U N I V E R S I T Y O F T A R T U

FACULTY OF MATHEMATICS AND COMPUTER SCIENCE

Institute of Computer Science

Computer Science speciality

Toomas Laasik

A Conjugate Gradient Solver Library for the Playstation 3

Master Thesis (30 EAP)

Supervisors: Ulrich Norbisrath, PhD

Eero Vainikko, PhD

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

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

Allowed to defence

Professor: : ..... "....." May 2010

TARTU 2010

Contents

Acknowledgements

I would like to thank people who helped me in the process of writing this thesis. Writing such a long text has been a challenge for me.

At first I would thank my supervisors Ulrich Norbisrath and Eero Vainikko for giving feedback and suggestions on how to improve the work. I would also thank Oleg Batrašev who provided material and helped with the implementations. He also reviewed the contents of the thesis thoroughly and pointed out places hard to understand. Lastly I would thank Aivi Kaljuvee for helping me to clean up the text from grammatic errors.

Chapter 1

Introduction

This master thesis focuses on my work with the multi-core processor architecture named Cell Broadband Engine (CellBE) used in Playstation 3 (PS3) gaming consoles. I mainly worked on solving large systems of linear equations using the conjugate gradient method. These systems usually arise from physics or other areas where differential equations need to be solved. As a result I present an optimized solver which can be used as a general purpose library running on the PS3. It has already been used in a real time wave simulator and visualizer application written together with Oleg Batrašev for the Teeviit 2009 education fair. As a further result, I present techniques, the library API and show how to use them for developing scientific applications for the CellBE architecture and also how to extend them in the future.

In the ever-growing hunger for more computing resources at cheaper prices and lower power consumption, the architecture and price of the PS3 makes it very attractive for scientific applications. It has a good peak FLOPS-per-watt and FLOPS-per-dollar ratio [20]. From the hardware design perspective, it stands somewhere between general purpose processors (CPUs) and graphics processors (GPUs). Not all problems can be implemented near that peak. These problems will be discussed in more detail in this thesis. Also solving systems of linear equations is effected by these problems mainly due to the inherent memory access patterns.

While being a general purpose computing platform, it turns out to be more difficult to develop optimized code for this distributed memory multi core platform than initially expected. The problem areas suitable are either limited or require extensive work to use the full potential of the hardware. During my work it happened multiple times that I had to abandon particular implementation strategies because of the too steeply growing complexity that made the code error prone and very hard to manage. In the end I learned that usually simple, elegant and performance-wise not on the very edge algorithms are practical enough for solving anything non-trivial. In this work you will find a few examples of what did not work, but mostly I focus on the solutions that worked, are reusable and portable to other CellBE applications.

The first hands on experience I got with the PS3 platform was during a research lab on 2008 spring at University of Tartu. It was organized by Ulrich Norbisrath and Eero Vainikko who are also supervisors of my thesis. After the course I started working at the Distributed Systems Group [4] at University of Tartu. The main topic was learning the platform by trying to implement an efficient conjugate gradient solver. The same topic, but different approach, was tried out by Lauri Tulmin, who also participated in that lab and started working at Distributed Systems Group afterwards. Until writing this thesis we have had similar progress timeline, but we both have worked quite independently.

The major milestone in my work was developing a real time surface wave simulation and visualizer together with Oleg Batrashev [12]. It was used at Teeviit 2009 fair [5] to introduce possibilities of this gaming system in scientific computing to potential future students at our faculty. It consisted of a server on a PS3 running the simulation and a regular computer doing hardware accelerated 3D visualization. These two communicated with each other over UDP. Inside the visualizer application you can control the whole simulation process primarily by resetting it with different parameters and creating waves on the surface interactively by using the mouse.

The main result of my work is an efficient code base for PS3 written in C [3] that can be integrated and used in other projects that require conjugate gradient solver. The code base has few dependencies and a simple API. To use it, it is only necessary to include a header file and link to a static library. The library does all the resource management and optimizations needed for CellBE platform so that you can write regular portable C code. In this thesis I will explain some of the approaches I tried and the one that finally got fully implemented. The API of the library is also covered to show how to use it properly.

Besides the usable conjugate gradient solver library, a simple independent resource management and program flow coordinator API emerged. It effectively abstracts away complexities of the underlying APIs without sacrificing performance. It can be done by limiting the options how to parallelize the problem to only one SIMD-like (single instruction, multiple data) approach. This is most suitable for problems that involve large vectors. In general the API is not PS3 specific and not even limited to one machine. If there existed an implementation of that API for a cluster of PS3s, then scaling your application from one PS3 to a cluster would require no source code changes, but just linking to a cluster-aware library. The effect on overall performance can then be tuned by setting different data distribution flags to your data items. The data transfer over network and caching on various levels of this distributed memory system is handled implicitly. At this time the API design is fixed, but only one-PS3 version of it is implemented. A cluster-aware version of it may be future work.

The thesis is organized as follows. At first in chapter 2, I will explain the motivation behind the work by introducing the hardware platform. In chapter 3, I will present some

of the works that influenced my work. In chapter 4, I will cover the conjugate gradient method and my approaches to optimizing it. As the matrix-vector multiplication being the most difficult part of the work, a whole chapter 5 will explain the details of the calculation. In chapter 6, I will present the C APIs I developed and also briefly describe the real time wave simulator written for fair Teeviit 2009. Chapter 7 concludes my work.

Chapter 2

Motivation

In this chapter I will introduce the CellBE architecture and one of its materializations - the Playstation 3 gaming console. At first, I will write a bit about the history of the console and what makes it attractive. Next, I will make a small introduction to the hardware and what a programmer needs to know about it. In general, the features that make CellBE so powerful, also make writing the code for it a lot more difficult than for traditional x86 or PowerPC architectures. Still, for writing non-optimized code, you can entirely overlook the CellBE architecture and write code for regular PowerPC core.

2.1 History

The Playstation 3 or officially "PS3" for short by Sony is currently one of the most powerful video game consoles on the market. The first version was released in November 2006 [9]. From there on, the console has been sold with different hardware configurations and colors all over the world. The main distinctive feature when compared to other consoles on the market and past Playstations is that Sony officially allows people to install other operating systems on it (feature called OtherOS). This has enabled cheap access to computing power for many who previously had to use supercomputers or clusters in their work.

In September 2009 a new revised Playstation 3 Slim hit the market. Together with re-branding and hardware upgrades, the OtherOS functionality was dropped. Sony explained it as focusing on entertainment content and that it would be overly costly to keep OtherOS functionality [8]. The OtherOS functionality was eventually dropped in every PS3 starting from system software update to version 3.21 which was released on 1st of April 2009 [7].

The CellBE architecture also appears in other computing systems. The IBM Roadrunner was the world's fastest supercomputer until November 2009 consisting of AMD Opteron and PowerXCell 8i processors and achieving over 1 PetaFLOPS [11]. There

also exist various add-on cards for personal computers for specific and general purposes like video encoding.

2.2 Hardware specification

For writing efficient programs you either need to know the hardware or alternatively use tools or frameworks that know the hardware. Since the PS3 specifications are easy to locate in the Internet, I will not just list them, but try to comment on the most important parts based on my experience.

3.2 GHz CellBE CPU. Having 1 PowerPC CISC core (PPE) and 7 specialized RISC cores (SPE) makes this a very unique CPU. Features like distributed memory in the chip and annoyances like data alignment in memory are discussed in more detail later on. To be absolutely correct, the CPU has 8 SPE units from which one is disabled in the factory and another one is reserved for the system. Therefore only 6 SPEs are available under OtherOS.

256Mb XDR DRAM. This amount is usually not enough for solving problems of large data, but having 25.6 Gb/s of memory bandwidth will usually not leave the CPU starving. Very roughly, the total amount of memory available limits the problem size for the solver I have implemented to about maximum 5 million matrix elements.

1Gb Ethernet. While being more than enough for home computing, it is quite slow and with high latency for using in GRID environments. Theoretically, when on average ~704 or more instructions are performed using one double precision float before moving it over the network, the network bandwidth will not become the bottleneck. This is because 1 Gbit/s = 0.125 GBytes/s and with peak 11 GFlop/s, we can do approximately $11 * \text{sizeof(double)}/0.125 = 704$ operations per float before we can move it.

550 MHz NVIDIA/SCEI RSX 'Reality Synthesizer' 256Mb GPU. Unfortunately access to graphics card and its memory is severely restricted by Sony. This can be used only as a framebuffer for Linux without no 2D or 3D hardware acceleration.

40Gb 2.5" SATA HDD. The console has a HDD from witch you can use 10 Gb less than the total HDD size. The remaining 10Gb is used by PS3 itself for games, save games, multimedia and such.

Other notable features are a Blue-ray drive, HDMA output, Bluetooth, USB, WiFi, but in my work these are not important.

2.3 Multicore PPU/SPU model

The CellBE architecture has a multicore processor with two kinds of cores [10]. These are called "Power Processor Element" (PPE) and "Synergistic Processing Element" (SPE). The PS3 has one PPE core with two hardware threads having duplicated CPU registries, but shared computation units. 6 SPE vector processor cores are available for programmer under Linux environment. These cores share the CPU clock and have means of communication, but they run on their own. Some sources call these cores PPU and SPU. The "U" stands for "Unit". Since I am more accustomed to the latter naming, I will use that from now on.

The PPU is optimized for running conventional programs. In fact, it is PowerPC compliant and therefore is capable of running any Linux PowerPC kernel and any program compiled for it. The programs do not need to know the underlying hardware, but usually they want, because they want to use specific PS3 hardware. When distributing code over PPU and SPUs, the PPU is usually used for control or task oriented parts, while SPUs handle SIMD operations. But being free of hard alignment restrictions and direct access to main memory sometimes makes PPU much more feasible over SPU for certain types of calculations.

Each SPU is an independent RISC SIMD vector processor having its own 256kB RAM called LS (Local Storage), program, stack, registers, MFC (Memory Flow Controller) and all it needs to run on its own. LS is initially empty and does not have any sort of operating system running on it. A SPU program is loaded into the LS by the PPU. During the execution of the SPU program, it has access to its LS and to the main memory by using explicit DMA calls. The SPU has also methods for communicating with the PPU and other SPUs over a high speed EIB (Element Interconnect Bus). The logical connectivity of the PPU and SPUs is drawn on the figure 2.1. Along with them various high level interfaces are mentioned.

SPU is powerful in computations, but not efficient in branching. Scalar calculations require all data to be aligned to work efficiently. It works only on 128 bit aligned chunks of data. There are a few special purpose (like stack, program pointer) and 128 general purpose registers each 128 bit, which give a lot room for the compiler to optimize out unnessecary moving of data. CPU operations are designed so that any register of these 128 can be used as source or destination with a few exceptions.

