UNIVERSITY OF TARTU

FACULTY OF MATHEMATICS AND COMPUTER SCIENCE

Institute of Computer Science

Computer Science speciality

Lauri Tulmin

A Conjugate Gradient Solver on the PlayStation 3 - a native approach

Master Thesis (30 EAP)

Supervisor : Ulrich Norbisrath, PhD

Author : .... "....." May 2010

Supervisor : .... "....." May 2010

Allowed to defence
Professor : : .... "....." May 2010

TARTU 2010

Table des matières

Introduction

The ability of modern computers to solve previously intractable problems and simulate complex real world processes using mathematical models gives scientists and engineers invaluable information that aids them in their research. Computer simulations of real world problems often result in solving large systems of linear equations. The conjugate gradient method [21] is a well-known algorithm for solving large sparse linear systems that are symmetric and positively definite. In this thesis, I will describe the Cell Broadband Engine Architecture (CBEA) [16, 6], that is the basis of the Sony PlayStation 3, and my own implementation of the conjugate gradient algorithm for this hardware architecture. The implemented solver uses double precision floating point numbers and only the basic programming library for the Cell platform [8], a similar solution based on higher level libraries, such as DaCS (Data Communication and Synchronization Library) [7], is described by Toomas Laasik in his master's thesis [18].

For a long time the capabilities of digital electronic devices have been linked to Moore's Law [20]. Moore's Law is based on an long term observation that the number of transistors that can be placed inexpensively on an integrated circuit doubles every 18 to 24 months. The law is named after Intel co-founder Gordon E. Moore, who introduced the concept in 1965. Although Moore said nothing about improved performance of the microprocessors, the exponential growth in the number of transistors roughly translated to the exponential increase of performance of microprocessors.

By 2006 the previously used ways of increasing processor performance, such as increasing the operating frequency by reducing the amount of work done per cycle while increasing the pipeline depth, had started giving diminishing returns [22]. Designing more performant single core processors was getting more and more difficult, mainly due to the increased power usage and heat generation. It was predicted that by 2005 high-end processors would radiate the same heat volume per square centimeter as a nuclear reactor shell, by 2010 – as a rocket nozzle, by 2015 – as the Sun's surface. Even if the chips could be made thermally resistant to such temperatures, for consumer electronics the size of the cooling system, the maximum airspeed and the maximum allowable temperature of air leaving the system would still impose a limit to tolerable power consumption. Although manufacturers had trouble squeezing more performance out of traditional uni-processor centric designs, the market still expected the performance of the processors to continue increasing at the same rate as previously. The processor makers had to find a new way to improve their chips and satisfy the market's expectations as further increasing the operating frequency would also increase the power consumption, which in turn would increase the heat dissipation. To overcome the challenge of delivering double performance

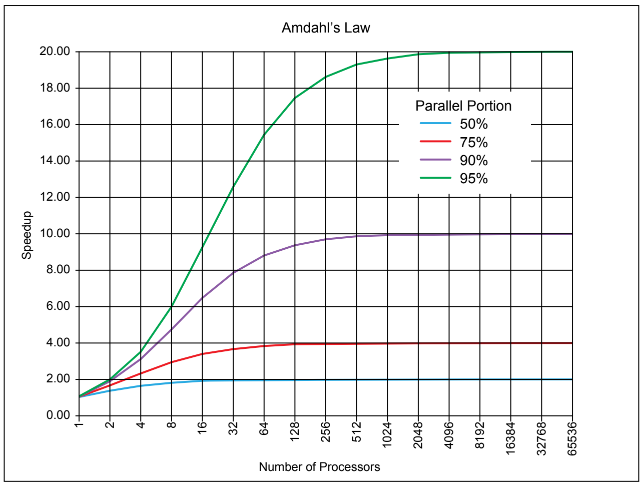

Figure 1 – Expected speedup for a program using multiple processors [1].

the focus shifted from building faster single-core processors to adding multiple cores to a single processor.

The amount of performance gained from using a multiple processors depends strongly on the software algorithms and their implementation. In particular the possible gains of using multiple processors are limited to the parts of the software that can be parallelized to run simultaneously on these processors. This effect is described by Amdahl's law [1, 3] which states that the expected speedup from using multiple processors is limited by the time needed for the sequential part of the program. Amdahl's law is best illustrated by Figure 1. In the best case, where the problem is embarrassingly parallel, the speedup factor may be near the number of processors used. In practice many applications are not easily parallelizable and therefore they gain a much smaller speedup from running on multi-core machines. Also, there is no getting around the fact that creating a properly multi-threaded application, that scales across many cores, is a significantly more complex task than writing traditional linear code. Although the Amdahl's law was postulated long before the emergence of the multi-core processors in a paper published in 2008 Mark D. Hill and Michael R. Marty [15] find that even in the multi-core era researchers should still seek methods of speeding up sequential execution. The authors of the previously mentioned paper also propose that for better performance to cost ratio it would be beneficial to design some cores more powerful than the others. This would allow for fast execution of the sequential part on the more powerful cores with wasting less resources on the idle cores while still allowing for executing the parallel part on all of the cores simultaneously.

Another major barrier hit by conventional microprocessor is the increased latency of accessing the system main memory. This is known as the memory wall [25]. The problem is that the frequencies of the dynamic random access memory (DRAM) have not been increasing at same rate as the processor's operating frequencies, thus the memory access latencies have been increasing with each new generation of microprocessors. A microprocessor with sequential execution model assumes that each instruction is completed before the execution of the next instruction begins. This means that in order to hide the memory access latencies, in case of a cache miss, the processing can proceed only if instructions are executed speculatively. This in turn makes the processor designs significantly more complex. The probability that useful work is being speculatively completed decreases every time the processor must speculate in order to continue [16].

The CBEA described in this paper was designed from the ground up to address the diminishing returns offered by frequency oriented single-core processor designs by exploiting application parallelism through multiprocessing. The CBEA design also tackles the memory wall by a processor organization that allows for more memory bandwidth to be used effectively by allowing more memory transactions to be in flight simultaneously.

The motivation of this paper is to study the unique hardware architecture used in the Sony PlayStation 3 and provide an implementation of the conjugate gradient algorithm that would utilize the computational resources of the hardware as much as possible. The first chapter gives an overview of the CBEA architecture and the programming models used on this platform. In the second chapter I will give a brief description of the conjugate gradient method. The third chapter introduces an implementation of the conjugate gradient method. The fourth chapter will discuss an implementation of conjugate gradient method for CBEA. The fifth and final chapter will provide the performance estimates for the previously described implementation.

Chapitre 1

Cell broadband engine architecture

In this chapter I will describe the hardware architecture used in the Sony PlayStation 3. This hardware is the Cell Broadband Engine Architecture (CBEA) [16, 6]. It is a processor design jointly developed by Sony Computer Entertainment, Toshiba and IBM. The CBEA came to existence from a challenge posed by Sony and Toshiba to provide a power and cost efficient high-performance processor for a wide range of applications. The initial development of the CBEA was carried out in Austin, Texas over a four year period starting from March 2001.

CBEA is a heterogeneous processor architecture that consists of a Power Processing Element (PPE), augmented with 8 specialized co-processors called Synergistic Processing Elements (SPE). The main building blocks of the CBEA are shown on Figure 1.1 - SPEs, PPE, the Memory Interface Controller (MIC) and the Element interconnect Bus (EIB), each of these will be explored in more detail throughout this chapter. The central piece in the CBEA is the PPE. On its own, the PPE acts as a conventional processor that does not provide many additional benefits when compared to other modern processor architectures. What sets Cell apart from other processor architectures are the additional SPEs. The SPEs use a novel Single Instruction Multiple Data (SIMD) design, which is well suited for data intensive processing like that found in multimedia, gaming, cryptography and scientific computing applications.

The high number of cores gives improved performance to applications with thread level parallelism regardless of their ability to exploit data level parallelism. On the PlayStation 3 one SPE is reserved for the operating system and one is locked to reduce manufacturing defects. The remaining six SPEs are available to the applications.

1.1 The Power Processing Element

The heart of the CBEA is the PPE, which is used to run the operating system, manage resources and handle input and output. It also distributes the workload between the SPEs and coordinates their operation. The PPE is an IBM 64-bit PowerPC [13] core with Symmetric Multi Threading (SMT) allowing two independent threads of execution, clocked at 3.2 GHz. It has a 32 KiB instruction and a 32 KiB data level 1 cache and a 512 KiB level 2 cache. It also includes a vector multimedia extension (VMX) unit with

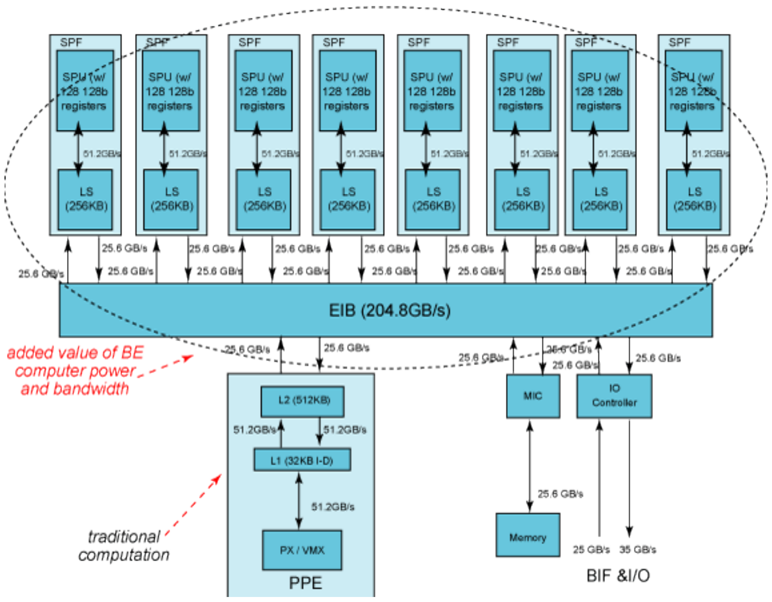

Figure 1.1 – CBEA overview[6].

SIMD floating point and integer instructions. Due to its similarity to other PowerPC based processors the PPE can be used to run a wide variety of applications already ported to the PowerPC architecture. The main design goals of the PPE were maximizing the performance/power ratio as well as the performance to area ratio.

Although the theoretical peak performance of single precision floating point operations is 25.6 GFLOPS and 6.4 GFLOPS for double precision floating point, the attainable performance is much lower. In the CBEA architecture the main task of the PPE is running the operating system and orchestrating the work of the SPEs that provide the bulk of the computational resources.

1.2 The Synergistic Processing Element

An SPE is a 128-bit RISC processor consisting of a Synergistic Processing Unit (SPU), that is the part of the SPE which actually runs your code, and a Memory Flow Controller (MFC), that is used to communicate with the system main memory and other SPEs. Each SPE is an independent processor running an independent application thread. Figure 1.2 shows the main components of the SPE.

