order execution that build up the chip complexity.

As a conclusion to this section, it should be remarked that while the physical limitations encountered by processor architects today have been dealt with in a novel, groundbreaking manner in the architecture of Cell BE, the employed solutions often traded improved crude performance for a significant decrease in programming comfort.

2.4 Performance

The following paragraphs discuss theoretical performance of the Cell BE CPU. The 3.2 GHz version from PS3 is considered. Even though that processor has eight cores physically, one of the SPEs is disabled to increase production yields so only seven cores can be used for computation [29].

Each SPU is capable of achieving 25.6 GFLOPS in single precision, yielding a theoretical total processing power of almost 180 GFLOPS. This number is astonishing compared to the top-class desktop CPUs with peak GFLOPS count around fifty [5].

The integrated memory controller can achieve a peak bandwidth of 25.6 GB/s to the XDR RAM, a speed three to four times higher than that of DDR2 memories used in PCs today [40]. The theoretical peak bandwidth of EIB is of 204 GB/s for intra-chip data transfers among the PPE, SPEs, the main memory and I/O controllers. The sustained EIB data transfer speed however fluctuates wildly depending on the physical distance of communicating devices [4].

Those numbers are impressive by the 2009's standards, but achieving a near-peak performance is complicated and not always possible in practice. There are reports of achieving the peak performance in single precision on the SPE, while the double precision performance is much worse, with the peak at 14.6 GFLOPS (however, this still compares favorably to other architectures) [4, 42]. The large performance drop between single precision and double precision tasks is due to the Cell BE's optimization to multimedia applications, which mostly use single precision operations.

2.5 Applications

The first objective of building the Cell BE architecture was to deliver an order of magnitude higher performance for a new generation of gaming consoles and, in 2006, Sony's PS3 thus became the first application of the new processor.

Besides boosting multimedia applications, there were however broader secondary objectives. In particular, the design of the architecture and its enormously high FLOPS per Watt ratio makes the processor fated for applications in cluster computing. IBM has been selling their *blade server* solutions based on Cell BE since 2007. Also built on a blade server technology, the worlds first petaflop system (Roadrunner[2]) uses commodity Cell and AMD processors.

Many other applications have been announced or are considered, among which is a home cinema solution, real time transcoder of high quality video or a PCI extension card

[2]http://www.lanl.gov/roadrunner/

with Cell. Combining tens to hundreds of PS3 machines has become a cheap possibility of cluster computing for science experiments or 3D rendering [2].

2.6 The Cell BE Software Development Kit

The provided development tools and documentation are the topic of this section.

Background

IBM is ambitious with the Cell BE architecture and, in order to create a strong base of developers and gain sympathy of the Linux community, they distribute the Cell SDK as a free download from their website[3].

The SDK package comes with all the necessary components to start development: documentation, toolchain, libraries and a system simulator.

The documentation is comprised from both guides and references. Tutorials are included, as well as source code samples. Particularly the *Programming tutorial* [22] can be recommended for a neophyte Cell programmer. There are documents describing the PPE and SPE domains of the architecture, as well as the assembly languages and the SPE ISA. All the supplied non-standard libraries (e.g. the SPE Runtime Library) are described.

Besides the documentation included with the SDK, IBM provides a lot of other resources targeted for Cell software developers like handbooks, forums and white papers. These are accessible free of charge from the above mentioned web.

Compilers

Since the SPE and the PPE support two different instruction sets, in principle there need to be two different variations of a compiler, targeted for the particular processor. The SDK currently provides two different compilers, a GCC-based compiler having two binaries, `spu-gcc` and `ppu-gcc` to produce object files for the two different targets and IBM XL/C Single Source compiler with only a single binary executable for both types.

When using the GCC compiler, the source code for PPE and SPE is compiled with their respective compilers. The PPE compiler will produce PPE object files. The SPE compiler will produce an SPE executable which is then embedded into a PPE object. This embedded SPE executable and the original PPE object files are finally linked together to produce a single executable binary. The whole process is shown on Figure 2.3. The SDK provides a Makefile framework (built on top of the standard GNU Make) that slightly alleviates complexity of the entire process.

The other compiler currently available for the platform is IBM XL C/C++ for Multicore Acceleration [13], based on IBM's XL C/C++ family of compilers. The particular compiler in the SDK is still in its alpha version. With this *single-source* compiler, code destined for the PPU does not need to be written and compiled separately from the SPU code, all source can be compiled within a single compiler invocation. Being a partial result of IBM's research effort in compiling for Cell [8], this compilers offers some advanced optimizations over

[3]http://www.ibm.com/developerworks/power/cell/

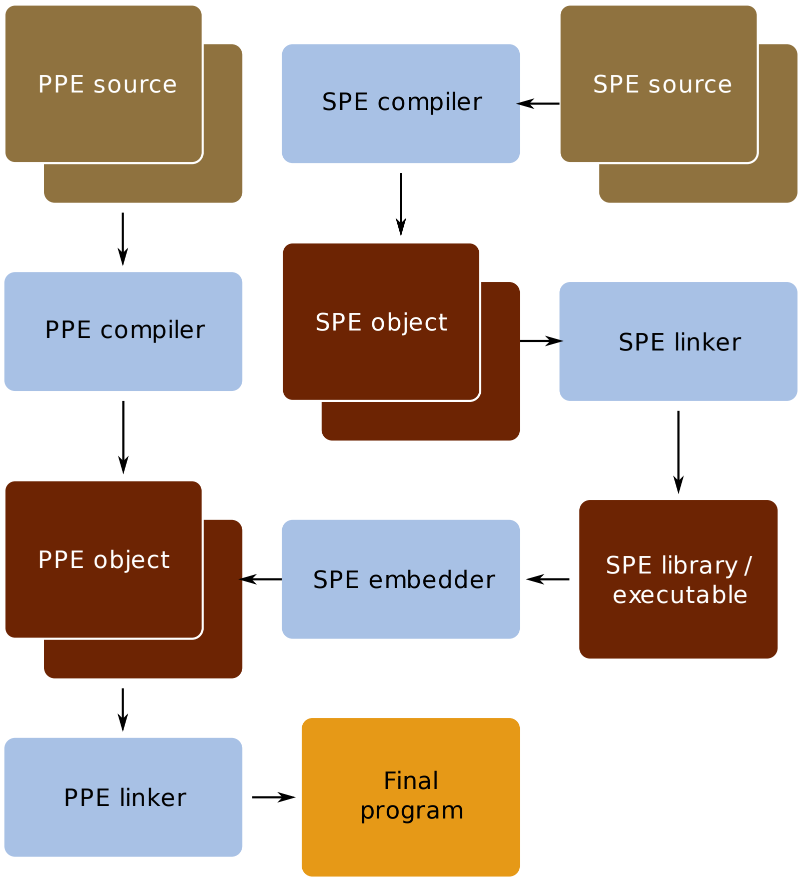

Figure 2.3: Building a Cell BE program

the GCC compiler like auto-vectorization of scalar loops. Most importantly, it allows the developer to instruct it about parallelization of routines using the OpenMP[4] directives.

Research in the field of compilers and other techniques to make programming Cell BE simpler is an interesting topic and we discuss its certain specifics in "Using the Single Source Compiler" on page 37.

Full System Simulator

After the compilation produces the resulting binary, it can be run on a live Cell BE system or executed using the Full System Simulator (or simply "simulator" below) that comes with the SDK. The simulator is cycle-accurate, meaning that it features a faithful reflection of the processor inner timing, instruction scheduling, pipelining etc. The developer can hence profile his code and study the performance bottlenecks almost as well as on a real machine.

[4]http://www.openmp.org

Interestingly, it is SPE's simplicity, the lack of pipeline speculation and of branching prediction, and the absence of nondeterministic caching that makes it possible to reach 90% or higher accuracy of the simulator [4].

2.7 Developing Software for Cell BE

Because Cell BE is based on the PowerPC architecture, programs written for other systems based on the architecture will run on a Cell BE system with no needed modifications. Alas, such programs will not utilize the power of SPEs and hence perform poorly.

This section is an introduction into the problematics of building programs capable of using much of the potential the microprocessor offers by effectively utilizing the SPE accelerators. Inevitably, for most applications to be deployed on Cell BE, that is going to become the goal of their development.

Currently, the only languages supported by the SDK are C/C++, Fortran and Ada (only PPE). We will focus on C/C++ point of view here, assuming similar features and limitations exist for the other languages.

The Compiler Intrinsics

Sooner or later while coding, the programmer will encounter *compiler intrinsics*. Intrinsics are C-language extensions that look like function calls in the source code. They allow the programmer to comfortably execute desired assembly instruction, without resorting to inlining any machine code using GCC's `asm`. There are four intrinsics categories.

- **specific**, mapping one-to-one to a single assembly-language instruction.
- **generic**, mapping one-to-one or one-to-many assembly instructions, depending on the types of the input parameters.
- **composite**, built from a sequence of specific and generic intrinsics.
- **predicates**, evaluated SIMD conditionals.

Intrinsics are typically used for vector operations (integer or floating point), DMA commands and MFC channel communication [16]. Suppose the programmer declared and initialized two float vector variables `a` and `b` and would like to compute their product and store it in vector variable `r`. Instead of inlining the assembly-language instruction `fm rr, ra, rb`, she can simply use the `spu_mul` intrinsic and assign `r = spu_mul(a, b)`.

Intrinsics provide an additional benefit of dynamically adapting to the type of their operands. Continuing the previous example, if the programmer instead of

```
vector float a,b;
```

declared `a` and `b` as

```
vector double a,b;
```

then the `spu_mul` intrinsic would correctly generate `fmd` instruction instead of `fm`.

Running code on the SPE

Generally, three additional steps need to be taken when constructing a Cell BE program, as compared to constructing a non-parallel program for a conventional architecture. The three steps are:

1. Define a parallelization strategy for the problem and deploy subproblems onto the SPEs.
2. Deal with the new memory hierarchy level by defining how and when data is to be moved between the main memory and the local store (see "Memory Access on the SPE" on page 24).
3. Vectorize the SPE code to use SIMD instructions (see "Data Vectorization", page 25).

The programmer's first task is to make his problem parallel and to find out how to distribute subproblems across as many available SPEs as desirable. Concrete methods for optimally spreading the workload across SPEs are described below, in "Programming Models" on page 26. After the design phase is finished, how is this done in practice? Because dual-source compiler is used, the code that will run on PPE has to be placed in different source files than the SPE code. The format of a source SPE program is similar to the standard program source in C, with the `main()` function defining the entry point.

To start a program on the SPE, the programmer calls a function from the SPE Runtime Management Library [18]. Because this function is blocking, the main PPE program usually spawns a POSIX thread, dedicated to "managing" the SPE program.

Once the SPE program is initialized and running, it typically blocks and waits for the input parameters to be sent to it via mailboxes. The input parameter usually contains a pointer to the input data in the main memory. The program then fetches the data into the SPE's local store, processes them and stores the results back to the main memory. The task then either terminates (freeing the SPE for other tasks) or waits for a next set of data to be assigned to it.

Memory Access on the SPE

It was mentioned in Section 2.2 that the SPE units can not access the main memory using their instruction set. When an SPE needs to access data in the main memory, it has to issue a DMA request through its MFC to move data between the main memory and its local store. Only from there can the data be loaded into registers using the load instruction. In other words, from SPE's point of view there are 3 levels which storage is organized into: the register file, the local store and the main memory. Further, a good care has to be taken not to run out of the memory budget as each local store is only 256 KB for *both code and data*. In extreme cases *code overlays* have to be employed to make ends meet with the space available [21].

Note that the PPE can access all 2^{64} addresses of the main memory using its load and store instructions. It can also, using MFC's MMIO problem-state registers, send a command to an MFC to initiate a DMA request into its local store. This option is however somewhat less frequently used than leaving the local store entirely under the control of its SPE.

Because SPE has got only its LS for both code and data, the program has to be preloaded into the SPE before it starts. If the remaining space is insufficient for a large set of data, SPE has to partition the data and then process it in batches—in fact, that is the typical scenario. On the other hand the code should always fit in the LS completely so the code overlays need only be used in rare cases.

What does requesting a DMA transfer look like in practice? On the SPE the programmer typically uses one of the composite intrinsics, for example the `spu_mfcdma32`, passing it five parameters: address in the local store, address in the main memory, total size of the transfer, a request *tag ID* and finally the type of the requested DMA operation itself. Compiler expands the intrinsic into several MFC channel operations that basically just forward the information to the MFC. From that point on, MFC handles the transfer and the SPU can continue other computations and some time later ask MFC whether the given transfer has finished yet. The mentioned tag ID is a number used to distinguish among transfers in case more of them were requested concurrently. There are two basic groups of commands: PUT (for moving data from LS to the MS) and GET (for moving data from MS to LS). Each of these exists in their basic form or can take different flavors to provide guarantees about ordering of the operations with the same tag id etc. The book "Cell Broadband Engine Architecture" [17] includes a complete reference documentation about the MFC, channels and DMA commands.

The need to micromanage DMA transfers is both a bless and an inconvenience. There are few rules on the size or address alignment of the transfer, for example the last four bits of the source address have to be naturally aligned with the transfer size if it is less than 16 kilobytes. The developers thus need to take special care when designing data types to be transferred over DMA. To make things worse, there are further recommendations on the alignment to ensure optimal performance. We will talk about these things more in Chapter 3.

Ideally, the programmer should overlap DMA with computation. Issuing DMA transfers synchronously (that is passively waiting for them to finish) is clearly a programming mistake. Using double buffering and multibuffering is encouraged to mitigate the impact of memory latency. The main memory access strategy is another issue that is to be talked about at a later time.

Data Vectorization

Both the PPE and the SPE support a single-instruction, multiple-data (SIMD) instruction set. While the PPE can be also programmed with a scalar PowerPC instruction set, SIMD is the only choice on the SPE. SIMD instructions perform the same operation on every element of their vector arguments and store results into a vector. Since SPEs support only a SIMD instruction set operating on 128 bits of data at time, scalar code has to be processed using vector operations too. That can lead to a loss of computational throughput, because the results of vector operations are only used in part.

Executing scalar code on SIMD is even more inefficient than that. Consider the C code `a = b + c`, where `a`, `b` and `c` are scalar variables; Typically, two vector loads are executed, yielding two 128 bit register values containing 32-bit values of `b` and `c`. To be able to add those values together using a vector addition, we need to ensure relative alignment of the values in the vectors is the same by permuting contents of one of the vectors to match the remaining one. Next vector addition is performed. Storing the result is not straightforward either: vector load, splicing the 32-bit result into the 128-bit value and vector store are needed. Even though compiler can generate this code automatically, it is often better to

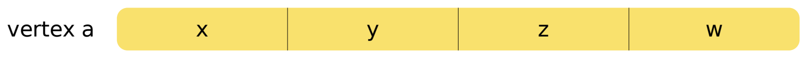

Figure 2.4: Array of structures

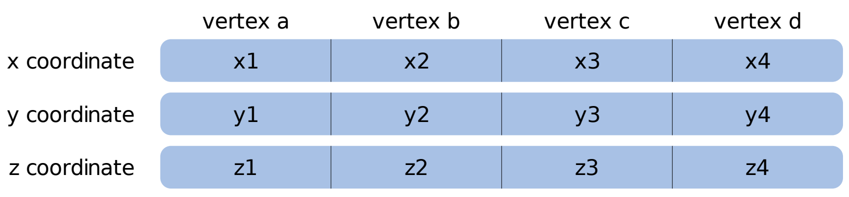

Figure 2.5: Structure of arrays

force the alignment of frequently used scalars to 128 bits. Some memory space is lost, but unnecessary vector manipulations are avoided.

Truly utilizing the SPE's potential is only possible when chunks of data are organized so they can be easily loaded and operated in vector registers. There are generally two methods of packing data into vectors [22]. We are going to document these on an example of vertexes in 3D space, each having three coordinates (x, y, z). The natural way of organizing data structure like this into vectors is simply by placing the coordinates next to each other in a vector, as shown in Figure 2.4. We call this arrangement *array of structures* (AOS).