The SPU has two instruction pipelines. Which pipeline the instruction goes into is determined on whether it treats the register as full 128 bit quadword or as a 128 bit vector of some data type (4 floats for example). Instructions in separate pipelines can be executed simultaneously. Therefore two continuous operations per CPU clock is achievable and code can be hand-optimized for the case when the compiler fails. In general, instructions that work on full quadwords go to odd pipeline (load and store,

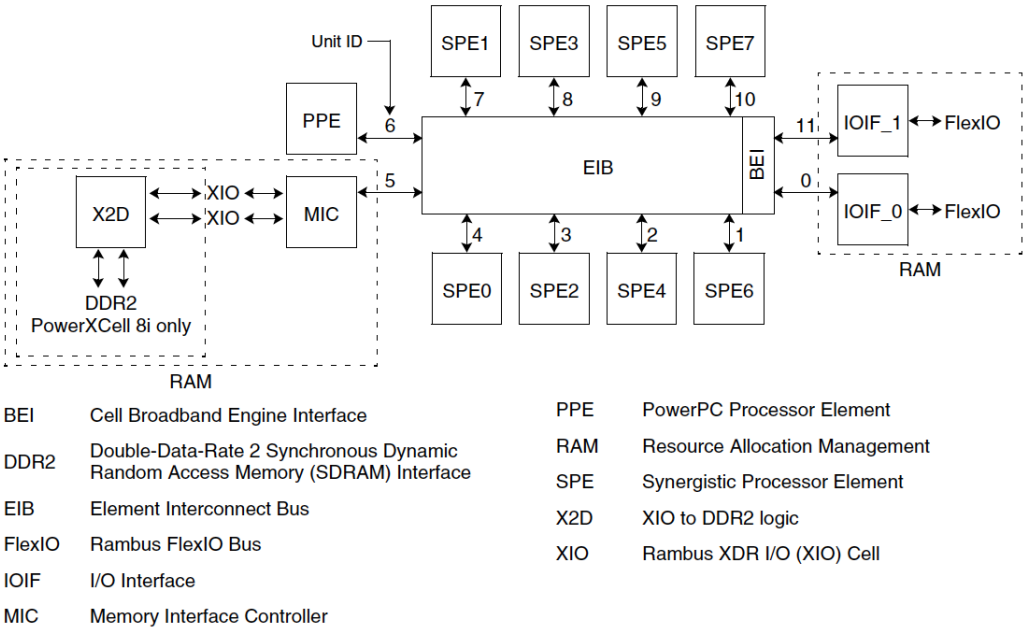

Figure 2.1: Connectivity diagram inside the CPU [10]

shift quadword, ...) and those working with vectors of 4 byte or less elements in it, go to even pipeline (4 floats, 16 chars, ...). Without the MFC (Memory Flow Controller) on the picture, the even and odd pipeline among with LS and SPU registries is laid out on the figure 2.2.

PPU and SPUs can be viewed as independent computers with means of communication. This model allows various approaches on how to coordinate work, but the most common is to have a star topology where the PPU is in the center and each SPU is in the following cycle:

1. wait for a command from PPU
2. execute it
3. send the result back to PPU and goto 1

In the search for possible bottlenecks, it is good to know how bus bandwidth and latencies compare to the computational resources available. Each SPU is capable of performing one instruction in both pipelines per one CPU clock cycle. For PS3 that is maximum 6.4 billion operations on 128bit vectors in SPU registry file when even and odd pipeline can be dual issued. An instruction usually takes more than one cycle to execute, but we can ignore that when assuming all instructions are properly pipelined. Moving a vector between LS and registry file usually has no latency, that is, it occurs during one CPU cycle as any other assembly instruction. Sometimes ongoing DMA transfers degrade load/store performance, but the effect is insignificant. The only exception for how long it takes to execute an instruction is when doing double precision

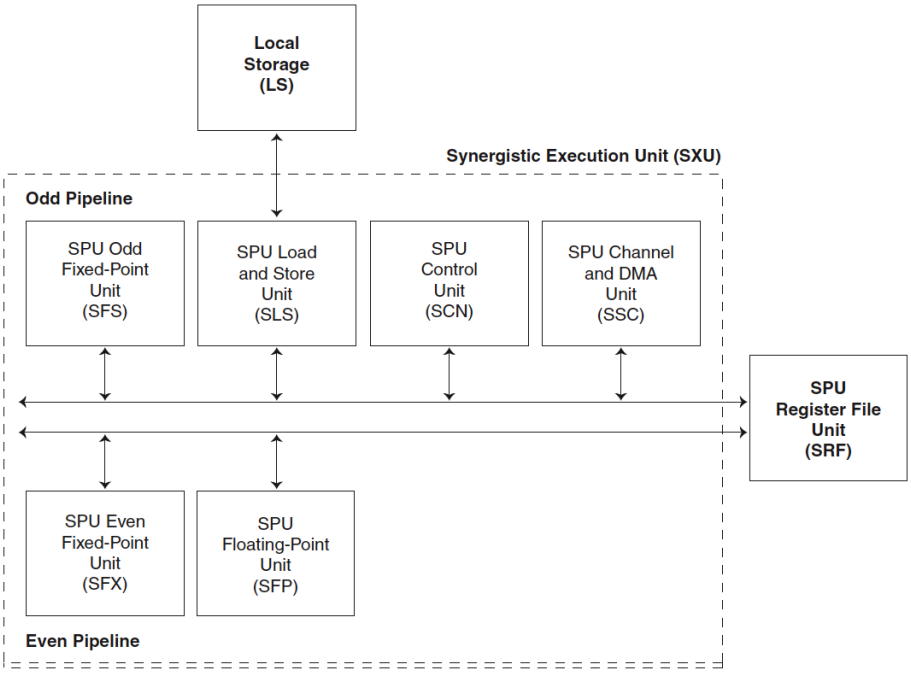

Figure 2.2: SPU local resources and pipeline [10]

arithmetic. It stalls instruction pipeline for 6 CPU cycles and therefore each double precision operation takes 7 times more time than any other. Since all instructions operate on 128 vectors, one instruction can add 16 8bit integers together at the same speed as for example multiplying 4 floats together. Given the performance in GFlops for single precision floating point FMA (fused multiply add) yields to 6*8*3.2=25.6 GFlops and for double precision 6*4*3.2/7~=11 GFlops.

The EIB (Element Interconnect Bus) connecting all SPUs, PPU, I/O and memory controller together consists of 4 rings that can hold up multiple transfers simultaneously. It is designed to provide enough bandwidth that will not leave endpoints starving for data. Each connected device can access the rings at 25.6 Gbytes/s. However the latency is somewhat traded for bandwidth. The main problem is related to mailboxes that are used to transfer 32bit integers between devices. The round trip time PPU->SPU->PPU is 7..8 microseconds which is slow. Compared to the almost achievable maximum main memory bandwidth that is 25.6 Gbytes/s, one can transfer 192 kbytes of data during the round trip time. One can deal with it is by making as much work as possible between using mailboxes and by using barriers for synchronization.

In this chapter I have introduced Playstation 3 as a computing platform. I have covered some of the hardware features that make it exciting and are relevant in my work. PS3 has a lot of potential while being very challenging to write code for. Exploiting the novel hardware features in PS3 is what motivates to work on conjugate gradient and other problems from HPC field.

Chapter 3

Related works

CellBE is a relatively new platform in the HPC (High Performance Computing) field. It is probably best known for being used in currently the world's second most fastest supercomputer IBM Roadrunner and also for being used in the Playstation 3 gaming console. Most of the related work is targeted to these two. In this chapter, I will give a small overview of projects that I have encountered and that have inspired me while working on the conjugate gradient solver.

There are multiple challenges when implementing solvers of linear equations on the PS3 platform. The simplest case is when direct methods like Gaussian elimination with dense matrixes and single precision are used. Jakub Kurzak and Jack Dongarra have implemented a mixed precision solver using Gaussian elimination in single double precision combined with iterative refinement to improve the result to full double precision accuracy [19]. The computationally complex parts LU factorization and forward and backward substitution are done in single precision having complexity $O(n^3)$ and double precision is used where complexity is $O(n^2)$ or less. The achieved performance was an impressive 98.05 GFlops when it was run on 3.2 GHz CellBE processor. Since it was not a PS3 they could use all the 8 SPUs and had more memory. Later on they improved their mixed precision solver by adding Cholesky factorization and improved parallelization scheme [18]. Their work is available to the public [**?**].

D. DuBois and others have compared a few supercomputing platforms [15] by evaluating how these perform when solving double precision sparse problems with non-preconditioned conjugate gradient solver. One part of it focuses on IBM Roadrunner that uses the same CellBE architecture as PS3, but is combined with AMD Opteron processors to form a large hybrid (two CPU types with mixed endianness) supercomputer. Their work is comparable to my work since they evaluate only one Opteron CellBE pair inside a TriBlade [17]. They try out many different approaches before settling down to one. The main differences compared to my work is that they use ELLPACK-ITPACK matrix storage format and they use purpose-designed software based cache on SPUs for matrix vector multiplication. The overall performance they

achieved for conjugate gradient was a bit over 1.7 GFlops. The code is not public.

In "A Rough Guide to Scientific Computing On the Playstation 3" [14] they implemented a conjugate gradient solver for tri-diagonal case with Jacobi preconditioner. Due to the nature of the problem, the parallelization scheme is static and easily vectorizable. They achieved a performance of 6.6 GFlops, but it is not stated whether it was single or double precision. To my knowledge the code has not been made public.

The core of conjugate gradient method is sparse matrix vector multiplication. Various matrix structures are evaluated by Williams [23]. They try out various optimization strategies for many multicore platforms including PS3. Their findings are that matrix vector multiplication is almost always SPU bounded on PS3, because of the relatively low double precision performance. On the other hand, explicit DMA will not leave SPUs in data starvation and therefore memory efficiency is very good. The mean performance they achieved over various matrix types was 3 GFlops. Although they have plans to integrate the results into OSKI (Optimized Sparse Kernel Interface) package, I could not find any CellBE related code in their publicly available code base [2].

Lauri Tulmin also worked on conjugate gradient solver [22] at Distributed Systems Group at Tartu University, but our approaches were different. The main difference is in matrix-vector multiplication method where I used tiny grained block format for matrix, but my colleague tried out a more conventional approach using a modified version of row major format. In the future it may be desirable to merge the best of these two approaches together to form a library.

In this small chapter I have provided an overview of works that I studied when formulating my approach. These works include dense solvers, sparse solvers and just sparse matrix-vector multiply kernels. All these tend to deal with either somewhat special cases or are proof of concept works without publicly available code base. In my work I tried to build a more robust and easier to use solver.

Chapter 4

Conjugate gradient method

In this chapter I will describe the conjugate gradient method from programmers perspective and show how I have parallelized it. I will not describe the mathematical background of the algorithm. To learn more, refer to this material [21].

4.1 What is the conjugate gradient method?

The conjugate gradient method is an iterative method for solving a large system of linear equations in the form

$$Ax = b$$

where given matrix $A \in R^{n,n}$ and right hand side vector $b \in R^n$, we can find $x \in R^n$ that is a good enough approximation of the real solution x_{real}. For the method to work, the matrix must be symmetric ($A^T = A$) and positive definite ($x^T Ax > 0$ for all non-zero vectors $x \in R^n$). Otherwise it may not converge, but if it still does, we can find a solution.

Algorithm 4.1 gives an iterative conjugate gradient method in Python (written by O. Batrašev). It takes as an input the matrix A and the right hand side vector b. It returns the result vector x when a precise enough solution in regards to predefined $ERROR$ margin has been found.

4.2 Parallelizing conjugate gradient for PS3

For my application I have reordered the operations inside the iteration which helped me to write better parallel code. It is still the same algorithm, but what I have done is that the vectors x and r are now computed in the beginning of the loop not at the end. Also the vector copy operation $p = r$ is removed from the loop. The pseudocode for the algorithm is on algorithm 4.3.