The SPEs do not have direct access to the main memory, but only a small local memory store for efficient instruction and data access. For current hardware the size of the local store is 256 KiB, but as the local store resides in a 32-bit address space, the architecture supports local store sizes up to 4 GiB. The local store is different from traditional processor caches, as it is not transparent to the programmer, neither does it prefetch data from the system memory. It can be viewed as a software managed cache, whose content is explicitly controlled by the programmer.

The SPU has 128 general purpose registers, each of them is 128 bit wide. However, the point of the SPU is not to do operations on 128-bit values. Instead the SPU is

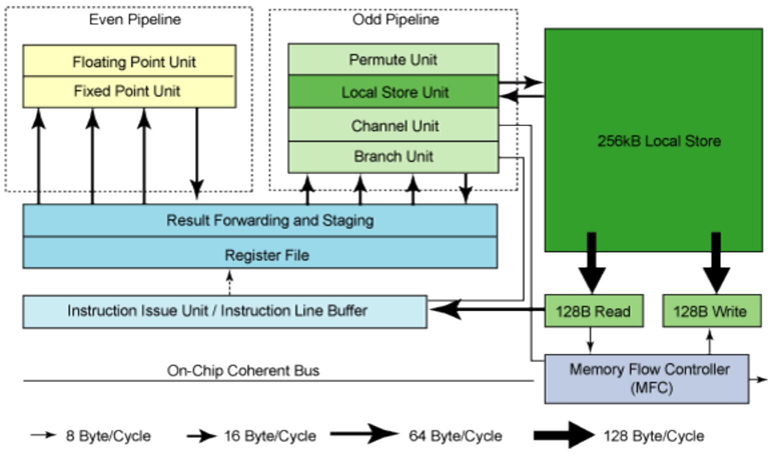

Figure 1.2 – SPE overview [6].

a vector processor, meaning that the registers are divided into multiple smaller values and the instructions operate on all of these values simultaneously. This is known as Single Instruction Multiple Data (SIMD), or data level parallelism. The registers may be treated as four distinct 32-bit values (32 bits is considered to be the word size on the SPUs), sixteen 8-bit values (byte), eight 32-bit values (halfword), two 32-bit values (doubleword) or a single 128-bit values (quadword). All instructions can use any of the 128 registers, there are no instructions that must use an instruction specific register or subset of registers. The high number of registers means that the SPU can keep lots of temporary and intermediate values around without having to put them back to the memory, to free some registers, like on most other architectures.

An SPU has two execution pipelines. The floating point and integer arithmetic units are on the even pipeline while the rest of the functional units are on the odd pipeline. The SPU can issue and complete up to two instructions per cycle, one on each execution pipeline. A dual issue (see Figure 1.3) occurs when a group of fetched instructions has two issuable instructions, one of which is executed by a unit on the even pipeline and the other by a unit on the odd pipeline [6].

The distinguishing feature of the SPU architecture is its simplicity. To save valuable die space and increase efficiency, SPU designers have omitted functionality, such as support for misaligned memory access, second level processor cache and branch prediction, which is commonly seen on other architectures. Any complexity reduction directly translates into better performance as the saved die space allows for additional cores per given area.

The lack of misaligned memory access means that both memory read and write operations return a single quadword by truncating the low order four address bits. When the application wishes to perform a load that crosses the quadword boundary it is possible to emulate it by loading to adjacent quadwords and perform a data merge operation using SPU shuffle instruction. The reasoning behind this design decision is that the support for unaligned load would have incurred a substantial cost. One possible implementation option would have been to preallocate bandwidth to perform two memory accesses at instruction issue time, thereby reducing the available bandwidth two times even if the access was properly aligned. Another option would have been to optimistically assume

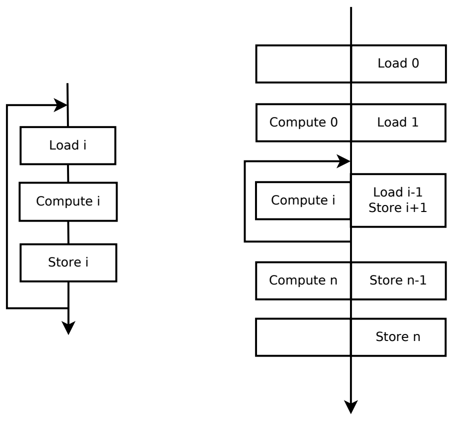

Figure 1.3 – Pipeline and dual issue .

aligned access and in case of an unaligned access perform a recovery sequence. However, this would mean a penalty to every unaligned access. Instead the designers chose to implement an efficient common case and let the compiler generate an explicit dual load and data merge sequence if the data alignment can not be determine at compile time [14]. For programmers using higher level languages this means that to create efficient code the compiler must be given enough information to properly recognize the aligned data accesses.

Instead of having a second level cache the SPUs have a programmatically controlled local store. This simplifies the architecture by eliminating complex cache hit/miss detection, miss recovery and cache coherence management. It provides low latency deterministic memory access to the SPU. Deterministic memory access aids static scheduling of the SPU instructions as the memory load/store instructions are guaranteed to take 6 cycles. Compilers can take advantage of the deterministic memory access and arrange the instructions so that the computation will start on the first cycle after the load completes.

While most of the modern microprocessor architectures include sophisticated logic to predict which code branch is executed (e.g. if-then-else construct) before this is known for sure, the SPU designers decided to not implement this feature. The purpose of the branch prediction is to improve the flow in the instruction pipeline. Without the branch prediction the processor would have to wait till the conditional jump instruction is executed to know where the next instruction to be executed is. Branch prediction tries to avoid this by guessing whether the conditional branch is more likely to be taken or not and speculatively fetches and executes instructions from one of the branches. If the branch instruction is reached and the prediction turns out to be wrong the speculatively executed instructions are discarded and the pipeline restarts from the correct branch. On the SPU the idea of the static instruction schedule has also been applied to branch prediction, which is implemented by a prepare to branch instruction. The compiler inserts this branch prediction hint instruction to predict the target branch so that the instruction prefetch from the target branch address can be initiated ahead of time. The prepare to branch instruction accepts two addresses, a trigger address and a target address, and fetches instructions from the specified target address into a branch target buffer. When the instruction fetch reaches the trigger address, the instruction stream continues execution with instructions from the target buffer to avoid a branch delay penalty. Both mispredicted and non hinted taken branches incur a misprediction penalty [14]. From a software development perspective this means that an unpredictable branch in critical code path will incur a significant performance penalty. To solve this issue the SPU has several conditional instructions, which allow conditional operations to occur without branching. The program will calculate answers for both branches and then use a conditional select instruction to pick the correct answer.

To increase processing power high performance microarchitectures try to increase Instruction Level Parallelism (ILP). ILP is the measure of how many instructions can be performed simultaneously. The techniques to increase ILP include :

- Instruction pipelining.
- Superscalar execution with multiple execution units. Superscalar processors can, for example, run multiple arithmetic operations in parallel, providing there are no data dependencies between the operations.
- Out-of-order execution where instructions are executed in any order if there are no data dependencies.
- Register renaming, which refers to a technique where operands are placed to a different register than indicated by the instruction. This is used to enable out-of-order execution.
- Speculative execution to allow execution of instructions before it is certain whether this execution path is taken.
- Branch prediction is used to avoid stalling while it is determined which code path should be taken. This is used together with speculative execution.

In contrast, the SPU executes instructions in the order they appear in the program code and only has one execution unit of each type. To reduce unnecessary stalls because of data dependencies programmers can take advantage of the large number of registers on SPU and loop unrolling. The goal of loop unrolling is to reduce the number of instructions that control the loop, such as the of the loop test on each iteration. Loops can be re-written as a repeated sequence of similar independent statements to reduce this overhead. Also this often helps instruction scheduling. By having more independent instructions the compiler can eliminate data dependency stalls.

The data parallel nature of SPU provides a major advantage to programs with even modest amounts of data parallelism over transforming data level parallelism into instruction level parallelism. Traditional cores often take the latter approach, which requires processing and tracking the increased number of instructions, and often yields significant penalties because parallelism must be rediscovered in the instructions [14].

Local store with deterministic memory access latency and static instruction schedule make SPU architecture highly predictable. This allows for a cycle accurate simulator [10] and static program analysis to determine the expected cycle count of the given algorithm. This, in turn, makes it easier for software developers to find inefficiencies in their implementations.

1.3 Communication between PPE and SPE

The SPEs only have direct access to their own dedicated local store. If the SPE needs to access the system main memory or the local store of another SPE, then it needs to initiate a DMA data transfer operation. Each SPE has its own Memory Flow Controller (MFC) that allows for streaming data in and out of the local stores in parallel with the program execution.

1.3.1 DMA

The SPE can initiate up to 16 DMA transfers in parallel using the DMA queue. Most of the DMA commands can be assigned a 5-bit identifier called the Tag Group ID. As the DMAs are processed out-of-order by the DMA engine, the software can use the tag group identifier to check or wait on the completion of all queued commands in one or more tag groups. To order the DMA commands within the queue one can use fences and barriers.

- Execution of fenced command option is delayed until all previously issued commands within the same tag group have been performed.
- Execution of a barrier command option and all subsequent commands is delayed until all previously issued commands in the same tag group have been performed.

The SPE DMA engine supports DMA transfers of sizes 1, 2, 4, 8 or a multiple of 16 bytes, with sizes up to 16 KiB per transfer. The source and target address of the DMA transfer must be aligned to 16 byte boundary, although 128 byte alignment is preferable. For maximum performance it is best to make sure that the source and destination addresses have the same quadword offset within a PPE cache line, where the PPE cache line size is 128 bytes. Quadword-offset-aligned data transfers generate full cache line bus requests for every transferred chunk of data, except possibly the first and last chunk. Transfers that start or end in the middle of a cache line transfer a partial cache line (less than 8 quadwords) in the first or last bus request, respectively [23].

1.3.2 DMA lists

SPE DMA engine also supports scatter-gather memory access via DMA lists. These operations take a list from the local store as argument. Each element in the list contains a transfer size and the address of the transferred element in main memory. For list transfers, the same alignment rules apply as to normal DMA operations. Each DMA transfer specified in a list has a size from one quadword up to 16 KiB. The maximum size of the list is limited to 2048 entries.

1.3.3 Mailboxes

While DMA transfer allows transfer of up to 16 KiB of data between the main memory and each SPE's local store, mailboxes are designed for transfer of 32-bit data between the PPE and the SPE. To put it another way, mailboxes offer a simple way to transfer small data chunks such as status information and parameters. The MFC provides three types of mailboxes, each with a different behavior and data transfer direction [19].

1. *SPU Inbound Mailbox* Used to send data from the PPE to the SPE. This mailbox has space for storing up to four 32-bit messages at a time. If no message is found when the SPE program accessed the queue, the SPE stalls until data is written by the PPE program.
2. *SPU Outbound Mailbox* Used to pass data from the SPE to the PPE. This mailbox has the capacity to accept only one 32-bit message. If the SPU outbound mailbox is full, writing of the next data is suspended until the PPE reads the data from the queue.
3. *SPU Outbound Interrupt Mailbox* Like the SPU outbound mailbox, this is used to send data from the SPE to the PPE. When this mailbox is written, however, an interrupt event is generated to notify the PPE when to read the data.