An alternative is to spread the dependent data (coordinates of a vertex in this case) across several vectors. This method called *structure of arrays* (SOA) causes each vector to contain independent data of the same type. See Figure 2.5.

In most cases (but depending on the algorithm used), programs sporting SOA tend to run faster then those with AOS. Also, as the picture indicates, we avoid the need for filling the unused space of a vector (w) which not only wastes space, but because the corresponding part of result of computation with such vectors is discarded, also processing time. Code sizes on the other hand are smaller with array of structures. With so many variables in play, obtaining the optimum with vectorizing strategy is often a question of experimenting, static analyzing and profiling.

Programming Models

Cell BE offers several programming models for partitioning the given problem among its nine processors. To ensure the resulting program is the most effective, several factors need be considered before the programmer commits to one of the partitioning strategies. These include the structure of the program, program data flow and the cost of communication that would possibly be required by each model.

Two major classes of programing models are:

- PPE-centric model where the main applications runs on the PPE and offloads certain tasks to the SPEs.
- SPE-centric model where the application is distributed among the SPEs with the PPE only acting as a centralized resource manager.

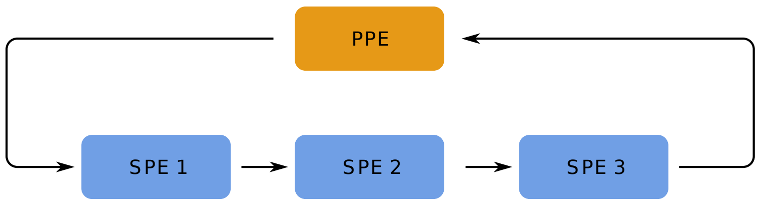

Figure 2.6: The multistage pipeline programming model

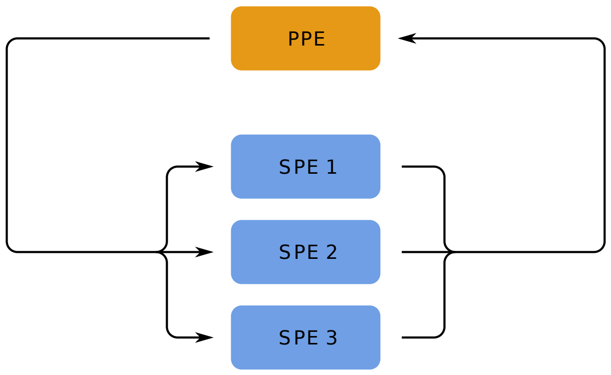

Figure 2.7: The parallel stages programming model

The PPE-centric model is further divided into three models that differ in the way the SPEs are utilized under them:

- The multistage pipeline model.
- The parallel stages model.
- The services model.

The *multistage pipeline model* (Fig. 2.6) is most suitable for tasks consisting of sequential stages, characterized by outputs of a previous stage being inputs of the next stage. An example of a scenario where multistage pipeline model can be applied is a rendering pipeline [38], where the first SPE can determine visibility, the second one perform texturing, the third one figure out the lighting, etc. A drawback of this model is that it can be difficult to equally balance the load among the processors.

The *parallel stages model* (Fig. 2.7) is a model best suited for tasks that process large amounts of data which can be easily partitioned and the parts can be processed in parallel. Different SPEs running the same program can then at the same time act on different parts.

The *services model* (Fig. 2.8) is a generalization of the parallel stages model. Here different SPEs can run different programs, providing different services to the PPE. The PPE requests services from appropriate SPEs as they are needed.

Finally, there is the *SPE-centric* model, where SPEs operate more or less independently of each other, somewhat resembling a machine with nine independent processors. Because the main memory is used as a shared memory under this model, mechanisms need to be

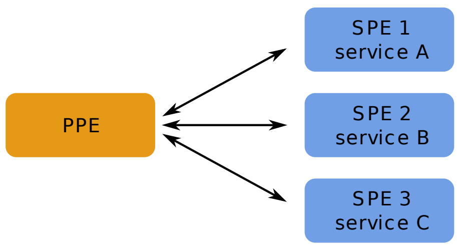

Figure 2.8: The services programming model

used to prevent potentially harmful concurrent data access. Since we have seen above that an SPE can not be used as a fully-fledged microprocessor, this model has limited practical usability.

Developing for Cell BE , Conclusion

The topic was discussed in its plain, elementary form, to show the general intricacies the Cell programmer should expect today. The next few sections then talk about the currently researched solutions that should take a part of the developer's burden off once finished.

Also the described work flow is relevant to using the GCC compiler and can be simplified greatly by using IBM's XL C/C++ compiler. Unfortunately, because it is not in the production version yet and the development community does not seem to be adopting the alpha version, there are currently no real alternatives to programming Cell BE then in C/C++ using the GCC toolchain if one has ambitions about the target performance.

2.8 Development difficulties

Writing well performing software for Cell BE is by and large difficult. The PPE and all the SPEs should be utilized by the program and that requires the programmer to write explicitly multi-threaded code. *Parallelization* is thus the first obstacle. Next, the SPEs only support SIMD instructions. Common scalar code can of course be trivially encoded as an SIMD program but that will waste most of the Cell BE's potential. We can thus identify *vectorization* as the second main development problem. The last major difficulty is the *memory hierarchy* or more precisely having to manually manage data transfers in it. While the remaining chapters elaborate the data access problem and its solutions, this section briefly introduces the problematics of parallelism and vectorization and gives references to further information.

Parallelization

During a parallel computation, many calculations are performed simultaneously. The higher number of cores a system has, the higher degree of program parallelism is required in order to utilize the processing power effectively. There are sequential programs which are straightforward to parallelize because the potentially simultaneous calculations in them are easy to identify. Other programs need to be modified considerably to support parallelization. Yet other programs deal with an inherently sequential problem and can not be parallelized. The computer science nowadays stands before the great problem of devising a universal approach to parallelization. Programming languages constructs and frameworks are proposed. Let us take a look at those of particular interest for a Cell BE developer.

The most general option is parallelizing the C code manually. The programmer chooses a suitable programming model (page 26) and decides what parts of the computation can be offloaded to the accelerators. Respective code has to be ported from the PPE to the SPE. The programmer then uses a multi-threading framework (POSIX threads for example) to start threads that carry out the parallelized computations by launching the SPE programs and providing them with input data.

The XL/C single source compiler [13] supports OpenMP directives [34] that let the programmer specify sections of code that shall be parallelized by the compiler. One of the OpenMP directives, `omp paralell for` is shown as an example:

```
#pragma omp parallel for
for (i = 0; i < N; i++)
    a[i] = 2 * i;
```

The code will run the for loop simultaneously in many threads. If desirable, the XL/C compiler will automatically run some of the parallel threads on the SPE processors. For the programmer, having these constructs at her disposal allows her to focus on the parallel algorithm itself instead of having to manage threads. Also the related code is placed at the same place instead of separated across different source files.

Besides the language-based approaches there are also parallelization-fostering frameworks. One such framework is ALF [15], a part of the Cell BE SDK. With ALF the programmer separates the application into tasks. Optionally dependencies can be set among tasks. For each task the set of input data and a computational kernel to process this data is provided. The input data is separated into work blocks that can be processed in parallel. The ALF runtime then carries out all the task in the correct order and by effectively utilizing all the available SPEs. Data transfers to and from the accelerators, parallel task management, double buffering and load balancing are all done by the runtime.

MapReduce [7] is a programming model for generating large data sets and processing them in parallel. In its nature it is more general than ALF since it was designed to be used on large distributed clusters rather than on a single multicore machine. Nonetheless, it offers an abstraction of *values mapping and reducing* which is powerful in terms of numbers of problems that can be expressed in it and it can be trivially parallelized. Suppose one would use MapReduce to write a program that counts appearances of each word in a large set of documents. The map portion of the program would simply traverse the documents and emit an *intermediate pair* $< w, 1 >$ for each word w. The MapReduce framework collects all the intermediate pairs with the equal first element (key) together and feeds them to the reduce part. In our case the reduce part simply sums the second elements of the intermediate pairs and yields a result for every word. MapReduce is not part of the SDK, although an experimental implementation for the Cell BE exists [6].

Vectorization

Vectorization is a code transformation that operates over several items of an array at the same time (that is, it treats the array as an vector), instead of processing the items one by one. We showed in "Data Vectorization" on page 25 how data has to be manually organized so vector operations over it are possible. Because SIMD instructions sets has become common in general purpose processors even before Cell's arrival, the area of performing the vectorization automatically has been seriously researched since the 90's with the goal of building compilers capable of *auto-vectorization* and, more recently, *auto-SIMDization*. The GCC compiler started to support vectorization in 2004. The techniques used there are described in [32] and [33]. The GCC compiler supplied with the Cell SDK is therefore able to perform certain vectorizing optimizations automatically. With the XL/C compiler the situation is similar, the ways for achieving vectorization and related problems that need to be overcome are summarized in [8].

We can state that vectorization and auto-SIMDization in compilers is maturing. The programmer however needs to be aware of its presence and write the program in a way that makes vectorization possible (e.g. by not introducing possible data dependencies).

Dealing with the Memory Access Problem

This chapter looks in more detail on the problem created by separating the SPE local stores and the main memory on Cell BE. The text below lists things the programmer has to deal with as well as the features offered by the architecture for achieving that. It discusses why this presents an extra burden for the developer should she handle all the issues manually and it discusses existing ways which can help manage the memory transfers. Other not yet existing possibilities are described next.

The second part of the chapter is dedicated to a new solution proposed by this thesis, a set of novel programming language constructs meant to reduce the programming complexity related to memory access management. The concepts mentioned in those pages are the core contribution of this work.

3.1 Main Memory, Local Memory

Perhaps the most surprising thing for a programmer new to the world of Cell BE is her new duty of managing the data flow between different places of the memory hierarchy. The new level of local store memory was introduced as the architecture's means of dealing with the *memory wall*, a major performance bottleneck of a modern CPU architecture that we described in Chapter 2.

The local store is a fast on-chip memory within the near proximity to the SPU and being 256 KB in size. Register loads on the SPEs are only possible from local stores. Access to the main memory is only possible by issuing asynchronous DMA commands that transfer the data into the local store first. It is meant to be the executing code that controls the DMA communication. In other words, unlike for example the L2 cache on x86-based systems, the local store is not transparent. Unless other software mechanisms are in place, it is up to the programmer herself to insert appropriate code performing the DMA.

Let us document this using a simplified pseudo-code example. Suppose there is a memory buffer of length `len` pointed to by `ptr` and we want to process this buffer in-place using function `process()`. On an x86 or similar architecture with direct main memory access, the code would simply be:

```
process(ptr, len);
```

However, on the Cell's SPE, something similar to the following needs to be done:

```
handle = dmaDataToLS(localPtr, ptr, len);
//.. some other computation
```

```
waitFor(handle);
process(localPtr, len);
handle = dmaDataFromLS(localPtr, ptr, len);
//.. more other computation
waitFor(handle);
```

Not only does the programmer need to remember to treat data on the main memory in a special way, she preferably should design her program to keep the SPE entertained with different things while the transfers are taking place (more on this later in Section 3.3).

How do the x86-based systems save the programmer from having to similarly manage their L1 and L2 caches, which are also a kind of extra levels of fast, on-chip memory? The answer is that those microprocessors have been *designed* with such a feature in mind and hardware is present to transparently cache data and instructions flowing into and out of the processor. An unwanted yet inevitable side effect of this feature is therefore a more complex design of the processor. Further, some help by the executing program is still required occasionally, as the more recent revisions of the x86 architecture include a PREFETCH instruction [12, pp. 4-221], purpose of which is to start reading data from a slow memory to a fast on-chip memory with a few (tens, hundreds, thousands) cycles head start, similarly to asynchronous DMA reads on Cell. Whether provided manually or (more often) compiler generated, its use can improve performance critical parts of an application [11, pp. 3-69].

There is one more thing that makes the design of the Cell chip simpler, but the developer's life harder: cache coherency. While the x86 multicore architectures have to provide sophisticated hardware mechanisms of ensuring that the on-chip caches stay coherent, Cell leaves this to the programmer. She has to, for example, ensure that while the local store holds a copy of data from the main memory, no undesirable write access occurs to the original.

Before all the ins and outs of effectively handling the transfers are explained, a look should be taken at what options and limitation for DMA the architecture provides.

3.2 The DMA Subsystem of the Cell BE

This section is going to be answering questions about what the programming features of the architecture's DMA subsystem are and what its limitations are. For a much more comprehensive technical discussion, refer to the architecture's reference manual [17].

An SPE program initiates and controls all DMA transfers by sending commands to the MFC through the MFC channels. The information that needs to be specified is:

1. Local store address.
2. Main memory address (effective address).
3. Size of the transfer.
4. Tag ID of the transfer (see below).
5. Command opcode (put, get, etc.).

Tags are used for distinguishing between different classes of DMA requests, defined by the programmer. We say that a set of commands labeled with the same tag ID forms a *tag* *group*. Status of all DMA operations within the same tag group can be queried at once with a single MFC command.

The DMA system also provides a way to specify several DMA transfers with a single MFC command, using the *DMA lists*. Every element of a list specifies an address in the main memory and the size of data to transfer. Only one LS address is specified, the target address for the first transfer in the list. Because the DMA list method uses only a single LS area and the data specified by the list items are stored in a continuous sequence in the LS, target address of any given item can be computed as a sum of the previous item LS address and its transfer size. While this scheme doesn't allow for a set of completely arbitrary transfers, it is sufficient for an efficient implementation of *scatter-gather lists*.

So far we have seen four different types of MFC DMA commands: PUT, GET, PUTL (PUT list) and GETL (GET list). Each of these commands can take two additional flavors modifying their semantics: barrier and fence.

Fenced command (e.g. PUTF) guarantees local ordering of this command with respect to all previously issued commands on the same SPE and within the same tag group.

Barrier command (e.g. PUTB) guarantees local ordering of this command *and all subsequent commands within the same tag group* with respect to all previously issued commands on the same SPE and within the same tag group.

Ensuring command ordering is important for example when there is a possibility that two consecutive commands target the same main memory or local store address. In such cases, barriers and fences allow the programmer to *queue* locally ordered commands without having to wait for a completion of any of them.

The SPE compiler supports composite intrinsics using which the programmer can pass all the parameters and the DMA command to the MFC in one go. Suppose the programmer wants to move 4096 bytes of data from the local store address `buffer` to the main memory `ea_ptr`, using the `t0` tag ID. The intrinsic to use in this case is `spu_mfcdma32`.

```
spu_mfcdma32(buffer, ea_ptr, 4096, t0, MFC_PUT_CMD);
```

To make sure that the transfer has finished, the programmer waits for the status bit given by `t0` to be updated. The last operation in the following snippet blocks the processor until there are no outstanding operations in tag group `t0`.

```
mfc_write_tag_mask(1 << t0);
mfc_read_tag_status_all();
```

Certain limitations apply when requesting a DMA transfer. There are rules any triple of a local store address, a main memory address and a transfer size used in a single DMA request has to satisfy [17, pp. 80–83], summarized in the following list:

1. The **transfer size** must be 0, 1, 2, 4, 8, 16 or a multiple of 16 bytes, but not more than 16 KB.
2. The last four bits of the **effective address** must be naturally aligned with the transfer size.
3. Finally, the last four bits of the **local store address** must be equal to the last four bits of the effective address.

