Multiprocessor Platforms for Natural Language Processing

Henrique Costa (53931)
Orientador: David M. Matos
MEIC, Alameda, Instituto Superior Técnico, Lisboa, Portugal

Abstract. When performance is an important requirement, parallelization is often used. With the ubiquity of multiprocessor and multicore machines, there is a need to identify the various existing paradigms and tools. In this document we present a description of the existing programming models, frameworks and toolkits for the Cell Broadband Engine Architecture, a heterogeneous multiprocessor chip, and evaluate their relevance and usefulness for algorithm parallelization in natural language processing systems. The Cell has gained notoriety both with its presence on the Playstation 3 and also its unfriendliness to beginner programmers. Through three case study applications we will position the Cell regarding the performance gains and effort required to obtain them, and compare the platform to other high performance computing alternatives.

1 Introduction

This section introduces some concepts of parallel programming and define the scope and goals of this work.

1.1 Motivation

When analyzing an algorithm, one might find that different parts have little to no dependencies between them, so they could be computed simultaneously instead of sequentially to reduce overall execution time. Many approaches have been taken to achieve this, like operating system threads, several cores on the same processor, several processors on the same silica blade, and several machines in a network, with each type of parallelization bringing its own set of limitations and strengths.

We chose the Cell Broadband Engine (Kahle et al., 2005), developed by Sony, Toshiba and IBM (STI), mainly because it has a presence both in high-end servers (IBM’s Bladecenter QS20/21/22 servers) (IBM, 2008a) and as a commodity, ubiquitous, machine (Sony’s Playstation 3)(Sony, 2008) and some very distinctive features worthy of exploration, such as a very high memory bandwidth and even higher bandwidth between the several heterogeneous processors.

Furthermore, games (the area the Cell was primarily designed for) are applications with high requirements in terms of speed and amount of data processed...
that are similar to those of spoken language systems, hinting that if they can be made to run extremely well on the Cell then maybe the solutions from our field can be reimplemented efficiently to this platform. Examples of the similarity between fields include matrix operations for collision detection in games and text summarization, neural networks for game AI, and audio identification.

1.2 Goals

The Cell is a high performance, heterogeneous multiprocessor chip directed mainly at applications that perform a large number of computations. However, its architecture imposes restrictions which introduce some programming complexity. And although the Cell’s technical specifications indicate a high execution speed, there are several multiprocessor high performance alternatives that must also be considered.

Therefore, this work intends to investigate the performance and usefulness of the Cell in the field of natural language processing, providing comparisons to other platforms, while at the same time considering the programming effort needed to attain gains in execution speed. From case studies implemented on this architecture, conclusions will be drawn with respect to what kind of problem types perform well on the Cell, since not all cases justify the development effort needed to implement optimized applications for this processor.

These cases will provide an evaluation both from the programmer’s perspective, which can design and implement code optimized for the Cell, and the user, which will use prebuilt, partially optimized, libraries to obtain performance gains.

For each use case there will also be comparison metrics taken with the same (or comparable) problem implemented in at least one of current multiprocessor architectures, either processors using Intel x86 architecture (homogeneous) or Graphics Processing Units (heterogeneous).

2 The Cell Broadband Engine Essentials

2.1 Single Instruction Multiple Data

This section introduces the concept of Single Instruction Multiple Data computing and describes the Cell Broadband Engine, providing a basis for the following sections.

2.2 Single Instruction Multiple Data

A vector is a data type containing a set of data elements packed in a one-dimensional array. In the Cell these arrays are 128-bit long and support fixed-point and floating-point values. By operating on all the elements of one vector simultaneously, e.g. four integers at a time, Single Instruction Multiple Data (SIMD) instructions perform Data-level parallelism, introducing performance
gains. This type of computation, however, requires the program to explore these functionalities by organizing the data as well as the computation in such a way that SIMD operations can be used.

The process of preparing a program to use SIMD operations is called **SIMDization**, and can be done manually by the programmer or by a compiler/tool that performs **auto-SIMDization**.

SIMD instructions are abundant on the Cell, as we will see ahead. The processor supports the *vector* concept both in the arithmetic processing units, operating on the 128-bit arrays, and in memory organization, with register files of 128 registers of 128 bits each.

### 2.3 The Cell Broadband Engine

The Cell Broadband Engine Architecture (CBEA), of which the Cell Broadband Engine (Cell) is the first implementation, is a definition of an architecture directed at compute-intensive applications.

The main components in the Cell are its Power Processing Element (PPE), eight Synergistic Processing Elements (SPE - six in the Playstation 3), a Rambus XDR (Rambus, 2008) controller (MIC) that interfaces with two banks of XDR memory and an Element Interconnect Bus (EIB), to which they all these elements are connected. An overview of this architecture can be seen in Fig. 1.

![Fig. 1. Organization of Cell components](image)

The CBEA is based on the 64-bit Power Architecture (Frey, 2005), which means that CBEA-compliant processors support both 32-bit and 64-bit POWER and PowerPC (IBM, 2008b) applications. CBEA design extends the Power Architecture in two main aspects (Hofstee, 2005):

- Memory flow control introduces the concept of “local storage” (LS) and DMA transactions to move data between local storage and main address space.
The SPE is introduced as a processor that works solely on its associated LS (there is one per SPE). The SPE is capable of SIMD operations and its LS is non coherent with main memory.

The Cell also supports the huge translation lookaside buffer (TLB) filesystem, that allows the programmer to reserve 16MB memory pages of pinned, contiguous memory. The use of these huge pages reduces stress on the TLB when using large sets of data, since memory allocated on these pages will not be thrashed when swapping occurs.

With a clock speed of 3.2 Ghz, the Cell has a peak theoretical performance of 204.8 Gflop/s for single precision numeric operations.

**Power Processing Element** The Power Processing Element is a dual-threaded 64-bit RISC processor (May, Silha, Simpson, & Warren, 1994; Tabak, 1986) with vector/SIMD extensions. The PPE is a general purpose CPU, and is responsible on the Cell for running the operating system. It consists of two units, the Power Processing Unit (PPU) and the Power Processor Storage Subsystem (PPSS).

The PPU is responsible for instruction execution, and has a level 2 cache coherent with main memory.