Algorithm 4.1 Iterative CG in Python

```
def cg(A,b):
        x = numpy.zeros(A.shape[1])
        r = b - dot(A,x)
        k = 0
        while dot(r,r)>ERROR:
                rho = dot(r,r)
                if k==0:
                        p = r
                else:
                        beta = rho/rho_
                        p = r + beta*p
                q = dot(A,p)
                alpha = rho/dot(p,q)
                x = x + alpha*p
                r = r - alpha*q
                rho_ = rho
        return x
```

To make sure that it is still the same algorithm, we need to observe how x , r and p are computed in the first and in the last iteration. In the first iteration we have $\alpha = 0$ which ensures that x and r stay the same as they were in the original loop. Making $p = r$ is trickier, but we can accomplish it by setting $rho\_ = \infty$. Any very large number is enough for practical purposes. The last iteration is also equivalent because we have placed the $rho < error$ check right after computing the norm of r.

Reordering is needed since we want to exploit the data level parallelism by dividing each vector into parts and logically binding each part exactly to one SPU. That SPU will be responsible for writing into that part and nowhere else. It can read any other vector part and it is up to the programmer to keep synchronization points so that the next operation will not start reading data from other SPU's part when it has not been filled yet.

This partitioning scheme and writing policy makes all vector operations needed for this algorithm parallel without the need for inter-SPU communication or write locking. There exists 3 synchronization points from which one can be almost eliminated in theory. Two of these are where α and β are evaluated, and the third is just right before the sparsedot method. For the sparsedot we need to access vector p elements that are out of range assigned to the SPU doing the calculation. For assuring that all of the vector p is filled, we wait for the last operation to complete before executing sparsedot. It is illustrated by the algorithm 4.3.

The vector part size is chosen to be a multiple of 128 bytes which is a 16 double precision vector element. When the vector size is not initially a multiple of elements

Algorithm 4.2 C-like pseudocode for conjugate gradient

```
vector solve(vector A, vector b, double error)
        vector r, x, p, q
        double rho, rho_, alpha, beta
        r = b
        x = p = q = 0vec
        rho = alpha = beta = 0
        rho_ = +inf

        do {
                x += alpha*p
                r -= alpha*q
                rho = dot(r,r)

                if(rho < error) break;

                beta = rho/rho_

                p = beta*p + r
                q = sparsedot(A,p)

                alpha = rho/dot(p,q)
                rho_ = rho

        } while (rho > error)
        return x
```

Algorithm 4.3 Parallel execution of conjugate gradient method

```
    PARALLELISM        PPU                        each SPU
do {
    +---+---+---+      do part 0 -> SPU
    |   |   |   |                                 my_x += alpha * my_p
    |   |   |   |                                 my_r -= alpha * my_q
    |   |   |   |                                 my_dot = dot(my_r, my_r)
    +---+---+---+      rho = <gather>
    | cg done?  |      if(rho < error) break;
    | beta=...  |      beta = rho / rho_
    +---+---+---+      do part 1 -> SPU
    |   |   |   |                                 my_p = beta * my_p + my_r
    |   |   |   |                                 --- sync ---
    |   |   |   |                                 my_q = sparsedot(A,p)
    |   |   |   |                                 my_dot = dot(my_p, my_q)
    +---+---+---+      tmp = <gather>
    | alpha=... |      alpha = rho / tmp
    |           |      rho_ = rho
} while(...);
```

	SPU1	SPU2	SPU3
r	0..31	32..63	64..80
x	0..31	32..63	64..80
p	0..31	32..63	64..80
q	0..31	32..63	64..80

Figure 4.1: Vector block distribution when vector size here is 80 and block size is 16

we want, we just pad it from the end with zeros. In memory the vector offset is chosen to be aligned to 128 bytes. This is needed for the cache-line to be efficient, because reading less than 128 bytes will still make a 128 byte fetch to the main memory. In practice it is usually better to transfer larger chunks of data per request to achieve better bandwidth. Therefore, I use vector chunk sizes that are larger than 16 elements. In my implementation the size is runtime-configurable and can be as large as 2048 elements which is the maximum size DMA can handle in one request.

After the vector has been divided into blocks, those blocks are distributed over available SPUs for block-cyclic distribution. When the block count is not a multiple of an SPU count, then some SPU-s will get 1 block less. To efficiently overlap computation and vector blocks DMA transfers, it is better to have as many blocks as possible per SPU. Since we also want to have vector block size as big as possible, the performance will not be good for small vectors. In the current implementation, the block size is manually specified, but it might be wise to adjust it according to the vector size. The figure 4.1 illustrates how the vector elements would be distributed if the chunk size was 16 and there were 3 SPUs.

When the main parallelization scheme has been sorted out, the last and the hardest part to optimize is the matrix vector multiplication operation (`sparsedot`). Since we know how the vector can be partitioned, we only need to decide how to store the matrix in memory so that it can be used efficiently on the CellBE architecture. As expected, it was the major challenge in my work. The chapter 5 is dedicated to describing how the matrix vector multiplication works. Before that I go through the rest of the steps in conjugate gradient.

4.3 Optimizing each CG step

For linear algebra there exists the BLAS library for both PPU and SPU [1]. At first I tried to use these, but soon found that they were not flexible enough. The reason why I could not use PPU BLAS library was because I needed to implement matrix vector multiplication on the SPUs myself. Mixing BLAS managed SPU threads and sparsedot threads was difficult without much benefit. The SPU BLAS library could have been suitable for a few methods, but I chose not to use it. This is because there

were operations that were not directly implemented in SPU BLAS and I also wanted to use combined methods. For the few BLAS methods I reimplemented, I got not significantly, but still 1-2% better performance and I set vector size to be a multiple of 16 while BLAS required 32.

The main profiling tool I used on SPU code was `spu_timing` from the IBM Cell SDK [6]. When you compile you program, you can also export the generated assembler for it. The `spu_timing` tool annotates that assembler file with static timing analyzes. For each operand you will know if it was dual issued, how many CPU cycles it took to execute and how many CPU cycles was it stalled because of registry dependencies. However, there are numerous parameters that `spu_timing` does not tell. The most important of those is the effectiveness of branch hinting. No hints or wrong hints may cause CPU stall for over 10 CPU cycles. Another less notable effect to actual CPU cycles used and those predicted by `spu_timing` is due to the LS (Local Store) accesses by the MFC (Memory Flow Controller). Most commonly an ongoing DMA transfer for example.

The main goal when using `spu_timing` tool was to ensure that the compiler does not generate any unnessecary assembly instructions. Using only 128bit aligned vector data types and SPU intrinsics was usually enough to achieve this. Sometimes it was nessecary to explicitly use X form of SPU load (`si_lqx`) operation when address could be computed by hand faster than compiler generated code did. Another big goal was to minimize dependency stalls by logically decoupling data, usually by unrolling loops. Then the compiler could reorder the operands and minimize stalls. For effective branch hinting, I simply kept inner loops as long as possible and let the compiler insert all the hints. It is interesting to note that using `__builtin_expect` in the for-loop condition gave better results only sometimes. In general, GCC did pretty good at reordering dependencies and branch hinting at optimization level `-O2` and `-O3`. Using higher levels never improved the performance of hand tuned code, sometimes it degraded over 10%. Occasionally it was nessecary to force variables into CPU registries, because it seemed that GCC could not handle large amount (total 128) of SPU general purpose registries efficiently. Combining the ability to force variables into CPU registries and write SPU intrinsics that are essentially SPU assembler instructions wrapped into convenient C functions, it was not nessecary to write assembler directly. There would have been no or tiny speed increase, but an even greater burden on the programmer.

Besides the inner loop optimizations in SPU code, synchronization and main memory access need to be effective. Here I mainly relied on DaCS (Data Communication and Synchronization Library) provided by IBM for CellBE [?]. Using it simplifies programming for PPU and SPU, because one does not have to deal with low level resources like MFC or SPU thread management. It provides you a small set of functions which fall into 3 categories: ecosystem (eg. loading SPU program into LS, resource allocation

and releasing), synchronization directives for master-worker model (sending messages, group barriers) and DMA transfers between main memory and SPU local store. On top of DaCS I built a few non CG specific lightweight functions to simplify double buffering, distributing workloads and gathering return values. More about it is written in the section 6.2.

For PPU code and DMA optimizations I referred mainly to CellBE Programming Handbook [10]. While DaCS library is designed to handle many tasks efficiently, I still had to put some effort into making it happen. One of the most important issues is the usage of huge pages which can be enabled in Linux kernel. It will make use of hardware provided 16Mb memory pages instead of the default 4kB pages. Each main memory access has to translate the process specific virtual address to physical memory address. For efficiency in chip TLB cache is used for that. Since this is a relatively scarce resource, accesses spanning a lot of pages will generate TLB cache misses that are very expensive to handle. These are processed by the Linux kernel and usually take over 1000 CPU cycles to execute. Another significant factor that contributes to the speed is how SPU mailboxes are checked for incoming data. For some reason that I have not investigated yet, busy loop polling for messages is multiple times faster than using blocking calls.

For each CG step I calculated how it would perform when theoretical CellBE main memory bandwidth was saturated at 25.6 Gb/s. Then I measured how fast SPU cores could do the calculation without any main memory access and finally I measured how fast the step performs when combining the calculations with all the nessecary data fetching-storing from the main memory. Each operation and bytes needed per FLOP and the theorical and measured performances are presented in the table 4.1.

SPU measurements were made with vector of 2048 elements (16kB). The first half of the table contains single steps and the second half contains combinations of steps to save memory bandwidth. For example `axpy` operation needs to transfer 2 doubles from main memory and put one 1 double back to evaluate one element in the result vector. That is transferring 8*3 bytes and doing 2 FLOPs on them, hence 12 bytes/FLOP. Since the memory bandwidth is 25.6 Gbytes/s we can not do better than 25600/12=2133 MFlops. How fast all 6 SPUs combined together can execute the `axpy` operation having all the data locally available in LS was measured to be 7140 MFlops, meaning that this operation is definitely main memory bandwidth bound. The last column in the table shows how well the main memory bandwidth could be utilized when all 6 SPUs run in parallel and work on main memory. In the case of `axpy`, I achieved 1691 FLOPs which reaches about 79% of peak memory bandwidth.

The memory limited bandwidth for `sparsedot` is calculated based on the assumption that matrix element count per vector element is infinity. Therefore we do not need to include vector elements in bytes/flop ratio. `Sparsedot5` and `comb_cgb5` use the

C mnemonic	operation	bytes/flop	memory limited	measured 6 SPU	measured actual
axpy	$y \leftarrow \alpha x + y$	$\frac{8*3}{2} = 12$	2133 MFlops	7140 MFlops	1691 MFlops
xpay	$y \leftarrow x + \alpha y$	$\frac{8*3}{2} = 12$	2133 MFlops	6882 MFlops	1691 MFlops
dot	$\leftarrow x^T y$	$\sim \frac{8*2}{2} = 8$	3200 MFlops	8016 MFlops	3084 MFlops
dot_self	$\leftarrow x^T x$	$\sim \frac{8}{2} = 4$	6400 MFlops	8749 MFlops	not measured
sparsedot	$y \leftarrow Ax$	$\frac{16}{4} = 4$	6400 MFlops	see table 5.1	3431 MFlops
sparsedot5	$y \leftarrow Ax$	$\frac{5*16+2*8}{5*4} = 4.8$	5333 MFlops	see table 5.1	2913 MFlops
comb_cga	$x \leftarrow x + \alpha p$ $r \leftarrow r - \alpha p$ $rho \leftarrow r^T r$	$\sim \frac{8*3+8*3+0}{2+2+2} = 8$	3200 MFlops	7956 MFlops	2578 MFlops
comb_cgb5	$q \leftarrow Ap$ $dot \leftarrow q^T p$	$\sim \frac{5*16+2*8+0}{5*4+4} = 4$	6400 MFlops	not implemented	not implemented

Table 4.1: Expected and measured performance of CG steps

model where there are approximately 5 matrix elements per 1 result vector element. In the byte/FLOP ratio value matrix macroblocks are used - per one 16-byte macroblock that contains 2 original matrix elements, we do 4 FLOPs. The macroblocks are essential in my design and are covered in the next chapter. The `sparsedot5` performs slower than `sparsedot` since the latter spends most of the time in the inner loop fetching matrix elements and using them to update the result vector in a double buffered way where calculations are overlapped with DMA transfers. In the case of sparsedot5 we do the same, but now we spend less time in the double buffered loop. That is, we need to fetch the first matrix block before we can enter that loop.