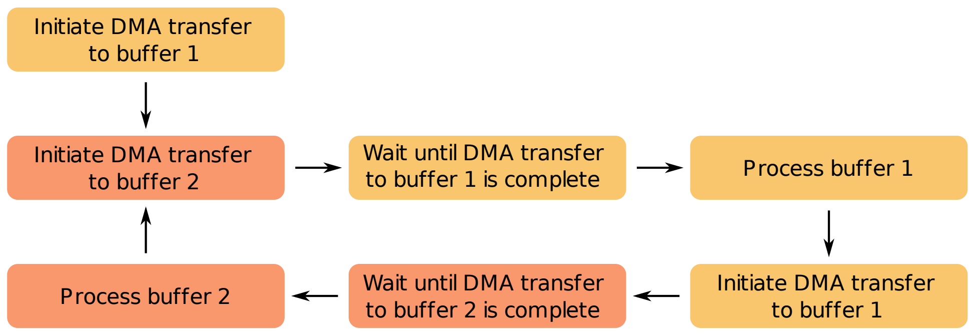
Initiate DMA transfer to buffer 1
Initiate DMA transfer to buffer 2
Wait until DMA transfer to buffer 1 is complete
Process buffer 1
Initiate DMA transfer to buffer 1
Wait until DMA transfer to buffer 2 is complete
Process buffer 2

Figure 3.1: Double buffering

Failure to obey these rules leads to undefined behavior. On both the simulator and on the PS3 machine this typically demonstrates itself as an immediate SPE program termination.

The DMA and memory subsystems provide optimum performance only when even stricter rules are followed, for example the addresses being aligned to 128 bytes or, ideally, transferring only entire cache lines.

3.3 Handling the Transfers Manually

Later on, this chapter talks about ways to fully or partially automate the DMA communication between the main memory and the local store. Before we get there however, this section describes the programmer's duties and discretions when she decides to take on these issues with a manually crafted code.

The manual approach lets the programmer completely micromanage the LS buffers and transfers to and from them as well as full control over when to stall the SPU until a given DMA transfer is finished. This gives her the necessary flexibility needed to design an SPE program able to process and move data in and out of the LS at the same time, a very important concept often encountered in the Cell BE development.

The technique typically used for implementing interleaving of a computation and a data transfer is known as *double buffering*. While a naive implementation would process data by synchronously loading them to an allocated buffer, processing them and synchronously storing them back to the main memory, double buffering (Figure 3.1) operates by using two buffers instead.

It starts by initiating fetching data into the first and second buffer at the same time and then synchronously waiting for the first buffer's transfer to finish to start computation there. By the time the computation on the first buffer's data is done, transfer into the second buffer has often either finished or progressed significantly. The program starts asynchronous transfer into the first buffer again and then, with little or no waiting time, continues the computation on the second buffer. It is straightforward to extend this technique for storing the computed results back into the main memory. Other variations of double buffering that employ more than two buffers (multibuffering) exist.

The way the data is organized in the memory will often have an effect on the fashion in which it will be processed, because of the performance constrains given by different access methods. Double buffering is the most effective approach for sequentially processing a stream of data and contrary, if the data is of a stream nature and unlimited in size, the programmer should use double buffering or a similar technique.

Clearly, not any combination of data and algorithm can use double buffering or use it effectively. Think for example processing an array at seemingly random indexes or with each successive index given by the value found at the previous access. In such cases the programmer can only speculate on which part of the array to asynchronously start fetching next. If a heavy computation has to be performed over relatively small amount of data that completely fits in the LS, it might even be the best solution to synchronously fetch all data up-front using a single DMA to avoid excessive DMA setup overhead in the future.

The important point of those considerations is that in an SPE program there is a relationship between the used algorithm, the organization of the data in the memory and the *optimal existing method* for accessing the data. In many cases the programmer might realize that changing the structure of the input data will allow for use of a more effective memory transfer strategy and sometimes she will actually be able to refactor the data type to embrace this change. As an example of this, think of storing a matrix in column-major order instead of row-major order to allow for a simple double buffering scheme during a column-wise access.

Other times however the same data type is used during a computation on other SPEs, where the programmer either has no control of the algorithm and access method used or where the original data organization is actually preferred over the new one. In the latter case, it is up to the programmer to carefully discover the "sweet spot" between all the used algorithms, access methods and the single data organization.

Still other times when the data organization is an obstacle to a good performance, one might employ the PPE to preprocess the input data into the desired format (as demonstrated for example in [39]). This arrangement has two possible drawbacks that the programmer should consider. First is the increased code complexity because more extensive synchronization is taking place between the SPE and the PPE. Second drawback is more important: due to the new computational load on the PPE, other tasks running there might run slower. Additionally, distributing the SPE task over more SPEs might not scale well as PPE could get choked with reorganizing all the inputs and outputs.

Manually programming LS data loads and writes is in practice error prone. It is so because the programmer needs to manage the LS buffers and issue DMA commands with changing address and size parameters, all that while keeping an eye on the transfer sizes and data alignments. In the common case of asynchronous DMA transfers, timing and synchronization concerns arise too.

A high number of buffer operations and DMA calls in the SPE source code tend to obfuscate it somehow, making debugging and/or extending the program a more complex task than it would be in the case of a similar program developed for an x86 architecture. In practice, it also is not easy to factor this code out into a library functions, as the SPE does not handle branching too effectively. Declaring the library functions using the `inline` keyword is not a reliable solution either, since it is still up to the benevolence of the compiler to actually perform inlining [30, pp. 112–113]. The only viable alternative in the C-language to repeating oneself within a performance critical SPE source code is therefore using preprocessor macros.

Finally, and that is perhaps the most unpleasant thing about manual memory access, the tight bond between the data organization, memory access strategy and the used algorithm makes the code hard to modify. To put it other way, if either the data organization or the used algorithm changes, the programmer will probably need to adjust the memory access strategy as well. Such a change is susceptible to create new bugs and new performance grievances.

Despite all the shortcomings of manual handling of the SPE/PPE DMA communication, its performance is superior to any other existing solution discussed next.

3.4 Current Alternatives of Managing the Transfers

While implementing all the DMA transfers by hand is the preferred option for applications trying to extract every GFLOP from an SPE, different possibilities demanding less programming effort exist. The most significant of those approaches are treated in this chapter. They are namely the SPU Software Cache from the SDK library, ALF (a framework for data and task parallel applications introduced in the previous chapter) and the XL/C single source compiler.

SPU Software Cache

The first method we will discuss is provided by the Example Library of the SDK [20]. To a certain degree the SPU Software cache can emulate the L1 and L2 caches of other processors, but it is of course not transparent. Its API provides two sets of interfaces to control it, safe and unsafe.

The *safe interface* is most useful for programs with a high-amount of hard to predict memory accesses and it operates synchronously. To use the cache `mycache` to read a data item at the main memory location `eaptr` using the safe interface:

```
i = cache_rd(mycache, eaptr);
```

If a given item is in the cache already (cache hit), it is returned immediately. In the opposite case (cache miss), the item is synchronously loaded from the main memory. This in effect makes the call to `cache_rd()` block until the data is ready. Similarly, call to `cache_wr()` modifies the cache contents and might also block if the cache line is not currently present.

Clearly, using the software cache through the safe interface can only provide reasonable performance when the cache hit ratio is high, otherwise the program will be blocked in DMA calls too often. Notice that during a typical read-modify-write cycle using the safe interface, the EA to LS address translation occurs, extraneously, twice, on call to `cache_rd()` and then again on call to `cache_wr()`.

A more efficient means of accessing the memory are provided by the *unsafe interface*, at the expense of a slightly higher programming complexity. The double address translation issue is avoided because in this case the programmer obtains pointer to the LS data via a call to (possibly blocking) `cache_rw()`. The pointer can be used to read or modify the data, as shown in the following snippet:

```
i_ptr = cache_rw(mycache, eaptr);
*i_ptr = *i_ptr + 42;
```

Under the unsafe interface, it is possible to steer clear of wasting time in a blocking call to `cache_rw()` by using its asynchronous version `cache_touch()` and `cache_wait()` and continuing computation on data already present. This is an important and useful feature of the software cache.

There is a downside to using the unsafe interface too. It is necessary to guarantee that a new (synchronous or asynchronous) request for an item will not evict data the program holds a pointer to. The cache system itself has no information on how to find out about such pointers. Therefore, it is the programmer's responsibility to *lock* any pointers she is still planning to use in the future by calling `cache_lock()` on them.

The software cache is implemented via a set of macros and inlined C functions. The programmer "instantiates" a cache by defining its parameters using `#define` directive and including the relevant API header file. To have the program utilize several caches at the same time, the procedure can be repeated This explains why the cache access functions need to be passed the cache name as one of their parameters.

ALF

ALF [15] was already mentioned in Section 2.8. The reason why we are going to talk about it here is that it presents its own way of dealing with the memory access problem.

The idea behind ALF is that the programmer specifies data and tasks that need to be run over the data. Each task is partitioned into work blocks. The programmer is responsible for defining functions doing that, that is one that prepares the input data for a given work block and one that processes the output data after the work block is finished. This process is called *data partitioning*.

The ALF runtime examines the tasks and their dependencies and decides on optimal strategy for work block scheduling. Before the computation kernel of each work block is executed, the corresponding data partitioning function is called. The computation kernel then process the data and finally the output data partitioning function is called.

ALF supports partitioning functions defined to run on either PPE or SPE[1]. Because effectively utilizing the accelerator memory is one of the ALF's design concerns, all the partitioned input data has to be stored in a physically contiguous buffer on the local store. The way the input/output data partitioning functions look therefore actually strongly resembles the scatter/gather DMA lists discussed in Section 3.2. ALF provides abstractions for the programmer to specify such transfers comfortably.

Before ALF schedules a task to run, it examines the programmer's specifications on the buffers used by its work blocks. If certain criteria are met, ALF uses double buffering when copying the data between the main memory and the LS.

An important point to highlight about ALF is that by letting the client partition the work blocks and then sending them to the SPEs automatically gets the programmer rid of mechanically repeating boiler plate buffering code and focus on the things specific to the problem. The repetitive code is provided by well-optimized ALF runtime libraries instead.

Using the Single Source Compiler

While the early Cell BE developers using the GCC compiler have been dependent mainly on hand crafted memory access methods, IBM has been extensively researching possibilities to avoid pains of such approach. Their advances up to 2006 are summarized in [8]. It

[1]With those running on PPE bearing performance problems resembling those of PPE data processing functions talked about in Section 3.3.

is not known how many of the schemes and optimizations presented in the paper has been really implemented in the IBM XL/C compiler. For the sake of completeness however, this section will more succinctly present the main ideas of their approach. It can be assumed that the XL/C compiler will catch up sooner or later.

Because the single source compiler does not let the user specify any memory transfers explicitly, the presented solution is built around the compiler's ability to analyze the accelerator source code and determine possible optimizations for every variable referenced there. Variables that are not shared among SPEs or otherwise used for communication are directly allocated at the local store. Access to the remaining variables needs to be propagated between the local store and the main memory and back. This is done on two levels. The basic level is implemented using *compiler-controlled software cache*. Besides being transparent, this cache is basically a more sophisticated version of the one offered by the Example library's safe interface. The particular implementation from the paper is a 4-way associative cache. Its great advantage is that all four ways are searched simultaneously, exploiting the SIMD parallelism of the SPE.

Even though the code for triggering the cache lookup and eventually the cache miss handler is optimized on many different levels throughout the compilation, those calls are still quite expensive. It is therefore important to reduce the cache usage whenever possible with the second level of optimizations that allow multiple elements to be fetched in a single operation.

The typical scenario permitting the use of those optimizations is the case of regular-stride access to elements of a large array. To increase data locality, loop-restructuring techniques such as *loop tiling* [37] are employed. By combining loop restructuring with explicit DMA transfers, the compiler can in certain cases overlap computation and data transfers.

Substantial compile-time analysis is required to be able to perform the discussed optimizations and the paper mentions a few more, not yet well explored possibilities. In the meantime, the team seems to have refocused on improving the caching schemes as described in their 2008 paper about *hybrid access-specific* software caching [9].

The caching scheme proposed depends on the compiler to distinguish between two main data access patterns:

High-locality access, using high-locality cache structure.

Irregular access, using transactional cache.

There is a higher chance that several operations with high locality will target the same cache line, therefore the cache lines are longer for this cache. For the same reasons, lines of this cache can be pinned to prevent them from eviction. Often the compiler can even deduce that several consecutive operations will be a cache hit and can entirely remove all the cache overhead and reference the memory of the given cache line directly instead.

On the other hand, it is expected that the irregular access will cache-miss most of the time. The transactional cache is designed to deliver very low hit and miss overhead and enable overlapped computation and communication.

Performance boost from using these caching structures is enforced by suiting code transformations. The experiments in [9] would indicate that the new hybrid access-specific caches offer significant improvements over the "traditional" caches in [8].

3.5 Pros and Cons

Before the new memory access solution will be presented, let us take a look again at all the existing solutions and contrast their features.

Clearly the best performance for the greatest programming effort is what the manual method offers. Besides the extra time spent designing and coding the transfer methods, the final program is more susceptible to bugs and so might incur further costs during the testing and maintenance period of its life cycle. Further, such code is not portable to other architectures.

Some of the programming discomfort can be tackled by using software cache. One such scheme is provided in the SDK, section "SPU Software Cache" introduces its two interfaces. Being synchronous and therefore slow for accessing memory in programs with a poor hit ratio is the main drawback of the safe interface. Even in the case of a good hit ratio is the performance affected by ubiquitous cache lookups.

The unsafe interface has none of those two caveats, however new programming tasks arise. First, the asynchronous features are manually controlled. Second, to prevent the cache from evicting useful data from parts of the LS, all data that is to be used later has to be manually pinned by the programmer. An error while carrying out either of the duties can make irregularly appearing bugs emerge.

From the programmers point of view, even though the DMA transfer internals are now hidden away from her code and in the safe interface's case it is possible to use only EA pointers, access to the main memory is still far from being transparent. It can be concluded that the software cache from the Example library is mainly useful as a first pass implementation of code being ported to the SPE.

There are many pros of using ALF (see "ALF" on page 37). A well developed ALF application delivers excellent performance benefiting from the well tuned ALF runtime. What is more, it scales well when running on a system with multiple processors—the runtime just spreads the tasks over all the available accelerators. Another advantage is that the supported platform is not just Cell BE, but in general any multicore system, even a heterogeneous one. The main problem related to ALF is that its framework nature imposes a certain structure of the program under development and is thus not suitable for all kinds of applications.

The single source compiler seems to be IBM's most promising hope for gaining support of developer masses. Its solid advantage is that memory access is entirely transparent just like when programming an x86 computer. Another positive is code portability to other platforms.

Unfortunately, the weaknesses are also many, all of them stemming from the fact that success of the used memory access strategies will above all depend on the compiler's shrewdness in analyzing the source code. If it fails to find the optimum strategy (for example by choosing synchronous caching where overlapping communication and computation is possible), the programmer has got very little control over the situation, besides restructuring her code slightly and hoping that the compiler will pick up on that! Indeed a problem of program analysis is that a large speed-up gained by optimization may go away after a seemingly minor change to the program, because the analysis is no longer able to conclude that the considered optimization is still correct [31].

Even the currently best possible analysis can not in certain situations compete with the manual optimization approach. The research team behind the technology admits that:

> General compiler-based solutions are often difficult to deploy due to the lack of sufficient information at compile time to generate correct and efficient code [9].

Also notice that if the programmer can optimize the memory access *based on a statistical knowledge of the data* (for instance, "in a typical dataset, 92% of memory references are to the last 128 items of the array"), it is *not algorithmically possible* to have a compiler automatically achieve the same result during the compile time.

For all those cons of the imperfect solutions today, let us venture on finding a language that allows the developer to *choose the memory access strategy* while having the compiler do as much work as possible to increase the programming comfort.

3.6 The Concept of a Memory Access Language

One can say that the XL/C compiler and the GCC compiler are on opposite sides of the SPE memory access automation spectrum, XL/C providing completely transparent access with no programmer input and GCC leaving everything up to the programmer. This thesis presents a previously non-existent solution to the problem, for it proposes a new language that offers both the comfort of the single source compiler and the degree of control of GCC.