**Synergistic Processing Element** The SPE is a 128-bit RISC processor that is designed for applications with a strong computation component, especially those that can exploit its SIMD features.

Two parts make up an SPE, its Synergistic Processing Unit (SPU) and its Memory Flow Controller (MFC).

An SPU is split into three functional units: the Local Storage (LS), the Synergistic Execution Unit (SXU), and the register file.

All of these are organized into two pipelines, as displayed in Figure 2 (adapted from (IBM, 2007b)). These two pipelines operate in parallel, as long as there are no dependencies between instructions and/or data. Instructions are assigned to their pipeline according to their type and the functional unit that executes them.

The Channel and DMA execution unit seen in Figure 2 is the interface to the MFC, being its responsibility to program the DMA controller in the latter, thus enabling communication, data transfer and control into and out of the SPU.

The MFC is the SPE’s interface to main memory and to other SPEs, via the EIB. It supports not only the transfer of instructions and data to the LS, but also a mailbox signaling service (which enables the transfer of 32 bits between two SPEs or to and from the PPE) and a DMA Proxy Queue, that allows another processing element (the PPE or a different SPE) to program DMA operations to or from the SPE.

Each DMA command issued is associated by the programmer to a number, called the tag ID, used to organize all the DMA operations done by one SPE into groups (called DMA tag groups), which are useful as they allow software to check or wait on the completion of commands.
The MFCs support naturally aligned transfers of 1, 2, 4, 8, 16 and multiples of 16 bytes, up to 16KB of data. MFCs also allow the use of DMA lists, where an SPU program passes to the MFC an LS address of list of eight-byte elements, each describing a DMA transfer. The DMA controller then issues all of the specified transfers while the SPU resumes execution.

Due to the existence of two distinct memory spaces, RAM and SPE local stores, context switching is a costly operation on the Cell. When this happens, the entire local store must be copied to main memory and replaced by the one belonging to the SPE thread that will now execute. Taking this into account, Cell developers tend to adopt a run to completion model instead of a multithreaded concurrent one.

Another execution concern is branch prediction. In the SPE, a wrongly predicted instruction causes a penalty of 18-19 clock cycles. A technique commonly used to avoid this issue is to avoid branches altogether by calculating both possibilities and then selecting the correct value with a native bit selection function called shuffle.

**Element Interconnect Bus** The EIB consists of four 16-byte wide data rings, two running clockwise and two counter-clockwise, with each ring transferring 128 bytes at a time. The internal maximum bandwidth of the EIB is 96 bytes per clock cycle, and more than one transfer can occur on each ring as long as their paths do not overlap.

**NUMA** In the BladeCenter QS20 and QS21, the two Cell processors operate in a Non-Uniform Memory Architecture (or Access) paradigm – NUMA –, where
each processor has its own memory bank but which are globally accessible to
SPE threads running on the machine. This allows for a single application to
use all 16 SPEs at once, but since inter-Cell memory accesses are slower than
on-chip ones, the programmer should split the data in two and programatically
use the SPE’s ID number (accessible in code) to determine which set of data is
the closest to it and work preferentially on it.

3 Existing Frameworks and Other Platforms

3.1 Introduction

The following section is divided into two main parts.

The first part of this section presents IBM’s SDK along with other frame-
works and tools available for the Cell, taking into account their limitations and
strengths.

Secondly, another high performance architecture with the same fundamental
characteristics as the Cell (heterogeneity, affordable versions, strong suite of
tools available), called CUDA, is presented. This platform will also be used as a
performance comparison.

**IBM Cell SDK and Simulator** IBM has developed, since the release of the
Cell processor, an SDK that supports the C, C++, and Fortran languages. It
consists of header files that provide access to assembly-level functionality in
DMA and SIMD operations, along with a suite of compilers and libraries.

These libraries include mathematical functions (BLAS for matrix operations,
MASS for general functions like sin and square root), communication and syn-
chronization helpers (DaCS (IBM, 2007c)) and a SPE runtime management API
for the PPE.

Another library worth considering is the Accelerated Library Framework
(ALF), designed to be integrated with the Eclipse IDE and that provides a
wizard-like interface for the data partitioning and overall application, generat-
ing code in the process.

Also available is a graphical full-syste Cell simulator (IBM, 2007d) that can
emulate a Cell processor, including all of the PPE, SPEs and memory, in a
functional simulation mode. It can also be used in performance simulation mode
to obtain exact timings for applications when developers have no access to Cell
processors.

The IBM SDK provides very powerful low-level access to the Cell’s potential,
since most of the interface provided consists of intrinsic functions, meaning that
they map one-to-one with assembly instructions. The experienced programmer
finds in these intrinsics the tools to fine-tune and thoroughly optimize the code.
The novice user, however, will have larger development and testing times than
in more traditional computational platforms.
IBM XL Multicore Acceleration for Linux V0.9 The XL is a common name for IBM compilers, available for most of the hardware architectures they sell. For the Cell, they have implemented a commercial version (IBM, 2007e) for Fortran, C, and C++. It has two main features, automatic code SIMDization and single source automatic generation of SPE and PPE code.

Regarding optimization, this compiler employs standard SIMDization techniques, like converting loops to SIMD operations (Eichenberger et al., 2006), and combines them with more advanced algorithms to take into account the critical path in the instruction pipeline and the dual issue capabilities of the processor. Optimization methods also include branch prediction since they are, as mentioned before, very expensive operations on the Cell.

The XL Multicore Acceleration V0.9 also includes the Toronto Portable Optimizer (TPO), which provides inter- and intra-procedural optimizations. If the programmer conforms to the OpenMP (OpenMP, 2008) programming model then the TPO will look for and identify parallelizable tasks and generate the PPE and SPE executables from a single source file.

Although this compiler is still in a development phase, it has the advantage of enabling the user to optimize his code where he has the skill to do it and rely on XL to improve performance on the rest of the application.

CorePy CorePy (Mueller, 2007) is a Python (Python, 2008) library to explore synthetic programming on the Cell platform. Synthetic programming consists of transforming (synthesizing) high-level code, usually in a scripting language, into a high-performance computational kernel. This project in particular combines low-level code with a high-productivity interpreted language, enabling the developer to produce applications in a short time whilst still exploring the computational performance of this processor.