How the SPU cores perform on sparsedot does not depend on how many matrix elements there are per vector element. Instead the matrix structure is important. How it affects the performance is covered in the section 5.2 and comprehensively shown in the table 5.1. The actual measured performance taking into account all the main memory accesses is evaluated on the most suitable matrix. The details about such matrixes are covered in the following chapter, but it is worth mentioning here that the absolute worst case matrix that still has elements near its main diagonal will achieve approximately 50% of this measured number.

Currently no measurements for `comb_cgb5` exist, since it is not implemented yet. Also `dot_self` is currently only available within `comb_cga` method. In general, all operations besides `sparsedot` are memory limited. Referring to the measured maximum combined 6 SPU performance of 4476 GFlops for `sparsedot` in table 5.1, the maximum of approximately only 75% of memory bandwidth can be used. In the reality however, the achieved performance is less than that (about 2913/5333=55%), suggesting that the DMA transfers are not performing as well as they should. This needs further investigation.

When these steps are combined to form conjugate gradient, the numbers for all measurements except sparsedot should be achievable for any input large enough. The sparsedot depends on the input matrix structure, but since it uses tiny grained ap-

proach, the effects of "bad" matrixes should be minimized as long as they are diagonal type.

The overall algorithm speed depends on the input data, but for rough comparison I achieved 2542 MFlops for the overall conjugate gradient method with diagonal matrix having 5 elements per row close to the main diagonal. The vector size was 131072 and matrix had 655360 elements before packing. The performance measurements taken with different matrix and vector block sizes can be found in the appendix. How well the matrix got packed, how many elements were not evaluated on SPU, how many groups were formed, how many groups were able use fast float to double conversion and how many packed even and odd elements were created can be seen in the following matrix structure dump.

```
-------------------------- MATRIX fffffad2a0 --------------------------
Format:   PACKED    Total size: 23.07M    Useful data size: 12.59M (54.55%)
Even elements:     0 327674  Compress ratio: 50.00%  Non-parallelizable: 1530 (0.47%)
Odd elements: 655348 458740  Compress ratio: 58.33%  Non-parallelizable: 3060 (0.67%)
Group count: 514 (even+odd)   Using fast f2d: 512 (100.00%)
Checksum: 131066   OK
------------------------- /MATRIX fffffad2a0 -------------------------
```

In this chapter I briefly introduced the conjugate gradient method for solving linear systems of equations. Then I showed my approach how to parallelize it and how it suits for PS3 or in theory, any other distributed memory multicore platform. After that I described primary methods I use to conjugate gradient operations together with the measurements. Next I will talk about the most challenging operation, that is matrix vector multiplication.

Chapter 5

Sparse matrix vector multiplication

In this chapter I will cover the challenges I was faced with when implementing matrix vector multiplication on CellBE platform. Then I will go through the multiplication core design principles and implementations with measured performances for each one of them. Some cores make use of custom fast float to double converter which is discussed next. Finally I will go through the process of restructuring the matrix to a format suitable for the cores.

5.1 Challenges

CellBE handles single precision dense linear algebra very well. The BLAS SGEMM routine in the Cell SDK provided by IBM [6] computing single precision matrix-matrix multiplication achieves close to peak computational performance which is $8*3.2 = 25.6$ GFlops per SPU when using pipelined FMA (*fused multiply add* which does 8 single precision floating point operations per CPU clock cycle) running at 3.2Ghz. For 6 SPUs on PS3 this totals to $25.6*6 = 153.6$ GFlops. Peak FMA performance is almost achievable because load and store operations between SPU local store and SPU registry are dual-issued and all data transfers from main memory to local store run in parallel with the computations. This also applies to other BLAS routines like matrix-vector multiplication, because bytes/FLOP is sufficient enough. For routines like adding two vectors, the memory bandwidth sets the upper performance bound. The performance for BLAS routines is usually bounded either by peak SPU computational performance or memory bandwidth. Peak values for both of these are almost achievable. In fact, any problem sufficiently data parallel can be implemented very efficiently on CellBE where

- computations can be vectorized
- large continuous memory areas are accessed predictably

- bytes per computation is low
- inner loop can be pipelined
- inner loops do not require branching
- synchronization point count it relatively low

The sparse linear algebra is much more challenging. Many of the CPU features that make SPUs so good at dense linear algebra, make it hard to do sparse linear algebra. To my knowledge, IBM has provided optimized BLAS routines and reference implementation for dense LINPACK on CellBE, but there are no sparse matrix kernels. Here I present the main problems that are listed in [14] and [13] and also add some more conjugate gradient specific ones. I will also describe in brief, how I approached each one of them.

Vectorization In general, it is pretty hard to vectorize operations on sparse data. One must either use some reorganized vector format in matrix representation or compose and decompose vectors on the fly while computing. Examples of the first approach are compressed sparse row/column, blocked compressed sparse row (BCSR), some triangular and diagonal storage formats. There are many papers covering specific matrix types, some of them are listed in the chapter 2. However, in general when the sparsity pattern is not known, computational overhead of zero valued elements will cut the performance. Building vectors on the fly is not very suitable for CellBE because of the memory alignment restrictions and too big computational overhead. In my work, I used the most small grained approach that can exploit *fused multiply add* (FMA) fully for double precision. For this I organize matrix elements into two types of macroelements (*even* and *odd*) each containing 2 original matrix elements. This way the wasted computations for the worst case are 50% and for a random matrix 25%.

Memory alignment On the CellBE you have to use main memory (and therefore DMA transfers) for any problem size bigger than what fits into the SPE local store. For the best bandwidth you have to align all memory accesses to cache-line 128 bytes and the bigger blocks you transfer the better. When data is already in SPU local store you should access it so that the address is 16 byte aligned. Otherwise, the compiler must use extra assembler instructions for each memory access that still do 16 byte aligned vector load and then unpack the portion you requested. In my work I have overcome this by using 16 byte matrix macroelements, that store 2 elements from original matrix. Some precision loss is unavoidable, but without it design complexity or memory usage would grow

multiple times. In fact I tried to implement inner loop kernels using macroelements that keep full precision, but packing matrix into such blocks proved to be too time consuming and complex with little benefits. How the precision loss affects the result needs further investigation, but it is equivalent to reducing the input matrix precision before giving it to the solver which operates only on double precision. Currently the precision used in stored elements is 32 bits, but by switching to custom binary floating point format, the precision can be raised to 52 bits. When casting to double is done with custom binary format instead of the standard format (IEEE-754), the performance will also rise. Unrelated to the precision used in matrix macroelements, all calculations are done in double precision.

Double buffering The explicit DMA transfer on SPU requires that the application knows beforehand what data it needs. This way transferring chunks of data and computation can occur simultaneously. In general, when doing sparse matrix to vector dot product, it will need one random access to main memory per element. In general, with block formats you need one random access to main memory per block. All these accesses must happen explicitly as DMA transfers and whenever possible, during computing the last element or block. In my work I group vector into chunks and organize matrix macroelements into groups such a way that I can have matrix data and vector data being transferred to SPU local store during last block computation most of the time. In current implementation there remain some matrix elements that are handled only by PPU which will drastically drop overall performance, but that is not a problem of design, but just not yet implemented on SPU side.

Unrolling and pipelining Loop unrolling may be a problem when loop length is not known or there are data dependencies inside the loop. To overcome this I use two techniques. One is that whenever I group matrix macroelements, I try to make sure the group size is multiple of 4. The other one is that I keep elements in group ordered such a way that I can unroll inner loop by a factor of 4.

Limited memory Having only 256Mb main memory and 256Kb local store per SPU, we need to use it efficiently. Having duplicate data in memory would be a waste. Even when it is temporary it may be a problem. Input binary matrix data may use up over half the memory needed for application which means we really want to avoid copying it and want to do as much operations as possible in place without copying the data when restructuring is needed.

Waiting at synchronization points Whenever the data is not optimal (mainly when matrix is unbalanced) or memory transfers are not completed in order so that

some SPUs are left idle, we want to minimize the effect this has to the whole computation. For matrix-vector multiplication I currently do not do any vector chunk size adjustments to balance matrix elements used for that chunk, but I have thought about this and it is possible without restructuring the program. For the conjugate gradient loop however, I group some steps together to minimize waits and also avoid transferring same data multiple times whenever possible.

Staying general purpose solver There are cases when a sparse problem has enough structure in it so that solving it can be reduced to a special case. How some of the special cases are handled is described in the chapter 3. In my solver design I wanted to stay as general as possible when it comes to matrix structure. This means not the highest peak performance (but still good enough), but support for all kinds of matrixes. Speed will gradually drop as the matrix gets less suitable for the optimizations I have chosen, which are for small-grained diagonal matrixes.

Being preconditioner ready To improve conjugate gradient method, one usually wants to add some sort of preconditioner. In the current implementation there is no preconditioner, but it is designed so that adding an optimized and parallelized Jacobi preconditioner would be as effortless as possible.

Double precision is slow To cut cost and complexity in PS3, double precision arithmetic operations are not fully pipelined on SPU. This means that all such instructions introduce 7 CPU wait cycles on SPU during which no other arithmetic instruction can start. For all other arithmetics, every operation takes 1 CPU cycle when properly pipelined. PS3 CPU successor named PowerXCell does have fully pipelined double precision arithmetics, but this is not available in PS3. In the inner loop for each matrix macrolement, two double precision operations are performed. One is the conversion from single precision to double precision and another is the *fused multiply add* (FMA). We can do nothing about FMA, but in some cases the single to double precision can be done only in 1 cpu cycle when properly pipelined. The same technique can be used to convert any precision any floating point number to double precision, but then we have to use nonstandard binary format that is not compatible with IEEE 754.

To sum up the design and the current implementation from user's perspective, you have a *vanilla* double precision conjugate gradient solver that has matrix elements stored in single precision. The best performing matrix structure is a balanced diagonal matrix. Element distribution pattern near the diagonal does not affect the performance much, but elements far from the diagonal significantly reduce it. Solving small problems trying to exploit various levels of parallelism will not succeed, but as vectors and matrix get bigger you get much better results. Maximum matrix and vector sizes are limited to

how many huge pages you can allocate. The input matrix is in coordinate format and during all the conjugate gradient preparing and computation the memory size taken under matrix and vectors will not be bigger than $A_{elements} * 16 + rhs_{size} * 4 * 8$ bytes. Minimal memory is used besides huge pages. The solver is exposed as C library with a few easy to use functions and it can handle any matrix where some perform better.