1.4 The SPU instruction set

The SPU Instruction Set Architecture (ISA) [11] is designed to achieve better performance to cost ratio than general purpose processors by reducing the power and chip area size requirements to implement the instructions. The key features of the architecture and implementation as outlined in Table 1-1 presented in the Synergistic Processor Instruction Set Architecture [11] specification are listed in Table 1.1

The SPU has 128 general purpose registers, each of these registers contains 128 data bits. The same registers can hold both floating point and integer data. The SPU uses a single SIMD instruction set for both scalar and vector data, this means that all instructions operate on the full register, by treating the register as vector of multiple elements of the same type. Scalar elements are handled by loading them to a *preferred slot* of the vector, computing on the full vector and extracting the result from the preferred slot.

The SPU is an in-order dual-issue statically scheduled architecture. It has two instruction pipelines with each instruction pre-assigned to execute on only one of those pipelines. Two SIMD instructions can be issued per cycle :

- one compute instruction on even pipeline, see Table 1.2
- one memory operation on odd pipeline, see Table 1.3

Dual issue is only possible when :

- there are no dependencies
- operands are available
- the even-addressed instruction is an even pipeline instruction
- the odd-addressed instruction is an odd pipeline instruction, and
- the instructions are ordered even pipeline followed by odd pipeline.

Although the SPU ISA supports integer arithmetics it is still geared towards floating point. For example the SPU ISA implements only 16-bit integer multiplication in hardware. 32-bit integer multiplication can be emulated by executing five instructions, three

Feature	Description
128-bit SIMD execution unit organization	Many of the applications previously mentioned allow for single-instruction, multiple- data (SIMD) concurrency. In an SIMD architecture, the cost (area and power) of fetching and decoding instructions is amortized over the multiple data elements processed. A 128-bit (most commonly 4-way 32-bit) SIMD has commonality with SIMD processing units in other general-purpose processor architectures and the existing code base to support it.
Software-managed memory	Whereas most processors reduce latency to memory by employing caches, the SPU in the CBEA implements a small local memory rather than a cache. This approach requires approximately half the area per byte and significantly less power per access, as compared to a cache hierarchy. In addition, it provides a high degree of control for real-time programming. Because the latency and instruction overhead associated with direct memory access (DMA) transfers exceeds that of the latency of servicing a cache miss, this approach achieves an advantage only if the DMA transfer size is sufficiently large and is sufficiently predictable (that is, DMA can be issued before data is needed).
Load/store architecture to support efficient static random access memory (SRAM) design	The SPU ISA microarchitecture is organized to enable efficient implementations that use single-ported (local storage) memory.
Large unified register file	The 128-entry register file in the SPU architecture allows for deeply pipelined high-frequency implementations without requiring register renaming to avoid register starvation. This is especially important when latencies are covered by software loop unrolling or other interleaving techniques. Rename hardware typically consumes a significant fraction of the area and power in modern high-frequency general-purpose processors.
ISA support to eliminate branches	The SPU ISA defines compare instructions to set masks that can be used in three operand select instructions to create efficient conditional assignments. Such conditional assignments can be used to avoid difficult-to-predict branches.

Table 1.1 – Key Features of the SPU ISA Architecture and Implementation [11].

Even pipeline instructions	**Latency (clocks)**
Single precision floating point operations	6
Double precision floating point operations	6+7
Integer multiplication Integer/float conversion Interpolate estimate	7
Immediate load Logical operation Integer addition/subtraction Sign extend Count leading zero Select bits Carry/borrow generate	2
Element rotates and shifts Special byte operations	4

Table 1.2 – Instructions for the even pipeline.

16-bit multiplies and two adds to accumulate partial products. For single precision floating point SPU only implements a subset of IEEE-754. The decision to no support full IEEE compliance was driven by the expected target applications of the hardware, for which features such as multiple rounding modes and IEEE-compliant exceptions where not deemed important. The full list of incompatibilities is available in section 9.1 for single precision and 9.2 for double precision of the SPU ISA specification [11]. The list of noncompliances with IEEE-754 for single precision arithmetics includes :

- the only supported rounding mode is truncation towards zero
- IEEE Inf and Nan are not recognizer by arithmetic operations
- there is no support for denormal numbers (denormal numbers are flushed to 0)
- overflows produce saturated results instead of +/-Inf

Instead of writing optimized assembly code or relying on the capabilities of the compiler to maximize the performance, the developer can take advantage of the language extensions provided by the SPU C/C++ compiler [9]. These extensions give programmers nearly full access to SPU assembly language instructions. To ease writing SIMD code the SPU C/C++ compiler also includes the `vector` keyword to describe a 128 bit vector of given type. For example `vector float` will declare a 128 bit vector containing four 32 bit single precision floating point numbers. Also the compiler provides a wide range of intrinsics and built-ins that allow developers to access the underlying hardware instructions from the C programming language. To use the SPU intrinsics `spu_intrinsics.h` must be included at the beginning of the code. Instructions that differ only by the data type of the operand are represented by a single C/C++ intrinsic, which selects the proper assembly language instruction according to the operand data type. For example `spu_add` when given two arguments of `vector unsigned ints` will map to 32-bit add instruction `a`, when given two `vector floats` will generate a floating point add instruction `fa`.

Some of the more common SPU intrinsics (roughly base on the list presented in [5]) :

Odd pipeline instructions	**Latency (clocks)**
Loads and stores Branch hints Channel operations Moves to/from SPRs	6
Shuffle bytes Quad-word rotates and shifts Estimate (reciprocal, reciprocal sqrt) Gather bits Form select mask Generate insertion control Branches	4

Table 1.3 – Instructions for the odd pipeline.

- `spu_add(val1, val2)`
  Adds each element of `val1` to the corresponding element of `val2`. If `val2` is a non-vector value, it adds the value to each element of `val1`. For integer vectors this intrinsic does not detect overflows.
- `spu_sub(val1, val2)`
  Subtract each element of `val2` from the corresponding element of `val1`. If `val1` is a non-vector value, then `val1` is replicated across a vector, and then `val2` is subtracted from it. Similarly to `spu_add` this intrinsic does not detect overflows.
- `spu_mul(val1, val2)`
  Because the multiplication instructions operate so differently, the SPU intrinsics do not coalesce them as much as they do for other operations. `spu_mul` handles floating point multiplication (single and double precision). The result is a vector where each element is the result of multiplying the corresponding elements of `val1` and `val2` together. To multiply integers one can use `spu_mulh`, `spu_mule`, `spu_mulo`, `spu_mulsr` intrinsics. All of these operate slightly differently, depending on the situation, one may need to execute multiple integer multiplication instructions and later combine their return values to get the final result.
- `spu_madd(val1, val2, val3)`
  Each element of `val1` is multiplied to the corresponding element of `val2` and added to the corresponding element of `val3`. Besides floating point vectors this intrinsic can also be used if `val1` and `val2` are vectors of 16-bit integers and `val3` is a vector of 32-bit integers. For integer vectors the odd elements of vectors `val1` and `val2` are sign extended to 32-bit integers prior to multiplication.
- `spu_msub(val1, val2, val3)`
  Each element of `val1` is multiplied to the corresponding element of `val2` and the corresponding element of `val3` is subtracted from the product. This intrinsic only works with floating point vectors.
- `spu_nmsub(val1, val2, val3)`
  Each element of `val1` is multiplied to the corresponding element of `val2`. The result

is subtracted from the corresponding element of `val3`. This intrinsic only works with floating point vectors.

- `spu_extract(val, element)`
  The element of the vector `val` that is specified by the `element` is extracted from the given vector.
- `spu_insert(val1, val2, element)`
  Scalar value `val1` is inserted into an element of the vector `val2` specified by the `element`.
- `spu_promote(val, element)`
  Scalar value `val` is promoted to an vector that contains `val` in the element specified by the `element`.
- `spu_splats(val)`
  Scalar value `val` is replicated across all the elements of the vector.
- `spu_and(val1, val2)`, `spu_or(val1, val2)`, `spu_not(val)`, `spu_xor(val1, val2)`, `spu_nor(val1, val2)`, `spu_nand(val1, val2)`, `spu_eqv(val1, val2)`
  Boolean operations operate bit-by-bit, so the type of operands the boolean operations receive is not relevant except for determining the type of value they will return. `spu_eqv` is a bitwise equivalency operation, not a per-element equivalency operation.
- `spu_rl(val, count)`, `spu_sl(val, count)`
  `spu_rl` rotates each element of `val` left by the number of bits specified in the corresponding element of count. Bits rotated off the end are rotated back in on the right. If `count` is a scalar value, then it is used as the count for all the elements of `val`. `spu_sl` operates the same way, but performs a shift instead of a rotate.
- `spu_rlmask(val, count)`, `spu_rlmaska(val, count)`, `spu_rlmaskqw(val, count)`, `spu_rlmaskqwbyte(val, count)`
  Although these operators are named "rotate left and mask" they are actually performing right shifts (they are implemented by a combination of left shifts and masks, but the programming interface is for right shifts). Both `spu_rlmask` and `spu_rlmaska` shift each element of val to the right by the number of bits in the corresponding element of `count` (or the value of `count` if `count` is a scalar). `spu_rlmaska` replicates the sign bit as bits are shifted in. `spu_rlmaskqw` operates on the whole quadword at a time, but only up to 7 bits (it performs a modulus on count to put it in the proper range). `spu_rlmaskqwbyte` works similarly, except that count is the number of bytes instead of bits, and count is modulus 16 instead of 8.
- `spu_cmpgt(val1, val2)`, `spu_cmpeq(val1, val2)`
  These instructions perform element-by-element comparisons of their two operands. The results are stored as all ones (for true) and all zeros (for false) in the resulting vector in the corresponding element. `spu_cmpgt` performs a greater-than comparison while `spu_cmpeq` performs an equality comparison.
- `spu_sel(val1, val2, conditional)`
  This corresponds to the `selb` assembly language instruction. The instruction itself is bit-based, so all types use the same underlying instruction. However, the intrinsic operation returns a value of the same type as the operands. As in assembly language,

`spu_sel` looks at each bit in `conditional`. If the bit is zero, the corresponding bit in the result is selected from the corresponding bit in `val1` ; otherwise it is selected from the corresponding bit in `val2`.

– `spu_shuffle(val1, val2, pattern)`
This instruction allows rearranging bytes in `val1` and `val2` according to a pattern that is specified in `pattern`. The instruction goes through each byte in pattern, and if the byte starts with the bits `0b10`, the corresponding byte in the result is set to `0x00` ; if the byte starts with the bits `0b110`, the corresponding byte in the result is set to `0xff` ; if the byte starts with the bits `0b111`, the corresponding byte in the result is set to `0x80`. Finally if none of the previous is true, last five bits of the pattern byte are used to choose which byte from `val1` or `val2` should be taken as the value for the current byte. The two values are concatenated, and the five-bit value is used as the byte index of the concatenated value. This is used for inserting elements into vectors as well as performing fast table lookups.