In the CorePy environment, the developer is exposed to the synergistic processing units (SPU) instruction set as a module containing native Python functions, along with other common components. These functions are analogous to the IBM SDK’s intrinsics, since they map one-to-one with the SPU’s assembly instruction set. There are four main modules inside the CorePy library:

**ISAs** are the instruction sets for the architectures CorePy supports.

**InstructionStreams** are wrappers for synthetic programs, i.e., containers for sequences of low-level instructions that manage the specific tasks needed to execute them.

**Processors** are the executors of the synthetic programs, either synchronously or asynchronously (the latter provide real multithreaded execution).

**Memory Classes** provide support for describing memory and moving data across memory boundaries.

Programs written with CorePy can interact with any other data available to the Python interpreter, so it is possible to use synthetic programs along with other Python libraries.
**MPI Microtask** Message Passing is a form of parallel programming that assumes three characteristics. First, the programs are in a “nothing shared” environment, i.e., they have independent memory spaces; second, all communication is made through a set of available message forms, transmitted generally in an asynchronous fashion and finally, data transfer between files requires cooperative operations, in the sense that for every `send` operation a matching `receive` must be executed.

One message passing specification is the Message Passing Interface (MPI) (MPI, 2008). It is a definition of an interface that has become a de facto standard in cluster computing (Al Geist et al., 1996).

The MPI Microtask project, a programming model proposed by IBM, implements the MPI specification. It relies on the programmer to divide the application and data into small tasks that fit in the local store of an SPE and analyzes the calls to the MPI interface to generate a fast runtime environment. The framework’s activity occurs at two different times of application development.

In a first stage, which happens at compile time, all tasks are identified and decomposed into basic tasks, elements of computation with communication happening only at their beginning and end, similar to computational kernels in stream programming languages. Basic tasks are grouped by inter-dependency, creating clusters of tightly coupled elements of computation in order to reduce the likelihood of a context switch. The compiler derives these dependencies from a dependency graph, where a dependency translates to MPI communication between two tasks, and uses this knowledge to identify parallelizable tasks and establish an execution schedule that explores the Cell architecture.

The second stage occurs at runtime, where the framework handles the message buffers and transmission defined by the programmer, along with the required synchronization and context switching of executing tasks.

This model brings benefits to the developers regarding the synchronization and communication between computational elements, as they are concealed or simplified, respectively, reducing the burden that is managing DMA transfers explicitly and programming all the control that is usually in one of the PPE threads. On top of that, it supports Multiple Program Multiple Data applications. However, there are still some drawbacks in using this tool (at least in its current version). The main issue is that it is still the programmers’ responsibility to clearly define the microtasks in the application and, more importantly, their dependencies. The preprocessor is not yet mature enough to do these tasks automatically.

**Rapidmind** RapidMind is a commercial “development and runtime platform that enables single threaded, manageable applications to fully access multicore processors” (Monteyne, 2007), and it consists of a C++ library and a set of header files that define new C++ types. The applications that use it are expressed as a set of computations applied to arrays (RapidMind arrays). Cur-
rently, backends exist for x86 multicore and multiprocessors, GPUs, and the Cell.

Developers use the platform first by replacing their existing C++ types with the RapidMind ones. This alone brings some parallelization, since operations on these types take advantage of the available processor cores for performance.

The next step is to express parallel operations using the Program concept. These are like C++ functions but only made machine-specific during the first execution of the application by the platform.

Apart from these two methods, RapidMind has features that enable more complex optimization techniques. They include support for a reduction operation, like the one discussed ahead in MapReduce, support for control flow with irregularity in execution times, and accessors to allow for customization on how data is read and written.

For performance tuning, the creators have also included in the platform profiling and data inspection tools, that record the duration of different operations and how each array of data was manipulated along the execution of the program.

Results published (Monteyne, 2007) by the RapidMind team indicate that the tool enhances performance on each processing element and fully uses the multiprocessor environment as a whole. However, and as expected since it favors programability over performance, the platform loses in a comparison with algorithms implemented with the low level frameworks available for that specific platform. On the other hand, low level code is inherently less portable and more cumbersome to develop. RapidMind seems very easy to use and we find particularly attractive that algorithms, when ported to this platform, are kept conceptually serial, instead of explicitly parallel like in other frameworks.

Sequoia Sequoia is a project by Stanford University for parallelization in machines where multiple address spaces are exposed to the programs. Its abstract machine model is one of a tree of memories, where the data is divided and control flow is done along the memory hierarchy.

At each tree level other than 0 (leaves), control tasks, called inner tasks, partition the data and do little or no computation. They enable parallelism by using special language constructs, namely mappar and mapreduce, to define which invocations should be made on which chunks of data and the kind of dependency they share.

Level 0 is the level that is mapped to the processing elements (PE), and is where tasks called leaf tasks do the actual processing. So, inner tasks express the decomposition of the algorithm and work usually on large datasets that are required to fit in main memory, while leaf tasks do the “actual computation” on smaller datasets that must fit in the PE’s memory. Finally, there is a third kind of task, the external task, which is a language construct to allow developers to call functions not written in Sequoia, as long as they conform to the Sequoia API.

All communication between tasks, memory allocation and task invocation and scheduling is generated by the compiler, depending on the architecture speci-
ified in a XML mapping file, and, in the particular case of the Cell processor, techniques like double buffering are used to optimize performance.

Sequoia is best suited for data-centric algorithms, and the mapping of tasks to levels can be cumbersome in large projects as an unexperienced programmer might not be able to identify the most high-performing mapping.

**Mercury Framework** The Mercury Multicore Framework (MCF) library (Bouzas, Cooper, Greene, Pepe, & Prelle, 2006) is a commercial product to program the Cell processor where, the PPE plays the role of manager and runs the control code, creating workers that run on a 12KB kernel in the SPEs.

Workers are often grouped into teams to perform a specific task at the manager’s command, and are synchronized using barriers or semaphores (both provided by the library). Data transfers between the workers and the manager are expressed using a concept of channels, where the participants involved simply issue “put” and “get” commands to communicate.