There are features that I have thought about, but have not yet implemented. Adding these would probably not need restructuring existing code. Performing all matrix-vector multiplication operations on SPU would eliminate the performance drop currently caused by processing some matrix elements on PPU side. Then we could remove one synchronization point right after matrix-vector multiplication and save time as retransferring two vectors for subsequent dot product is not needed. By switching to custom floating point format, we can save time in matrix-vector multiplication kernel and also increase matrix element storage precision up to 52 bits. A data aware matrix and vector distribution logic can be added to make unbalanced matrixes perform better. The initial matrix restructuring process is not fully optimized and runs only on PPU. Some of the steps can be parallelized between SPUs. Last, but not least, a Jacobi preconditioner can be added that works on the same matrix structure.

5.2 Matrix-vector multiplication kernel

The matrix-vector multiplication kernel is designed for matrixes with unknown sparsity pattern. That means we cannot just use block format, because it may introduce too much overhead with redundant zero-valued elements inside a block. In addition, we have to handle the data in such a way that we can still use DFMA (double precision fused multiply and add) assembly instruction on a vector of two doubles. Otherwise we can not achieve good performance, because double precision operations stall the SPU execution pipeline for too long. Working with data out of 16 byte alignment would also contribute to wasted CPU cycles spent on reordering the data. The DFMA operation would use two doubles from vector p, two matrix elements and update two result vector q values as shown on the figure 5.1. This section explains how I organize matrix elements so that we can use DFMA in such a way. With these being the main points, I first constructed the kernel for one SPU and then built all the data distribution and parallelization around it.

Since this multiplication is done as part of conjugate gradient I use the same variable names as there. I use the same names throughout this thesis for consistency. Some markup may not be conventional, since I try to stay close to the actual C implementation. For example I use x and y as indexes instead of i and j, because these are very often used as loop variables in C. Let the multiplication be defined as follows:

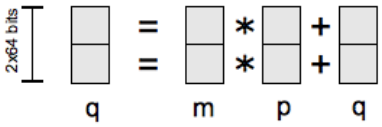

Figure 5.1: DFMA operation does 4 FLOPs on 128bit vectors each containing two double precision values

$$
\begin{array}{cccc}
 & q & = & A \cdot p \\
 & \begin{pmatrix} q_0 \\ \vdots \\ q_{n-1} \end{pmatrix} & = & \begin{pmatrix} a_{0,0} & & a_{n-1,0} \\ & \cdots & \\ a_{0,n-1} & & a_{n-1,n-1} \end{pmatrix} \cdot \begin{pmatrix} p_0 \\ \cdots \\ p_{n-1} \end{pmatrix} \\
q \leftarrow 0, & \forall x, y \quad q_x & \leftarrow & q_x + a_{x,y} p_y
\end{array}
$$

Running this directly on SPU is very inefficient, because all the vector elements in double precision are 8 byte aligned and we also can not use DFMA efficiently, because the elements are not blocked in any way. The simplest way to overcome these two problems at the same time is to block matrix elements into 2x2 macroelements and optimize SPU code for it. However in the worst case this method gives only 25% of efficiency. However, we can break the 2x2 block further to get two 2 element macroblocks which are to relatively simple to implement and give 50% efficiency at worst case. I call these *even* and *odd macroelements* or *packed elements* depending on context. The macroelements are defined and used in the multiplication as follows.

$$
\begin{array}{cccc}
 & q & \leftarrow & 0 \\
\forall A_{even} & \begin{pmatrix} q_x \\ q_{x+1} \end{pmatrix} & \leftarrow & \begin{pmatrix} q_x \\ q_{x+1} \end{pmatrix} + \begin{pmatrix} a_{x,y} & 0 \\ 0 & a_{x+1,y+1} \end{pmatrix}_{even} \cdot \begin{pmatrix} p_y \\ p_{y+1} \end{pmatrix} \\
\forall A_{odd} & \begin{pmatrix} q_x \\ q_{x+1} \end{pmatrix} & \leftarrow & \begin{pmatrix} q_x \\ q_{x+1} \end{pmatrix} + \begin{pmatrix} 0 & a_{x+1,y} \\ a_{x,y+1} & 0 \end{pmatrix}_{odd} \cdot \begin{pmatrix} p_{y+1} \\ p_y \end{pmatrix}
\end{array}
$$

By having x and y values macroblock aligned all vector accesses are 16 byte aligned as desired and we can also use DFMA directly on vector data loaded to SPU registries for *even* macroelements. For *odd* we need to execute one extra instruction to flip doubles for vector p macroelement.

As it can be seen, I switched to looping over matrix macroelements A_{even} and A_{odd} instead of x and y. This is because I do not want to have nested loops inside SPU code, because branching is expensive and the code would be more complex to optimize. However, the problem it raises is that I can not unroll the loop without worrying about

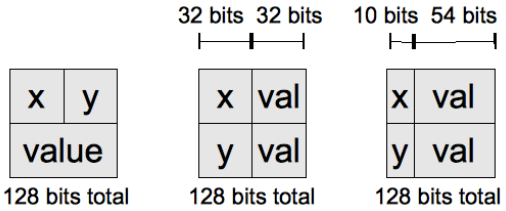

Figure 5.2: Matrix elements in memory: unpacked, currently used macroelement and optimal macroelement format

data dependencies. When two or more q macroelements with the same x are processed in the same inner loop iteration , the output will be corrupted. I avoid this by having matrix macroelements sorted by x. This way, when the matrix element count that the inner loop processes (also matrix block size from section 4.2) is big enough, I can divide these matrix elements into equally sized parts and unroll the inner loop so that each unrolled thread works on one part. Unrolled threads are safe to execute when each one has unique set or x values. Is such unrolling safe is determined during initial matrix prepare per group. When it is safe, a flag is set for the group so that SPU can know beforehand, which matrix-vector multiplication kernel it can use.

I have not yet talked about data structures that I use. For vectors this is trivial, since they are just array of double precision floats. All accesses to it are by macroelement offset which means they are all 16 byte aligned.

Matrix macroelements are more complex. For even and odd elements the binary format is the same, but values are handled differently. We want access to matrix macroelements also to be 16 byte aligned. For that we need each element size to be multiple of 16 bytes. Since hardware limits maximum size I can transfer with one DMA operation to 16kbytes which is 1024 vector macroelements, I need only 20 bits of addressing room for coordinates ($2^{10} = 1024 \Rightarrow 10$ bits per one coordinate) in optimal case. Combining this with the fact that we can avoid expanding input matrix while forming macroelements even in the worst case, it is desirable to keep one matrix macroelement not bigger than one unpacked element. To achieve that we must give away about 10 bits of precision, but that is toleratable, when all the calculations can still happen in double precision. The formats described are illustrated on figure 5.2

Currently, I drop the precision to as low as single precision, because then it is directly binary compatible with hardware and I can use FESD assembly instruction to convert single precision to double precision right before DFMA. But it turns out, unfortunately, that FESD is even slower operation than DFMA. Fortunately there are some lucky cases when we can do this much better. I call it simply *fast float to double* conversion here. It can be also modified to support custom precision numbers with the same conversion speed as for single to double has.

Since SPU is bad at branching, all the combinations of the methods described here,

type	optimizations	time	per 1 SPU	per 6 SPU
even	unroll by 4 + fast to double	5.492 s	746 MFlops	4476 MFlops
odd	unroll by 4 4 + fast to double	5.572 s	735 MFlops	4410 MFlops
even	unroll by 4	7.096 s	577 MFlops	3462 MFlops
odd	unroll by 4	7.416 s	552 MFlops	3312 MFlops
even	fast to double	11.248 s	364 MFlops	2184 MFlops
odd	fast to double	12.531 s	327 MFlops	1962 MFlops
even	-	12.531 s	327 MFlops	1962 MFlops
odd	-	13.814 s	297 MFlops	1782 MFlops

Table 5.1: Matrix-vector multiplication kernel optimization benchmarks on SPU

must be hard coded together and be hand-optimized in SPU matrix vector multiplication kernel inner loop. There are total $2^3 = 8$ of them by combining:

- odd or even matrix macroelement
- no unrolling or unrolling into 4 threads
- does not use *fast to double* or uses *fast to double*

The actual measured performances of these combinations are listed in table 5.1. Since the SPU instruction timing is static and the computational kernel does not access any shared CellBE resources, the result is accurate and always achievable. For the tests I use 1024 matrix macroelements, because this is the largest block of macroelements SPU can fetch from main memory with a single DMA operation. Using small matrix macroelement count will be somewhat slower, because loop initialization time is not amortizised out. Vector sizes do not affect the result. Since the code is heavily hand tuned, compiler optimizations between -O2 to -O4 did not change the result and using -O5 produced even slightly worse results. All the benchmarks are compiled with -O3 option, execution time is measured with unix `time` utility and the operation in test is repeated 1 000 000 times.

From the table 5.1 it can be seen that the actual measured speed is two or more times slower than the theoretical maximum given in section 2.3 which is approximately 11 GFLOP/s total per 6 SPUs. Dense linear algebra can use memory access patterns where load/store operation operate on continuous memory locations. In sparse matrix-vector multiplication kernel this is not so. Per one FMA instruction, at first matrix element must be unpacked (probably using *fast float to double* described in the next section) and inspected to locate source and destination vector addresses. Then these vector macroelements are loaded, FMA is executed and result is stored back. For dense cases, the unpacking and address computation is not nessecary, thus the difference. This is the cost of fine grained approach.

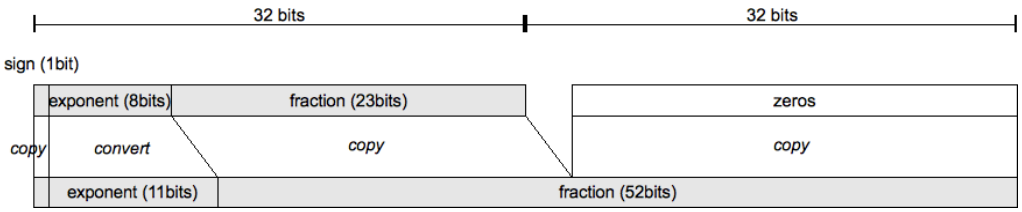

Figure 5.3: Converting float to double (half of the quadword)

5.3 Fast float to double conversion

Current packing the matrix elements will loose the floating point precision from double to single. In the future versions the precision loss may be smaller, but a loss of a few bits is still inevitable unless bigger than 16 byte macroelements are used. In this chapter I will present a way how in some lucky cases two single precision numbers can be converted to double precision with only one CPU cycle on SPU when pipelined properly.

The DFA assembly instruction that converts two floats to two doubles inside a 128bit vector is just as slow as DFMA. It takes 19 CPU cycles to execute from which it stalls execution pipeline for 6 cycles. Therefore even in the best case, the conversion takes 7 CPU cycles. However, when the float is known to be in the range $(-131071.98... - 0.000030517580, 0, 0.000030517580...131071.98)$, then you can use a much more efficient conversion that requires 2 SPU instructions. Those will execute in separate SPU pipelines, so that compiler can usually reorder them for dual issue unless there are dependency stalls. Therefore it usually takes only 1 CPU cycle to execute. Even in the worst dependency stall case, it will not take more than 6 CPU cycles for the whole conversion to complete.

The CellBE uses IEEE-754 floating point binary format consisting of 1 bit sign, exponent and fraction. 32 bit float has 8 bit exponent and 23 bit fraction while 64bit float has 11 bit exponent and 52 bit fraction. The conversion process copies the sign and fraction, but the exponent part must be recomputed because it is encoded differently. To get the actual exponent, one has to decode it by subtracting exponent bias from it, which is 127 for single precision and 1023 for double precision. The process on conversion is visualized on figure 5.3.