In addition if the developer needs even more fine grained control it is possible to use a specific assembly language instruction with *specific intrinsics*. All specific intrinsics are of form `si_assemblylanguageinstructionname` where `assemblylanguageinstructionname` denotes the name of the instruction in SPU assembly language. For example `si_a` corresponds to the 32-bit integer add instruction `a`.

1.5 The Memory Subsystem

The integrated memory interface controller (MIC) connects the system memory to the rest of the chip. On Sony PlayStation 3 256 MiB of RAMBUS eXtreme Data Rate Dynamic Random Access Memory (XDR DRAM) is used. The access to memory is provided through two XIO channels that operate at 3.2 GHz. Both RAMBUS channels can have eight concurrently operating banks. The CBEA memory subsystem has 16 banks, interleaved on cache line boundaries (on CBEA the cache line size is 128 bytes). Addresses 2 KiB apart access the same bank. System memory throughput is maximized if all memory banks are uniformly accessed.

With both XIO channels operating at 3.2 GHz, the peak raw memory bandwidth is 25.6 GiB/s. However, normal memory operations such as refresh, scrubbing, and so on, typically reduce the bandwidth by about 1 GiB/s. The peak bandwidth assumes that all the banks are kept active all the time by the incoming request streams, and requests are all of the same type (read or write), and each is exactly 128 bytes in size. If streaming reads and writes are intermingled, the effective bandwidth can be reduced to about 21 GiB/s ; the bandwidth loss in this case arises from the overhead of turning around the MIC-to-XIO bidirectional bus [6].

The CBEA supports three concurrent memory page sizes - 4 KiB and any two of 64 KiB, 1 MiB, or 16 MiB. For maximum memory throughput it is advisable to allocate large data sets from large pages. Using larger pages helps to reduce Translation Lookaside Buffer (TLB) miss penalties. TLB is a cache used to improve the speed of virtual address translation. It has a fixed number of entries that translate virtual addresses into physical addresses. If a virtual address is not in the TLB then external memory must be accessed to find the needed address information. With larger pages there is a bigger chance that the needed page is already present in TLB and thus no extra lookups are necessary for address translation.

1.6 The Element Interconnect Bus

Communication between the SPEs, PPE, external I/O devices and the system main memory goes through the Element Interconnect Bus (EIB) [14, 17]. As EIB interconnects all the processing elements and the system memory, the main design goals of the EIB where to provide a bus with enough throughput to satisfy the needs of all the units on the bus.

The EIB consists of a address bus and four 16 byte wide data rings. Two of these rings run clockwise and the other two counter-clockwise. Each of these rings can support up to three concurrent data transfers as long as their paths do not overlap. Scheduling of the transfers is handled by the EIB data bus arbiter. The arbiter selects on of the two rings that travel in the direction of the shortest transfer path. This ensures that the data will not need to travel more than halfway around the ring to reach its destination. The arbiter also makes sure that the transfers will not interfere with other in-flight transactions. The EIB operates at the half of the processors frequency and its maximum theoretical bandwidth is 204.8 GiB/s.

The actual bandwidth achieved on the EIB ranges from 78 to 197 GiB/s (see Table 1.4) and depends on several factors, including the position of the source and destination units, the chance of a new transfer interfering with the transfers in progress, the number of Cell chips in the system, whether data transfers are to or from memory or between local stores in the SPEs, and the data arbiter's efficiency.

Reduced bus bandwidths can result in the following cases :

- All requestors access the same destination, such as the same local store, at the same time.
- All transfers are in the same direction and cause idling on two of the four data rings.
- A large number of partial cache line transfers lowers bus efficiency.
- All transfers must travel halfway around the ring to reach their destinations, inhibiting units on the way from using the same ring.

1.7 Computing performance

Operating at 3.2 GHz each SPE has a theoretical peak of 25.6 GFLOPS for single precision floating point operations. This level of performance can be attained only if an instruction that produces two floating point operations per vector element is used. An example of such an instruction is the fused multiply add which computes $d = a*b+c$. On SPE the intrinsic that corresponds to this instruction is `spu_madd`. The 25.6 GFLOPS comes from the following formula $3.2*4*2 = 25.6$ where 3.2 is the operating frequency

Test configuration	Aggregate EIB bandwidth

SPE1 <-> SPE3, SPE5 <-> SPE7
SPE0 <-> SPE2, SPE4 <-> SPE6
	186 GiB/s

SPE0 <-> SPE4, SPE1 <-> SPE5
SPE2 <-> SPE6, SPE3 <-> SPE7
	197 GiB/s

SPE0 <-> SPE1, SPE2 <-> SPE3
SPE4 <-> SPE5, SPE6 <-> SPE7
	197 GiB/s

SPE0 <-> SPE3, SPE1 <-> SPE2
SPE4 <-> SPE7, SPE5 <-> SPE6
	197 GiB/s

SPE0 <-> SPE7, SPE1 <-> SPE6
SPE2 <-> SPE5, SPE3 <-> SPE4
	78 GiB/s

SPE0 <-> SPE5, SPE1 <-> SPE4
SPE2 <-> SPE7, SPE3 <-> SPE6
	95 GiB/s

SPE0 <-> SPE6, SPE1 <-> SPE7
SPE2 <-> SPE4, SPE3 <-> SPE5
	197 GiB/s

Table 1.4 – Sustained EIB bandwidth achieved for some SPE-to-SPE DMA transfers.

of the SPE, 4 is the number of vector components, and 2 is the number of flops produced for each vector component. Similarly for operations that do one floating point operation per vector element, such as multiplication (`spu_mul`) and addition (`spu_add`), the peak is $3.2 * 4 = 12.8$ GFLOPS.

For double precision floating point the SPE used in the Sony PlayStation 3 has a theoretical peak of 1.83 GFLOPS. Each double precision operation takes 13 cycles to complete and can be issued every 7 cycles. This is because, unlike the single precision instructions, the double precision instructions are not fully pipelined on the SPE. After issuing a double precision instruction there is a 6 cycle stall, during which no other instruction can be issued. A double precision instruction does a maximum of 4 FLOPS (fused multiply add on a vector containing 2 doubles). For this $3.2/7 * 4 \sim 1.83$ GFLOPS (round the first division 6 places after decimal because of the GHz). For real world applications this level of performance may be out of reach because no other instruction can be issued at the same cycle as a double precision instruction and for the next 6 cycles. This gives a effective 7 cycle stall during which no other instruction can be issued. Therefore, if a double precision instruction is issued immediately after the stall of the previously issued instruction ends there is no room for load/store as a new stall begins. Assuming that we will need to perform one load and one store for each arithmetic operation then we will get that a double precision can be issued every 9 cycles, this gives us roughly 1.42 GFLOPS. For newer generation Cell processors, that have fully pipelined double precision arithmetics, the theoretical peak is $3.2 * 2 * 2 = 12.8$ GFLOPS per SPE.

Clocked at 3.2 GHz, the PPE can theoretically deliver $3.2*2 = 6.4$ GFLOPS of double precision floating point performance from its fully pipelined floating point unit using the fused multiply add operation. It can also deliver $3.2 * 4 * 2 = 25.6$ GFLOPS of single precision floating point performance from its VMX unit using 4-way SIMD fused multiply add operation. Although the PPE looks like quite a potent processor, its main purpose is still to serve as a controller and supervise the work of the other cores on the chip. Thanks

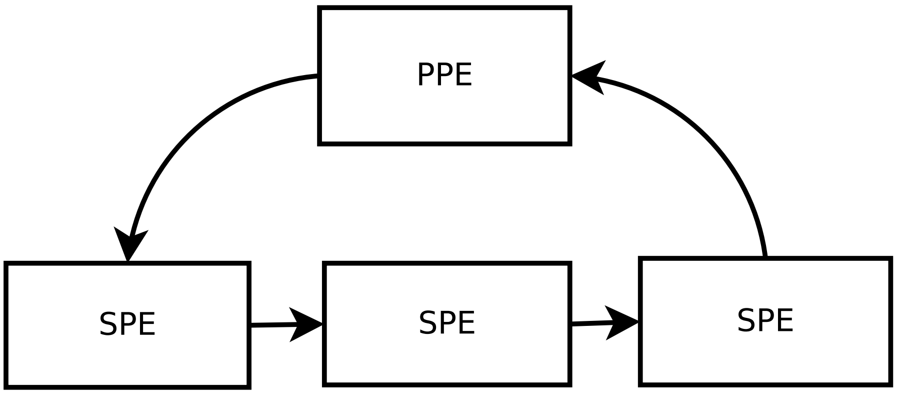

Figure 1.4 – Multistage pipeline.

to the PPE's compliance with the PowerPC architecture, existing applications can run on the Cell out of the box, and be gradually optimized for performance using the SPEs, rather than written from scratch.

1.8 Programming model

While Cell offers huge potential for computing performance, it does not come free. Just recompiling existing applications for Cell will not magically make them run faster, as the recompiled code will only run on the PPE, but the bulk of the Cell's performance is provided by the SPEs. To make full use of the Cell's computing power the application has to be partitioned across the processing elements. For this there are two main ways : the PPE-centric and the SPE-centric.

In the PPE-centric model, the main application runs on the PPE, and individual tasks are off-loaded to the SPEs. This approach is suitable for most of the applications, as only performance critical parts need to be implemented on the SPE and most of the application control logic can be left unchanged. The PPE will coordinate the work by dispatching the tasks to SPEs and collect the results after the tasks have completed. There are three ways in which the SPEs can be used in the PPE-centric model :

Multistage pipeline model (Figure 1.4). If a task requires sequential stages, the SPEs can act as a multistage pipeline. Different program is loaded into each of the SPEs and the SPEs are chained together so that data is passed through them sequentially. The stream of data is sent from the system main memory to the first SPE, which performs the first stage of the processing. The first SPE then passes the data to the next SPE for the next stage of processing. After the last SPE has done the final stage of processing on its data, that data is returned to the system main memory. As with any pipeline architecture parallel processing occurs with having various portions of data in different stages of being processed. This approach is well suited if processing the data can be divided into multiple steps that can be performed in serial order. As the available bandwidth between the SPEs is much larger than the bandwidth between the system main memory and the SPEs, avoiding transferring the same data to and from the main memory is likely to result in a

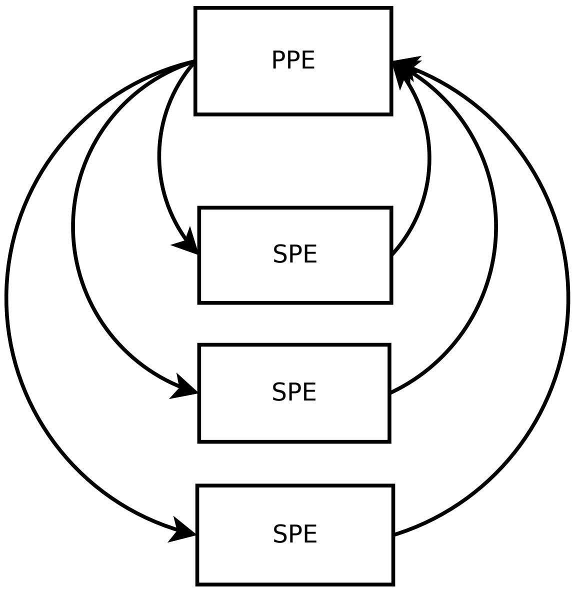

Figure 1.5 – Parallel stages model.