Data partitioning is made relatively simple, since the programmer must only allocate the memory on the manager code and specify the size of a tile – which is the basic unit of data sent to the worker – together with the number of tiles processed by each worker at a time and the overlap between each worker’s data set.

The MFC is sold by itself or with the Mercury Multicore Plus SDK, which also contains a Scientific Algorithm Library, a performance analyzer called Trace Analysis Tool and Library, an image processing library and a SPE Assembly Development Kit that is an optimizer for SPE assembly.

With this framework, there is a much simpler interface to data transfer between different processors than the one in IBM’s SDK, and data partitioning seems much easier to do. However, it is admissible that the PPE might become a performance bottleneck due to the use of the FOE model.

**MapReduce** MapReduce (Dean & Ghemawat, 2004) was created by Google and is a programming model and an associated implementation for processing and generating large data sets. All input data is organized according to a key/value scheme, leaving to users only the task of specifying a map function that processes a key/value pair to generate a set of intermediate key/value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key.

The runtime system partitions the input, schedules the program’s execution across a set of processors and manages the required inter-processor communication.

In broad strokes, the design for this particular implementation (Kruijf & Sankaralingam, 2007) on the Cell is described in five phases.

The algorithm starts with a Map stage, where data is transferred to the SPEs’ local stores and the user-specified Map function is executed on it. The data is partitioned and pointers to these sections are sent to each of the SPEs, which in turn fetch the maximum amount that fits into their local stores. The
output of each Map invocation is sent by DMA to main memory and the PPE is notified of completion.

The Partition part of the algorithm groups identical keys and into the same hash table bucket. It is done mostly by PPE helper threads and takes place in global memory and while the Map phase is still executing. The hash table buckets are structured as linked lists of buffers sized to fit in the local stores of the SPEs.

Each of the partitions is sorted in a third step called Quick-sort, with the work being distributed to the SPEs as much as possible. For partitions larger than the local store size, each of the sorted buffers must be merged to form a single sorted partition. This is done in the Merge-sort step, and it is executed mostly on the SPEs but also on the PPE.

Finally, the Reduce phase is quite similar to the Map operation, with the framework executing the user-specified Reduce function on all the values available for each key and returning to memory buffers with key/value pairs. These buffers are passed on to the user and are identical in structure to the ones input to the Map phase, enabling chained uses of the MapReduce tool where the output of one invocation is used as input in the next one.

This model is potentially very simple to use. Once the potential difficulty of transforming an algorithm into one defined by map and reduce operations has been overcome, the framework itself is quite user friendly.

However, there are some limitations to this implementation. First, it is necessary to consider the cases when an application simply is not portable to the model. Then, there are the cases where the Map operation is computationally weak or where the ratio between the map operation intensity and the number of intermediate keys generated is too low, making the PPE the serialization point and performance is critically hit.

3.2 Comparison

The frameworks presented have strengths and limitations, which may determine their usefulness in solving a particular type of problem. On top of that, more generic limitations arise, like availability or cost.

**Programmability** RapidMind and Sequoia seem to be the favorites when it comes to programmability because they abstract all of the multiprocessor characteristics from the programmer. MPI and Mercury ease communication but leave exposed the parallelism in the application, with the creation and management of runtime elements. Furthermore, MPI Microtask and IBM XL require expertise on the MPI and OpenMP paradigms, respectively.

MapReduce also exempts the programmer from dealing with communication and helps with memory allocation. However, it is built on top of the IBM SDK so some caveats exist, namely in memory alignment (there are some alignment requirements for DMA operations) and performance (memory-mapping input files, for example).
Using CorePy or the IBM SDK by themselves can be a strenuous task as they both provide only very low-level access to the Cell.

**Optimizations** Several of the analyzed tools, namely RapidMind, IBM XL and Sequoia, optimize the application automatically via SIMDization, loop and memory transfer optimization techniques. MPI has automatic optimized scheduling of SPE tasks and, like MapReduce that optimizes communication, requires the programmer to optimize his own code.

Viewing optimization from another perspective, we notice that Rapidmind and Sequoia allow for little to none fine-tuning, while an experienced Cell programmer using CorePy or IBM’s SDK can optimize at assembly level taking into account the semantic particularities of the data. The issue is how experienced a programmer must be to out-optimize an equivalent Sequoia/RapidMind algorithm.

**Completeness** Most of the described platforms demand the algorithm or data to conform to a specific model or be organized a certain way. So, it may occur that a particular algorithm cannot be implemented in the MapReduce paradigm, or in the OpenMP standard. Therefore the programmer that intends to use one of these tools should first validate the applicability of these metamodels to the particular application to implement. For example, in a project where the processing of a chunk of information depends on “adjacent” data, the MapReduce algorithm may be hard to apply since each unit is processed independently.

**Commercial and Maturity Issues** Apart from the previous points, issues like the availability and maturity of these frameworks are very relevant. For instance RapidMind, although a very attractive platform, is prohibitively expensive and will not be further investigated, and Sequoia shows several shortcomings since some important language constructs remain unimplemented.

MapReduce development has ceased (but some support exists) and, although some changes that would bring much better performance have been identified, they will not be implemented in the near future. Also, MPI Microtask is not available to the public, as only a simple prototype has been implemented as proof of concept. Finally, some tools are only compatible with the IBM SDK version 2.1, which may cause some limitations to users who wish to take advantage of newer features.

### 3.3 CUDA by NVIDIA

NVIDIA, a graphics card maker, has created a development platform to leverage the power of their GPUs in general-purpose (i.e. not just graphics-related) applications. Compute Unified Device Architecture (CUDA) is a suite of compiler and tools to enable developers to code algorithms in a streaming model and deploy them in recent GPUs. The NVIDIA GPU GeForce 8800GTX, which supports CUDA, is organized into 16 Streaming Multiprocessors (SM), each with 8
processing units, with a global theoretical peak performance of 345.6 GFLOPS for single precision data. All these processing units are, like the Cell SPEs, SIMD-capable and, also like what happens with the Cell, have much higher performance on single precision floating point operands than with double precision. This graphics card was available for testing, so it was used to provide comparative data.

