Artificial Neural Networks on the Cell Broadband Engine Architecture

A study of the Cell processor's performance with a memory intensive algorithm

Per Hasselström

Master´s Thesis in Computer Science
at Stockholm University, Sweden 2008

Stockholms universitet

Artificial Neural Networks on the Cell Broadband Engine Architecture

A study of the Cell processor’s performance with a memory intensive algorithm

Per Hasselström

Master´s Thesis in Computer Science (30 ECTS credits)
Single Subject Courses
Stockholm University year 2008
Supervisor at Nada was Mikael Djurfeldt
Examiner was Anders Lansner

TRITA-CSC-E 2008:079
ISRN-KTH/CSC/E--08/079--SE
ISSN-1653-5715

Department of Numerical Analysis and Computer Science
Royal Institute of Technology
SE-100 44 Stockholm, Sweden

Abstract

The Cell Broadband Engine Architecture is a new multi-core CPU architecture designed to overcome, among other things, the bottle neck of memory bandwidth. The Cell processor can be found in the gaming console Playstation 3, and in IBM Bladecenter servers. Simulations of artificial neural networks are usually memory intensive, which makes it uncertain whether the Cell's high potential computational performance can be achieved running such simulations. In order to test this, two different implementations of a Bayesian Confidence Propagation Neural Network are made. The first implementation puts the main communication load on the main memory interface. The other puts the main communication load on the bus that connects the Cell's internal CPU-cores with each other. The first implementation reached the memory bandwidth limit relatively quickly, and performance was bound by it. The other implementation did not reach the memory bandwidth limit, and performance was not restricted by the memory bandwidth. Performance of the Cell processor with this second implementation was about 100 times faster than a conventional Intel Pentium 4 2.60 GHz CPU.

Referat

Artificiella Neuronnätverk på Cell Broadband Engine-arkitekturen En studie av Cell-processorns prestanda med en minnesintensiv algoritm

"Cell Broadband Engine Architecture" är en ny CPU-arkitektur designad för att, bland annat, komma över problemet med minnesbandbredd. Cell-processorn finns i spelkonsollen Playstation 3, och i IBM Bladecenter-servrar. Simuleringar av artificiella neuronnätverk är ofta minnesintensiva, vilket gör det osäkert om huruvida Cell-processorns höga potentiella beräkningskapacitet kan uppnås vid körande av sådana simuleringar. För att testa detta har vi implementerat två olika versioner av ett "Bayesian Confidence Propagation"-neuronnätverk. Den första versionen sätter kommunikationsbelastningen på kommunikationen med huvudminnet, den andra versionen sätter den på kommunikationen inne i själva Cell-processorn, i datatrafiken mellan dess kärnor. Den första versionen uppnådde minnesbandbreddens gräns relativt snabbt, och prestandan blev begränsad av kommunikationen med huvudminnet. Den andra versionen nådde inte upp till minnesbandbreddens tak, och prestandan blev inte begränsad av kommunikationsbelastning. I jämförelse med en Intel Pentium 4 2.60 GHz-processor kom Cell-processorn (i den andra versionen) upp i en hastighet som var cirka hundra gånger snabbare än den konventionella processorn.

Contents

List of Figures

1 Introduction **1**
1.1 Background 1
1.2 Problem definition 1
1.3 Report composition 2

2 Theory **3**
2.1 Introduction to the architecture of the Cell Broadband Engine 3
2.1.1 The PPE 4
2.1.2 The SPU 5
2.1.3 The memory flow controller 6
2.1.4 The reasoning behind the design of the CBEA 6
2.2 Programming the CBEA 7
2.3 Spiking Bayesian Confidence Propagation
Neural Network (BCPNN) with hypercolumns 8
2.3.1 Auto-associative artificial neural networks 9
2.3.2 The BCPNN 9