The main design principle of the new language is allowing the programmer to specify everything important and then let the compiler generate everything else. In practice this means that it should be sufficient for given array `arr` within SPE code to declare what kind of access we wish to use for it (e.g. caching or buffering), the sequence of subscripted indexes (e.g. starting at the end of array going towards the beginning and subscripting every item) and optionally other constrains (e.g. read-only access). Using this information, the compiler will take care of the DMA access and the LS management behind the scenes and the programmer can, given an index `i`, subscript the array in the standard C notation. To get a concrete idea of what is meant here, consider the following code on SPE:

```
long sum(int* arr, unsigned size)
{
    long s = 0;
    for (unsigned i = size - 1; i >= 0; --i)
        // defining the access strategy to arr
        : DoubleBuffering(array: arr,
            startAt: size - 1,
            step: -1,
            readOnly: true)
    {
        s += arr[i];
    }
    return s;
}
```

The declaration in the for loop will make the compiler substitute any array subscription in the block with a reference to the local store buffer and insert double buffering code at the correct places that asynchronously keeps this buffer filled with parts of the array that are going to be accessed next. Later in this chapter, after the new language is properly introduced, an entire sample program is given, discussed and compared to an equivalent C implementation.

Access to any variable would always be entirely transparent, but once the programmer of an SPE function declares a *contract on the access* to a certain variable, it becomes efficient too. Double buffering will often be possible since many SPE programs go through an array from the start to the end, only changing the index slowly, In other cases it might be a cache mechanism with parameters optimized for the specific occasion (associativity, size, write back).

When no access method is specified for a particular variable, the compiler can always fall back on a synchronous cache or similar. An important feature of such organization is that it can benefit from improvements in the compiler's analysis, impact of which might be that the user will not have to specify an access strategy at all in certain cases but obtain a good execution speed still.

It also makes it possible for the developer to sketch her core algorithm quickly and with no extra programming effort. Once done with the debugging, she can focus on the performance critical parts of the code and increase the speed by suggesting optimal memory access strategies. On the other hand it is also possible to do the opposite for rarely executed code and for example decrease its cache size to trade-off worse execution speed for saved local store space. Doing so will be relatively cheap labor-wise, unlike in the case of manual memory access when communication and computation are tightly entangled in the code.

On the ideological level, using such a language would not violate the manual control over memory hierarchy principle. If Cell BE's design decision of exposing the local stores to the programmer is a revolutionary idea, it would be somewhat reactionary to hide it again at the compiler level, as XL/C does.

3.7 New Extensions

To be able to experiment with the notions from Section 3.6, a simple imperative language *Dali* (Data Access Language Interface) with memory accessing extensions was designed and partially implemented.

Dali provides the same basic control structures like other imperative programming languages, including for example conditional branching or a loop. Several simple data types including integral and floating-point numbers and an array data type are supported. A *Dali* program has a structure similar to a C program with functions acting as the basic blocks of execution. Since one of the goals was to be able to specify the entire program in one source file, *Dali* needs to provide mechanisms for designating the code that should run on the accelerators. This is supported through the `spe` function modifier. Such a function can be called from a function running on the PPE, with similar semantics to calling any other function, that is the execution of the calling function is suspended, the called function is passed the parameters, it runs and returns a value that is communicated back to the calling function so it can continue computation. *Dali* can be extended with high level parallel programming abstractions like asynchronous calls and futures seen in the X10 programming language [3] and others, although none of these features are considered or implemented.

A major extensions to other imperative languages are the *accessor declaration* and the *accessor application*[2]. Accessor declaration takes place outside of any function and is a named

[2]The word "accessor" used here has a different meaning than a mutator method from the C# language.

definition of how a data transfer for a certain data structure should be handled. In an SPE function then, the accessor can be applied to an effective address pointer. By doing so the programmer declares that she is aware of all the possible limitations of using the accessor and declares that the code respects them. The compiler then generates code relevant to the chosen accessing strategy whenever the effective address is used for dereference. For the programmer this has the desired effect of a complete transparency of access. For the compiler it means it only has to decide on how to implement this access, but does not need to speculate what access is the best.

To demonstrate these extensions in practice, Section 3.10 shows and discusses a simple *Dali* program. First however we will take a more formal look on the syntax and semantics of a *Dali* program.

3.8 Syntax and Semantics of *Dali*

A *Dali* program is a list of variable definitions, constant definitions, function definitions and accessor declarations:

```
program ::=
    (variable_definition |
    const_definition |
    function_definition |
    accessor_declaration)*
```

Every definition, declaration or statement in the source has to be terminated by a semicolon. Variables are defined using the traditional C syntax, which is by specifying their type and their name. Only integer constants are allowed and they are defined similarly.

```
variable_definition ::=
    (type id ';' |
    type id '[' integer ']') ';'
const_definition ::=
    'const' id '=' integer ';'
```

The second alternative in variable definition creates an *array* of the specified size. Array is a homogeneous data structure stored contiguously in the memory. Variables declared at the top level (outside of any function) are *globally declared variables*. These variables are allocated in the main memory and can only be referenced within PPE functions (see below). Constants are accessible from any location. The type of a variable is simply the name of the type or a pointer to the type denoted by the name of the type followed by a star *:

```
type ::=
    ('float' |
    'unsigned' |
    'int' |
    'unsigned long long' |
    'char' |
    'float' |
    'void') ['*']
```

Functions are defined by specifying their return type, their name and their arguments, followed by the function body as a block:

```
function_definition ::=
    [function_modifier] type id
    '(' [function_argument (',' function_argument)*] ')'
    block
function_argument ::=
    type id
function_modifier ::=
    'spe'
block ::=
    '{' variable_definition* statement* '}'
```

Functions are the basic units of program structure. When one place in a program calls a function, the given parameters are passed to it and the execution is transferred to the function entry point. Function is executing code within its body until a *return statement* is encountered upon which the control is transferred back to the caller with an optional return value. By default, all functions are running on the PPE code. When the program starts, function named `main` is called. Hence the entry point of `main` is also an entry point of the program. There is only one function modifier, `spe`. This makes the corresponding function run on one of the Cell BE's accelerators. We say that such function is an SPE function. All other functions are PPE functions.

The remaining top-level structure to be explained is the *accessor declaration*. As the syntax suggests, declaring an accessor is a matter of giving a name to a particular access method configuration.

```
accessor_declaration ::=
    'accessor' id '{'
    [accessor_option ';' (accessor_option ';')*] '}'
accessor_option ::=
    ('first_element' integer) |
    ('array_size' integer) |
    ('array_width' integer) |
    ('write' boolean) |
    ('prefetch_extent' integer ) |
    ('access_method' access_method)
```

All the options with an exception of the access method are optional. The initial version of *Dali* focuses mainly on array manipulation therefore most of the access options are quite specific to accessing arrays. The *first element* option specifies the first element that will be fetched when the accessor is used with the array. Because it is often not possible to decide this at the compile time, this option can also be passed to the accessor at the place of its application (see below) or even deduced automatically by an advanced compiler. The *array size* option specifies the number of elements of an array and *array width* specifies number of elements in a single row for two dimensional arrays (matrices). Like the first element option, these options can also be passed in dynamically. (Alternatively, a next version of the language could allow Java-style arrays that always remember their dimensions. We discuss this possibility in Section 3.11.) *Write* indicates that the given accessor can also be used on the left side of an assignment and that elements changed in this way should be written back from the local store to the main memory. The *prefetch extent* option specifies how aggressively should data be prefetched ahead. Notice that it depends on the selected access method whether and how the compiler interprets the information specified through of the options above. Specifying the method itself is mandatory:

```
access_method ::=
        'naive' |
        'linear_stencil' |
        'double_buffer' |
        'block2d' |
        'binary_search'
```

The method declares the general strategy that is to be used for any array the declared accessor will be applied to. Discussion about the different access methods is given in Section 3.9. The access method listed here are only those that were actually implemented for this work. Declaring an accessor has no semantic effect per se. How an accessor is declared only becomes important when the accessor is applied.

The statements in *Dali* are the if statement, the expression, the return statement, the for loop, the assignment and the block. Since a block can enclose a statement we see that blocks can be nested arbitrarily.

```
statement ::=
    (if_statement |
    expression ';' |
    assignment |
    return_statement |
    for_loop |
    block)
```

The *if statement* transfers the control to one of the two blocks depending on whether the expression evaluates to zero or a nonzero value.

```
if_statement ::=
    'if' '(' expression ')'
        block
    'else'
        block
```

Expressions can consist of constants, variable references, array subscriptions, arithmetic operations and function calls:

```
expression ::=
    constant |
    id |
    id '[' expression ']' |
    expression ('+' | '-' | '*' | '/') expression |
    id '(' [expression (',' expression)*] ')'
```

The only semantic way *Dali*'s expressions differ from expressions in C-like imperative languages is array subscriptions. If an array subscription appears as a part of an expression in the PPE code, the compiler uses array's base address offset by the index to retrieve the requested element. On SPE, however, the compiler first checks if an accessor has been applied to the array variable in the local scope. If so, the relevant accessor code is generated at the place of the subscription to retrieve the requested element. Otherwise the compiler falls back to the naive accessor to retrieve the element.

Assignment is similar to the expression statement, but the value the expression evaluates to is not discarded, instead it is assigned to a variable.

```
assignment ::=
    (id | id '[' expression ']') = expression ';'
```

Again, for assignments to an array element on PPE, the value is simply written to the memory at the relevant offset. On SPE, applied or naive accessor is used.

The *return statement* causes the function it appears in to terminate and pass the return value back to the caller.

```
return_statement ::=
    'return' expression ';'
```

Finally, the *for loop* is a construct that allows code to be repeatedly executed.

```
for_loop ::=
    'for' '(' expression ';' expression ';' expression ';' ')
    [':' accessor_application]
    block
```

The first expression in the for loop is executed before the loop starts. The code in the loop's block is executed repeatedly. After each iteration the last expression is executed and then the second expression is evaluated. If the yielded value is nonzero, the block is executed again, otherwise the loop stops. Optionally, the loop can include a list of accessor applications:

```
accessor_application ::=
    id '(' id ',' expression ',' expression ')'
    [',' accessor_application]
```

The name before the parenthesis references a declared accessor, the name inside the parentheses must be an array variable. The two expressions have meaning depending on the access method, usually the first expression is taken to be the size of the array, the second expression advices what index is the prefetching going to start from. Some access methods might ignore those parameters completely. If an accessor is used over an array variable in this way, we say that *the accessor is applied on the variable*. For SPE function this means that for every subscription of the array that occurs within the scope of the loop a special code is generated for handling the DMA transfers between the main memory and the local store. This transparently changes access to the main memory to access to the local store. The exact mechanics of when and how the data is transferred between the two memories depends on the used type of an accessor and is described in Section 3.9. The type of the used accessor is found in the relevant accessor declaration as was shown earlier in this section. All options specified there are also respected by the compiler when it generates the accessor code.

In case of nested for loops that both apply an accessor for the same variable, lexical scoping applies. This means that the application from the closest for loop enclosing the particular expression is used. Notice that in a PPE function, accessor applications are ignored.

3.9 Accessor types

What kinds of accessors can be used in *Dali* programs? This section will describe those that would probably become the most commonly used. In theory however there are no limits on the design of new accessors not mentioned here.

Below is a summary of accessing methods for array types. Note that one program can easily declare two accessors using the same method, but having other method-dependent options different.

Naive Accessor is the most trivial of all the accessors. It synchronously downloads given data item at a place of its reference or synchronously stores it back to the main memory when it is modified. Present for debugging purposes when the programmer suspects memory access problems.

Permanent Accessor immediately downloads the data items entrusted to it and keeps them on the local store until the end of its scope when it optionally writes the data back. It can be a performance booster, especially if all the data fits the limit of one DMA transfer (16 KB) since then no other special treatment needs to be done as all the references are simply pointed to the local store. It is especially useful for smaller arrays that are repeatedly accessed at random positions, like for instance lookup tables.

Multibuffering Accessor is based on techniques similar to double buffering, described in Section 3.3. Multibuffering is only useful for an algorithm with almost absolutely regular access patterns exhibiting high data locality. When this condition is met, multibuffering is almost always the best choice for arrays that do not fit in one DMA transfer because the computation can start immediately after the first chunk of data arrives at the local store. The remaining chunks are copied asynchronously, allowing computation/communication overlapping. Possible parameters of a multibuffering accessor are size and number of the buffers and size of DMA chunks.

Caching Accessor is an accessor providing a software cache (see "SPU Software Cache" earlier in this chapter). It is particularly handy for random access patterns in places of the program that are not performance critical.

Speculative Accessor is similar to the caching accessor, but it has some extra information about the data provided by the programmer. Based on that information it speculatively prefetches memory areas or makes decisions about which cache lines to evict first and which to pin. It is an interesting alternative for seemingly random data access patterns where double buffering methods are not possible.

Matrix Accessor handles effectively access to arrays representing matrices. It is an example of accessor that is more tightly bound to the data structure it is used with. It is useful in scenarios where access to the matrix is regular, yet traditional double buffering can not be applied, such as column-wise access in matrix stored in row-major order. To be able to function matrix accessor needs information about how the matrix is stored in the memory (row-major, column-major, blocks) and its dimensions.

One can also wonder about how data-structures other than arrays can be handled. Dynamic data structures are spread around the heap with no data locality, therefore buffering methods are useless in this case. Solutions for traversing these data types however exist. One of them might be an accessor that given the last accessed node n of the structure automatically prefetches nodes reachable from n in certain number of steps.

Another solution involves accessors that operate on both the PPE and the SPE. The accessor on the PPE can for example preprocess the dynamic data structure into a format that can be accessed as a contiguous sequence. Alternatively, it can simply collect address of the nodes in the memory and send that to the SPE. The accessor there can then gather the structure using DMA list commands.

Accessors running on the PPE have a major issue with scalability for if the programmer parallelizes the computation over several SPEs the PPE having to preprocess data for each of them might not be able to keep up. This problem does not plague only *Dali*'s accessors, but also any other SPE program that leaves some work to the PPE.

At this moment enough has been shown to present an actual program that uses some of the accessors from this section.

3.10 Sample *Dali* Program

The following program uses *Dali* to implement an algorithm that takes a matrix on the input and returns sums of elements in each of its rows as the output, that is the output is a one-dimensional array. While the matrix is declared in the main program, the computation is offloaded to an SPE.

```
const MSIZE = 32;
accessor MatrixRowWise
{
    access_method(double_buffer);
    buffer_size(2048);
    direction(increasing);
}
accessor SmallArray
{
    access_method(permanent);
    write(true);
}
spe void compute_sums(float* in, float* out)
{
    unsigned i;
    unsigned j;
    // this is where the accessors are applied
    for (i = 0; i < MSIZE; i++)
        : MatrixRowWise(in, MSIZE * MSIZE, 0),
          SmallArray(out, MSIZE, 0);
    {
        out[i] = 0;
        for (j = 0; j < MSIZE; j++)
        {
            out[i] += in[i * MSIZE + j];
        }
    }
}

int main ()
{
    float matrix[MSIZE * MSIZE];
    float sums[MSIZE]
    read(matrix, stdin);
    compute_sums(matrix, sums);
    print("results: ");
    print_array(sums, MSIZE);
}
```

The accessors are declared at the beginning of the listing. The first accessor will handle the input matrix, employing double buffering to do that. It then sets the size of its buffers and declares that the array is traversed in the direction of increasing index, that is from lower indexes to higher.

Because the second accessor will be used with the results array, permanent accessors can be used. The programmer can reason here that the results array is always going to be rather small (square root the size of a square input matrix). The `write` option tells that the programmer wants the changed data to be written back to the main memory.