To use CUDA, the developer allocates memory on the device and copies the data to those locations. Then, he runs GPU threads, which then execute (usually small) computational kernels on the data in parallel (NVIDIA, 2008). Only one particular kernel can be executed by a device at a given time, but many instances of that same kernel can run simultaneously on different data.

CUDA provides simple extensions to the C language in the form of annotations to distinguish between GPU and CPU code, along with functions to manage the memory allocation and transfer between host (CPU) and guest (GPU) devices. To define a computational kernel, the programmer annotates a C function with the \texttt{global} declaration specifier and to invoke it the syntax element is added to the function name with two arguments, specifying the organization of the work into threads in blocks and of blocks in the grid respectively.

Apart from these language extensions the CUDA platform defines, to take advantage of the SIMD features of the processors, vector data types with 1, 2, 3 and 4 dimensions. These data types can be manipulated with the usual arithmetic operators (+, −, /, ∗).

Like the Cell and its NUMA architecture, NVIDIA’s latest products are capable of being used cooperatively with their Scalable Link Interface (SLI) technology (NVIDIA, 2009), assuming the GPUs are mounted on a SLI-compatible motherboard. According to NVIDIA, the SLI connectors allow for a maximum transfer rate between cards of 1GB/s, and a maximum of 3 cards can be interconnected.

4 Case Studies

So far in this document the Cell has been described both through an overview of its hardware characteristics and trough its programming requirements and available tools. But to evaluate its usefulness, especially in the natural language processing field, some common problems in the area were approached using Cell-based solutions.

This section consists of a description of the applications implemented on the Cell, more precisely their purpose, architecture and optimizations. The first application presented was implemented in a more naïve fashion, and the remaining two were done taking into account some optimization techniques like loop unrolling, multibuffering, computation/data transfer overlapping, and others.

In the next section the performance of the implementations described here will be analyzed with this in mind, in order to determine the degree of optimization needed to obtain significant performance gains in relation to similar solutions on other platforms.
4.1 Neural Networks

Neural networks (Jain, Mao, & Mohiuddin, 1996) are a tool that is frequently used in the field of natural language processing. One example is an application (Santos Meinedo, 2008) built with the purpose of identifying jingles in an audio stream to detect the beginning/end of broadcast news, along with commercial breaks and filler segments.

This jingle detector consists of a pipeline of different tools. In the first step, features are extracted from the audio stream. From each sample, a total of 26 parameters are extracted into a feature vector.

The resulting information is input to the second block, a neural network classifier of the type Multi-Layer Perceptron (MLP) that classifies these feature vectors and outputs the probability of each frame being a certain type of jingle. To increase classification accuracy, evaluation of one sample takes into account the value of adjacent ones, called context frames.

The output of this step is then smoothed by a median filter and compared to a threshold value (pre-determined).

The neural networks described are implemented using a series of matrices and linear algebra operations. Each layer of the network is represented by a weight matrix and the input/output values are also stored into matrices. So, to calculate the output of a layer, three matrices are involved in a $A \times B + C$ operation (called SGEMM in the Basic Linear Algebra Subprograms – BLAS (Blackford et al., 2001) interface):

**Input (A)** This is a rectangular $(m, n)$ matrix, where $m$ is the number of frames currently being processed and $n$ is the number of perceptrons of the previous layer or, in the case of the first layer, the number of features that represent the set of context frames.

**Weights (B)** A rectangular matrix with as many lines as the number of outputs (perceptrons) from the previous layer, and as many columns as the number of perceptrons in the current layer.

**Bias (C)** To apply a bias to the perceptrons output, this matrix is added to the product of the other two.

After this operation a sigmoid function is applied to each of the output values for normalization, and the result is fed to the next layer as input.

IBM’s SDK provides a BLAS library that is partially optimized for the Cell. Since one of the optimized functions was the SGEMM routine, already used in the original implementation of the jingle detector, this library was chosen to port the application to the Cell. Since one of the goals of this work is to quantify in some way how much programming expertise is needed to attain significant speedups on the Cell, initially no alteration to the jingle detector code was made, the application was simply linked with the new BLAS library instead of the one it had been developed with.
4.2 Matrix Multiplication Server

Matrix multiplication is a common but computationally expensive operation in natural language systems. As seen before, it was the SGEMM function that was most useful for neural network programs.

Different implementations of the operation exist in the many linear algebra libraries available, including the IBM BLAS library provided with the Cell SDK. Having no knowledge of how this function is implemented by IBM, we decided to investigate the performance gains of a highly optimized, low level, version of this linear algebra operation in comparison to IBM's BLAS.

Since the original application this case study was based on showed promising results, a choice was made to create a server that could be invoked by code running on other platforms, using an RPC-like communication protocol.

This section describes the architecture and optimization of the application. More precisely, it receives three matrices (A, B and C) as input and then performs the operation A*B+C, destructively changing matrix C. This application was based on Daniel Hackenberg’s (Hackenberg, 2008) implementation.

Data Organization  In this application the input data is partitioned into square blocks of 64 single precision floating point elements. These may be organized in memory in the traditional (C language) row major layout (RML) or in block data layout (BDL). This choice influences performance since when RML is used all DMA operations must use the scatter/gather facilities that DMA lists provide (recall 2.3), while in BDL an entire block may be transmitted in a single DMA operation (Kistler, Perrone, & Petrini, 2006).

Work Partitioning  In this application the PPE has only setup and statistics collection duties, with all the calculations being performed on the SPEs. To improve the (calculation time)/(DMA latency) ratio, each SPE works on four output blocks at a time (equivalent to a square 128x128 element block).

Each output block is assigned to a SPE according to an algorithm based on the block and SPE number. Blocks are numbered first horizontally and then vertically, and each SPE processes the blocks whose number is equal, modulus the number of SPEs used, to the block number.

Computational Kernel  The code utilized in this application was the assembly language implementation by Daniel Hackenberg, which was presented by the author as being highly optimized. It makes frequent use of the Fused Multiply-and-Add operation, which performs two arithmetic operations on four floats per cycle, meaning that the six available SPEs can execute 48 operations per clock cycle, in total. Also, its main loop is unrolled, providing additional optimization.