3 Implementation **13**
3.1 Development-tools 13
3.1.1 Hardware 13
3.1.2 Software 13
3.1.3 Lessons learned from other CBEA-applications 14
3.2 Test-implementation of a non-spiking BCPNN-net 14
3.2.1 Initial scalar version 14
3.2.2 SIMDizing the scalar version 15
3.2.3 DMA-transfer considerations 18
3.2.4 Distributing the workload to multiple SPEs 19
3.3 Description of two different implementations of spiking BCPNN 19
3.3.1 Implementation of Net 1 21
3.3.2 Implementation of Net 2 25
3.4 Implementation-, and measurement-tools 34
3.4.1 How to measure performance 34
3.4.2 Tests with the full system simulator 34
3.4.3 Code analysis with the spu-timing static timing analyzer 37
3.4.4 The benefits from using huge translation look aside buffers with the nets 39
3.4.5 Tests on a “normal” CPU 39

3.4.6 Performance tools 40

4 Results **41**

4.1 Net 1 41

4.1.1 Timed results for Net 1 41

4.1.2 Systemsim statistics for Net 1 46

4.2 Net 2 49

4.2.1 Timed results of Net 2 49

4.2.2 Systemsim statistics for Net 2 51

4.3 Results from a conventional computer, and comparisons 52

5 Conclusions and discussion **55**

5.1 Future work 58

Bibliography **59**

List of Figures

2.1 The Cell Broadband Engine architecture. 4
2.2 The PowerPC Processor Element. 5
2.3 The Synergistic Processor Element. 6
2.4 Hypercolumns 11

3.1 Multiply and add performed on weight and activity. 15
3.2 How summation of the elements in a vector works. 16
3.3 Reorganization of the weight-matrix. 17
3.4 Reorganization of the activity. 17
3.5 An illustration of double-buffered memory-transfers. 18
3.6 Principal illustration of Net 2. 20
3.7 The final development of Net 2. 20
3.8 The weight-matrix as stored in memory. 21
3.9 Transformation of the weight-matrix. 22
3.10 Reorganizing the activity. 23
3.11 First implementation of Net 2. 27
3.12 Further development of Net 2. 27
3.13 Problems with Net 2. 28
3.14 Mailbox problems. 29
3.15 The final version of Net 2. 30
3.16 Scheme of the process in Net 2. 31
3.17 Synchronization using signals. 32
3.18 The assembly visualizer. 38
3.19 Daniel Hackenberg's matmul. 39

4.1 Results of Net 1, with a net-size of 32x32. 42
4.2 Results of Net 1, with a net-size of 52x52. 43
4.3 Results of Net 1, testing different solutions. 44
4.4 Results of Net 1, testing different solutions. 45
4.5 Results of Net 1, with smaller networks. 45
4.6 Results of Net 2, with different configurations. 49
4.7 Results of Net 2, larger net-size. 50
4.8 Results of Net 2, smaller net-size. 50
4.9 Results of normal CPU. 52
4.10 Results of normal CPU against Net 2. 53

Chapter 1

Introduction

The background to the work will be outlined, and the problem defined. The introduction concludes with a description of the report's composition.

1.1 Background

An ever increasing amount of microprocessors are developed with heavy, and specific computational loads in consideration. The graphical processor units in modern 3D-accelerator cards, and the CPUs in the XBOX 360 and Playstation 3 gaming consoles, are examples of specialized architectures where the heavy computational loads are offloaded.

With this background the Cell Broadband Engine Architecture makes its entrance. With nine processor units unified on a single CPU-chip, it is designed to be able to handle large amounts of calculations of general types. This stands in contrast to for example the GPUs, whose design is specialized to handle 3D-graphics calculations. Artificial neural networks are often parallel in nature, which enables network implementations to be designed for parallel architectures. The question addressed in this master's thesis is, how well the Cell Broadband Engine Architecture will be able to handle a large-scale network, where the high amounts of data-transfers to and from the CPU very well may become a significant bottle-neck.

Two different implementations of the Bayesian Confidence Propagation Neural Network (BCPNN) is made on the Cell, in order to test what kinds of bottle-necks are suffered with the BCPNN artificial neural network.

1.2 Problem definition

The purpose of this master thesis is to investigate how well the Cell processor handles a large-scale neural network. Or negatively put, to investigate how much of a bottle-neck the data transfers in a large-scale net will become. For this we will implement a spiking Bayesian Confidence Propagation Neural Network (BCPNN) with hypercolumns. This is a net that stores patterns in a manner which gives predictable results, even when over storing patterns (Sandberg, 2003). The net will also be using spiking neurons, since simulations with spiking neurons give more interesting results.