The syntax of accessor applications is borrowed from the C++ constructor initialization list, but in *Dali* it is applied to blocks. What the two particular applications from the sample code achieve is:

1. Generate code that accesses the `in` array using double buffering. Its size is `MSIZE`^{2} and the first item to be accessed is at index 0. The generated code starts fetching the first two buffers before the loop is entered and makes sure the transfer of the first buffer is completed before the array is referenced for the first time. It changes all references to the array so they point to the buffers in the local store. Finally, it manages swapping the buffers and starting new DMA transfers as necessary.

2. Before the loop is entered, transfer the entire `out` array into the local store and direct all references to it to the new location. Upon exiting the loop, write the updated array back into the main memory.

An important point demonstrated in this code is that in order to keep the accessor declarations general and not static, some information has to be passed to them from their place of application. It is of course better if the passed values are constants because then the compiler can perform more optimizations, e.g., if it knows the size of the array it can compute the optimum size of the local store buffers etc.

A manually written C equivalent of the summing program is listed on pages 77–80. The code was placed in an appendix rather than here for space reasons. Only the more interesting parts of it are mentioned here and commented on. While the entire *Dali* implementation takes 39 lines of code in a single file, the equivalent C implementation has 140 lines in total, spread across two source files and one header file.

The first place where a lot of boilerplate code adds many lines to the C implementation is the SPE function call. To pass input and output parameters for the SPE, programmer typically defines a structure with correct padding and alignment:

```
typedef struct
{
    float* matrix;
    float* sums;
    char   _padding[8];
} argsum_t __attribute__((aligned(16)));
```

This structure is instantiated and initialized with the respective parameters. To create an SPE program, several API calls have to happen. The first one initializes the SPE context—a data type used for handling the SPE programs (see [18] for a description of its input parameters).

```
spe_context_ptr_t speContext;
speContext = spe_context_create(0,NULL);
```

Next, this context has to be given a pointer to the code we want to run on the SPE:

```
spe_program_load(speContext, &spu_sum);
```

Finally, the program can be started using another call to the Cell BE runtime:

```
unsigned entry = SPE_DEFAULT_ENTRY;
spe_context_run(speContext, &entry, 0, spuArgsp, NULL, NULL);
```

Notice that typically we do the handling of the SPE program from the PPE program within a separate thread to allow the two programs execute concurrently, which further increases the source size and complexity.

The first task of the newly started SPE program is obtaining its input and output parameters. Because the program only has a pointer to the parameter structure in the main memory, it needs to place a DMA request and synchronously wait for its completion using a previously reserved group ID `tag`:

```
spu_mfcdma32((void *)(&args),
    (unsigned int)argp,
    sizeof(argsum_t),
    tag[0], MFC_GET_CMD);
```

```
spu_writech(MFC_WrTagMask, 1 << tag);
spu_mfcstat(MFC_TAG_UPDATE_ALL);
```

Knowing the locations of the input and output arrays, the SPE program can finally start fetching the data and processing them. Double buffering is used and therefore a pair of sufficiently big buffers and a pair of DMA tag groups is defined together with a buffer for the results:

```
unsigned tag[2];
volatile float buffer[2][ROWS_IN_BUF * SIZE] __attribute__((aligned(128)));
volatile float sums_buffer[SIZE] __attribute__((aligned(128)));

unsigned buffer_index = 0,
    next_index = 1;
unsigned base_row = 0;
unsigned current_row;
```

The two index variables keep track of the index of the buffer that is currently processed and the index of the buffer that will be processed next. It is up to the programmer to initialize and update these two values accordingly. The other two integer variables maintain the index of the first matrix row contained in the currently processed buffer and the index of the row that is currently processed (within the current buffer). These values also have to be kept up-to-date painstakingly because the DMA operations derive the memory addresses from them.

The computation starts by initiating a DMA command to fetch the first chunk of data:

```
spu_mfcdma32((void *)(&buffer[0]),
    (unsigned int)args.matrix,
    BUFFER_LENGTH,
    tag[0],
    MFC_GET_CMD);
```

The main program loop works as the double buffering scheme from Section 3.3 dictates, that is it initializes fetch of the next part, waits for the transfer into the current part to finish and then processes the current buffer:

```
for (;base_row < SIZE; base_row += ROWS_IN_BUF)
{
    float* nextRowAddress = args.matrix + (base_row + ROWS_IN_BUF) * SIZE;
    spu_mfcdma32((void *)(&buffer[next_index]),
        (unsigned int)nextRowAddress,
        BUFFER_LENGTH, tag[next_index],
        MFC_GET_CMD);
    spu_writech(MFC_WrTagMask, 1 << tag[buffer_index]);
    spu_mfcstat(MFC_TAG_UPDATE_ALL);
    for (current_row = base_row; current_row < base_row + ROWS_IN_BUF;
        ++current_row)
    {
        unsigned i;
        for (i = 0; i < SIZE; ++i)
        {
            sums_buffer[current_row] +=
                buffer[buffer_index][(current_row - base_row) * SIZE + i];
        }
    }
    buffer_index = next_index;
    next_index ^= 1;
}
```

When the loop is finished, the result array has to be uploaded back to the main memory:

```
spu_mfcdma32((void *)(sums_buffer),
    (unsigned int)args.sums,
    sizeof(float) * SIZE,
    tag[0],
    MFC_PUT_CMD);
spu_writech(MFC_WrTagMask, 1 << tag[0]);
spu_mfcstat(MFC_TAG_UPDATE_ALL);
```

Compare the last few code listings with the loop in the *Dali* example. What it shows is that by making the main memory access transparent the new language constructs are eliminating a lot of repetitive boilerplate code, making the interesting part of the program itself more visible.

3.11 Other Considerations

Dali is intended to be an experimental language, focusing purely on the memory access problem. That is why it has no support for concurrent computation or vector data types, the two other essential traits of a language that could exploit the potential of the Cell BE architecture. If on the other hand the data memory access concepts this language represents prove practical, they can be built into a language that already has those features, or *Dali* can be extended with them.

The discussion about the language in this chapter and its implementation described in the next chapter is concentrating mainly on the class of SPE programs that process large arrays inside loops. This is the case of the most important SPE programs and covers many large classes of problems, e.g., matrix computations, computer graphics, scientific simulations and others. The accessor concept is however general and Section 3.9 talks about using accessors with other data types then array.

An interesting extension could be arrays that always know their dimensions. In C, an array is simply a pointer to a contiguous memory area. Once this pointer starts to be passed around to functions, it is impossible to determine length of the array, except for manually maintaining the value in a separate variable. Most modern languages however has a mechanism in place to retrieve array sizes at any point (Java's `length` for example). Cyclone [10] is a language dialect of C that keeps track of array sizes during the runtime. Besides the reason of a better programming comfort, knowing the exact length of an array would allow a *Dali* accessor to safely prefetch parts of the array without risking memory violations.

A tempting idea is making the accessors uniquely belong to a data type similarly to how a method belongs to a class in an object oriented language. Whenever such a data type would be accessed from the SPE, the compiler would know which access to invoke. There are however two reasons why another approach was chosen. First is the reuse pattern. It is clear that the same accessor is going to be reused by more than one instance of a given data type. Let us say we are performing matrix addition over two matrices that are stored in the memory in row-major order. They both will use the same access method, probably some variation of prefetching buffers from the main memory just in the order they are stored. This would mean that an access method is indeed bound to a type. Now suppose we are doing matrix multiplication using the algorithm based on the matrix product definition. In this case, the left operand still uses the same simple form of buffering, whereas the right operand is now accessed column-wise and so we have to use a different access strategy reflecting that!

This shows that in general more than one accessor can be needed by a single data type, depending on the algorithm the data type appears in. We might instead try to define several accessing methods as part of the data type and then choose among them at the place of accessor application, just like when calling a method in an object oriented language. There is however the second reason why we can not quite make this case either. That is to say, accessors do not simply cause some code to be executed at the place they are instantiated, they have broader semantics than that. To keep access to arrays transparent, for example, some memory accessing code has to be inserted everywhere an array is subscripted. Accessors generally can induce the compiler to make program transformations, as Section 3.12 illustrates.

3.12 Loop Transformations with Accessors

It is a sad fact that the SPE programmer has much more to worry about than the memory hierarchy. One of her concerns is producing code with as little branching instructions as possible, which means that loops should be unrolled and conditional statements avoided at all cost.

This is unfortunate news for implementing *Dali*'s accessor mechanisms. To see this, imagine how an access to an array element using double buffering accessor will be implemented. The *Dali* code summing all the elements of an array might look similar to the following:

```
int sum;
for (int i = 0; i < LENGTH; i++)
    : DoubleBuffering(a, LENGTH, 0)
{
    sum += a[i]
}
```

What target code is generated for the statement in the loop? Most of the time, the value of `a[i]` is already in the local memory (active buffer), this is what double buffering ensures. However, there are values of `i` for which `a[i]` is the first element of the buffer that is just being fetched from the main memory (non-active buffer). Because the code inserted in place of `a[i]` by the compiler is always the same, what it does is:

1. See whether `i` is an index that maps to the active buffer and if it does continue with Step 3, else continue with Step 2.
2. Start transfer into the active buffer and wait until the transfer into the non-active buffer is finished. The non-active buffer now becomes active and the other way round. Continue with Step 3.
3. Return the requested item from the active buffer.

The Step 1 contains conditionally executed code and an `if` statement can not be avoided there. This is frustrating since conditional branching within a loop is likely to severely degrade performance.

Fortunately it shows that, at least in simple cases, the `for` loop can be transformed so a smarter version of the double buffering accessor can be used. The following steps describes how is such transformation done, assuming the total length of the array is divisible by the number of elements that fit a buffer, `BUF_ELEMS`.

1. Increase the stepping of `i` from 1 to `BUF_ELEMS`.
2. At the start of the loop's body, insert code that starts DMA transfer to the non-active buffer and ensures that the active buffer is ready.
3. Insert another loop with control variable `j` starting at `i`, going on while less than `i+BUF_ELEMS` and being incremented by one. The body of this loop is the body of the original loop with all occurrences of `i` replaced by `j`.
4. At the end of the loop's body, insert code that swaps pointers to the active and the non-active buffer.

According to the procedure, the original example can then be transformed by the compiler like this:

```
int sum;
fetch(B0, 0);
for (int i = 0; i < LENGTH; i += BUF_ELEMS)
{
    //dma operations:
    fetch(B1, i + BUF_ELEMS);
    wait(B0);
    for (j = i; j < i + BUF_ELEMS; j++)
    {
        sum += B0[j];
    }
    swap(B0, B1);
}
```

Coincidentally, this scheme is similar to the way manual implementations of multibuffering are done [22, pp. 111–112].

The technique was demonstrated in this section only to show that the branching issue can be avoided and is currently not implemented. It would however be possible to do so as part of the future work (Section 6.1). In practice, this code needs to be a little more complex to also handle array sizes not divisible by the number of elements in the buffer. The original loop is split into two loops, the first one to handle the largest possible part of the array with size divisible by `BUF_ELEMS` and the second one for the remaining elements.

Implementation

To see what sorts of practical problems and what performance can be expected from a language-based solution of memory access on Cell BE accelerators, a subset of the *Dali* language has been implemented as a part of this work.

This chapter discusses the technical matters of the implementation, experiments with it are covered in Chapter 5.

4.1 Source-to-source Translator

Two approaches for having the extensions running with a Cell BE program were considered. The first one was extending any available C compiler for Cell. In fact, this gives us only a single option, GCC, since the IBM single-source compiler is proprietary and its source code is not publicly available.

There are plenty of arguments talking against GCC at this point. It is evident that many layers of the compiler would need modifications, including the grammar due to the language extensions and extensive tree manipulations due to the non-local meaning of accessor application. GCC is a notoriously complex system and learning its internals would be costly resource-wise. The main problem, however, is that it is still only a C compiler, whereas *Dali* offers more abstract language constructs than C and when the implementation started it was not possible to tell how vast changes would be necessary and what features will be included.

Implementing a source-to-source translator for a small domain-specific language was a more viable idea. Initial subset of the *Dali* language was selected for implementation. Next a translator accepting programs in this language and producing C/C++ code for the GCC toolchain was built.

Advantages of this approach are full control of the accepted language and freedom to extend it in any way. Further, using the right tools building such a translator from ground does not necessary need to be more complicated than extending GCC directly, as Section 4.3 reveals later, but first let us take a look at what the implemented subset of the language is.

4.2 The Extent of the Implementation

Generally, the syntax and semantics specification from Section 3.8 is fully supported. The programmer in the first *Dali* implementation writes the program similarly to a C program, that is as a set of functions. Global numeric constants and global or local variables of integral or floating point types can be declared. One dimensional arrays over all those types are also supported. Accessors are declared as a top-level structure, with the word `accessor` followed by accessors name and accessor's parameters inside braces. Within functions, one looping construct (`for`) and one control flow construct (`if-else`) are available.

Currently accessors can only be applied in a for loop. This decision has been made based on the notion that performance-critical parts of program are most often concentrated in loops. Besides, the access patterns that the accessors exploit are hard to spot in sequences outside loops. This does not mean that the SPE code outside of any for loop can not access the main memory. It can—compiler uses naive accessor as a default in those cases, similarly to accessing a variable within a loop, but with no accessor applied to it. The lexical scoping for accessors has been implemented. Not all accessor options from Section 3.8 are supported.

Accessors are applied by adding

```
: AccessorName(variable, size, index)
```

after the declaration part of a for loop. The three parameters passed to the accessor are:

`variable` is the name of the array the accessor should be applied to.

`size` is the size of the array; For accessors doing prefetching this is important so they know "how far" can they go with it.

`index` is present for synchronization purposes; Typically it tells the accessor what index to start prefetching the array from.

Not all those parameters are always necessary and contrary there could be accessors which might need more runtime parameters than those. A more general concept of passing runtime parameters to the accessors would be nice to have, and is actually specified by the syntax, but the current solution was chosen as a compromise that sufficed our experimental purposes. Because applying an accessor triggers generation of code and possibly also specific code optimizations, accessors are always static despite some of their parameters being set runtime. This implies that accessors can not be decided on during runtime or, for example, be assigned to a variable.

SPE functions are called like ordinary functions, with the same synchronous semantics. The function return value is not supported for SPE functions, but they can of course communicate information back to the PPE by modifying arrays they are passed. A separate pointer types for the main memory and the local store were considered too, but were not implemented in the end. Therefore all pointers in SPE code are considered as pointing to the main memory. This prevents the programmer from allocating and using her own dynamic data structures in the SPE code. Again, this has proved sufficient for our experimental purposes. Extending the *Dali* translator to support syntax for local storage pointers would be straightforward.

Besides creating separate C++ source and header files with the PPE and the SPE code, the translator also generates a Makefile so the programmer can build the final Cell BE binary in a simple way.