4.3 Euclidean Distance Calculator

In the field of music analysis, various techniques have been used for information extraction (Logan, 2000; Makhoul, 1975). At INESC-ID there is such a project,
led by Prof. David Matos and Ricardo Santos, that extracts features from songs, performs similarity detection between them, and finally groups them to obtain information such as location and duration of choruses, openings, and similarities to other songs.

The similarity detection portion of the algorithm works by calculating a distance metric (currently the Euclidean distance) between feature arrays that represent each song. Since it is somewhat alike to the matrix multiplication problem, it was re-implemented on the Cell with a similar strategy. However, the matrix multiplication implementation dealt with square matrices of around 4000 by 4000 elements, while in this problem the typical input is approximately 46 (the number of features extracted) by 30,000 elements. Therefore, the scheduling of work units was changed in this project.

Given that there are only 256MB of available RAM in the PS3 (the Cell instance more easily accessible to us), special care was taken in designing a processing workflow that operated efficiently without exceeding the memory limit, otherwise the application would incur page swapping and stall on disk I/O.

The SPEs process areas of the matrix called “lines” that are 128 rows by \( m \) columns, where \( m \) is the length of the side of the matrix. The atomic work unit for a SPE consists of four blocks from input (two from each matrix) and four blocks to output. This is similar to what happened in the matrix multiplication application, however in this case the two blocks from one of the input matrices are reutilized for the processing of the whole “line” and remain in the SPE for this entire period.

The algorithm for each SPE is presented in Figure 3.

When a SPE finishes a line, it must wait for PPE notification. This is because of another strategy derived from the small memory available, where the PPE periodicallyflushes the output buffers to the result file.

The PPE maintains two buffers, each large enough to hold one output line from each SPE. To reduce the stall time when the SPEs finish one iteration and the result must be written to disk, a double buffering technique was used. This translates into the result of one iteration being flushed while the next is being calculated. To accommodate this, the SPEs receive two addresses for main memory output as argument, and after each line is processed the designated output pointer is swapped.
input: Address of input matrices (MainA, MainB), the id of the SPE (spe_id), number of SPEs being used (num_spes) and the length of the output matrix side in blocks (num_blocks)

\[ m \leftarrow 2 \times \text{spe}_{\text{id}}, \quad n \leftarrow 0; \]

// the 2nd argument of DMAGetBlock is the DMA Tag of the transfer
LocalA_a \leftarrow \text{DMAGetBlock}(\text{MainA} \ m, 0, 1);
LocalA_b \leftarrow \text{DMAGetBlock}(\text{MainA} \ m + 1, 0, 1);
LocalB_a \leftarrow \text{DMAGetBlock}(\text{MainB} \ 0, n, 1);
LocalB_b \leftarrow \text{DMAGetBlock}(\text{MainB} \ 0, n + 1, 1);
for m \leftarrow 1 \text{ to } \text{num}_{\text{blocks}} \text{ do }

// the last column will be treated separately
for n \leftarrow 1 \text{ to } \text{num}_{\text{blocks}} - 1 \text{ do }

WaitForDMATag(1);
LocalC_a \leftarrow \text{CalculateDistance}(\text{LocalA}_a, \text{LocalB}_a);
\text{DMAStoreBlock}((\text{LocalC}_a, m, n));
WaitForDMATag(2);
LocalC_b \leftarrow \text{CalculateDistance}(\text{LocalA}_a, \text{LocalB}_b);
\text{DMAStoreBlock}((\text{LocalC}_b, m, n + 1));
WaitForDMATag(3);
LocalC_c \leftarrow \text{CalculateDistance}(\text{LocalA}_b, \text{LocalB}_a);
\text{DMAStoreBlock}((\text{LocalC}_c, m + 1, n));
LocalB_a \leftarrow \text{DMAGetBlock}(\text{MainB} \ 0, n + 2, 1);
WaitForDMATag(4);
LocalC_d \leftarrow \text{CalculateDistance}(\text{LocalA}_b, \text{LocalB}_b);
\text{DMAStoreBlock}((\text{LocalC}_d, m + 1, n + 1));

WaitForDMATag(1);
LocalA_a \leftarrow \text{DMAGetBlock}(\text{MainA} \ \text{m}_{\text{fornextline}}, 0, 1);
WaitForDMATag(3);
LocalC_e \leftarrow \text{CalculateDistance}(\text{LocalA}_a, \text{LocalB}_a);
\text{DMAStoreBlock}((\text{LocalC}_e, m, n));
WaitForDMATag(2);
LocalC_f \leftarrow \text{CalculateDistance}(\text{LocalA}_a, \text{LocalB}_b);
\text{DMAStoreBlock}((\text{LocalC}_f, m, n + 1));
if m_{\text{fornextline}} < \text{num}_{\text{blocks}} \text{ then }

\begin{align*}
\text{LocalA}_b & \leftarrow \text{DMAGetBlock}(\text{MainA} \ \text{m}_{\text{fornextline}}, 0, 1) \\
\text{WaitForDMATag}(3); \\
\text{LocalC}_g & \leftarrow \text{CalculateDistance}(\text{LocalA}_b, \text{LocalB}_a) \\
\text{DMAStoreBlock}((\text{LocalC}_g, m + 1, n));
if m_{\text{fornextline}} < \text{num}_{\text{blocks}} \text{ then }

\begin{align*}
\text{LocalB}_a & \leftarrow \text{DMAGetBlock}(\text{MainB} \ 0, 0, 1) \\
\text{WaitForDMATag}(4); \\
\text{LocalC}_h & \leftarrow \text{CalculateDistance}(\text{LocalA}_b, \text{LocalB}_b); \\
\text{DMAStoreBlock}((\text{LocalC}_h, m + 1, n + 1));
if m_{\text{fornextline}} < \text{num}_{\text{blocks}} \text{ then }

\begin{align*}
\text{LocalA}_b & \leftarrow \text{DMAGetBlock}(\text{MainA} \ \text{m}_{\text{fornextline}}, 1, 2) \\
\text{LocalB}_b & \leftarrow \text{DMAGetBlock}(\text{MainB} \ 0, 1, 2); \\
\text{m} & \leftarrow \text{m}_{\text{fornextline}};
\end{align*}

\end{align*}

SignalPPEViaMailbox(); // notify the PPE of line completion
GetSignalFromPPEViaMailbox(); // wait for PPE signal to proceed

Fig. 3. SPE-side algorithm for the Euclidean Distance project
5 Results and Evaluation

5.1 Goals and Methodology

This section consists in a description of the performance and scalability tests done on the previously described case studies, in an attempt to ascertain what types of problems and, more specifically, the problem dimension intervals where the programs had considerable speedups.