To convert a quadword containing 2 floats to doubles, we must initially have the floats at the same position as DFA instruction requires. In addition the unused space must be contain zeros like this {float, 0, float, 0}. If you do not make sure that there are the zeros, the conversion would still work, but the result would contain a small fraction of "randomness".

The conversion uses two assembly instructions. The first shifts the input quadword by 3 bits to the right to get the whole fraction part and 7 least significant bits of

Algorithm 5.1 Fast conversion from floats to doubles in C

```
qword f2d_maskb = (qword)((vec_uint4){ // constant
        0x07FFFFFF, 0xFFFFFFFF,  0x07FFFFFF, 0xFFFFFFFF});

qword f_in = (qword)((vec_float4){32.1f, 0,
-0.017f, 0}); // 0-s are important
qword d_out;

d_out = si_rotqmbii(f_in, 5);
d_out = si_selb(f_in, d_out, f2d_maskb);

// d_out contains here (vec_double2){32.1f, -0.017f}
```

exponent correct. The second instruction selects bits from input quadword and shifted quadword to get the sign and the 4 most significant bits of exponent correct. Using the SPU intrinsics in C the exact algorithm is shown on algorithm 5.1.

5.4 Restructuring the matrix

The restructuring process consists of 6 distinguishable steps that are currently all performed on PPU, but in theory some of them can be also done on SPUs in parallel fashion. I call these steps as follows: splitting, sorting, packing, grouping, fixing group offsets and resorting groups. Despite the number of steps needed, the overall complexity of it is low when the input matrix is already sorted. When it is not sorted, then the first sorting will take most of the time, because it currently uses vanilla quicksort algorithm. Also when the matrix rows get longer, packing step (matrix macroelement forming) will take more time, because it needs to scan elements from the next row.

The overall procedure of reorganizing the matrix is shown on figure 5.4. Blocks represent data structures - the structure type, name and size are given in C notation. Arrows accompanied by bold text show how the data is processed in one or two words.

Split step takes the matrix elements and divides them into two parts named *even* and *odd* where *even* elements have $(x+y) \bmod 2 = 0$. This is needed for the following packing step which can use 2 *even* elements to form one *even* macroelement or 2 *odd* elements to form one *odd* macroelement. Those macroelements will provide as fine granularity as possible while being directly able to fully use DFMA (double precision fused multiply and add) on SPU. Throughout the preparing stage the *even* and *odd* ranges and handled separate from each other except the very last one. The splitting is done in 2 steps: first we count range sizes for both element types and then we just walk through the elements and put them where they

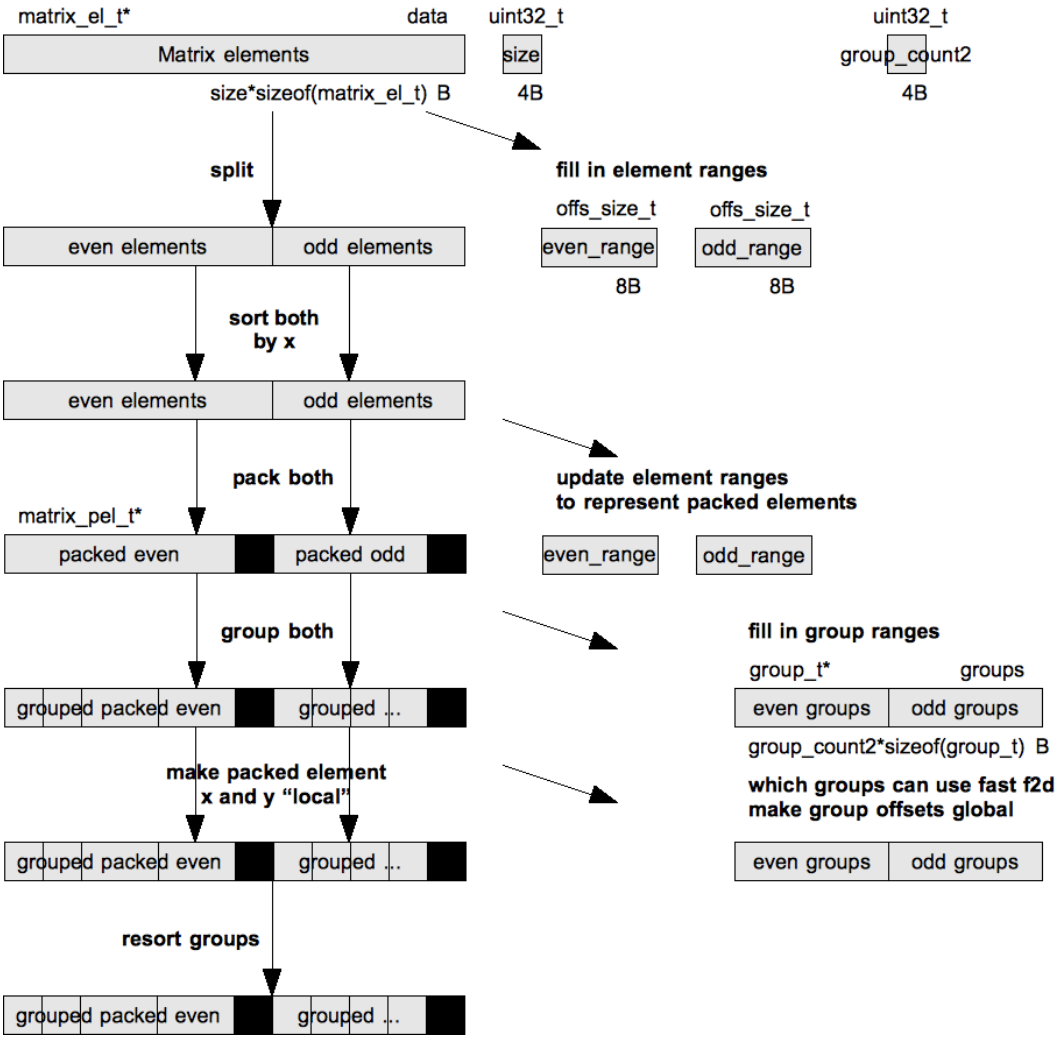

Figure 5.4: Overview of matrix reorganization

belong. The complexity of this is $O(n)$.

Sorting step takes matrix elements and orders them by x ascending. The sorting does not care about y and also it does not have to be stable. This process is needed for the packing step and also combined with the last step makes it possible to pipeline inner loop in SPU code. This is described in more detail later on. When the input matrix elements are already ordered then this step just checks the ordering and completes fast. It should be noted that this is indeed so, because splitting did not change the ordering by x. The sort is implemented by quicksort so in general the complexity is $O(n\ logn)$ which is the worst among other matrix preparing steps when initial matrix was not ordered already.

Packing is a step where matrix elements are combined into 2 element macroelements. I call it packing, because in this step memory usage of the matrix can only decrease. This is so because each macroelement holds one pair of coordinates and two matrix element values which will all fit in 16 bytes. Original matrix elements also used 16 bytes. Some precision loss is inevitable with this approach, but it greatly simplifies overall implementation. I also have constructed matrix-vector multiplication kernels for more complex macroelements using full double precision and respecting the 16 byte alignment, but packing original matrix into those proved very complex, time consuming and error prone. Currently I use single precision in macroelements, but it can be increased up to 52bit by using custom floating point format.

The packing step will search for the following patterns in the input matrix. Here $a_{x,y}$ stands for original matrix element value and 4 number groups written down as $\{...\}_{even}$ and $\{...\}_{odd}$ denote packed element (macroelement).

$$\begin{pmatrix} a_{x,y} & 0 \\ 0 & a_{x+1,y+1} \end{pmatrix} \rightarrow \left\{\tfrac{x}{2}, \tfrac{y}{2}, a_{x,y}, a_{x+1,y+1}\right\}_{even}$$

$$\begin{pmatrix} 0 & a_{x+1,y} \\ a_{x,y+1} & 0 \end{pmatrix} \rightarrow \left\{\tfrac{x}{2}, \tfrac{y}{2}, a_{x+1,y}, a_{x,y+1}\right\}_{odd}$$

$x \bmod 2 = 0$

$y \bmod 2 = 0$

All the packing step can be done in place and indipendantly for *even* and *odd* elements by just walking over the matrix elements ascendingly. For each matrix element we search for the match. Since we have sorted the matrix, we only need to look no further than $r_x + k_{x+1}$ elements ahead where r_x is the remaining element count in the current row x and k_{x+1} is the element count in the next row. The complexity for this

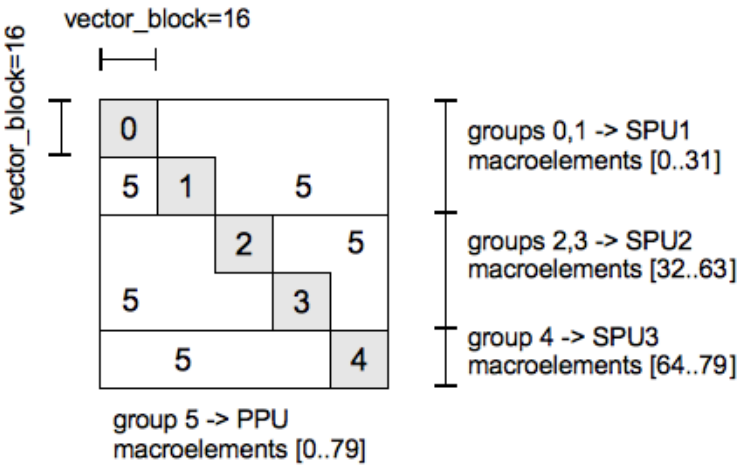

Figure 5.5: Grouping of matrix macroelements

is $O(n * k_{avg})$ which is good because usually average row length k_{avg} is small. After this step the matrix and vector coordinates we work on are macrocoordinates which assures that all memory access for vector data will be 16 byte aligned.

Grouping step groups matrix macroelements in such way that all groups (except last one in current implementation) can be processed in matrix vector multiplication in parallel. For simplicity the last group currently contains macroelements that are evaluated by PPU after all SPUs have done their work in parallel, but this does not have to be so. To get the macroelements grouped efficiently, we do it in two passes. At first we walk the macroelements ascendingly to count how many elements each group will contain. Then we can use that data to move each macroelement to right place by moving each one only once. The group number is determined by the the following rule:

$$\text{group\_nr} = \begin{cases} \lfloor x \div \text{vector\_block} \rfloor & \text{, if } \lfloor x \div \text{vector\_block} \rfloor = \lfloor y \div \text{vector\_block} \rfloor \\ \text{group\_count} - 1 & \text{, otherwise} \end{cases}$$

The outcome of the grouping is illustrated on figure 5.5. Assigning groups to SPUs is not done during matrix prepare, but to see how the groups will get eventually divided between SPUs and PPU, can also be seen from that figure. The grouping step has complexity of $O(n)$.

Fixing group offsets step will convert macroelement x and y values from global space to local space. This way we can transfer block of vector data with DMA into SPU local store and we can address it directly. This adjustment is not done for the last group, since PPU does not do any DMA and accesses vector data directly. During this walking through matrix elements in this simple step we also mark which groups can use fast float to double conversion. This is discussed in

detail in section 5.3. Complexity of this step is $O(n)$ and the equation used for offset fixing is the following.

