# Linux on the Cell processor

## Linux Kernel Hacking Free Course — IV edition

Paolo Palana

System Programming Research Group — University of Rome Tor Vergata

*palana@sprg.uniroma2.it*

April 23, 2008

# What is Cell?

# What is Cell?

- Cell is a multiprocessor system on single chip developed by IBM in collaboration with Sony and Toshiba

# What's New in Cell?

# What's New in Cell?

- Many other multiprocessor architectures today:

# What's New in Cell?

- Many other multiprocessor architectures today:
- Intel Core duo
- Intel Xeon
- AMD Athlon 64 X2
- AMD Opteron

# What's New in Cell?

- Many other multiprocessor architectures today:
- Intel Core duo
- Intel Xeon
- AMD Athlon 64 X2
- AMD Opteron
- All homogeneous architectures

# What's New in Cell?

- Many other multiprocessor architectures today:
- Intel Core duo
- Intel Xeon
- AMD Athlon 64 X2
- AMD Opteron
- All homogeneous architectures
- Cell has a non homogeneous architecture

# What's New in Cell?

- Many other multiprocessor architectures today:
- Intel Core duo
- Intel Xeon
- AMD Athlon 64 X2
- AMD Opteron
- All homogeneous architectures
- Cell has a non homogeneous architecture
- One general purpose processor (*PPE*)

# What's New in Cell?

- Many other multiprocessor architectures today:
- Intel Core duo
- Intel Xeon
- AMD Athlon 64 X2
- AMD Opteron
- All homogeneous architectures
- Cell has a non homogeneous architecture
- One general purpose processor (*PPE*)
- Eight special purpose processors (*SPE*)

## What's New in Cell?

- Many other multiprocessor architectures today:
- Intel Core duo
- Intel Xeon
- AMD Athlon 64 X2
- AMD Opteron
- All homogeneous architectures
- Cell has a non homogeneous architecture
- One general purpose processor (*PPE*)
- Eight special purpose processors (*SPE*)
- To fully exploit the Cell architecture a new programming approach is required

# Architectural Overview

# Power Processor Element (*PPE*) architectural overview

The Power Processor Element:

# Power Processor Element (*PPE*) architectural overview

The Power Processor Element:

- The main processor: it executes both the operating system and the general purpose applications, and it spawns tasks to *SPE*

# Power Processor Element (*PPE*) architectural overview

The Power Processor Element:

- The main processor: it executes both the operating system and the general purpose applications, and it spawns tasks to *SPE*
- A dual-threaded general purpose processor

# Power Processor Element (*PPE*) architectural overview

The Power Processor Element:

- The main processor: it executes both the operating system and the general purpose applications, and it spawns tasks to *SPE*
- A dual-threaded general purpose processor
- Based on a 64 bit RISC architecture conforming to the PowerPC Architecture version 2.02

# Power Processor Element (*PPE*) architectural overview

The Power Processor Element:

- The main processor: it executes both the operating system and the general purpose applications, and it spawns tasks to *SPE*
- A dual-threaded general purpose processor
- Based on a 64 bit RISC architecture conforming to the PowerPC Architecture version 2.02
- Has vector/SIMD multimedia extensions

# *PPE* simple block diagram

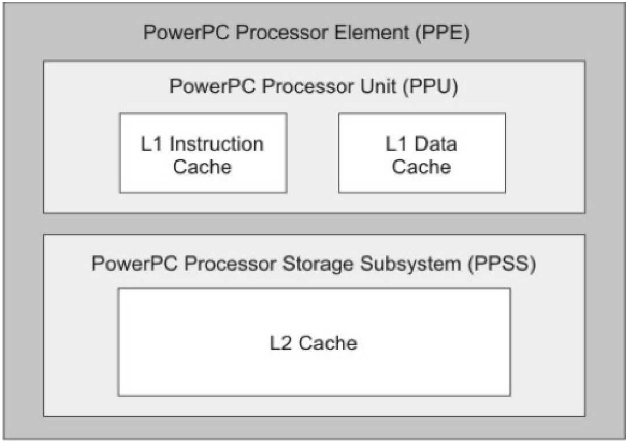



Image taken from CBE Programming Tutorial v. 3

# Synergistic Processor Element (*SPE*)

Each *SPE* is:

# Synergistic Processor Element (*SPE*)

Each *SPE* is:

- Slave processor: it execute tasks spawned from the *PPE*

# Synergistic Processor Element (*SPE*)

Each *SPE* is:

- Slave processor: it execute tasks spawned from the *PPE*
- Based on a 128 bit RISC architecture specialized for computing intensive SIMD applications

# *SPE* simple block diagram

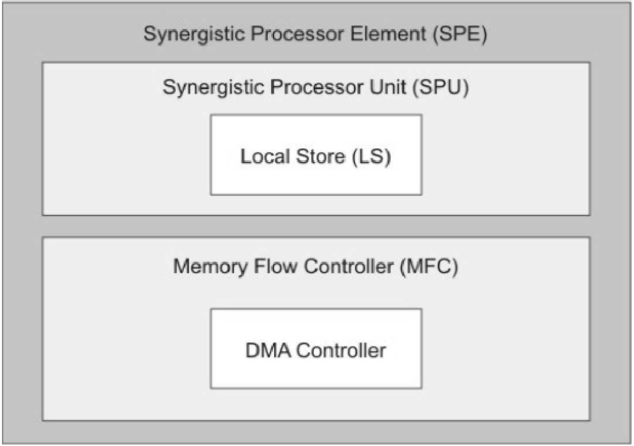



Image taken from CBE Programming Tutorial v. 3

# Synergistic Processor Unit (*SPU*)

# Synergistic Processor Unit (*SPU*)

- Deals with instructions execution and control

# Synergistic Processor Unit (*SPU*)

- Deals with instructions execution and control
- Single (unified) register file with 128 registers

# Synergistic Processor Unit (*SPU*)

- Deals with instructions execution and control
- Single (unified) register file with 128 registers
- Unified 256 KB local memory for instructions and data named Local Store (*LS*)

# Synergistic Processor Unit (*SPU*)

- Deals with instructions execution and control
- Single (unified) register file with 128 registers
- Unified 256 KB local memory for instructions and data named Local Store (*LS*)
- New SIMD (Single Instruction Multiple Data) instruction set

# Local Store (*LS*)

# Local Store (*LS*)

- Each *SPE* is an indipendent processor with its own program counter

# Local Store (*LS*)

- Each *SPE* is an indipendent processor with its own program counter
- The *SPU* fetches instructions and load/store data from/to its own Local Store

# Memory Flow Controller (*MFC*)

# Memory Flow Controller (*MFC*)

- It's the interface between the SPE and the other system processors

# Memory Flow Controller (*MFC*)

- It's the interface between the SPE and the other system processors
- Contains a DMA controller for DMA transfers support

# Memory Flow Controller (*MFC*)

- It's the interface between the SPE and the other system processors
- Contains a DMA controller for DMA transfers support
- In order to support the DMA controller, the MFC maintains a queue of DMA commands

# Memory Flow Controller (*MFC*)

- It's the interface between the SPE and the other system processors
- Contains a DMA controller for DMA transfers support
- In order to support the DMA controller, the MFC maintains a queue of DMA commands
- After a DMA command has been queued, the SPU can continue to execute instructions while the MFC processes the DMA command

# DMA tranfers

# DMA tranfers

- Each DMA transfer can move up to 16 KB.

# DMA tranfers

- Each DMA transfer can move up to 16 KB.
- The *SPU* associated with *MFC* can issue a DMA-list of up to 2048 DMA

# High level programming

# High level programming — an introduction

# High level programming — an introduction

- SPE and PPE are independent processors

# High level programming — an introduction

- SPE and PPE are independent processors
- To fully exploit the Cell performance you must write two different software programs:

# High level programming — an introduction

- SPE and PPE are independent processors
- To fully exploit the Cell performance you must write two different software programs:
- *PPE program* — a program running on PowerPC core that offloads task to SPE

# High level programming — an introduction

- SPE and PPE are independent processors
- To fully exploit the Cell performance you must write two different software programs:
- *PPE program* — a program running on PowerPC core that offloads task to SPE
- *SPE program* — a program running on SPE processor that uses the SPU Instruction Set