The tests presented here are intended to define how the Cell performs in comparison with both a homogeneous processor machine and a GPU. Performance will be evaluated as raw execution time and also as number of operations per second (GFLOPs).

The results in this section are based on timings on a number of runs of the applications over several platforms, and we have chosen to display them either as execution time versus data size/CPU, as GFLOPs versus data size/CPU or as relative speedup of one platform using another as a comparison basis.

5.2 Testing environment

In all tests, the Cell instance used was a PS3 running the Fedora 7 operating system, with 256MB of RAM of which 32MB were reserved for huge page allocations. This particular implementation of the Cell only has 6 SPEs available to the programmer instead of the standard 8, but in all other relevant aspects (memory bandwidth, EIB clock speed, etc.) it is identical to the BladeCenter QS20 and QS21 versions of the processor. For the matrix multiplication case study the implementation was also run on QS20 machines, with 1GB RAM and 16 SPEs, provided by the Georgia Tech Institute.

To provide a comparison, two machine configurations were used. One, configuration A, consisted of an Intel Q6600 Core2 Quad CPU, with 8GB of DDR2 RAM at 600MHz. The other machine, configuration B, used the same processor, but with an NVIDIA 8800GT graphics card and a higher memory clock rate of 800MHz. In configuration A the BLAS library used was Intel MKL version 10.0.5.025 and in configuration B the tests used the cuBLAS library provided with the CUDA SDK.

5.3 Neural Networks

As explained before, this application was first ported to the Cell in a naïve fashion, replacing only the BLAS library used in the original implementation with IBM’s version and leaving the rest of the code unchanged.

This program was tested using both a network with 9 context frames of 26 features, using two hidden layers of 75 perceptrons each and an output of 4 values, and a network with 13 context frames, using two hidden layers of 2000 perceptrons and an output of 39 values. From here on forward the first will be called the small network and the latter the big network.
IBM’s BLAS library on the PS3 recognizes environment variables that can be used to control the launching of SPEs and control memory allocation (see (IBM, 2007a), Chapter 3). The variables used by us were:

**BLAS.NUMSPES** This variable controls the number of SPEs to be used by the library. The application was run with 1 through 6 SPEs.

**BLAS.USE_HUGEPAGE** Specifies if the library should use either heap or huge pages for temporary memory allocations, if it has a value of 0 or 1 respectively. This memory is used in BLAS level 3 routines as extra space to reorganize matrices. The application was tested with either option for each variation of SPE number.

**BLAS_SWAP_SIZE** To reduce the time spent in memory operations across several invocations to level 3 routines, the library can reuse the space allocated in the huge pages. In our tests with BLAS.USE_HUGEPAGE active the swap size was set to 16384KB (16MB - the size of one huge page as defined in the system).

The small network was run 100 times per test on the PS3 and configuration A with a test input of 420,000 frames. The results are shown in Tab. 1.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>PS3</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1 SPE</td>
</tr>
<tr>
<td>Time (s)</td>
<td>3.26</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 1.** Timing results in seconds for the small neural network

Similarly to the small network tests, the application was run on an input of 420,000 frames for the big network. However, given the long running time of each non-huge page trial, only 20 executions were done per SPE without huge pages. The results are shown in Tab. 2.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>PS3</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1 SPE</td>
</tr>
<tr>
<td>Time (s)</td>
<td>178.94</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 2.** Timing results in seconds for the big neural network

In Tab. 1 there is an evident increase in execution time as more SPEs are used. This increase is attenuated when huge pages are used, but is still noticeable. According to IBM BLAS documentation, SPE threads created during the first
invocation to a BLAS function are reused through subsequent calls to the library. Therefore, the overhead of thread creation is not the main factor behind the increase.

One factor that clearly has a big influence on performance is memory allocation. As mentioned before, the library uses extra memory space for optimizations. Reusing memory allocations for this data reorganization lessens the performance hit, as seen in Tab. 1 and Tab. 2. Considering that in these tests there were a total of 13125 calls to the matrix multiplication routine in the BLAS library, the repeated allocation and freeing of memory blocks becomes a significant performance grinder. Supporting this are the results for runs using huge pages and swap space, which showed up to 80% faster times, which supports the theory that one of the main issues with this type of problem – many calls to SGEMM with small matrices – is this extra space allocation overhead, when not using huge pages.

Using huge pages and swap space overcomes the memory allocation problem, but even when these features were active, there was still a loss in performance for small matrices when more SPEs were used. This indicates that there is another overhead, internal to the BLAS library, that is related to the management of running SPEs and their associated resources. With the big matrices run this effect is not as noticeable, but still present.

This case illustrates how the Cell may not be a friendly platform for newcomers. A programmer simply porting an application to this new platform in search of easy acceleration of his code by using one of the level 3, Cell-optimized, BLAS functions, may find that some reengineering of the original implementation and experimentation with the available tools (e.g. huge pages) is needed to obtain performance gains.

5.4 Matrix Multiplication Server

Tests for this application were done on all three platforms. However, since both the PS3 and the GPU have strong memory limitations, the maximum matrix dimensions tested on these machines were lower. In the case of the PS3, only matrices of up to 4096x4096 were considered for the results presented as the next larger size (4224x4224) caused memory swapping and therefore much worse results. The results for configuration B do not show values over 6272x6272 since larger matrices hit the memory limit for the device and originated program errors.

The average execution time for each matrix size and number of SPEs used, displayed in Fig. 4, especially the clear increase in performance for 6 SPEs and 4096x4096 matrices, provide interesting indicators. With the increase of computational power from 1 to 6 SPEs, the time needed to calculate the product of two 4096x4096 matrices was reduced by more than 70%, which suggests that an overhead exists that reduces the performance-wise scalability. This overhead lies in the DMA transfer time to SPE execution time ratio, with the SPEs occasionally stalling while input data is transferred to the LS or results are sent DMA-ed.
to main memory. These results support the idea that the Cell needs a very high arithmetic intensity (Harris, 2005) to provide good performance.