$x = x - \mathbf{group\_nr} * \mathbf{vector\_block}$

$y = y - \mathbf{group\_nr} * \mathbf{vector\_block}$

Resorting groups fixes macroelement order that may have been lost during grouping. The grouping step will not preserve the sort order, but it usually does not significantly reorder the elements either. We care about the sorting order because we want to unroll matrix-vector multiplication kernel on SPU. For this we must not have data dependencies between matrix elements evaluated in one inner loop iteration. It turns out that by having macroelements sorted by x makes it easy to determine at runtime whether SPU can unroll matrix-vector multiplication kernel or not. How it can be done in written in section 5.2.

In this chapter I gave an overview of the matrix vector multiplication and how it is implemented on PS3. At first I show a few challenges that anyone implementing it must face. Then I show how the multiplication works algorithmically and how what kind of inner loops did I implement. All inner loops are accompanied by measurements of their performances. Then I presented a way how to do single precision (or any precision lower than double) conversion to double much faster in some special cases than it is normally done on SPU. For the matrix vector multiplication kernels to work, the matrix needs to be restructured. How this is done was covered lastly in this chapter. In the following chapter I will briefly show how the solver can be used.

Chapter 6

Usage examples

In this chapter I will introduce the solver C implementation and show how to use the APIs. One is the conjugate gradient solver API and the other is a more general purpose API that I designed to simplify using the resources provided by CellBE. Lastly I will show one application where the API is used - a real time wave simulation for Teeviit 2009 education fair. While describing current working version of the solver, it may change as code matures.

6.1 Conjugate gradient C API

The API is currently usable through a static library or by directly including source code. In general one needs to create the solver giving matrix (in coordinate format) and right hand side vector with function `solver_create(...)`. The solver needs to be freed with `solver_free(...)` before exiting. The solver may be created and destroyed multiple times during the execution of main program. The creation step allocates SPU resources and prepares all data for following conjugate gradient. Among this matrix is also restructured. The freeing releases SPU resources held and clears up any memory allocated.

After the solver is set up, one can let it run for `max_iterations` or until result is accurate enough regards to `error` with function `solver_solve(...)`. When this function returns with error larger than specified, then it could not find good enough solution in `max_iterations`. To improve the result, one may call this function as many times as needed, but keeping it from running forever by limiting `max_iterations`.

When the same matrix can be used with many right hand side vectors, the solver can be reset without fully destroying and recreating it. A `solver_reset(...)` method clears solver internal state. Calls to `solver_solve(...)` will make it start from the "beginning". Using this can avoid wasting time on solver creation and matrix reorganization.

The data types used are `dataset_t` for solver state, `matrix_t` for matrix and

`vector_t` for vector. These are available through library headers. The underlying structures are simple, so that they can be easily accessed and modified between solver iterations. There are also a few functions to simplify matrix and vector allocating, freeing, loading from disk and saving to disk. Supported formats are currently Matrix Market format, simple text format and binary format.

6.2 Ecosystem C API

The conjugate gradient solver is implemented against a more general purpose star topology master-worker model-providing API that I call either "ecosystem" or by code-name "LW" (derived from lightweight). It basically provides two main features, one is resource management (SPU reserving, memory sharing, initializing group barriers, ...) and the other is sending orders to SPUs and gathering results back. It can be used to handle DMA memory transfers too with double buffering for overlapping DMA with computation. The ecosystem is built on top of DaCS, which makes it possible to use most of the DaCS features without conflicts.

The main design goal of this ecosystem is to make writing parallelized algorithms much easier than it would be with either DaCS or lower level approaches. Then it is possible to concentrate on tuning the algorithm cores or overall data distribution schemes. Otherwise a considerable amount of time may go to writing glue and data masquerading code. The ecosystem is designed to act nice with the surrounding application by posing as little requirements to coding and runtime environment as possible. The only requirement is having DaCS library present at runtime.

The LW consists of two parts - PPU and SPU part. These must be used together. Calling `lw_startup(...)` will "start" the LW ecosystem, load the SPU program into each available SPU and put them into loop where they wait for commands from PPU. Calling `lw_cleanup(...)` will instruct all SPUs to exit gracefully freeing any resources used. After that `lw_shutdown(...)` will "close" the LW ecosystem. When running, one can use `lw_sig...(...)` functions to give orders to SPUs who will receive it through `on_signal(...)` callback.

To use the ecosystem one has to do the following in minimal:

- include `lw.h` in PPU code and `lw_spu.h` in SPU code. For double buffering methods, include `dbuf.h` on SPU side too
- use `lw_startup(...)`, `...`, `lw_cleanup(...)`, `lw_shutdown(...)` sequence in PPU code
- call `lw_main()` in main loop and implement `on_startup(...)`, `on_signal(...)` and `on_cleanup(...)` on SPU code

on PPU side	on SPU side	description
startup		
`lw_mem_share`	`lw_mem_accept`	shares a main memory region to all SPUs
`lw_set_signal_range...`	-	instructs LW about how to split vectors among SPUs
cleanup		
`lw_mem_destroy`	`lw_mem_release`	releases previously shared memory
normal operation		
`lw_signal`	`on_signal`	gives and order to all SPUs
`lw_send_int`	`lw_recv_int`	sends integer to all SPUs
-	`dbuf_pull...`	pulls data from main memory in double buffered way
-	`dbuf_push...`	pushes data to main memory in double buffered way

Table 6.1: LW ecosystem methods

Now when the code layout is set, use the methods provided by LW on both PPU and SPU side to share resources, order SPUs to do work and do double buffered DMA. An overview of what methods are directly available is listed in table 6.1.

The current LW implementation works on one PS3, but the model itself is not limited to one PS3. In theory it supports any distributed memory model where computational units form a tree structure. The approach is closely related to a paper analyzing explicitly managed memory hierarchies [16]. In the cluster setup of PS3s one would have 3 level memory hierarchy. On the top of the tree there would be one machine that has enough RAM to hold the whole problem, RAM on PS3s would serve as local copies of the data which are managed by PPUs and accessed by SPUs. Implementing this would be a future work.

6.3 Real time interactive surface wave visualizer for Teeviit 2009 fair

One major milestone in my work was providing a conjugate gradient solver for a real time interactive surface wave simulation for Teeviit 2009 student fair. It was developed together with Oleg Batrashev and consists of 2 parts. One is a simulator server running on PS3 and the other is visualizer client. Actually we have two visualizers. Unity3D powered one which runs on Windows and Mac OS X and is written by me. And another written by Oleg in Python and OpenGL. The Unity3D visualizer can be seen on figure 6.1.

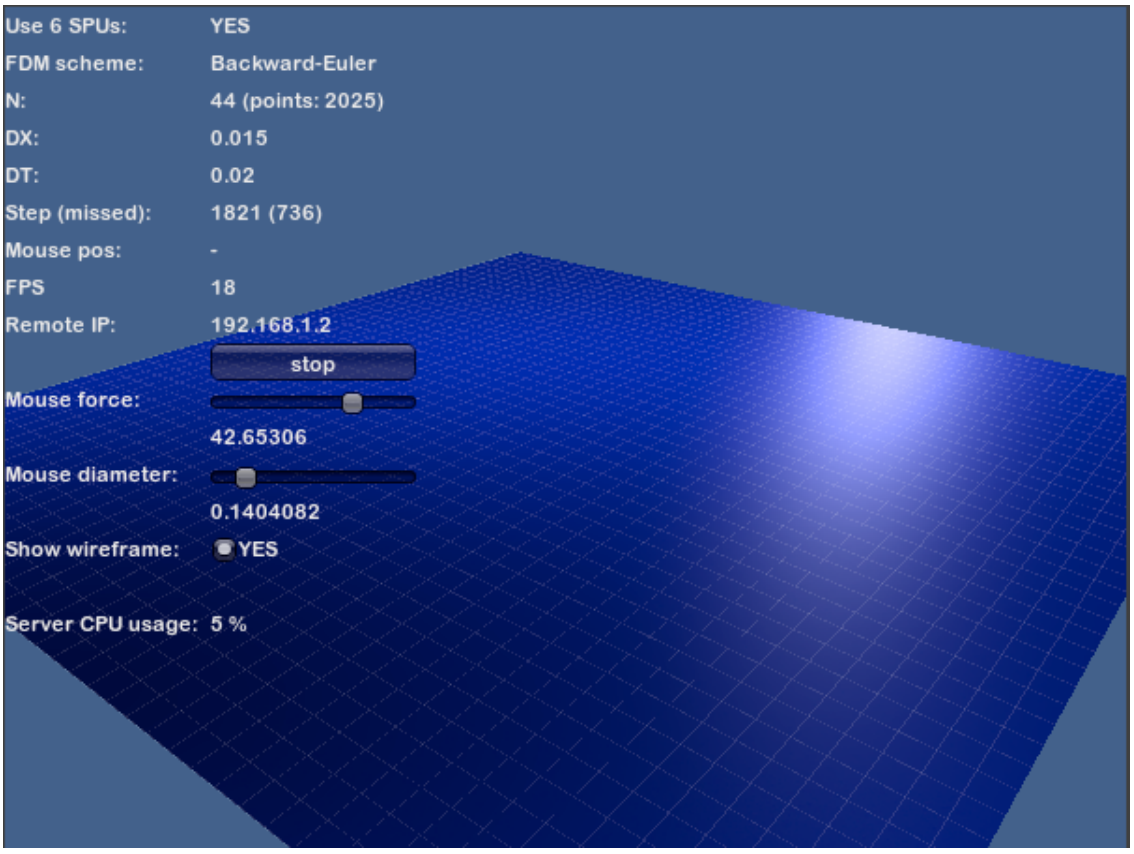

Figure 6.1: Interactive wave simulation real time visualization

From the architecture point of view, this is a client server application with not too big code base. The system works as follows. A server running on PS3 listens on a known UDP port. A client visualizer can (re)start the server by sending an UDP packet. After that a server creates a surface and starts the simulation on it. The heightmap is sent in fixed discretized time intervals to visualizer using UDP packets. The visualizer assembles the packets and draws the surface on screen. Pressing mouse button on the surface seen on the screen will send mouse position back to server over UDP which generates disruptions in wave horizontal speed vector.

At the time of the fair the solver was only using a fraction of power it does now. However the approach used there was the same, but some parts in matrix vector multiplication had to be switched off because of too many bugs. At the moment the solver is capable of providing data faster than it can be sent over network to visualizer and rendered on screen. To get it working smoothly again, the visualizer needs to be revised.

Chapter 7

Results and future

This thesis tries to collect together my experience and results from working with the conjugate gradient solver algorithm on the PS3 platform. The biggest goal from my work perspective is to have a general purpose solver that can be used in various projects needing the processing power offered by PS3 console. Currently the library is in getting ready for early testing as missing functionality is added.

Despite the difficulties, the PS3 is a desirable hardware platform for solving large systems of linear equations. This is mainly because of the price per flop per watt ratio, but also because of the explicit multicore architecture that can reach very high percentages from peak performance. For dense matrixes and for some special cases like tridiagonal matrixes, there already exist pretty good optimized double precision and mixed precision solvers. When appropriate, one can choose an existing implementation and use it. Unfortunately only little code is available to public because most implementations are proof of concept.

To salvage the situation, I have created a double precision conjugate gradient solver design and implementation with matrix elements stored as single precision. It uses a very fine grained (only 2 elements in block) matrix format to minimize the performance drop for low locality matrixes. It performs best with diagonal matrixes, but can handle any other type with degrading performance as the matrix gets less suitable. The solver is available as a static library with easy to use C API.