# Creating a SPE thread from PPE

# Creating a SPE thread from PPE

- A PPE program spawns a task to an SPE by creating a thread on the SPE. It uses the following functions:

# Creating a SPE thread from PPE

- A PPE program spawns a task to an SPE by creating a thread on the SPE. It uses the following functions:
- *spe_context_create()* – creates a context for the SPE thread

# Creating a SPE thread from PPE

- A PPE program spawns a task to an SPE by creating a thread on the SPE. It uses the following functions:
- *spe_context_create()* – creates a context for the SPE thread
- *spe_program_load()* – load an SPE program into the context

# Creating a SPE thread from PPE

- A PPE program spawns a task to an SPE by creating a thread on the SPE. It uses the following functions:
- *spe_context_create()* – creates a context for the SPE thread
- *spe_program_load()* – load an SPE program into the context
- *spe_context_run()* – execute a context on a physical SPE

# Creating a SPE thread from PPE and libspe2

- The function aboves are in libspe2, which is an implementation of the *SPE Runtime Management Library* developed by IBM under GPL license and downlodable from http://sourceforge.net/projects/libspe

# SPE Program

# SPE Program

- Conceived and compiled for execution on the SPE

# SPE Program

- Conceived and compiled for execution on the SPE
- This program can use SPE (vectorial) data types and SIMD instructions

# SPE Program

- Conceived and compiled for execution on the SPE
- This program can use SPE (vectorial) data types and SIMD instructions
- SIMD instructions are defined in the *SPU C/C++ language extensions* and are named *intrinsics*

# SPE Program

- Conceived and compiled for execution on the SPE
- This program can use SPE (vectorial) data types and SIMD instructions
- SIMD instructions are defined in the *SPU C/C++ language extensions* and are named *intrinsics*
- A SPE program transfers data from/to main memory to/from Local Store through DMA transfers

# Tips for performance improvements (SPE side)

# Tips for performance improvements (SPE side)

- Use vector data type instead of scalars

# Tips for performance improvements (SPE side)

- Use vector data type instead of scalars
- Perform loop unrolling

# Tips for performance improvements (SPE side)

- Use vector data type instead of scalars
- Perform loop unrolling
- Use double buffering

# Scalar and vector data types

# Scalar and vector data types

- The SPE processor has a vectorial architecture. The SPU loads and stores one quadword at time

# Scalar and vector data types

- The SPE processor has a vectorial architecture. The SPU loads and stores one quadword at time
- Scalar types are stored in the left-most word in the register (*Preferred Slot*)

# Scalar and vector data types

- The SPE processor has a vectorial architecture. The SPU loads and stores one quadword at time
- Scalar types are stored in the left-most word in the register (*Preferred Slot*)
- We must avoid as much as possible scalar types because operations on scalar types are inefficient

# Scalar and vector data types

- The SPE processor has a vectorial architecture. The SPU loads and stores one quadword at time
- Scalar types are stored in the left-most word in the register (*Preferred Slot*)
- We must avoid as much as possible scalar types because operations on scalar types are inefficient
- For example a scalar load must be rotated into the preferred slot

# Loop unrolling

# Loop unrolling

- Loop unrolling is a common technique for increasing the performances

# Loop unrolling

- Loop unrolling is a common technique for increasing the performances
- SPE processors have 128 registers

# Loop unrolling

- Loop unrolling is a common technique for increasing the performances
- SPE processors have 128 registers
- Using loop unrolling can improve register utilization

# Loop unrolling

- Loop unrolling is a common technique for increasing the performances
- SPE processors have 128 registers
- Using loop unrolling can improve register utilization
- PROBLEM

# Loop unrolling

- Loop unrolling is a common technique for increasing the performances
- SPE processors have 128 registers
- Using loop unrolling can improve register utilization
- PROBLEM
- Loop unrolling increases the size of code

# Loop unrolling

- Loop unrolling is a common technique for increasing the performances
- SPE processors have 128 registers
- Using loop unrolling can improve register utilization
- PROBLEM
- Loop unrolling increases the size of code
- Data and code must fit in 256 KB Local Store

# Double buffering (1/2)

# Double buffering (1/2)

- The SPU moves data from/to main memory only with DMA tranfers

# Double buffering (1/2)

- The SPU moves data from/to main memory only with DMA tranfers
- The communication bus between SPE's and PPE is a bottleneck