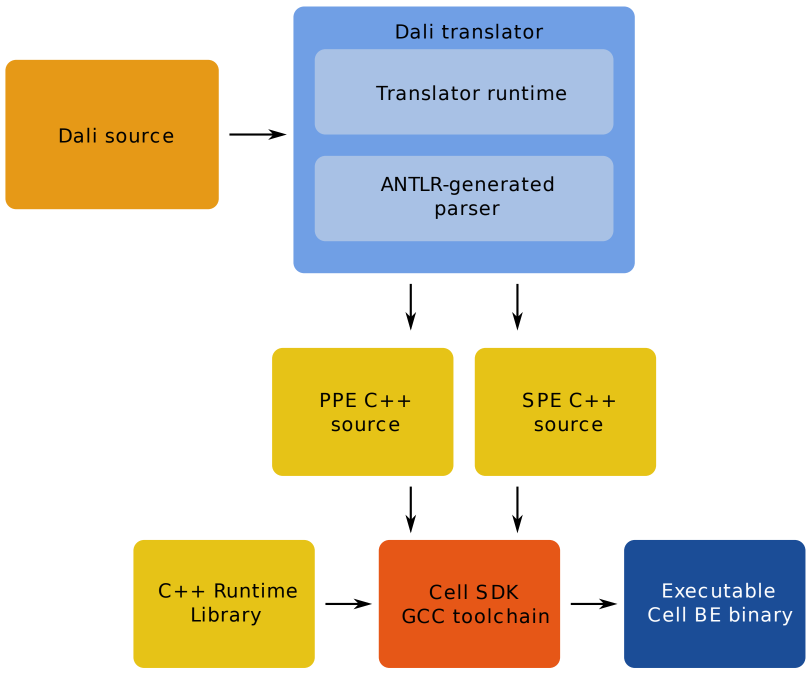

Figure 4.1: *Dali* to executable translation

4.3 Implementation Details

The core of the translator was built using ANTLR [36]. ANTLR is an LL(*) parser generator typically used for implementation of domain-specific languages (DSLs) interpreters, compilers and other translators. ANTLR is distributed under BSD License and supports several languages to generate the parser in, including Java which was used for translator implementation in this project. It also has several disadvantages, among them lacking support for AST rewriting forcing the user to copy the entire grammar per each implemented tree-rewriting pass[1], and occasional bugs. ANTLR turned out to be a good tool for the task. It is straightforward to specify a C-like language in it and the parser can be integrated with other components in a clean way. The ANTLRWorks tool [1] is also freely available to assist in debugging ANTLR grammars.

A *translator runtime*, also developed in Java, is providing structures like symbol table and output generator. The parser generated by ANTLR is depending on this runtime to pass information about the code (e.g. function and accessor declarations, scopes) between the passes. ANTLR has a built-in support for StringTemplate, a template engine enforcing strong separation between the model and the view layers of an application [35]. In combination with the runtime classes it is used to generate the target C++ code. Once the target code is generated, is can be built using the GCC toolchain supplied with the SDK (see Section 2.6).

[1]Terrence Parr, the project leader, has announced a feature dealing with the issue to be released mid-2009

As Figure 4.1 suggests, the generated code is using a C++ runtime library. This library contains code for every provided accessor, few support components the accessors share, routines for spawning SPE functions on the accelerators and other utility routines used for debugging and measuring performance. It would probably also be possible to simply inline all this code instead to the generated C++ program, but the preferred solution was using C++ templates. That way the runtime could be built and debugged separately and the critical methods of the libraries can still be hinted for inlining using the C++ `inline` keyword.

The second part of this section highlights some of the particular aspects of the implementation.

The Language Grammar Implementation

Three different grammars were defined for the translator. The second and the third one accept output of the previous grammar for their input. The first grammar is parsing the *Dali* source and generates the corresponding abstract syntax tree, AST. Like in other parsers, some of the information extracted from the source needs to be kept in a separate *symbol table*. Our symbol table keeps track of:

- global variables
- declared functions, their signatures and their SPE modifier
- accessor declarations
- accessor applications
- global variables and constants

Because output of the *Dali* translator is fed into a C compiler which performs parsing and semantic checking by its own means, the symbol table in the translator needs not maintain all the available information only that which is significant for the *Dali* specific extensions. Information on local variables, for instance, is therefore not included in the symbol table.

The second grammar in the translator rewrites the AST source tree to a form better suitable for code generation. That mainly consists of inspecting places of accessor applications and modifying nodes accordingly, as is explained below in “Accessor Table” and “Accessor Implementation”. The final grammar does not generate a tree but instead uses the template system to generate the target C++ code. At this point the AST is already in the state that makes the code generation rather straightforward.

Accessor Table

This is how the original SPE code reading and modifying a main memory array variable `a` might look like:

```
x = a[i];
a[i] = x * 2;
```

To turn this code into a code that actually performs the DMA transfers, two basic steps have to be taken by the translator. First, the context of the statement is inspected to find which accessor the programmer wanted to be used in this situation. This section discusses how it is done. Second, the statement is replaced with code that uses the accessors (and through them the local store buffers) to carry out the read or write, as will be described later.

There is a necessity to maintain lexical scoping of the accessors in for loops, as the language's semantics dictates (Section 3.8). This is handled in the second, rewriting grammar and is done by having every nested for loop push information about accessors applied at its level onto a stack of accessor applications. For every main memory variable reference the stack is inspected and the *topmost* accessor handling the current variable is selected to be used at the current place. The translator pops entries corresponding to a loop after the loop is processed. Two other things are done at the for loop declaration by the rewriting grammar. First, when the list of accessor applications is encountered the translator inserts a new node into the AST just before the for loop. The code generator expands this node into code that initializes the accessor with the dynamic parameters (in our simple implementation it is always the size of the array and the current index). Second, a node is inserted into the AST at the function level so the code generator knows it needs to instantiate the given accessor when the function starts.

Accessor Implementation

Accessors themselves are implemented as C++ template classes parameterized by the type of the variable the specific accessor instance will manage and conforming to the same interface. This arrangement was chosen purely for reasons of implementation simplicity and in the C++ templates can in theory be replaced by inlined C code optimized for the specific accessor application.

The previous subsection explained that when an accessor is applied in the *Dali* code, the translator generates code that instantiates the particular accessor and calls methods to set it up with the parameters given to it statically, within the accessor declaration. The runtime parameters are passed to the accessor just before the loop of its scope is entered. The last thing that has to be explained about accessor application implementations is how the code manipulating with arrays is translated into code that uses these accessors. When the parser encounters an expression containing an array subscription, it first checks whether the code appears within a PPE or an SPE function. In the former case the AST remains unchanged as array subscription on PPE has the same semantics in *Dali* as it does in C. In the latter case special accessor code is generated, more specifically the code generator outputs code that calls methods of the C++ accessor object. To be more exact, the concrete method generated depends on the context. When a value of the array is read, i.e.., on the right side of an assignment, the `get()` method is used. When a value is written into the array, like it is if the subscription appears on the left side of an assignment, then the `put()` method is inserted.

Let us summarize this on an example. All the array subscriptions within an SPE code are replaced by calls to the accessor methods. Therefore for instance,

```
arrA[i] = 2 * arrB[i-1]
```

becomes,

```
arrA_accessor.put(i, 2 * arrB_accessor.get(i));
```

Internally, the accessor objects are managing the LS buffers and DMA transfers. This simple solution is only possible because we do not perform any optimizations on the accessors based on their options. The loop transformations with accessors described in Section 3.12 have not been implemented, therefore the double buffering accessor object for instance needs to decide when to switch the buffers and start prefetching next segment. It does with this code:

```
template <typename T> inline T DoubleBufferAccessor<T>::get(unsigned offset)
{
    unsigned relativeOffset = offset - start;
    if (__builtin_expect(relativeOffset >= BUFFER_ELEMENTS, 0))
    {
        start += BUFFER_ELEMENTS;
        fetch(activeBuffer, ifelse(
            start + BUFFER_ELEMENTS < arraySize,
            start + BUFFER_ELEMENTS, 0));
        activeBuffer ^= 1;
        actbuffer = buffer[activeBuffer];
        relativeOffset = offset - start;
        waittag(tag[activeBuffer]);
    }
    return actbuffer[relativeOffset];
}
```

The code uses three different techniques for reducing impact of branching, those are in detail discussed in Section 4.5. At the beginning the relative offset into the current buffer is calculated from the absolute offset into the array. If the calculated offset is beyond the current buffer, the execution enters the if branch and starts prefetching the next buffer. The `ifelse` expression inside the call to `fetch()` ensures, without introducing branching, that the accessor never accesses the main memory outside the bounds of the array it is applied to. The current buffer is updated and the program makes sure that the new current buffer has been completely fetched. Finally, in code common to all cases, the correct item is returned as was requested.

SPE Call Implementation

There are several options of implementing the mechanism of SPE function calls. The approach chosen in this implementation was to generate a unique C-structure per each SPE function. For example, for a function with the signature

```
void vector_product(float* v1, float* v2, float* res);
```

the translator generates the following structure:

```
struct SFunction_dot_product
{
    float* v1;
```

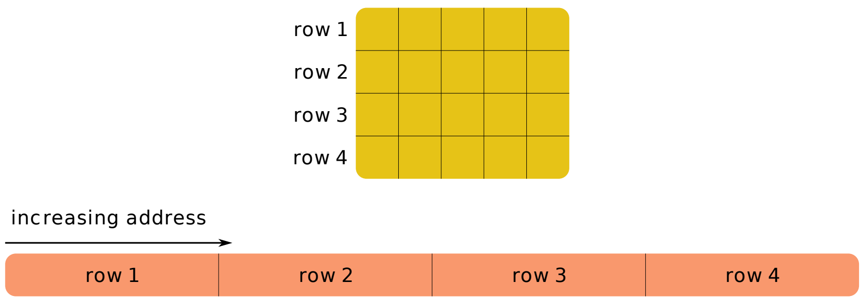

Figure 4.2: A matrix and its row-major storage in the memory

```
    float* v2;
    float* res;
    char _padding[20];
};
```

The structure is padded to 32 bytes so we can transfer it using DMA. For the translator to be able to do so it has to be aware of the type sizes. The structure is then placed in a header file shared between the PPE and the SPE code. Compiler maintains the spe modifier for each function in the symbol table so it can always determine when a PPE caller calls an SPE function. When it happens the structure is instantiated and filled with the actual parameter values. Next the SPE thread is started and a pointer to the structure is passed to it. The SPE runtime code then fetches the whole structure to the local store and finally calls the desired SPE function with the correct parameters.

4.4 Available Accessors

From the list of suggested accessors in Section 3.9, four were implemented. They are the naive accessor, a simple speculative accessor, a double buffering accessor (a special case of the multibuffering accessor) and a matrix accessor used for column-wise matrix access.

The only thing the implemented speculative accessor does is prefetching n items immediately following the currently accessed item in an array. The idea is that during a high-locality access there is a high chance that a program accessing `a[i]` will want to access `a[i+1]`, `a[i+2]` and so on. The reason for creating this accessor was to be able to spot accessor-attributed performance gains early during the development. As the next chapter documents, this accessor indeed offers a great improvement over the naive accessor, but is still very slow on the absolute scale.

The matrix accessor was implemented in order to explore memory access possibilities for data types with suboptimal memory layout given the program they are used in. For this particular case, suppose the program is storing matrices in memory in row major order (Figure 4.2).

As long as the matrix is processed row-wise, that is $(a_{1,1},\ a_{1,2},\ldots)$, simple double buffering can be used. But as soon as the programmer needs to process a large $N \times N$ matrix column-wise, accessing elements $a_{1,1},\ a_{2,1},\ldots$, the same approach is of no help as for buffers of size e.g. $2N$ the buffers need to be swapped every second access. Worse, utilization of those buffers is only $1/N$ and a lot of the EIB bandwidth is thus wasted.

What is needed instead is *double buffering over columns*. In practice this means that the e.g. the first buffer of size $2N$ needs to get items from offsets $0,\ 1,\ N,\ N+1,\ 2N,\ 2N+1, \ldots$, or the first two columns. Even though it is not possible to fetch all those locations in one DMA transfer, it is possible to setup the architecture's DMA subsystem for fetching them using a DMA list and a single DMA command. The matrix accessor is internally doing exactly that and by using it the programmer can save large amounts of DMA setup costs while effectively obtaining column-wise double buffering.

4.5 Eliminating Branching in SPE Code

The final section of this chapter will focus on a practical aspect of writing C code that runs on the SPE processor. As Section 2.7 has mentioned, one thing the SPE deals particularly bad with is branching. Ideally, the programs should not have any loops or `if` statements and all function calls should be completely inlined or, better, defined as macros. Such programming conditions would be largely unacceptable and, luckily, there are methods that remedy the situation somewhat. The remnant of this chapter will present different techniques for improving the program's performance by eliminating branches. The reason why we discuss it here is that the *Dali* runtime implementation uses some of them for improving the performance of the accessors.

Loop Unrolling

Loop unrolling is a well known way of limiting branching in a loop by decreasing the number of loop iterations. For example, instead of processing one item of an array per loop iteration and incrementing the loop control variable by 1, four items are incremented at once and the control variable is incremented by 4.

Loop unrolling can be compiler automated but most often it is manual. Despite the SDK documentation stating otherwise [22, pp. 89], we have found the effects of manual loop unrolling to be quite substantial. This is also confirmed by the SDK sample programs and even a code snippet from the same publication [22, pp. 110].

Care must be taken when loop unrolling is exploited, because the code that has to fit in the LS will increase in size.

Select Bits Instruction

There are situations where an algorithm chooses between two ways to obtain a given value. Suppose the final value should be stored in an unsigned integer variable `val`, and the decision is made based on a boolean condition `cond`. The programmer writes:

```
if (cond)
    val = [expression A];
else
    val = [expression B];
```

This introduces conditional branching and thus for the SPE it is frequently faster to compute both the expressions and then choose among them using the select bits instruction [23] like in the snippet below.

```
unsigned val1 = [expressionA];
unsigned val2 = [expressionB];
vector unsigned v1, v2, vr, vb, vs;
const vector unsigned vzeros = spu_splats(0u);
v1 = spu_promote(val1, 0);
v2 = spu_promote(val2, 0);
vb = spu_promote((unsigned)cond, 0);
vs = spu_cmpgt(vb, vzeros);
vr = spu_sel(v2, v1, vs);
val = spu_extract(vr, 0);
```

Because the select bits instruction is invoked through a compiler intrinsic accepting only vectors, we first need to promote the scalar values into the preferred slots of vectors. The condition vector `vb` is then compared with a zero vector, resulting in the vector `vs` that in its preferred slot either has only zeros if the boolean condition was `false`, or only ones if it was `true`. The select bits intrinsic takes `vs` to select either preferred slot bits from `v2` or `v1`, thus effectively choosing between the result of the two original expressions.

There are two caveats to this method of branch elimination. Obviously, with regard to the performance, the total cost of computing the expressions `A` and `B` needs be less than the total cost of computing each of them and performing branching, weighted by the frequency of the condition occurring in favor of the given expression. Secondly, to ensure correctness, the expressions should not have any side effects.

Branch Hinting Instruction

Since Cell BE has no branch prediction in hardware, a software method is provided for those cases where the programmer does not manage to get rid of branching in another way. This is done by using the ISA's hint for branch instruction (HBR).

HBR instructions notify the processor in advance about a soon to be encountered branch instruction. Specifically they inform about the address of the branch instruction and the address of the branch target that should be prefetched. To be effective the hint has to be provided soon enough, 11 cycles and four instruction pairs ahead, according to [14, pp. 288]. Branch instructions that are not hinted in this way are expected not to be not taken. The penalty for branching at an address not previously hinted is the same as branching at address hinted incorrectly—the already prefetched instructions are flushed and new ones are prefetched from the actual branch target, in total taking 18–19 cycles. An incorrectly hinted branch will therefore not affect the computation results, yet it can harm performance.

There is no intrinsic directly mapping to HBR. The compiler typically does so in obvious cases like at the end of a loop. The programmer can however help the compiler to predict outcomes of conditional branching using the `__builtin_expect` directive taking two parameters: The condition to be evaluated and probable outcome of the evaluation. If `cond` in the following code happens to be true only rarely

```
if (cond)
    doSomething();
else
    doSomethingDifferent();
```

then the directive can be used

```
if (__builtin_expect(cond, false))
    doSomething();
else
    doSomethingDifferent();
```

to instruct the compiler to generate code hinting the else branch.

Empty Calls

Branches including a function call with side effects are difficult to eliminate. The select bits trick by itself will not tackle the branching here—in one branch the function must be called, in the other branch it must not. In special cases, however, branching can still be eliminated. It is so when certain parameters can be passed to the function that stop the side effect from happening. The C Standard Library function `memcpy()` is such an example, consider:

```
if (cond)
    memcpy(dest, src, size);
```

This code can have its branch eliminated by employing the select bits method described in "Select Bits Instruction" (here abstracted into the function `ifelse_select(v1, v2, cond)` returning `v1` if `cond` is true and `v2` otherwise) combined with "empty call" to the function:

```
cond_size = ifelse_select(size, 0, cond);
mempcy(dest, src, cond_size);
```

That is, in case `cond` evaluates to false, `mempcy` is still called but with 0 bytes to copy.