1.3 Report composition

Chapter one gives a theoretical introduction. An overview of the Cell Broadband Engine architecture is given, along with an overview of a software development kit (SDK) developed by IBM, whose tools were used in this work. An overview of the Bayesian Confidence Propagation Neural Network (BCPNN) will also be given here.

Chapter two will describe two different implementations of the BCPNN on the Cell. First the tools and hardware used is described. The two different versions for the testing is then described. The different optimizations performed will also be described, and the tools producing the final results are shown.

Chapter three shows and explains the results. Chapter four finally discusses the conclusions drawn from the results, and gives suggestions for future work on this problem.

Chapter 2

Theory

The theory begins with an introduction to the architecture of the Cell-processor, where the architecture is described, and the tools for programming for it is explained. The artificial neural network to be implemented is also explained in this section.

2.1 Introduction to the architecture of the Cell Broadband Engine

The Cell Broadband Engine (CBE) was developed by IBM, Toshiba and Sony, and the first wide commercial version of the CPU was released as the CPU of the gaming console Playstation 3. It is structured towards distributed processing, containing multiple functional components geared towards processing of large amounts of floating point numbers. Although the CBEA was designed for application in games, and other media-rich environments, a much broader use of the architecture was envisioned, and it was designed to enable fundamental advances in processor performance (IBM, 2007g).

The CBEA is a single chip multiprocessor. The architecture is structured into one “main” processor, and eight “secondary” ones. The “main” processor, called the PowerPC Processor Element, interacts with the main memory. The “secondary” processors, called the synergistic processor elements, does not interact directly with the main memory, but instead works with own memories, called local stores.

The main purpose of the PPE is that of a controller of sorts, while the heavy computational burden lies with the SPEs, of which there are eight. Each SPE have a dedicated local memory storage, and a dedicated MFC, which handles the memory-communication between SPEs and the PPE, and the main memory storage.

Figure 2.1 illustrates the typical CBE architecture. Architectures vary slightly. For example, while the Playstation 3 has eight SPEs available, under Linux one is reserved as a backup SPE, and the last reserved for the duty of a hyper-visor. Toshiba has as of date not yet released any commercial products with the CBE, but sample shipping of a Full HD streaming processor called SpursEngine, which is a derived from the CBEA, began in April 2008. IBM has released two Blade-center servers Q20 and Q21, which use an architecture with two Cell processors. IBM is scheduled to release a further developed version of the Cell SPU in June 2008, with the IBM Blade-center Q22, which will have hardware support for double precision floating point operations, and support for more memory (32 GB, compared to 2 GB that the previous version Q21 supported). The support for faster double-operations is significant, since the Cell CPUs to date has had poor double-point performance by design.

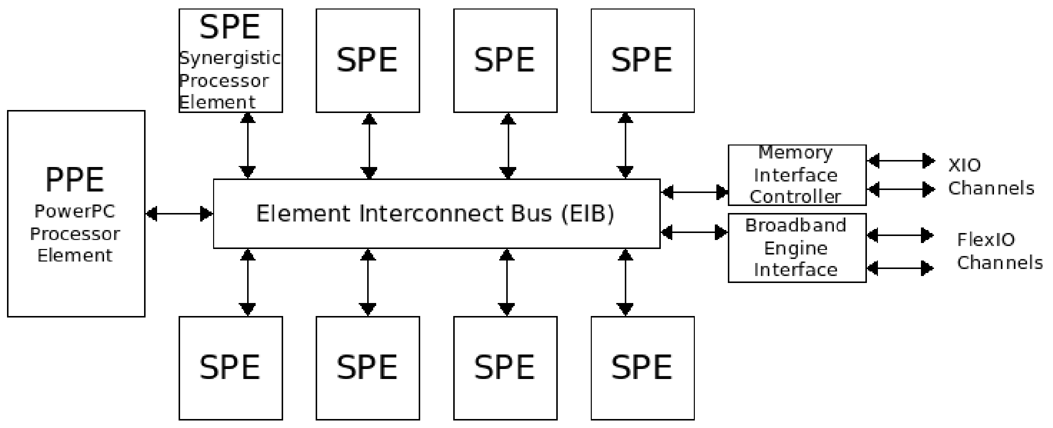

SPE
Synergistic Processor Element
SPE
SPE
SPE
PPE
PowerPC Processor Element
Element Interconnect Bus (EIB)
Memory Interface Controller
Broadband Engine Interface
XIO Channels
FlexIO Channels
SPE
SPE
SPE
SPE