# Double buffering (1/2)

- The SPU moves data from/to main memory only with DMA tranfers
- The communication bus between SPE's and PPE is a bottleneck
- In the Cell architecture DMA transfers are asynchronous

# Double buffering (1/2)

- The SPU moves data from/to main memory only with DMA tranfers
- The communication bus between SPE's and PPE is a bottleneck
- In the Cell architecture DMA transfers are asynchronous
- This feature allow the programmer to schedule the transfers so that the latency of memory accesses can be hidden by overlapping the transfers in one buffer with computations in another

# Double buffering (2/2)

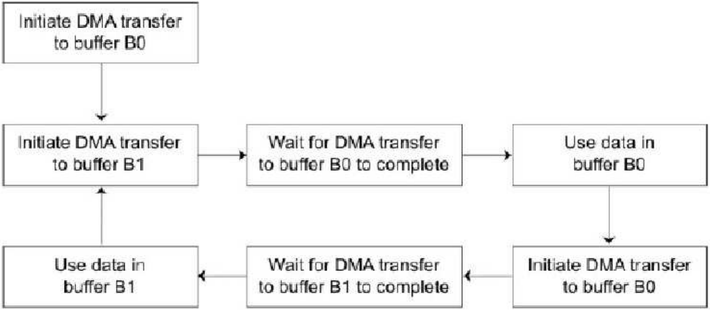



Image taken from CBE Programming Tutorial v. 3

# Cell and the Linux Kernel

# Linux Kernel support for Cell

# Linux Kernel support for Cell

- The Cell processor is fully supported by the Linux Kernel

# Linux Kernel support for Cell

- The Cell processor is fully supported by the Linux Kernel
- Cell is a PowerPC-based architecture

# Linux Kernel support for Cell

- The Cell processor is fully supported by the Linux Kernel
- Cell is a PowerPC-based architecture
- If you look in /<path_linux_source>/arch/powerpc/platforms you can find two folders (among many others) named:

# Linux Kernel support for Cell

- The Cell processor is fully supported by the Linux Kernel
- Cell is a PowerPC-based architecture
- If you look in /<path_linux_source>/arch/powerpc/platforms you can find two folders (among many others) named:
- cell

# Linux Kernel support for Cell

- The Cell processor is fully supported by the Linux Kernel
- Cell is a PowerPC-based architecture
- If you look in /<path_linux_source>/arch/powerpc/platforms you can find two folders (among many others) named:
- cell
- ps3

# Linux Kernel support for Cell

- The Cell processor is fully supported by the Linux Kernel
- Cell is a PowerPC-based architecture
- If you look in /<path_linux_source>/arch/powerpc/platforms you can find two folders (among many others) named:
- cell
- ps3
- The first folder includes code for supporting the native Cell

# Linux Kernel support for Cell

- The Cell processor is fully supported by the Linux Kernel
- Cell is a PowerPC-based architecture
- If you look in /<path_linux_source>/arch/powerpc/platforms you can find two folders (among many others) named:
- cell
- ps3
- The first folder includes code for supporting the native Cell
- The second folder include code for supporting the Cell on Sony PlayStation 3

# Differences between native Cell and Cell on ps3

# Differences between native Cell and Cell on ps3

- In native Cell the Linux kernel runs directly on hardware

# Differences between native Cell and Cell on ps3

- In native Cell the Linux kernel runs directly on hardware
- In ps3 the Linux kernel runs in a virtualized environment

# Why different platforms?

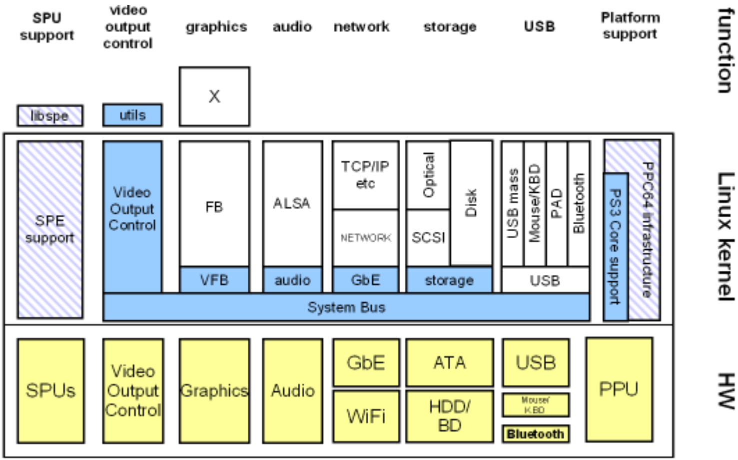
# Why different platforms?