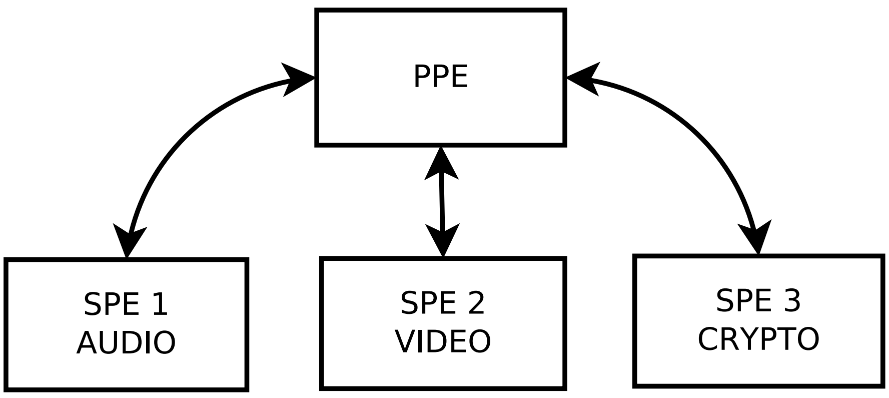

Figure 1.6 – Services model.

performance boost. As a downside, to keep all of the pipeline busy the processing stages should be of roughly the same size. Dividing the processing algorithm in such a way may be a difficult task.

Parallel stages model (Figure 1.5). If a task has a large amount of data that can be partitioned and acted on at the same time, then SPEs can be used to process different portions of that data in parallel. Here, the same program is loaded into each of the SPEs and the data is divided evenly between them. This approach is suitable for data-parallel problems. Partitioning data into chunks of roughly equal size is usually a lot simpler than dividing the processing stages so that they would take about the same amount of time, therefore, using this method is generally easier than constructing a pipeline from the SPEs.

Services model (Figure 1.6). The third way in which SPEs can be used in a PPE-centric paradigm is the services model. In the services model, the PPE assigns different services to different SPEs, and the PPE's main process calls upon the appropriate SPE when a particular service is needed. This approach is suitable if the SPEs are only used to accelerate some specific tasks, for example encryption or decoding a video stream.

In the SPE-centric model, most of the application code is distributed among the SPEs. The PPE acts as a centralized resource manager for the SPEs. The PPE may provide some services to the SPE code that can not be implemented on the SPE. Each SPE fetches its next work item (what function to execute, data to use etc.) from main storage (or its own local store) when it completes its current work.

The Cell processor allows for multiple different programming models, each with their own pros and cons. Application developers should choose carefully to find the model that matches their needs. For existing application using the PPE-centric models let developers leave most of their program structure intact and only port a small part of the functionality that would benefit the most from the extra computation power provided by the SPEs. Applications built specifically for the Cell architecture from the ground up can leverage both programming models to maximize their performance.

Chapitre 2

Conjugate gradient method

In this chapter I am going to give a brief overview of the Conjugate Gradient (CG) method. For a complete description of the method refer to Jonathan Richard Shewchuk's paper titled An Introduction to the Conjugate Gradient Method Without the Agonizing Pain [21].

CG is an iterative numerical method for solving systems of linear equations. The system is represented by the formula

$$Ax = b$$

where A is the *system matrix*, b is a known vector called the *right hand side* (RHS) vector, x is the vector of unknowns that represents the sought solution. For the CG algorithm to work the matrix A must be square, symmetric ($A^T = A$) and positive definite ($x^T Ax > 0$ for every non-zero vector x).

The quadratic form is described by the equation

$$f(x) = \frac{1}{2} x^T Ax - b^T x + c$$

where A is a matrix, x and b are vectors, and c is a scalar constant. If A is symmetric and positive definite then $f(x)$ is minimized by the solution to $Ax = b$. The gradient of the quadratic form is defined to be

$$f'(x) = \begin{bmatrix} \frac{\partial}{\partial x_1} f(x) \\ \frac{\partial}{\partial x_2} f(x) \\ \vdots \\ \frac{\partial}{\partial x_n} f(x) \end{bmatrix}$$

The gradient points to the greatest increase of $f(x)$ for every point x. The solution of the $f(x)$ is minimized at the point where the gradient is is zero ($f'(x) = 0$). From the previous equations one can derive

$$f'(x) = \frac{1}{2} A^T x + \frac{1}{2} Ax - b$$

In case A is symmetric this can be reduced to

$$f'(x) = Ax - b$$

Setting gradient to zero in the last equation yields $Ax = b$ which is the formula for the initial linear system. Hence, the solution to $Ax = b$ is a critical point of $f(x)$. In case of a symmetric and positive define matrix this is the minimum of $f(x)$.

The idea of the CG is to find the minimum of $f(x)$ instead of solving the linear system $Ax = b$. To do this the CG algorithm starts with a initial guess x_0 and gradually on each iteration moves closer to the solution. The main improvement of CG over the similar Steepest Descent [21] method is in the faster convergence, which is achieved by better correction of the guessed solution on each iteration of the algorithm.

The starting condition can be a rough estimate of the solution. If there is no estimation available the zero vector will also do as CG will eventually converge to the solution of the system. The ending condition of the CG algorithm is when the convergence is complete and the exact result is found, but due to the rounding errors CG implementations usually stop when the residual falls below a specified value. The conjugate gradient method in pseudocode appears in Algorithm 2.1.

Iterative methods like CG are well-suited for use with sparse matrices. For dense matrices solvers based on factoring the matrix are able to quickly solve the system for different values once the factorization is found. In case of a sparse matrix the triangular matrices produced while factoring may contain a lot more non-zero elements than the initial matrix. Due to the increased memory usage the factoring may be impossible or very time consuming. Iterative methods on the other hand are memory efficient with spares matrices and run quickly.

2.1 Practical uses of the conjugate gradient method

CG can be used to solve a wide variety of practical problems that require solving sparse systems of linear equations. For example many of the engineering problems contain partial differential equations that are numerically solved through linear systems. One of the problems that can be solved by CG is the steady-state heat equation. This equation deals with the problem where we have a physical system with a mix of materials, heat sources and heat sinks and we want to know the temperature of different parts of the system when the system has stabilized. The steady-state heat equation is a second order partial differential equation

$$-k\nabla^2 T = f$$

where

- ∇ is nabla operator, 3 dimensional case $\nabla = i\frac{\partial}{\partial x} + j\frac{\partial}{\partial y} + k\frac{\partial}{\partial z}$ where i, j, k are unit basis vectors
- k is a material specific quantity depending on the thermal conductivity, the density and the heat capacity
- T is a temperature function

– f is a heat source and sink function

Algorithm 2.1 The conjugate gradient algorithm in pseudocode.

```
 1: $A$ : matrix of linear equation system
 2: $b$ : know right hand side vector
 3: $x$ : initial guess
 4: **if** initial guess is a zero vector **then** $r := b$
 5: $r := b - A \cdot x$
 6: $p := r$
 7: $k := 0$
 8: **while** $k < maximum\,number\,of\,iterations$
 9:     $d := A \cdot p$
10:     $dr := r \cdot r$
11:     $\alpha := \frac{dr}{d \cdot p}$
12:     $xn := x + \alpha \cdot p$
13:     $rn := r - \alpha \cdot d$
14:     $drn := rn \cdot rn$
15:     **if** $drn < acceptable\,precision$
16:         soultion found after k iterations
17:         **return** $xn$
18:     $\beta := \frac{drn}{dr}$
19:     $pn := rn + \beta \cdot p$
20:     $k := k + 1$
21:     $x := xn$
22:     $r := rn$
23:     $p := pn$
```

Chapitre 3

Programming conjugate gradient

This chapter will introduce a C language implementation along with the necessary data structures for the conjugate gradient method that was described previously. Besides the implementation of the conjugate gradient method there will also be some discussion about sparse matrix formats and parallelizing the conjugate gradient method.

3.1 Matrix representation

The linear equation system that is solved by the conjugate gradient method can be very large, even so large that storing the entire system matrix in memory is impossible. Fortunately the system matrix is usually a sparse matrix. A matrix is called sparse if it mostly contains zeroes. Therefore, instead of storing all of the values in the matrix as an array it is practical to store only the non-zero values of the matrix. As most of the values in the matrix are zeros this will consume a lot less memory and also increase computational efficiency by avoiding doing computations on the zeros. One way is to encode the non-zero values of the matrix as an array of triplets (i, j, v) where

- i is the row number
- j the column number
- v the value at the specified matrix position

Depending on whether the matrix values are ordered firstly by the row and then column or vice versa, this is either called the row-major or the column-major format. For symmetric matrices, it is sufficient to only store the upper or lower triangular half of the matrix. Therefore, the following matrix

$$\begin{matrix} 1 & 2 & 0 & 0 \\ 0 & 3 & 4 & 0 \\ 0 & 0 & 5 & 6 \\ 0 & 0 & 0 & 7 \end{matrix}$$

in row-major format is represented by

$$(1,1,1),(1,2,2),(2,2,3),(2,3,4),(3,3,5),(3,4,6),(4,4,7)$$

Similarly in column-major format by

$$(1,1,1),(2,1,2),(2,2,3),(3,2,4),(3,3,5),(4,3,6),(4,4,7)$$

Instead of triplets the matrix could also be described with three arrays, each containing values of one triplet component.