Some parts of the solver are currently not yet implemented. Before the library can be used in practice, some of them need to be eliminated. Currently the greatest concern is that PPU evaluates some matrix elements in `sparsedot` method. This could be done much more efficiently on SPU side and it currently creates big performance drops when there are a lot of elements far from main diagonal.

The conjugate gradient solver is built on top of a more general purpose minimalistic ecosystem C API that emerged during the work. Its primary feature is that it simplifies programming for multicore platforms with distributed memory hierarchies. The API is currently realized for running on one PS3, but it can be extended to run on cluster

of PS3 machines. This transition does not need changing the programs written for this API. Still, declarative style flags are usually needed for optimal memory management.

I implemented a showcase of the solver for educational fair Teeviit 2009. The solver was used in conjunction with interactive real time surface wave visualizer. One PS3 was used as a server and a usual laptop was used to show interactive 3D surface which could be altered with mouse to create waves. The current solver implementation has improved a lot after its use at Teeviit. The main difference is that now matrix-vector multiplication is calculated mostly on SPUs, while in Teeviit server code, did not utilize the PS3 SPU cores as much as now.

A next step to add features like a Jacobi preconditioner and continue testing and finding and fixing more bugs. Also integrating the best parts of conjugate gradient solver implemented by Lauri Tulmin into library is be desirable. The API documentation needs to be extended, especially from usage perspective. After that, a demo reference implementation of primary use cases should be implemented. Maybe in conjunction with `netcat`, NFS (Network File System) or some other widely deployed technology, it can be used as a service on the local network.

In conclusion I can say that the current prototype works well for diagonal matrixes and it can be further improved to support other types of matrixes efficiently too. I hope that the emerging solver library will benefit for people trying to use PS3 as a solver for large problems.

Kaasgradientidel põhinev lineaarvõrrandisüsteemide lahendamise teek Playstation 3 konsoolile

Magistritöö (30 EAP)

Toomas Laasik

Kokkuvõte

Antud lõputöö võtab kokku minu senise töö Tartu Ülikooli Hajusüsteemide Uurimisrühmas, kus ma peamiselt tegelesin kaasgradientide meetodil töötava lineaarvõrrandisüsteemide lahendaja implementeerimisega Playstation 3 mängukonsoolile. Töö tulemustena esitan ma hetkel veel arendusjärgus oleva teegi C keele jaoks, mis mähib selle funktsionaalsuse mugavasti kasutatavasse pakendisse. Seda teeki on kasutatud juba Teeviit 2009 haridusmessil laine pinna interaktiivses simulatsioonis. Lisatulemusteks võib lugeda veel C liidest, mis lihtsustab programeerimist puukujuliste mäluhierarhiatega süsteemidele, nagu on üks PS3 või ka klaster PS3 konsoolidest. Hetkel on sellest kasutatav vaid ühte PS3 konsooli kasutav versioon.

PS3 on odav platform omades head jõudluse ja hinna suhet ja ka head jõudluse suhet eralduva soojusega. Seal kasutatavad keskprotsessorid on sama arhitektuuriga, mida kasutatakse ka hetkel maailmas kiiruselt teises superarvutis IBM Roadrunner. See teeb PS3 platvormi attraktiivseks teadlaste seas, kellel on vaja tegeleda suuremahuliste probleemide lahendamisega.

Siiski, PS3 mängukonsoolide laialdast kasutuselevõttu teadusarvutustes takistab arhitektuuriline omapära, mis ohverdab programmeerimise lihtsuse jõudluse nimel. Riistvaraliste tuumade haldus (*SPU*), protsessorisisene lokaalne mälu (*Local Store*) kasutamine, tuumadevaheline kommunikatsioon, arvutuste ja DMA mälupöörduste samaaegsus, piirangud mäluaadresside kasutamisele (*cache-line and quadword alignment*) - kõige sellega peab programmeija ise eksplitsiitselt tegelema, kui ta soovib saavutada tulemusi, milleks PS3 võimeline on.

Töö kõigus realiseeritud kaasgradientide meetodit kasutav lahendaja paralleliseerib arvutused kasutades ühe PS3 konsooli ressursse, sealhulgas ka hajusa maatriksi ja vektori korrutis. Selle arvutuse optimeerimiseks kasutatakse väikeseteralist blokkformaati, mis küll ei paku parimat jõudlust, aga halvimal juhul on ebaeffektiivsus suhteliselt väike. Samuti ei seata piiranguid maatriksi struktuurile, aga hetkel on optimeeritud vaid diagonaalsed maatriksid. Andmestruktuuride mälus hoidmine ja töötlemine on optimeeritud suhteliselt väikese kasutatava muutmälu hulgaga, mis on 256 Mb. Kasutada saab sellest vaid niipalju, kui mitu suurt lehekülge (*huge pages*) allokeerida on võimalik. Kõik arvutused teostatakse 64 bitiste ujukomaarvude täpsusega, aga kasutatava maatriksi elementide täpsust vähendadakse implementeerimise ja mälukasutuse huvides hetkel 32 bitini ja kuni 52 bitini ilma suuremate koodi muudatusteta.

Töö tulemusena on olemas esialgne versioon lineaarvõrrandisüsteemide lahendamise teegist. Selle edasiarendamisel tekib kergesti kasutatav ja kõigile kättesaadav lahendaja.

Bibliography

[1] Basic linear algebra subprograms library programmer's guide and API reference. http://public.dhe.ibm.com/software/dw/cell/BLAS_Prog_Guide_API_v3.1.pdf. Last accessed in 2010.05.24.

[2] BeBOP optimized sparse kernel interface (OSKI). http://bebop.cs.berkeley.edu/oski/. Last accessed in 2010.05.02.

[3] CgSolver project Trac wiki. http://dougdevel.org/ps3/trac/wiki/CgSolver. Last accessed in 2010.05.23.

[4] Distributed Systems Group at Tartu University. http://ds.cs.ut.ee/. Last accessed in 2010.05.24.

[5] Education fair Teeviit 2009. http://www.teeviit.ee/. Last accessed in 2010.05.24.

[6] IBM cell SDK. http://www.ibm.com/developerworks/power/cell/documents.html. Last accessed in 2010.05.24.

[7] PS3 firmware (v3.21) update – PlayStation blog. http://blog.us.playstation.com/2010/03/28/ps3-firmware-v3-21-update/. Last accessed in 2010.05.24.

[8] Sony press release Aug 18, 2009.

[9] Sony press release May 8, 2006. http://www.scei.co.jp/corporate/release/pdf/060509ae.pdf. Last accessed in 2010.05.24.

[10] Programmer's guide - cell BE programming handbook including PowerXCell 8i. https://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/1741C509C5F64B3300257460006FD68D, April 2010. Last accessed in 2010.05.24.

[11] Kevin J. Barker, Kei Davis, Adolfy Hoisie, Darren J. Kerbyson, Mike Lang, Scott Pakin, and Jose C. Sancho. Entering the petaflop era: the architecture and performance of roadrunner. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11, Austin, Texas, 2008. IEEE Press.

[12] Oleg Batrašev. Laine simulatsioon lõplike diferentsiaalide meetodiga (LDM). http://dougdevel.org/ps3/trac/attachment/wiki/WaveSimulation/description.pdf. Last accessed in 2010.05.24.

[13] A. Buttari, J. Dongarra, and J. Kurzak. Limitations of the PlayStation 3 for high performance cluster computing. *Manchester Institute for Mathematical Sciences, School of Mathematics, July*, 2007.

[14] A. Buttari, P. Luszczek, J. Kurzak, J. Dongarra, and G. Bosilca. A rough guide to scientific computing on the PlayStation 3.

[15] David DuBois, Andrew DuBois, Thomas Boorman, and Carolyn Connor. Non-Preconditioned conjugate gradient on cell and FPGA based hybrid supercomputer nodes. In *Proceedings of the 2009 17th IEEE Symposium on Field Programmable Custom Computing Machines*, pages 201–208. IEEE Computer Society, 2009.

[16] T. J Knight, J. Y Park, M. Ren, M. Houston, M. Erez, K. Fatahalian, A. Aiken, W. J Dally, and P. Hanrahan. Compilation for explicitly managed memory hierarchies. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, page 236, 2007.

[17] A. Komornicki, G. Mullen-Schulz, and D. Landon. Roadrunner: Hardware and software overview. *IBM Redpaper*, 2009.

[18] J. Kurzak, A. Buttari, and J. Dongarra. Solving systems of linear equations on the CELL processor using cholesky factorization. *IEEE Transactions on Parallel and Distributed Systems*, page 1175–1185, 2008.

[19] J. Kurzak and J. Dongarra. Implementation of the mixed-precision high performance LINPACK benchmark on the CELL processor. *LAPACK Working Note*, 177:06–580, 2006.

[20] R. J Meuth and D. Wunsch. A survey of neural computation on graphics processing hardware. 2007.

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

[22] Lauri Tulmin. *A Conjugate Gradient Solver on the PlayStation 3 - a native approach*. Master's thesis, University of Tartu, Tartu, 2010.

[23] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. *Parallel Computing*, 35(3):178–194, March 2009.

Appendix A

Measured performances

```
Vector size: 131072
Matrix size: 655360
  matrix elements per vector element: 5
  even macroelement: 327674
  odd macroelement:  458740
Format: <vector block size> <matrix block size>
  <measured FLOP/s> (<DMA transfer utilization>)

Benchmark: sparsedot (matrix-vector multiplication)
  512   256  2.694882 GFlop/s     13.719417 Gb/s
  512   512  2.772619 GFlop/s     14.115169 Gb/s
  512   768  2.772619 GFlop/s     14.115169 Gb/s
  512  1024  2.669930 GFlop/s     13.592385 Gb/s
 1024   256  2.720306 GFlop/s     13.848845 Gb/s
 1024   512  2.883524 GFlop/s     14.679776 Gb/s
 1024   768  2.883524 GFlop/s     14.679776 Gb/s
 1024  1024  2.826984 GFlop/s     14.391937 Gb/s
 1536   256  2.645435 GFlop/s     13.482716 Gb/s
 1536   512  2.854974 GFlop/s     14.550653 Gb/s
 1536   768  2.912651 GFlop/s     14.844606 Gb/s
 1536  1024  2.883524 GFlop/s     14.696160 Gb/s
 2048   256  2.621385 GFlop/s     13.345251 Gb/s
 2048   512  2.826984 GFlop/s     14.391937 Gb/s
 2048   768  2.854974 GFlop/s     14.534432 Gb/s
 2048  1024  2.854974 GFlop/s     14.534432 Gb/s
```

```
Benchmark: conjugate gradient method
  512    256   2139.920918 MFLOP/s
  512    512   2173.183938 MFLOP/s
  512    768   2173.183938 MFLOP/s
  512   1024   2150.894872 MFLOP/s
 1024    256   2317.262431 MFLOP/s
 1024    512   2410.485632 MFLOP/s
 1024    768   2410.485632 MFLOP/s
 1024   1024   2396.711429 MFLOP/s
 1536    256   2375.415254 MFLOP/s
 1536    512   2458.763158 MFLOP/s
 1536    768   2473.226471 MFLOP/s
 1536   1024   2473.226471 MFLOP/s
 2048    256   2424.419075 MFLOP/s
 2048    512   2541.966667 MFLOP/s
 2048    768   2541.966667 MFLOP/s
 2048   1024   2541.966667 MFLOP/s
```