- Why there are different platforms for Cell native and Cell on ps3?

# Why different platforms?

- Why there are different platforms for Cell native and Cell on ps3?
- The presence of a virtualization layer imposes different low level interactions between hardware devices and kernel

# Kernel execution overview on native Cell



# Kernel execution overview on PlayStation3

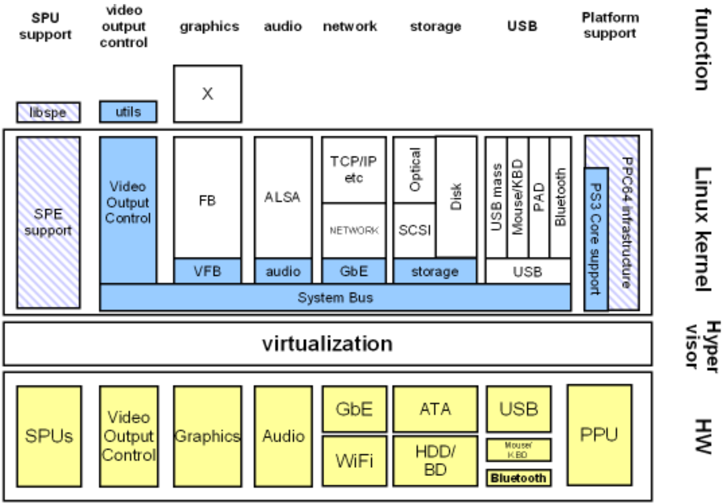



Image taken from http://www.kernel.org/pub/linux/kernel/people/geoff/cell/

# How libspe2 create a SPE context – *spu_create()* syscall

# How libspe2 create a SPE context – *spu_create()* syscall

- The *spu_create_context()* of libspe2 creates an *SPE context*

# How libspe2 create a SPE context – *spu_create()* syscall

- The *spu_create_context()* of libspe2 creates an *SPE context*
- An SPE context is, essentially, a directory in *spufs* pseudo file system (see later)

# How libspe2 create a SPE context – *spu_create()* syscall

- The *spu_create_context()* of libspe2 creates an *SPE context*
- An SPE context is, essentially, a directory in *spufs* pseudo file system (see later)
- The *spu_create_context()* function creates an entry in *spufs* through the *spu_create()* system call and maps some file created by *spu_create()*. For example the *mem* file (see later)

# How libspe2 create a SPE context – *spu_create()* syscall

- The *spu_create_context()* of libspe2 creates an *SPE context*
- An SPE context is, essentially, a directory in *spufs* pseudo file system (see later)
- The *spu_create_context()* function creates an entry in *spufs* through the *spu_create()* system call and maps some file created by *spu_create()*. For example the *mem* file (see later)
- The *spu_create()* system call creates a spu context in kernel memory and return an open file descriptor for the directory (in /spu) associated with it.

# SPU File System (*spufs*)

# SPU File System (*spufs*)

- Similar to procfs and sysfs

# SPU File System (*spufs*)

- Similar to procfs and sysfs
- Purely virtual file system

# SPU File System (*spufs*)

- Similar to procfs and sysfs
- Purely virtual file system
- By convention mounted in /spu

# SPU File System (*spufs*)

- Similar to procfs and sysfs
- Purely virtual file system
- By convention mounted in /spu
- Directories in /spu represent SPE contexts whose properties are shown as regular files

# SPU File System (*spufs*)

- Similar to procfs and sysfs
- Purely virtual file system
- By convention mounted in /spu
- Directories in /spu represent SPE contexts whose properties are shown as regular files
- Interaction with these contexts can happen through file operations like open, read, write, etc.