Alternatively in the compressed row storage format [4] the matrix is represented by three arrays. All the non-zero values of the matrix are placed in the first array named *values*. The second array named *row_start* contains the start index of each row in the first array. The third array named *coulmn_indices* holds the column index of each matrix element. For the same matrix as used previously we get $values := (1,2,3,4,5,6,7)$, $row\_start := (1,3,5,7)$ and $colum\_indices := (1,2,2,3,3,4,4)$

3.2 Basic operations

Besides computations with scalar values the CG algorithm contains vector-vector and matrix-vector operations. Implementation of these is relatively straightforward. Given the following data structures

```
typedef struct {
    unsigned int x;
    unsigned int y;
    double value;
} sparse_matrix_el_t;
typedef struct {
    sparse_matrix_el_t *data;
    unsigned int size;
} sparse_matrix_t;
typedef struct {
    double *data;
    unsigned int size;
} vector_t;
```

the operations necessary for the CG algorithm can be implemented as follows :

– vector-scalar multiplication

```
void multiply(vector_t* result, double k, const vector_t* a) {
    for (unsigned int i = 0; i < a->size; ++i) {
        result->data[i] = k * a->data[i];
    }
}
```

– inner product of two vectors

```
double dot(const vector_t* a, const vector_t* b) {
    double sum = 0;
    for (unsigned int i = 0; i < a->size; ++i) {
        sum += a->data[i] * b->data[i];
    }
    return sum;
}
```

– vector-vector addition

```
void add(vector_t* result, const vector_t* a,
        const vector_t* b) {
    for (unsigned int i = 0; i < a->size; ++i) {
        result->data[i] = a->data[i] + b->data[i];
    }
}
```

– vector-vector subtraction

```
void sub(vector_t* result, const vector_t* a,
        const vector_t* b) {
    for (unsigned int i = 0; i < a->size; ++i) {
        result->data[i] = a->data[i] - b->data[i];
    }
}
```

– matrix-vector multiplication

```
void multiply(vector_t *result, const sparse_matrix_t *smat,
        const vector_t *vec) {
    unsigned int j = 0;
    for (unsigned int i = 0; i < result->size; i++) {
        double sum = 0;
        while (j < smat->size && smat->data[j].y == i) {
            sum += smat->data[j].value
                * vec->data[smat->data[j].x];
            j++;
        }
        result->data[i] = sum;
    }
}
```

3.3 Implementation of the Conjugate Gradient algorithm

Using the data structures and primitive operations presented in the previous section the conjugate gradient algorithm can be implemented in the following way :

```
int cg(vector_t* result, const sparse_matrix_t *A,
        const vector_t *b, const vector_t *x0,
        double precision, unsigned int iterations) {
    // # Start with initial guess x0
    vector_t *x = clone_vector(x, x0);
    // # r := b - A * x
    vector_t *tmp1 = multiply(A, x);
    vector_t *r = sub(b, tmp1);
    free_vector(tmp1);
    // # p := r
    vector_t *p = new_vector(r->size);
    copy_vector(p, r);
    // # Iteration counter
    int k = 0;
    vector_t *xn = new_vector(r->size);
    vector_t *d = new_vector(r->size);
    vector_t *rn = new_vector(r->size);
    vector_t *pn = new_vector(r->size);
    while (k < iterations) {
        multiply(d, A, p);
        double dr = dot(r, r);
        // #            r * r
        // # alpha = ---------
        // #          p * A * p
        double alpha = dr / dot(d, p);
        // # xn = x + alpha*p
        multiply(xn, alpha, p);
        add(xn, xn, x);
        // # rn = r - alpha*dot(A,p)
        multiply(rn, alpha, d);
        sub(rn, r, rn);
        // # If acceptable precision is reached finish
        double drn = dot(rn, rn);
        if (drn < precision) {
            swap_vector(x, xn);
            break;
        }
        // #          rn * rn
        // # beta = -------
        // #           r * r
        double beta = drn / dr;
        // # pn = rn + beta*p
        multiply(pn, beta, p);
        add(pn, rn, pn);
        k++;
        swap_vector(x, xn); swap_vector(r, rn); swap_vector(p, pn);
    }
    // # Copy the result
    copy_vector(result , x);

    free_vector(d);
    free_vector(xn); free_vector(rn); free_vector(pn);
    free_vector(x); free_vector(r); free_vector(p);
    // # Return the number of iterations performed
    return k;
}
```

3.4 Parallelization

The time-consuming parts of the conjugate gradient method are
- inner products
- vector updates (addition, subtraction and multiplication with scalar)
- matrix-vector product

Out of these the vector update operations can easily be parallelized by dividing the vectors to segments and letting each processor do the update only on the segment assigned to it. Inner products can be parallelized in a similar fashion by letting each processor compute the product for its assigned segment. When processors have completed computing on their segment the results would be collected and summed to get the final result. The summing of the partial results introduces a synchronization point where all processors have to wait for the final result before they can proceed with the computation. Variations of the conjugate gradient method [4, 12] have been proposed that allow for overlapping the communication needed for inner product with subsequent operations.

On shared memory machines the matrix-vector product can be parallelized, similarly to the vector update operations, by letting each processor handle a segment of the result vector. For distributed-memory machines the whole input vector would need to be distributed to each of the processors, which can cause a large communication overhead. However, many sparse matrix problems have a matrix where element a_{ij} is nonzero only if i and j are close. In such a case, each processor may need input vector values only from the processors computing on neighboring matrix blocks. If the number of needed values is small then the computation can be overlapped with communication.

Chapitre 4

Implementing conjugate gradient on the Cell

This chapter demonstrates how the CG algorithm can take advantage of the Cell hardware. To accelerate CG on the Cell one way would be to implement the whole CG algorithm on the SPEs. The matrix and the vectors used in the algorithm would need to be divided into segments so that each SPE would get its own data range. This design would be similar to a CG implementation on distributed memory machines only instead of the Message Passing Interface (MPI), that is typically used for communication between nodes, DMA between SPEs and PPE would be used. Similarly to the distributed memory solution vector inner products would be a synchronization point where all SPEs have to exchange data to reach the final result. The second approach would be to delegate the time consuming parts of the algorithm to the SPEs. With this approach the main CG algorithm would be implemented on the PPE while the computation intensive parts such as :

- vector inner product
  - inner product between two different vectors
  - inner product of one vector (so we wouldn't need to move the same vector twice to SPEs)
- vector update operations
  - multiply vector with scalar and add to another vector, uses `spu_madd` intrinsic
  - multiply vector with scalar and subtract the result from another vector, uses `spu_nmsub` intrinsic
- vector - sparse matrix product

would be implemented on the SPEs. These operations would split the computation between the SPEs and when all of the SPEs are done with their part the PPE could gather the results and move on with the next operation. The implementation of all of these functions follows roughly the same outline.

- On PPE

   1. Submit work to each of the SPEs. Every SPE gets a separate data range to compute on.
   2. Wait while the SPEs complete their computations.

3. Post process the results from the SPEs. For example add together the vector inner product results computed by the SPEs to get the final result.

- On SPE
  1. Block and wait a notification from the PPE to start working.
  2. Transfer a small control structure from the main memory that is prefilled by the PPE with the details of the requested task, such as the operation type and memory addresses of the input/output data.
  3. Compute the result and DMA the result back to the system main memory.
  4. Notify the PPE that work is done.

On the SPE side the general idea is to overlap computation with data transfer by using double or multi buffering. This is needed due to the limited size of the local store, which makes it impossible to have all the necessary data ready on the SPE. The buffer size is limited by the maximum size of the DMA transfer, which is 16 KiB. This means that each buffer can contain up to 2048 double precision floating point values, or 1024 128-byte quad-words. To make the SPE code simpler and more efficient all the vectors should be padded with zeros so that we could always compute on full blocks and would not need to worry about figuring out whether the current block is a bit shorter than the other blocks or not. The outline of the implemented operations follows.

Inner product between two different vectors.

```
-- given source vectors a and b
i := 0
start transfer of the first block of vector a to buffer A[i]
start transfer of the first block of vector b to buffer B[i]
do
    start transfer of the next block of vector a to buffer A[1-i]
    start transfer of the next block of vector b to buffer B[1-i]
    wait for buffers A[i] and B[i] to complete transfer
    compute on buffers A[i] and B[i]
    i := 1 - i
while no more work
transfer result
wait for the result to be transferred
```

Inner product of one vector. This is similar to the inner product between two different vectors case, the only difference is that there is only one source vector. Thus the amount of data transferred is cut in half.

```
-- given source vector a
i := 0
start transfer of the first block of vector a to buffer A[i]
```

```
do
    start transfer of the next block of vector a to buffer A[1−i]
    wait for buffer A[i] to complete transfer
    compute on buffer A[i]
    i := 1 − i
while no more work
transfer result
wait for the result to be transferred
```

Multiply vector with scalar and add to another vector, multiply vector with scalar and subtract the result from another vector.

```
−− given source vectors a and b and target vector result
i := 0
start transfer of the first block of vector a to buffer A[i]
start transfer of the first block of vector b to buffer B[i]
while no more work
    start transfer of the next block of vector a to buffer A[1−i]
    start transfer of the next block of vector b to buffer B[1−i]
    wait for buffers A[i], B[i], C[1−i] to complete transfer
    compute on buffers A[i] and B[i] to C[1−i]
    start transfer of the result buffer C[1−i]
    i := 1 − i
wait for the result to be transferred
```

Vector - sparse matrix product. The problem with implementing sparse matrix multiplication on Cell is that at least some data used for computation has bad locality. For example if we divide the output vector ranges between the SPEs similarly as done before, then the matrix can be ordered so that each SPE also gets a non-overlapping region of the matrix values, but each of the SPEs needs potentially all of the input vector values (or at least it is likely that the values needed for the block we are computing on come from different parts of input vector). As the SPE local store size is limited to 256 KiB we cannot usually fit the whole input vector in the local store. At the worst case we may even need to transfer some parts of the input vector multiple times to complete the computation. Also we need to make sure that all the DMA alignment and size requirements imposed by the hardware are met.

Assuming we have a spares matrix in row-major format there are two ways of representing it. One way is to store the matrix as an array of triplets $(row, column, value)$ and the other one is to create three arrays one for row indices, the second one for column indices and the third one for values. The first way is call array of structures (AOS) and the second one structure of arrays (SOA). The SOA layout is usually a better fit for SIMD computations as with this layout similar data is stored in memory contiguously, which allows for computing on two consecutive matrix values simultaneously.

As the first step we will need to prepare the input matrix so that it would be easier to compute on in parallel by the SPEs.

1. Add zeros to matrix so that each SPE could compute on full blocks. This means that when we divide the output vector between SPEs, we take the last row index computed by the first SPE and search the matrix for the last element on that row and insert enough padding so that the next SPE would get it's first matrix block on suitable boundary (DMA start addresses must be properly aligned).
2. To compute a block of output vector values we may need to transfer a lot more matrix values (multiple blocks), therefore, the matrix blocks should be aligned with the output vector blocks so that when computing of an output block is done we could throw away the corresponding matrix elements. This is again achieved by padding the matrix with zeros.
3. For each iteration make a copy of the input vector and arrange it so that we could compute on it sequentially.

Now we could implement the SPE part in the following way :

```
foreach output block
    transfer input vector block
    for each matrix block in current output vector block
        transfer matrix block
        compute
transfer output block
```

While this algorithm is rather simple it will not perform too well as rearranging the input vector for each iteration has a rather big overhead, so we will need to get rid of this.

As an alternative we could use DMA gather operation to fetch the input vector values corresponding to the matrix values. In this case the biggest issue with the Cell DMA gather is that the minimum size of a item is 16 bytes, which corresponds to two doubles. In our case we only need one of them. Up to 2048 items can be transferred at a time.

```
foreach output vector block
    foreach matrix block in current output vector block
        transfer matrix block
        transfer needed input vector values via DMA list
        compute
transfer output block
```

Now to make things more complicated on the SPE we need for each output vector block :

1. Matrix values (may be multiple blocks)
2. Matrix row indices (which matrix value goes to which row in output vector)
3. For each matrix block we need
   (a) values from input vector.