**Fig. 4.** Variation of matrix multiplication times on the PS3 against variation of matrix size and number of SPEs

To make a comparison, we have presented timing and performance graphs in Fig. 5 and Fig. 6. There are several aspects worthy of note:

First, although Configuration A’s performance is the worst out of the three tested for large sizes, it was the one with the smallest variation in computing speed. This was because performance, in this case, hit the CPU wall. Configuration B showed poor results for small matrix sizes, caused by the overhead of thread and block initializations, but for larger sizes the GeForce’s speed trumped these fixed costs. The execution times indicate that the GPU operates on a much smaller problem space, memory-wise.

The PS3 was the most limited platform of the three in regard to the amount of available memory. However, within these limits it showed results close to the theoretical peak speed for the Cell (25 GFLOPS per SPE). The QS20 showed higher speeds, proportional to the number of SPEs used.
5.5 Euclidean Distance Calculator

Since the main objective for this implementation was to test the reutilization and how well the problem adjusted to the Cell, the comparison for this application was simply a naïve, single-threaded, application implemented in C++ and run on the configuration A machine. The main purpose of these tests was to determine if an adaptation of the matrix multiplication scheduling could still provide an efficient implementation, since the two problems are quite similar.

With input files varying from 2304 samples to 26,000 (using 64 features per sample on the PS3 and 40 on configuration A), the application was run 30 times for each input size. On the PS3 6 SPEs were used and memory allocation was made from heap and not in huge pages.

The results are displayed in Fig. 7.

The application on the PS3 outperformed the naïve implementation and showed inferior complexity growth. However, a careful analysis reveals that the PS3 never exceeded the 25 GFLOPs mark. Using VampirTrace, a tracing library available in a Cell version, the PPE was identified as the bottleneck. According to the trace information, an average of 70% of SPE execution time was spent waiting for mailbox communication. More specifically, waiting for the PPE to signal the start of processing of another set of lines.

In Fig. 8 is depicted a graph of a part of the execution created using Vampir, a trace analysis tool.

In Fig. 8 the full lines connect the instant when a mailbox message was sent and the instant when it was read from the channel. The oblique lines show the
Fig. 6. Variation of platform computational speed (in GFLOPS) when the size of matrices increases

SPEs signaling the PPE they are ready to process another set of lines, and the almost vertical full lines (visible at around 10s, 24s, 30s, 32.5s and 35s) show the PPE informing the SPEs that processing can resume.

SPU cycles are mostly spent waiting for a PPE signal (the areas marked with green SPU_MBOX_WAIT). These waits occur because the PPE spends a greater amount of time writing buffers to disk than it takes the SPEs to process their assigned lines of data. Since the results and tracings shown are for an implementation that uses two output buffers, a solution for this problem would be to increase the number of output buffers, so that the SPEs could keep working even during writes to disk. This parameter should be configurable, as for different numbers of samples the optimal number of output buffers may vary.

In spite of these problems, the application was implemented in a way that it can use a custom computational kernel, is easy to use, and provided better results for an end-user than a naive implementation on a homogeneous processor computer.

6 Conclusions

The applications tested provided results that shed a light on the applicability of the Cell in some areas of computing. The neural networks case represented the naive approach to the Cell, where no SPE or optimized code was written by the programmer, and only the porting to the Cell-optimized BLAS library was
done. The results showed that it may not be trivial or immediate to increase an application’s performance simply by moving it to a Cell processor. In this case, there were not enough calculations done on the SPEs to mask the overheads of memory allocation and processor management. The SPE setup overhead can also be mitigated via optimization techniques like the reuse of resources done by the BLAS library across invocations and can be made less significant when there are large volumes of data to process.

Analysis on the matrix multiplication reveals this to be a problem well adjusted for the Cell’s capabilities. It has a simple scheduling scheme and a good balance between the amount of data transmitted between PPE and SPE and the calculations done on those chunks of data. However, the code used to attain the performance gains presented was quite complex, especially the computational kernel, and for large matrices there may be cases where using cuBLAS on a GPU may be a good compromise between high performance and programming effort.

The current version of our BLAS server performs no type of load balancing or job queueing, but in the future this application could become a simple interface for users on other platforms (like Configuration A used for testing) to perform fast matrix multiplication. In this perspective, of the user and not the programmer, the PS3 would provide the performance gain at little expense in terms of programming complexity. This scenario, however, is only true in the
case that network speed is high enough so that the transmission time does not outweigh the decrease in processing time.

The euclidean distance case showed how the Cell’s complexity and limitations can hinder even an optimized program. Although care was taken to overlap DMA transfers with computation and to program a SIMD-ized kernel, performance was lower than expected due to the low memory capacity of the PS3 and the amount of control code left on the PPE.

Reutilization of the scheduling from one Cell program to another is a possible solution to lessen the programmer’s work complexity. However, two distinct problems may have data dependencies that require an entirely new scheduling algorithm. On the other hand, applications programmers whose data is organized and processed in a similar manner to the euclidean distance problem, for example, could use this application’s code and simply program a new computational kernel, as it was isolated from the rest of the SPE code for flexibility.

One other factor of more and more relevance today is the energy consumption of computers. Cell-based clusters are currently those with the lowest Watt per GFLOP ratio ¹.

Concluding, the Cell is, as a platform, interesting to different types of stakeholders, each with their concerns. To the end users of optimized applications, the

¹ www.green500.com
Cell provides both the advantage of having cluster software like MPI available and therefore being able to be integrated into the existing resources as well as a significant performance increase. To developers, it requires a strong analysis of data organization and significant amount of effort put into learning the platform and designing the application. Finally, for the financial investors, it provides good ratios in terms of GFLOPS per $ and GFLOPS per Watt, helping contain the maintenance costs of a cluster or supercomputer.

References


OpenMP. (2008). *The OpenMP specification for parallel programming.* (http://www.openmp.org/ (Visited on 2008/01/07))


27