It is hard to say in case like `memcpy` whether this code transformation would be beneficial since entering the function certainly involves further branching. Nonetheless, the DMA transfer intrinsics that do not map to any branching can also be passed 0 transfer size. In fact, the accessors implementation in the *Dali* runtime library uses this technique at several places.

Smart Arithmetics

Sometimes the code of a computation can be restructured by witty usage of arithmetic and bitwise operations so that it no longer needs conditional branching. Consider a function that for a given matrix index (i, j) returns index of the following item column-wise way, that is $(i+1, 0)$ if j is the last column of the matrix or $(i, j+1)$ otherwise. If M^2 is the matrix size and indexing starts at zero then the implementation in C++ might look like:

```
void next_index(unsigned& i, unsigned& j)
{
    i++;
    if (i == M)
    {
        j++;
        i = 0;
    }
}
```

The branching in such a short, simple code is unpleasant. Here is what a matrix multiplication sample in the Cell BE SDK does instead (extracted from a vectorized version).

```
void next_index(unsigned& i, unsigned& j)
{
    unsigned last;
    i++;
    last = cmpeq(i, M);
    i &= ~last;
    j -= last;
}
```

The `cmpeq()` function returns an unsigned integer with all bits set to one if its two operands are equal, zero otherwise. This means for the more common case where `i` does not equal `M`, `i` is incremented and then left unchanged, `j` is unchanged. In the opposite case, `i` is ANDed with zeros, itself becoming zero and the maximal possible unsigned integer is subtracted from `j`, increasing it by one.

This branch elimination technique requires the greatest creativity plus the programmer needs to have a good overview of the available instructions and be comfortable with bitwise operations, but it can be very rewarding if such a code transformation can be found.

5

Experiments

Three experiments were devised to evaluate the presented method of tackling the memory access problem from Chapter 3 and its first implementation we talked about in Chapter 4. The first experiment should give us an overview of the pure access speed into the main memory using different accessors, the second experiment then uses a real computation to measure the language performance in a more real situation. The last experiment shows a situation where a simple speculative accessor is applied to speed up a binary search.

All the experiments were carried out on a PS3 machine manufactured in 2006. The console was running Yellow Dog Linux 5.0, kernel version 2.6.22. Cell SDK version 3.0 was used to compile all the programs and link the runtime libraries against. Namely the version of the PPU and SPU GCC compilers supplied with it is 4.1.1. The compilers were always invoked with the maximal optimizations parameter `-O3`. To measure timing on the SPE, some basic library support is provided as a part of the *Dali* runtime library. Internally, those routines are exploiting the SPE decrementer [19, pp. 384].

The second experiment involves measurements made on an x86. The machine in question had Intel Core 2 T7200 CPU with the clock speed of 2.0 GHz and the memory clock speed of 667 MHz. It was running Ubuntu Linux 8.04 with the kernel version 2.6.24 and the GCC version 4.2.4. Here also was GCC invoked with the `-O3` option and the timing was measured using the `times()` function of the standard C library[1].

Because the current *Dali* implementation does not support parallelism, all the measurements were taken on a single SPE processor. As long as the accessors do not choke EIB or the main memory controller there is no reason to believe that spreading computation over several SPEs would not scale well, almost linearly.

5.1 Access Speed Experiment

The first experiment should help us obtain a crude image of how fast the different accessors are and how they compare to each other, to a manually written benchmark and to theoretical limits of the architecture. What this test has told us is *the upper bound of the access speed*, specifically in the direction from the main memory to the local store. The results can be seen in Table 5.1.

Since accessors are implemented as C++ templates, it was possible to benchmark their speed by writing a small C++ function that instantiates them and then accesses memory locations in a manner that suits the given accessor the most, e.g. item by item for the double buffering accessor or accessing items with constant step for the matrix accessor.

[1]http://www.gnu.org/software/libc/manual/

Access type	Avg. access speed
naive accessor	0.04 GB/s
simple speculative accessor	0.16 GB/s
matrix accessor	0.57 GB/s
double buffering accessor	1.52 GB/s
ideal DMA transfer	12.1 GB/s
memory throughput (2 channels)	25.6 GB/s

Table 5.1: Main memory read access speeds depending on the access type.

The maximum DMA speed is measured simply by fetching sequences of continuous blocks of 16 KB, the maximum size allowed per single DMA transfer and so the most efficient one, the same approach that the DMA benchmark in the SDK uses. It was mentioned in Section 2.4, that the peak memory access speed is 25.6 GB/s, but the first results of this benchmark were rather slow, presenting access speeds around only 1.1 GB/s. Inspecting the benchmarking program from the SDK has showed that much higher speeds can be achieved if we ensure that the page tables and TLBs are already loaded when we attempt the DMA access. After the program was extended with a small loop that "touches" all pages we are subsequently using, the benchmark has shown more realistic speeds, virtually identical to the SDK DMA benchmark.

5.2 Matrix Multiplication Experiment

The next set of benchmarks focused on measuring FLOPS performance in *Dali* programs implementing different algorithms of matrix multiplication. Both operands of the operation were square matrices of size 512×512. All the matrices used for the *Dali* benchmarks are stored in the memory in a row-major order (see Section 3.3). This is so because with this benchmark we are not trying to achieve the best performance as much as trying to see the relative gains of using different accessors. The results of the test can be seen in Table 5.2.

The first *Dali* program (1) implements the simple matrix product algorithm based on the operation's definition [41]. It does not specify any access methods and that is why the compiler generates only naive accessor code. The same method is used in program (2), this time however the matrix accessor is applied on the second operand and double buffering on the first two operands. If we invert the two innermost loops of the basic algorithm we arrive at program (3), which can now use double buffering accessors on both the operands and also the result matrix. To see how much we could improve the performance of this program, we vectorized the generated code manually and mildly unrolled loops in program (4).

To put the computation rates in perspective, additional results are present in the table. The program (5) is written entirely manually with optimal accessing methods, vectorization and mild unrolling. Listed is also the performance for the same algorithm implemented in pure C and run on an x86 processor (6). Result (7) is a Python[2] implementation of the basic algorithm. Finally, the performance of a highly optimized matrix multiplication

[2]http://www.python.org/

#	Implementation	Access type	Time	Computation rate
1	matrix product definition	naive accessor	44.3 s	0.006 GFLOPS
2	matrix product definition	matrix accessor on second operand	2.97 s	0.09 GFLOPS
3	inverted loops	accessor double buffering	3.66 s	0.07 GFLOPS
4	inverted loops, manually vectorized	accessor double buffering	0.166 s	1.61 GFLOPS
5	inverted loops	manual, double buffering	0.088 s	3.04 GFLOPS
6	inverted loops, x86	transparent	0.194 s	1.38 GFLOPS
7	matrix product definition, Python	transparent	80.9 s	0.0033 GFLOPS
8	highly optimized SDK implementation	manual, double buffering, block memory structure		23.21 GFLOPS

Table 5.2: Matrix multiplication benchmark.

program from the SDK that employs block memory layout and enormous loop unrolling is also present as result (8).

5.3 Evaluation of the First Two Experiments

Performance Evaluation

By and large the numbers in Table 5.2 are disappointing. They show that generating effective DMA access code is difficult. To put the unflattering comparison with test (8) to a better perspective, however, more background needs to be provided. The matrix multiplication implementation is present in the SDK to prove that achieving the marketed 25 million operations in single precision per second is actually possible in practice. To do that, several special steps were taken. The matrices are stored in the memory as a sequence of submatrices, each submatrix having precisely 16 KB so they can be transferred effectively. The main computation loops are so much unrolled that the two inner loops disappear completely. Further, by extremely unrolling the loops, all vector loads from the local store to a register can happen tens of instructions ahead of actual use of the register.

Still, why is the best implementation in *Dali* (4) slower by a factor of more than ten? The main contributing factor is branching in the accessor code (albeit hinted) that is encountered on each array subscription. A solution to the problem exists and was described previously (see Section 3.12). Unfortunately it has not been implemented. Besides only mild loop unrolling, the remaining performance loss can be attributed to poor optimization of the accessor code, for instance inlining the code instead of using templates could be helpful as well as optimizing the buffer sizes and frequently executed routines. A detailed analysis would require proper profiling on the target Cell BE machine which is not well supported on the PS3.

A positive sign are the relative speed-ups of factor 10 to 15 between the naive accessor and the matrix and double buffering accessor in results (1), (2) and (3). Also notice in results (4) and (6) that the vectorized and slightly unrolled version of the generated double buffering implementation is already faster than the x86 implementation. This is a great news when one realizes that the x86 implementation takes more or less the entire processor whereas our implementations in this experiment only use one of eight available accelerators of Cell and leave the main core idle. The remaining SPEs could at the same time be used for other tasks or be performing the matrix multiplication in parallel which could speed the computation up several times.

We can also clearly see from the results (3) and (4) that for a production deployment the *Dali* language would first need a proper support of vector data types and vector operations and perhaps also constructs that let the programmer semi-automatically control loop unrolling.

Programming Comfort Evaluation

There is an ugly side-effect of the high performance implementation (8): the SPE code has around 1300 source lines and most of it are dense, repetitive macros. Compared to this, the *Dali* sources have all between 45 and 70 lines for the entire program (PPE and SPE), which is similar to the C implementation. Python implementation has only around 30 lines of source code. Additionally, it is much more straightforward to build a *Dali* program into an executable than to build the C programs using the two GCC compilers. Again, Python as a scripting language that does not need to be compiled offers even greater level of comfort.

The experimenting proved the strong advantage of abstracting the access methods like *Dali* does with its accessors. The programmer can get the basic algorithm working first, perhaps only to receive a meager performance and then start to think about what best accessing methods can be fitted on the different memory accesses the program does.

5.4 Speculative Accessor Experiment

The following experiment shows an example of using a speculative accessor. Binary search algorithm [26, pp. 409–426] is a technique for locating particular value in a *sorted list*. First, the searched key is compared with the element in the middle of the array. If the found value is greater than the value we are looking for, only part of the array left of the middle is used for further search, if the found value is smaller then only the right part is used. The search continues analogously by finding the middle item of the selected part of the array and comparing it with the value we are looking up.

On the SPE, using a naive accessor will cause the binary search to stop and wait for a synchronous DMA request whenever the searched value needs to be compared with an item of the array. Possibly a better solution can be devised. The accessor for binary search works by observing the index of the current middle element and prefetching the next two candidates ahead (Figure 5.1). By the time the algorithm loop reaches the next iteration and asks for one of them, they will already be on their way to the local store.

We suppose that computing the key value is nontrivial. In the opposite case, for example where computing a key is only question of retrieving it from the array item, there won't be enough time for the speculative accessor to prefetch the items and the program will often stall on the DMA request again, but this time waiting for a transfer of two items instead of one. This experiment is therefore hashing contents of the items before comparing with the searched key.

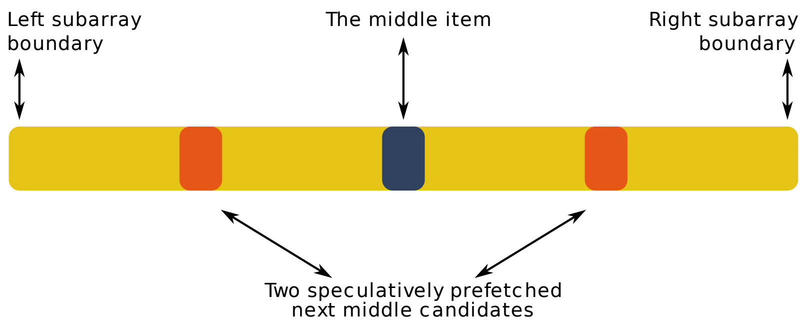

Figure 5.1: Binary search accessor

No. of array elements	Time using the naive accessor	Time using the speculative accessor	Speculative accessor speedup
4096	1.117 ms	1.147 ms	-2.67 %
8192	1.294 ms	1.212 ms	6.77 %
16384	1.399 ms	1.278 ms	9.42 %
131072	1.691 ms	1.459 ms	15.84 %
262144	1.776 ms	1.525 ms	16.45 %

Table 5.3: Binary search performance achieved with the naive and the specialized speculative accessor.

Table 5.4 summarizes the result. Each test was run in a loop for 128 times to reduce the error. Because of the logarithmic complexity of the binary search technique, great increases in array sizes produce only marginal differences in the observed results. Notice that gains in performance are only possible with relatively large arrays. This is probably because the used speculative accessor needs to dynamically allocate buffers on the local store and perform other initialization work that the basic accessor is spared doing. Clearly there would be no point using the binary search speculative accessor if the arrays were smaller or in case when almost no computation is taking place between two consecutive accesses. The synchronous naive accessor would be a better choice then—but notice that it is probably what the programmer would choose to do if she was manually implementing the technique on the SPE anyway. We can expect to see more encouraging results of speculative accessors with more advanced implementation of the *Dali* translator, for example because the accessors could internally share statically allocated buffers.

6

Conclusion

This thesis aspires to show that the software engineering problems arriving with the new multicore architectures can be overcome by designing programming languages which are explicitly aware of the architecture's specifics. It also shows that using such languages does not need to be more complex for the programmer then using other high level languages yet it can allow for better utilization of the architecture's features and so for a better performance.

Specifically, we studied the problem of memory access on the Cell BE architecture which has appeared on the market just recently. Along with its cheap price and immense computing power measured in hundreds of GFLOPS it bears idiosyncrasies which make developing software for it more elaborate and error prone, more so when unleashing the full potential of the processor is an ambition. In Chapter 2 we introduced this architecture together with the traps of software development for it. The technical rationale behind the architecture design which was also discussed in that chapter gives us reasons to believe that upcoming architectures might share some of their traits with the Cell.

One of the problems that always needs to be tackled when programming Cell BE is controlling the local stores, a newly introduced memory type that is present directly on the chip. Chapter 3 talks about the topic in great detail. The contribution of this thesis is surveying the existing methods of doing so and finally proposing a new solution based on language constructs that dictate what access strategies should the compiler use. This new solution is embodied in a new language, *Dali*, that was designed specifically for the purpose. The programmer defines access strategies in two steps. First she declares general access strategies to be used throughout the program and then, in functions running on the accelerators, binds the declared access strategies with variables that should be handled by them.

As we discuss in Chapter 4, ANTLR, a compiler generator, was used to implement translator from *Dali* to C++. The generated code can then be compiled using the standard Cell BE GCC toolchain and run on the target hardware, in our case a PlayStation 3 game console. Performance of the language was evaluated in Chapter 5. We experimented with different access speeds attainable with different access methods the language currently supports. Tests were also run to observe the extent of performance degradation of *Dali* programs as compared to a C implementation running on PS3, a C implementation running on an x86 processor and a Python implementation.

While the current language implementation can hardly deliver a performance competitive with equivalent but manually built and optimized C programs, it offers a much more efficient and high-level way of taming the architecture. Instead of spending time on tedious tasks of initiating and finishing DMA transfers between memories and painstakingly managing temporary local store buffers, she can fully concentrate on the implemented problem itself, just like one would when programming on an x86 platform. Once the program is correctly working, specific memory access methods can be can be used to boost the performance of critical places or perhaps globally.

We see this approach as taking the middle way between the two existing Cell BE compilers: the GCC toolchain and the IBM's XL/C compiler. The former leaves all the DMA access work to the programmer, giving him all the power that the architecture offers but many responsibilities as well. The latter tries to determine during compile time what the optimal strategy is and then use it. We argue in Section 3.4 that there are scenarios under which no compile time analysis can decide the optimal memory access strategy in the given program. Having said that, the *Dali* compiler can benefit from the same techniques used by the IBM's compiler. In fact, they can help in many cases suggest a better than the default strategy when the accessor specification is omitted in the source code. The programmer would then only need to intervene in exactly those cases where the analysis fails.