(b) as we can only get 16 bytes minimum aligned at 16 byte boundary we also need to know if it was the low or high double-word we actually needed. Alternatively we could add zeros between the matrix values so we could always assume that two consecutive matrix values correspond to two consecutive input vector values.

We can also use the fact that some input vector values are used multiple times during the computation on a matrix block (or output vector block) and precompute what is needed during initialization and cache them on the SPE, so that they would not be transfered multiple times. This kind of initialziation and precomputation needs to be done once per matrix. The worst case number of needed input vector values per matrix block is twice the size of the matrix block (because we can only get two input vector values at a time), but on average much less is needed.

```
-- given source matrix m and vector v and target vector result
i := 0
current := 0
next := 1
fetch := 2
-- fetch matrix and imput vector values
start transfer of the first block of matrix to A[current]
wait for buffer A[current] to complete transfer
start transfer of vector values for matrix in A[current]
start transfer of the next matrix block to A[next]
wait for buffer A[next] and vector for A[current]
start transfer of vector for matrix block in A[next]
start transfer of the next matrix block to A[fetch]

foreach output vector block
    wait for buffer C[i] to complete transfer
    foreach matrix block in current output vector block
        compute on A[current] and C[i]
        tmp := current
        current := next
        next := fetch
        fetch := tmp
        wait for matrix in A[next_buf_nr] and vector in A[current]
        start transfer vector for matrix in A[next]
        start transfer of the next matrix block to A[fetch]

    start transfer of the result buffer C[i]
    i := 1 - i
wait for the result to be transferred
```

The biggest challenge in implementing vector-sparse matrix product is finding a good balance between data preparation and the SPE code complexity and speed. If the matrix values are rearranged on each iteration the SPE computation can be fast and simple, but the rearranging itself will be time consuming.

On the other hand if matrix is only prepared so that the SPE could always DMA blocks of some fixed size then the SPE could potentially need input vector values from all over the vector. This in turn can cause a lot of data transfer for the input vector values. In this case data preparation is simple but the SPE code needs to do too much work to find input vector values.

The third option is to use a DMA list to fetch the input vector values. With this only the necessary input vector elements are fetched, but as the minimum transfer size of a list element is two double precision floating point numbers (16 bytes) instead of the one double (8 bytes) that is need the inner loop of the computation needs to figure out which of the fetched doubles was actually needed. This in turn slows down the inner loop. To improve the inner loop's performance by eliminating the expensive conditional operations the matrix data can be further padded with zeros so that the inner loop could assume that two consecutive input vector and matrix values can be multiplied by each other in one SIMD operation. This allows for a simpler and faster inner loop at the cost of increasing the size of the input matrix.

As a further improvement it is possible to precompute which input vector values are needed at which computation stage and use as much of the SPE local store space that is available as a cache. Without caching the number of transferred input vector elements would depend on the number of the matrix values. With caching it is likely that only a few vector elements will need to be transferred more than once.

The SPEs will only give optimal performance if data transfers can be overlapped with computations. Therefore the inner loop should be fast enough to complete before the data for the next computation is fetched from the main memory. Similarly the data size should be small enough that the DMA transfers can complete before the data is needed for computations. As the vector-sparse matrix product is the most time consuming operation in the CG, the overall performance of the CG implementation will depend heavily on the implementation of this operation.

Chapitre 5

Estimating conjugate gradient performance on the Cell

This chapter gives an estimate on how fast the Cell implementation of the CG will perform and compare it with the actual results. A common estimate of computer hardware performance is Floating point Operations Per Second (FLOPS). Like many other commonly used performance metrics such as frequency in GHz and Million of Instructions Per Second (MIPS) the FLOPS usually does not give a good overview of the actual processor speed. While the theoretical peak FLOPS can be quite easily computed if the processor frequency, clocks per instruction and number of parallel instructions are known, the number of FLOPS attained by running real applications may be very different. Besides raw floating point performance, the actual performance is also influenced by cache misses, memory access latencies and other hard to model details. This makes estimating the performance of the algorithm without benchmarking complicate. In contrast to most commonly used processor architectures Cell SPEs behave very predictably and are capable of achieving near theoretical peak performance.

5.1 Flops and CG

To estimate the performance of the CG we will firstly need to know the number of FLOPS performed on each iteration of the method. On every iteration the CG algorithm makes :

- one vector-sparse matrix product
- three vector inner products
- three vector-scalar multiplications
- two vector additions
- one vector subtraction

Given a vector of size n and a matrix with m non-zero elements these operations need :

- a vector-sparse matrix product needs roughly m multiplications and m additions
- a vector-vector dot product needs n multiplications and $n - 1$ additions
- a vector scalar multiplication needs n multiplications
- a vector addition needs n additions

– a vector subtraction needs n subtractions

This adds up to approximately

$$(m+m)+3(n+n-1)+3n+2n+n=2m+12n-3\approx 2m+12n$$

floating point operations on each iteration.

Secondly we will need to know if the operations performed in the CG algorithm are bound by the data transfer or by the computation speed. As described previously, SPE can issue one double precision instruction in 7 cycles so $3.2/7\sim 0.45$, which means that a SPE can execute 0.45×10^9 double precision instructions per second. If we take that one double precision instruction in 9 cycles then we get $3.2/9\sim 0.36$. Assuming that the data transfer rate is 25.6 GiB/s and 6 SPEs gives $25.6/6\sim 4.27$ GiB/s of bandwidth per SPE. $25.6/6/8\sim 0.54$, which means that we can move at most 0.54×10^9 doubles between the main memory and the SPE in one second. Taking 21 GiB/s as the peak transfer rate gives $21/6/8\sim 0.44$. To estimate whether an operations is bound by the transfer or computation we will need to compare the number of double precision instructions with the number of double precision floating point values transferred.

Vector inner products.

- inner product between two different vectors
  This operation needs two input vectors and has a single output value. Elements are processed two at a time from both inputs. The number of double precision instructions needed to process the data is $0.54/4=0.135$. $0.135<0.45$ which means that this operation is bound by the data transfer speed.
- inner product of one vector
  Only one input vector is needed, which gives us $0.54/2=0.27<0.45$ double precision instructions needed.

Vector update operations.

- multiply vector with scalar and add to another vector,
- multiply vector with scalar and subtract the result from another vector
  Both of these operations have two input vectors and one output vector, which gives us $0.54/3/2=0.09<0.45$

Vector-sparse matrix product. Matrix has m elements, vector has n elements, transfer overhead is t (we need to transfer matrix element coordinates too). From this $m+2n+t=0.54$. As we are doing two floating point operations with each matrix element we get $2m/2=0.45$ which gives $m=0.45$ from this $n+t=0,045$. This means that if transfer overhead is low and the size of the vector is considerably smaller than the matrix, then this operation could be bound by the speed of the computation. In my implementation the transfer overhead is at least $t=\frac{m}{4}+\frac{m}{32}=\frac{0.45}{4}+\frac{0,45}{32}=0,127$. Thus we can assume that this operation is also bound by the data transfer.

As it turns out all the operations should be bound by data transfer rates. This is due to the fact that the operations used usually do only one or two floating point operations for each value. While estimating the required bandwidth is relatively easy, accurate estimation of instructions is a bit more difficult as the floating point instructions are not the only instructions executed.

The inner loops of the computations should be arranged so that the overhead of the non floating point instructions is minimal (the ratio of floating point instructions to other instructions is large), there are no branch prediction misses, there are no data dependencies between floating point instructions and SIMD is used to compute on two values simultaneously. Assuming that these conditions hold, then the expected GFLOPS number for CG implementation on the SPE is determined by the data transfer rate.

Given a vector of size vs and a matrix with ms elements lets denote the vector size after padding as pvs and the matrix size as pms, the number of input vector values used on each iteration as ivs. Then the amount of data transferred on each iteration of CG can be described with formula

$$ts = 14pvs + pms + ivs + t$$

where t is transfer overhead, t is an implementation dependent quantity, which in my version of CG is $t = \frac{pms}{4} + \frac{pms}{32}$. Knowing the data size on each iteration we can estimate the running time of CG with

$$rt = i \times \frac{ts}{tr}$$

where rt is the running time, i the number of iterations and tr is the transfer rate. The performance of the CG can now be represented by the formula

$$flops = \frac{2ms + 12vs}{rt}$$

where $flops$ is the number of floating point operations performed by CG in one second.

5.2 Actual and estimated performance

Besides the speed of the computation operations the actual performance attained is also influenced by the ratio of useful to total work done. The wasted work arises from padding the data with extra zeros that do not influence the end result but are still used in computations. The observed performance may also be affected by uneven data distribution to the SPEs where some SPEs recieve a larger data set and need more time to complete their operations.

For testing I used matrix s3dkt3m2 obtained from the matrix market [2]. This matrix originates from finite element analysis on a cylindrical shell. The s3dkt3m2 matrix is a symmetric positive definite matrix, which makes it suitable for CG. It has dimensions of 90499 by 90499 and contains 3753461 elements (including explicit zero elements present in source data). The average number of non-zero elements in a row is 21.25. The right hand side vector used in CG was obtained by multiplying the matrix with vector $(1 \ldots 1)^T$.

After padding the matrix contained 4051968 elements (8% growth), as the vector was of suitable length no padding was necessary. The number of input vector elements transferred on each iteration was 110576 (22% overhead). The estimated performance figures for running 10000 iterations of CG are brought out in Table 5.1.

Transfer rate (GiB/s)	Expected time (s)	Expected GFLOPS
10	48.9436388	1.75566717
18	27.19091045	3.160200912
19	25.7598099	3.33576763
20	24.4718194	3.51133435
21	23.30649467	3.68690107
22	22.24710855	3.86246778
25.6	19.11860891	4.49450796

Table 5.1 – Expected performance for 10000 iterations.

	Time (s)	MFLOPS effective	MFLOPS done	Transfer (GiB/s)
CG	857.530171	3631.458393	3884.019071	20.685565
on SPE	829.024153	3756.326187	4017.571172	21.396839
dot1	20.317253	6453.856791	6457.210407	23.740455
dot2	20.586428	3184.735051	3186.389935	23.740455
madd	77.681847	1687.970144	1688.847264	18.874339
nmsub	38.703142	1693.979248	1694.859490	18.941531
sparsemul	671.735483	4050.271704	4372.383605	21.677794

Table 5.2 – Actual performance for 362427 iterations, with huge pages.

The actual results were obtained by running CG for 362427 iterations. The test was run with keeping data in large 16 MiB memory pages to maximize memory throughput. Results of the test are presented in Table 5.2 . Table 5.3 shows the results for the same test with normal 4 KiB memory pages. For comparison the expected performance figures for the same number of iterations can be viewed in Table 5.4 .