# Examples of files in a spufs sub-directory

# Examples of files in a spufs sub-directory

- *mem* – The local memory of a SPE context. Mainly used to load the executable file of the program to be run onto the SPE

# Examples of files in a spufs sub-directory

- *mem* – The local memory of a SPE context. Mainly used to load the executable file of the program to be run onto the SPE
- *regs* – The general purpose registers of an SPE. Normally can't be accessed directly but they can be saved in a context in kernel memory

# SPE context

# SPE context

- It is a data structure which represents a SPE task

# SPE context

- It is a data structure which represents a SPE task
- A context has all properties of a physical SPE

# SPE context

- It is a data structure which represents a SPE task
- A context has all properties of a physical SPE
- The kernel can use this structure to save the state of a SPE thread

# SPE context

- It is a data structure which represents a SPE task
- A context has all properties of a physical SPE
- The kernel can use this structure to save the state of a SPE thread
- Context switching on SPE is very inefficient

# How libspe2 load a program in to SPE

# How libspe2 load a program in to SPE

- The *spe_program_load()* function of libspe2 loads an SPE ELF object file in an SPE

# How libspe2 load a program in to SPE

- The *spe_program_load()* function of libspe2 loads an SPE ELF object file in an SPE
- This function does not call any syscall

# How libspe2 load a program in to SPE

- The *spe_program_load()* function of libspe2 loads an SPE ELF object file in an SPE
- This function does not call any syscall
- Instead, it makes use of a file memory mapping of the *mem* file

# How libspe2 load a program in to SPE

- The *spe_program_load()* function of libspe2 loads an SPE ELF object file in an SPE
- This function does not call any syscall
- Instead, it makes use of a file memory mapping of the *mem* file
- Thus, the SPE ELF object file is loaded into the context directly from user space

# Running a SPE program – spu_run() syscall

# Running a SPE program – spu_run() syscall

- The *spe_context_run()* function runs a SPE program previously loaded into a SPE context

# Running a SPE program – spu_run() syscall

- The *spe_context_run()* function runs a SPE program previously loaded into a SPE context
- It calls the *spu_run()* system call

# Running a SPE program – spu_run() syscall

- The *spe_context_run()* function runs a SPE program previously loaded into a SPE context
- It calls the *spu_run()* system call
- *spu_run()* starts the SPE thread execution. The PPE thread that called *spu_run()* blocks in that system call

# Running a SPE program – spu_run() syscall

- The *spe_context_run()* function runs a SPE program previously loaded into a SPE context
- It calls the *spu_run()* system call
- *spu_run()* starts the SPE thread execution. The PPE thread that called *spu_run()* blocks in that system call
- Each SPE thread is associated with one PPE thread

# Example of Cell application

# Example of Cell application

- Cell is increasingly used in accademic and scientific world

# Example of Cell application

- Cell is increasingly used in accademic and scientific world
- With ps3 Cell is incredibly low cost

# Example of Cell application

- Cell is increasingly used in accademic and scientific world
- With ps3 Cell is incredibly low cost
- The University of Massachusetts Dartmouth uses a cluster of sixteen ps3 for astrophysics analysis

# Performance scaling with implementation

- 2048x2048 float matrix multiplication on single SPE

| Implementation | Execution time (ms) |
|---|---|
| Scalar | 338687.230514 |
| Vectorial | 336059.746404 (-0,77%) |
| Vectorial - Unrolling | 280815.662356 (-17%) |
| Vectorial - Unrolling con spu_madd | 262594.693659 (-23%) |
| Vectorial - spu_madd | 75076.210915 (-78%) |
| Vectorial - spu_madd - spu_gcc -O3 | 18072.911028 (-95%) |
| Vectorial - spu_madd - with Double Buffering | 10509.868133 (-97%) |

# Conclusions

# Conclusions

- Cell is a very interesting and (potentially) powerful architecture

# Conclusions

- Cell is a very interesting and (potentially) powerful architecture
- Each SPE is capable of 25.6 GFLOPS in integer and single precision arithmetic

# Conclusions

- Cell is a very interesting and (potentially) powerful architecture
- Each SPE is capable of 25.6 GFLOPS in integer and single precision arithmetic
- Fully exploiting Cell capabilities is not easy