We are concluding that despite a flawed prototype implementation of the *Dali* language, the ideas it represents—namely automatic, programmer directed controlling of the memory hierarchy—are substantial, making important next steps towards better programmability of Cell BE. For if we fail to deliver an effective methodology of programming the architecture it will always stay a domain of only few scientific and game development specialists, despite its remarkable performance potential.

6.1 Future Work

In this final section the most interesting prospects of continuing the presented work are given. The first obvious area could be the access methods, description of which we have certainly not exhausted. When one starts to consider other programming models other than the services model (see "Programming Models" on page 26), it becomes clear that to allow for a local store to local store data movements, the accessors need to be extended to support storing processed data to a different place that they fetched them from. This could possibly involve accessors ensuring that unwanted concurrent read and write access to the same location leading to data corruption does not occur.

On a slightly higher level it can be studied how to extend *Dali* with other features that accommodate developing for Cell BE, particularly constructs for handling parallel execution and (auto)vectorization. Indeed, before fully realizing the importance of the memory hierarchy problem, we did a preliminary research into the possibility of adopting the X10 language for Cell [27]. While adopting the full language for the platform did not seem to be completely viable at the time, some interesting aspects can definitely be borrowed from it. Specifically the concept of *futures*, or expressions which are evaluated asynchronously but no later than at the moment their result is actually used, seems to be potentially an abstraction for accelerator computation. Another construct known from X10 and other parallel programming languages is the *foreach* loop that performs parallel computation on all elements of an array passed to it.

Exploiting the known compiler techniques for increasing data locality and managing the memory transfers at places where the programmer did not explicitly request concrete strategies would be an important task that overlaps with the ongoing research on IBM's XL/C compiler [8]. Besides, smart restructuring of loops with accessors allows for making the DMA transfer code simpler and reduce branching. Section 3.12 shows this for the double buffering accessor, but other accessors would perhaps benefit from similar techniques too. Accessor parameters could be defined in the language that would allow the programmer drive these optimizations in cases where the compiler alone is unable to decide. Once more powerful compile-time analysis techniques are implemented they can also be used to check logical errors in the programmer specified access strategies.

There is a lot to be done on the current *Dali* compiler too. In general, many basic features of other modern languages are missing such as defining new data types and classes. Accessors for dynamic data structures that speculatively prefetch nodes or preprocess the structure on the PPE side are described in Section 3.9, but the *Dali* implementation does not involve them. Later during the project it also showed that the concept arrangement between static and runtime accessor parameters is not quite clear and could be insufficient in certain scenarios—the implementation should be fixed to let any accessor parameter be dynamic. Better utilization of the static parameters can lead to optimizations in the accessing code too, for instance many small DMA transfers from nearby locations in the main memory could be merged into one. Experiments showed that optimizations and fine-tuning are missing noticeably. Making sure that the generated code is effective could be one of the first tasks in the area as well as the loop unrolling discussed earlier in this chapter.

A

Sample C implementation

This appendix gives a complete listing of the C program from Section 3.10.

common.h

```
#ifndef COMMON_H
#define COMMON_H

// size of one dimension of the matrix
#define SIZE 128

typedef struct
{
    float* matrix;
    float* sums;
    char   _padding[8];
} argsum_t __attribute__((aligned(16)));

#endif
```

main.c

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <libspe2.h>
#include "common.h"

extern spe_program_handle_t spu_sum;

void *thread_starter (void *arg)
{
    argsum_t* spuArgsp = (argsum_t*) arg;

    // create context, load SPU code
    spe_context_ptr_t speContext;
    if ((speContext = spe_context_create(0,NULL)) == NULL)
    {
        perror("spe_context_create() error.");
        exit(1);
    }
    if (spe_program_load(speContext, &spu_sum) != 0)
    {
        perror("spe_program_load() error.");
        exit(1);
    }

    // start
    unsigned entry = SPE_DEFAULT_ENTRY;
    if (spe_context_run(speContext, &entry, 0, spuArgsp, NULL, NULL) < 0)
    {
        perror("spe_context_run() error.");
        exit(1);
    }
    pthread_exit(NULL);
}

float matrix[SIZE * SIZE] __attribute__((aligned(128)));
float sums[SIZE] __attribute__((aligned(128)));

int main(int argc, char** argv)
{
    pthread_t posThread;
    void *status;
    argsum_t spuArgs;
    read(matrix, stdin);
    // start the spu program
    spuArgs.matrix = matrix;
    spuArgs.sums = sums;
    if ((pthread_create(&posThread, NULL, thread_starter, (void*)&spuArgs)) != 0)
    {
        perror("pthread_create() error.");
        exit(1);
    }
    // wait for the spu thread to terminate
    pthread_join(posThread, &status);
    // print results
    print_array(sums, SIZE, 1);
    return 0;
}
```

spu/spu_sum.c

```
#include <stdio.h>
#include <spu_mfcio.h>
#include <libmisc.h>
#include "common.h"

#define ROWS_IN_BUF 2
// each buffer is for two rows of the matrix
volatile float buffer[2][ROWS_IN_BUF * SIZE] __attribute__((aligned(128)));
volatile float sums_buffer[SIZE] __attribute__((aligned(128)));
```

```
#define BUFFER_LENGTH (ROWS_IN_BUF * SIZE * sizeof(float))

int main(unsigned long long id, unsigned long long argp)
{
    unsigned tag[2];
    volatile argsum_t args;

    tag[0] = mfc_tag_reserve(),
    tag[1] = mfc_tag_reserve();

    // synchronously fetch the arguments
    spu_mfcdma32((void *)(&args),
        (unsigned int)argp,
        sizeof(argsum_t),
        tag[0], MFC_GET_CMD);
    spu_writech(MFC_WrTagMask, 1 << tag[0]);
    spu_mfcstat(MFC_TAG_UPDATE_ALL);

    unsigned current_row;
    unsigned base_row = 0;
    unsigned buffer_index = 0,
        next_index = 1;
    unsigned i;

    // initialize the sums
    for (i = 0; i < SIZE; ++i)
        sums_buffer[i] = 0;

    // start fetching the first few rows of the matrix, no need to fetch the sums
    // array cos it will be overwritten
    spu_mfcdma32((void *)(&buffer[0]),
        (unsigned int)args.matrix,
        BUFFER_LENGTH,
        tag[0],
        MFC_GET_CMD);

    for (;base_row < SIZE - ROWS_IN_BUF; base_row += ROWS_IN_BUF)
    {
        // start fetching the next row
        float* nextRowAddress = args.matrix + (base_row + ROWS_IN_BUF) * SIZE;
        spu_mfcdma32((void *)(&buffer[next_index]),
            (unsigned int)nextRowAddress,
            BUFFER_LENGTH, tag[next_index],
            MFC_GET_CMD);
        // wait for the current buffer transfer to finish
        spu_writech(MFC_WrTagMask, 1 << tag[buffer_index]);
        spu_mfcstat(MFC_TAG_UPDATE_ALL);
        // go through the rows in the current buffer and do the adding
        for (current_row = base_row; current_row < base_row + ROWS_IN_BUF;
            ++current_row)
        {
            unsigned i;
            for (i = 0; i < SIZE; ++i)
            {
                sums_buffer[current_row] +=
                    buffer[buffer_index][(current_row - base_row) * SIZE + i];
            }
        }
        // the current buffer is now the next one
        buffer_index = next_index;
        next_index ^= 1;
    }
    // one last buffer remains to be processed
    spu_writech(MFC_WrTagMask, 1 << tag[buffer_index]);
    spu_mfcstat(MFC_TAG_UPDATE_ALL);
    for (current_row = base_row; current_row < base_row + ROWS_IN_BUF;
        ++current_row)
    {
        unsigned i;
        for (i = 0; i < SIZE; ++i)
        {
            sums_buffer[current_row] +=
                buffer[buffer_index][(current_row - base_row) * SIZE + i];
        }
    }

    // store the sums array
    spu_mfcdma32((void *)(sums_buffer),
        (unsigned int)args.sums,
        sizeof(float) * SIZE,
        tag[0],
        MFC_PUT_CMD);
    spu_writech(MFC_WrTagMask, 1 << tag[0]);
    spu_mfcstat(MFC_TAG_UPDATE_ALL);

    mfc_tag_release(tag[0]);
    mfc_tag_release(tag[1]);

    return 0;
}
```

Bibliography

[1] Jean Bovet and Terence Parr. Antlrworks: an antlr grammar development environment. *Softw., Pract. Exper.*, 38(12):1305–1332, 2008.

[2] Alfredo Buttari, Piotr Luszczek, Jakub Kurzak, Jack Dongarra, and George Bosilca. Scop3, a rough guide to scientific computing on the playstation 3. Technical report, Innovative Computing Laboratory, University of Tennessee Knoxville, May 2007.

[3] Philippe Charles, Christian Grothoff, Vijay A. Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In Ralph E. Johnson and Richard P. Gabriel, editors, *OOPSLA*, pages 519–538. ACM, 2005.

[4] Thomas Chen, Dr. Ram Raghavan, Jason Dale, and Eiji Iwata. Cell broadband engine architecture and its first implementation. `http://www.ibm.com/developerworks/power/library/pa-cellperf/`, November 2005.

[5] Intel Corporation. Intel microprocessor export compliance metrics. `http://www.intel.com/support/processors/sb/cs-023143.htm`, April 2009.

[6] Marc de Kruijf. Cell/b.e. challenge '07 submission, mapreduce for the cell architecture. `http://www-304.ibm.com/jct01005c/university/students/contests/cell/r1w2proposal.pdf`, 2007.

[7] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[8] Alexandre E. Eichenberger, Kevin O'Brien, Kathryn M. O'Brien, Peng Wu, Tong Chen, Peter H. Oden, Daniel A. Prener, Janice C. Shepherd, Byoungro So, Zehra Sura, Amy Wang, Tao Zhang, Peng Zhao, Michael Gschwind, Roch Archambault, Yaoqing Gao, and Roland Koo. Using advanced compiler technology to exploit the performance of the cell broadband engine^{tm} architecture. *IBM Systems Journal*, 45(1):59–84, 2006.

[9] Marc González, Nikola Vujic, Xavier Martorell, Eduard Ayguadé, Alexandre E. Eichenberger, Tong Chen, Zehra Sura, Tao Zhang, Kevin O'Brien, and Kathryn M. O'Brien. Hybrid access-specific software cache techniques for the cell be architecture. In Andreas Moshovos, David Tarditi, and Kunle Olukotun, editors, *PACT*, pages 292–302. ACM, 2008.

[10] Dan Grossman, Michael Hicks, Trevor Jim, and Greg Morrisett. Cyclone: A type-safe dialect of c. *C/C++ User's Journal*, January 2005.

[11] Intel Corporation. *Intel 64 and IA-32 Architectures, Optimization Reference Manual*, 2009.

[12] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 2B: Instruction Set Reference, N-Z*, 2009.

[13] International Business Machines Corporation. *Using the single-source compiler*, 2007.

[14] International Business Machines Corporation. *Programming the Cell Broadband Engine, Architecture Examples and Best Practices*, August 2008.

[15] International Business Machines Corporation, Sony Computer Entertainment Incorporated, Toshiba Corporation. *Accelerated Library Framework for Cell Broadband Engine, Programmer's Guide and API Reference*, 2007.

[16] International Business Machines Corporation, Sony Computer Entertainment Incorporated, Toshiba Corporation. *C/C++ Language Extensions for Cell Broadband Engine Architecture*, 2007.

[17] International Business Machines Corporation, Sony Computer Entertainment Incorporated, Toshiba Corporation. *Cell Broadband Engine Architecture*, 2007.

[18] International Business Machines Corporation, Sony Computer Entertainment Incorporated, Toshiba Corporation. *Cell Broadband Engine Architecture JSRE, SPE Runtime Management Library*, 2007.

[19] International Business Machines Corporation, Sony Computer Entertainment Incorporated, Toshiba Corporation. *Cell Broadband Engine, Programming Handbook*, 2007.

[20] International Business Machines Corporation, Sony Computer Entertainment Incorporated, Toshiba Corporation. *Example Library API Reference*, 2007.

[21] International Business Machines Corporation, Sony Computer Entertainment Incorporated, Toshiba Corporation. *Software Development Kit for Multicore Acceleration, Programmer's Guide*, 2007.

[22] International Business Machines Corporation, Sony Computer Entertainment Incorporated, Toshiba Corporation. *Software Development Kit for Multicore Acceleration, Programming Tutorial*, 2007.

[23] International Business Machines Corporation, Sony Computer Entertainment Incorporated, Toshiba Corporation. *SPU Assembly Language Specification*, 2007.

[24] International Business Machines Corporation, Sony Computer Entertainment Incorporated, Toshiba Corporation. *Synergistic Processor Unit Instruction Set Architecture*, 2007.

[25] James A. Kahle, Michael N. Day, H. Peter Hofstee, Charles R. Johns, Theodore R. Maeurer, and David J. Shippy. Introduction to the cell multiprocessor. *IBM Journal of Research and Development*, 49(4-5):589–604, 2005.

[26] Donald Knuth. *The Art of Computer Programming, Volume 3*. Addison-Wesley, third edition, 1997.

[27] Aleš Kozumplík. X10 for cell be. Technical report, Aalborg University, Department of Computer Science, December 2008.

[28] Edward A. Lee. The problem with threads. *IEEE Computer*, 39(5):33–42, 2006.

[29] Martin Linklater. Optimizing cell core. *Game Developer Magazine*, April 2007.

[30] WG14 Committee Members. ISO/IEC 9899:1999 (C99 standard), draft. Technical report, ISO/IEC, 2005.

[31] Alan Mycroft. Programming language design and analysis motivated by hardware evolution. In Hanne Riis Nielson and Gilberto Filé, editors, *SAS*, volume 4634 of *Lecture Notes in Computer Science*, pages 18–33. Springer, 2007.

[32] Dorit Naishlos. Autovectorization in gcc. *Proceedings of the GCC Developers' Summit*, pages 105–118, 2004.

[33] Dorit Nuzman and Ayal Zaks. Autovectorization in gcc—two years later. *Proceedings of the GCC Developers' Summit*, pages 145–158, 2006.

[34] OpenMP Architecture Review Board. *OpenMP Application Program Interface*, May 2008.

[35] Terence John Parr. Enforcing strict model-view separation in template engines. In Stuart I. Feldman, Mike Uretsky, Marc Najork, and Craig E. Wills, editors, *WWW*, pages 224–233. ACM, 2004.

[36] Terence John Parr and Russell W. Quong. Antlr: A predicated-LL(k) parser generator. *Softw., Pract. Exper.*, 25(7):789–810, 1995.

[37] Gabriel Rivera and Chau-Wen Tseng. Tiling optimizations for 3d scientific computations. In *SC*, 2000.

[38] Dave Salvator. Extremetech 3d pipeline tutorial. `http://www.extremetech.com/article2/0,2845,9722,00.asp`, June 2001.

[39] Peter Seebach. The little broadband engine that could: Use multiple spes for a single task. `http://www.ibm.com/developerworks/power/library/pa-tacklecell4/index.htm`, September 2007.

[40] Martin Simonsen. Phylogenetic inference on cell processors. Master's thesis, Department of Computer Science, University of Aarhus, Denmark, October 2008.

[41] Eric W. Weisstein. Matrix multiplication. MathWorld, A Wolfram Web Resource, `http://mathworld.wolfram.com/MatrixMultiplication.html`, April 2009.

[42] Samuel Williams, John Shalf, Leonid Oliker, Shoaib Kamil, Parry Husbands, and Katherine A. Yelick. Scientific computing kernels on the cell processor. *International Journal of Parallel Programming*, 35(3):263–298, 2007.