The total time in the results was measured by CPU clock for the whole CG algorithm, time spent on reading the matrix from the disk and preparing data was not measured. The time spent on the SPEs during various computations was measured using SPE hardware timer facilities with `spu_decrementer` function. On PlayStation 3 the hardware timer operates withe frequency of 79.8 MHz. `spu_decrementer` counts the number of ticks elapsed while performing a certain operation, to get the actual time the number of ticks is multiplied with the timer frequency.

As can be seen from the results about 97% of total time is spent on the SPEs, the

	Time (s)	MFLOPS effective	MFLOPS done	Transfer (GiB/s)
CG	945.692228	3292.916072	3521.931808	18.757156
on SPE	914.761040	3404.260786	3641.020325	19.391399
dot1	22.284961	5883.996836	5887.054337	20.401350
dot2	23.955825	2736.800715	2738.222839	20.401350
madd	80.660381	1625.638732	1626.483462	18.177369
nmsub	39.916310	1642.494496	1643.347985	18.365845
sparsemul	747.943563	3637.588922	3926.880793	19.469041

Table 5.3 – Actual performance for 362427 iterations, without huge pages.

Transfer rate (GiB/s)	Expected time (s)	Expected GFLOPS
10	1773.849618	1.755667174
18	985.47201	3.160200912
19	933.6050621	3.33576763
20	886.924809	3.511334347
21	844.6902943	3.686901065
22	806.2952809	3.862467782
25.6	692.910007	4.494507964

Table 5.4 – Expected performance for 362427 iterations.

remaining 3% is used on the PPE for running the algorithm's main loop and orchestrating the work of the SPEs. By far the most time consuming operation is the vector-sparse matrix product which accounts for 78% of the total execution time and 81% of the SPE execution time. The vector-sparse matrix product is performed at a speed a bit over 4 GFLOPS. Although the matrix used for testing is not the same as used by Williams et. al. in [24] the performance of vector-sparse matrix product is at a comparable level.

Computations on the SPE attain 84% of the theoretical peak due to the reduced data transfer rates compared to the maximum of 25.6 GiB/s. Degraded data transfer rates can be explained by the overhead of mixing streaming memory read and write operations, which can reduce the attained throughput down to 21 GiB/s [6]. Further benchmarking would be necessary to determine the maximum possible memory bandwidth for given workload on PlayStation 3 hardware. In case we consider the maximum available throughput to be 22 GiB/s then the computations operate at 97% from the peak. Using large memory pages gives a roughly 10% boost over small 4 KiB pages. For the test matrix useful work accounts for 94% of the total work done.

Conclusions

In the context of high performance computing the emergence of multi-core architectures makes it important to understand the most effective designs to utilize these systems. To harness the full power of these new architectures the code needs to be parallelized across several forms of parallelism, ranging from data level SIMD parallelism to multi-threading. In this paper I examined implementing conjugate gradient method, a popular iterative solver for sparse linear systems, on Cell Broadband Engine, a novel heterogeneous multi-core processor architecture, that exploits the parallel execution capabilities of the underlying hardware.

One of the most attractive features of the Cell processor is its simple architecture that gives programmers full control over the processors functionality. On most common processors performance depends heavily on the cache memories over whose behavior programmers have no directly control. The predictable nature of Cell gives programmers better understanding of the bottlenecks in their code, which in turn makes it possible to write highly efficient applications. As a result Cell offers developers a straightforward and easy to model development platform that allows real applications to attain performance near the theoretical peak performance of the processor, much more than what can be achieved on traditional cache based systems. However the high performance and the predictability comes at a cost. Implementing efficient and fast code for the Cell processor is a difficult task, that forces programmers to constantly pay attention to such low level details as data alignment, data dependencies and branch prediction. For high performance it is often necessary to write code using intrinsics which are on the border line between high level languages and the assembly language. Unleashing the full potential of the Cell requires a lot of effort, good knowledge of the underlying hardware and careful planning. Without investing time into learning the peculiarities of the processor it is only possible to release a small fraction of Cell's potential performance.

Despite the difficulties of programming Cell the proposed CG solver delivers good performance, being mostly limited by the available memory bandwidth. For further work it would be interesting to add a preconditioner [21] for faster convergence of the CG and parallelize the computation over multiple PlayStation 3 machines.

Kaasgradientide meetodil lineaarvõrrandisüsteemide lahendamine PlayStation 3 mängukonsoolil

Magistritöö (30 EAP)

Lauri Tulmin

Kokkuvõte

Teadusarvutuste seisukohast on mitmetuumaliste protsessorite esilekerkimise ja järjest laieneva levikuga muutunud oluliseks selliste süsteemide põhjalik uurimine. Uute mitmetuumaliste protsessoritega arvutite võimalikult efektiivseks kasutamiseks on vaja programmid paralleliseerida mitmel tasemel, alustades andmeparalleelsusest ja vektorarvutustes kuni mitmelõimelisuseni. Sony PlayStation 3 mängukonsool sisaldab endas uuenduslikku Cell protsessorit, kus ühe tavaprotsessoriga on seotud mitu piiratud võimalustega, kuid suure arvutusvõimusega, eriotstarbelist vektorprotsessorit. Käesolevas magistritöös uurisin lineaarvõrrandisüsteemide lahendamist kaasgradientide meetodil kasutades PlayStation 3 mängukonsooli. Antud töö eesmärgiks oli realiseerida kaasgradientide meetod viisil, mis võimaldaks maksimaalselt ära kasutada riistvara arvutusvõimsust.

Tänu uuenduslikule arhitektuurile on Cell protsessoril lisaks suurele arvutusvõimsusele ka hästi ette ennustatav käitumine, mis lihtsustab programmide analüüsimist ja nendest kitsaskohtade leidmist. Erinevalt enamikest tavaprotsessoritest on Cell protsessoril töötavatel rakendustel võimalik jõuda väga lähedale teoreetiliselt maksimaalsele võimalikule arvutusjõudlusele. Antud riistvaral on kõrge jõudluse ja etteaimatavuse huvides ohverdatud lihtne programmeeritavus, seetõttu on efektiivsete programmide kirjutamiseks oluline süsteemi ülesehituse tundmaõppimine.

Käesolevas magistritöös antakse ülevaade PlayStation 3 mängukonsoolis kasutatavast riistvarast ja kaasgradientide meetodil lineaarvõrrandisüsteemide lahendamiseks. Lisaks kirjeldatakse topeltäpsusega ujukomaarve kasutavat kaasgradientide meetodi realisatsiooni PlayStation 3 riistvarale ja tuuakse välja valminud programmi jõudlusnäitajad.

Bibliographie

[1] Amdahl's law - wikipedia, the free encyclopedia. http ://en.wikipedia.org/wiki/Amdahl%27s_law. Last accessed in 2010.05.02.

[2] Matrix market. http ://math.nist.gov/MatrixMarket/. Last accessed in 2010.05.22.

[3] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference on - AFIPS '67 (Spring)*, page 483, Atlantic City, New Jersey, 1967.

[4] Richard Barrett. *Templates for the solution of linear systems : building blocks for iterative methods*. SIAM, Philadelphia, 1994.

[5] Jonathan Bartlett. Tech tips : SPU vector intrinsics at your fingertips. http ://www.ibm.com/developerworks/library/pa-tipspu1/index.html, May 2007. Last accessed in 2010.05.21.

[6] Thomas Chen, 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. Last accessed in 2010.05.02.

[7] International Business Machines Corporation. Data communication and synchronization for cell BE programmer's guide and API reference, version 3.1. https ://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/EDEC4547DFD111FF00257353006BC64A, October 2007. Last accessed in 2010.05.23.

[8] International Business Machines Corporation. SPE runtime management library, version 2.2. https ://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/1DFEF31B3211112587257242007883F3, October 2007. Last accessed in 2010.05.20.

[9] International Business Machines Corporation. C/C++ language extensions for cell broadband engine architecture, version 2.6. https ://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/30B3520C93F437AB87257060006FFE5E, August 2008. Last accessed in 2010.05.22.

[10] International Business Machines Corporation. IBM Full-System simulator user's guide, version 3.1. https ://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/B494BF3165274F67002573530070049B, May 2009. Last accessed in 2010.05.20.

[11] International Business Machines Corporation, Sony Computer Entertainment Incorporated, and Toshiba Corporation. Synergistic processor unit instruction set architecture. https ://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/76CA6C7304210F3987257060006F2C44, 2007. Last accessed in 2010.05.22.

[12] James W. Demmel, Michael T. Heath, and Henk A. van der Vorst. Parallel numerical linear algebra. *Acta Numerica*, 2 :111, 2008.

[13] Brad Frey. PowerPC architecture book, version 2.02. http ://www.ibm.com/developerworks/systems/library/es-archguide-v2.html, February 2005. Last accessed in 2010.05.08.

[14] M. Gschwind, H.P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki. Synergistic processing in cell's multicore architecture. *IEEE Micro*, 26(2) :10–24, 2006.

[15] Mark D. Hill and Michael R. Marty. Amdahl's law in the multicore era. *Computer*, 41(7) :33–38, 2008.

[16] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the cell multiprocessor. *IBM Journal of Research and Development*, 49(4) :589–604, 2005.

[17] David Krolak. Unleashing the cell broadband engine processor : The element interconnect bus. https ://www.ibm.com/developerworks/library/pa-fpfeib/, November 2005. Last accessed in 2010.05.08.

[18] Toomas Laasik. *A Conjugate Gradient Solver library for the PlayStation 3*. Master's thesis, University of Tartu, Tartu, 2010.

[19] Geoffrey Levand. Advanced cell programming. http ://www.kernel.org/pub/linux/kernel/people/geoff/cell/ps3-linux-docs/ps3-linux-docs-08.06.09/CellProgrammingTutorial/AdvancedCellProgramming.html. Last accessed in 2010.05.20.

[20] Gordone E. Moore. Cramming more components onto integrated circuits. *Electronics Magazine*, April 1965.

[21] J. R Shewchuk. *An introduction to the conjugate gradient method without the agonizing pain.* 1994.

[22] V. Srinivasan, D. Brooks, M. Gschwind, P. Bose, V. Zyuban, P.N. Strenski, and P.G. Emma. Optimizing pipelines for power and performance. In *35th Annual IEEE/ACM International Symposium on Microarchitecture, 2002. (MICRO-35). Proceedings.*, pages 333–344, Istanbul, Turkey, 2002.

[23] Vaidyanathan Srinivasan, Anand K. Santhanam, and Madhavan Srinivasan. Cell broadband engine processor DMA engines, part 1 : The little engines that move data. http ://www.ibm.com/developerworks/library/pa-celldmas/, December 2005. Last accessed in 2010.05.02.

[24] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. Optimization of sparse matrix-vector multiplication on emerging

multicore platforms. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing - SC '07*, page 1, Reno, Nevada, 2007.

[25] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall. *ACM SIGARCH Computer Architecture News*, 23(1) :20–24, 1995.