Figure 2.1. An illustration of a typical Cell Broadband Engine Architecture.

An explanation of terminology might be appropriate here. PowerPC Processor Element (PPE) refers to the element of the CBE architecture that performs the defined architectural duty. The PowerPC processor unit (PPU) refers to the actual processor of the element. In the same way the SPE refers to the whole unit, including the actual processor (synergistic processor unit, SPU), local storage, and memory flow controller (MFC).

2.1.1 The PPE

Figure 2.2 shows the PPE and its parts. It has a 64-bit PowerPC processor unit (PPU). The PPU conforms to the typical PowerPC architecture, with associated caches etc. It has single instruction multiple data (SIMD)-functionality, but its purpose is of a more general nature - management and allocation of tasks for the SPEs. (IBM, 2007g)

The PPU deals with instruction control and execution, it includes:

- the full set of 64-bit PowerPC registers,
- 32 128-bit vector registers,
- a 32-KB level 1 instruction cache,
- a 32-KB level 1 data cache,
- an instruction-control unit,
- a load and store unit,
- a fixed-point integer unit,
- a floating-point unit,
- a vector unit,
- a branch unit,
- a virtual-memory management unit.

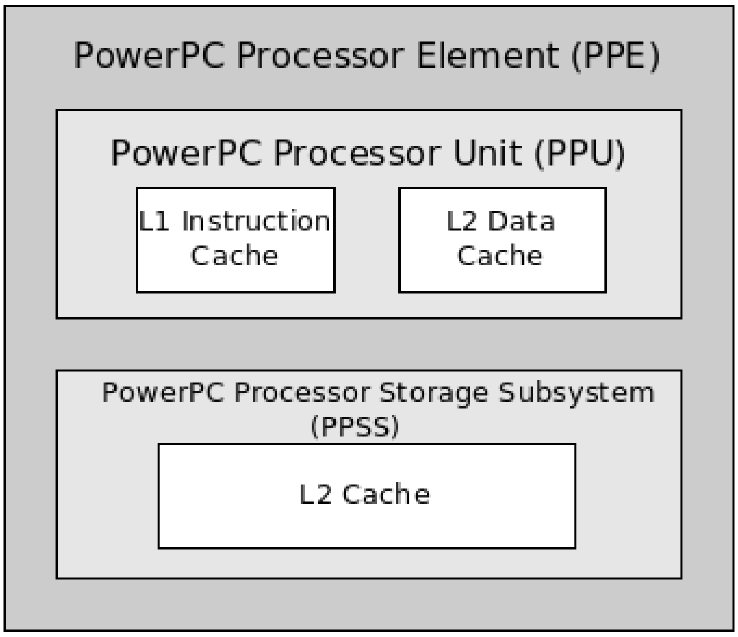

PowerPC Processor Element (PPE)
PowerPC Processor Unit (PPU)
L1 Instruction Cache
L2 Data Cache
PowerPC Processor Storage Subsystem (PPSS)
L2 Cache

Figure 2.2. The PPE and its parts.

It supports two simultaneous threads of execution, and can be viewed as a 2-way multi-processor with shared data-flow.

Part of the PPE is also the PowerPC Processor Storage Subsystem (PPSS). It handles memory requests from the PPE and external requests to the PPE from other processors or I/O devices. It includes:

- a unified 512-KB level 2 instruction and data cache,
- various queues,
- a bus interface unit that handles bus arbitration and pacing on the EIB.

2.1.2 The SPU

The SPU is less complex than the PPU. Its typical purpose is to process data and do the required data transfers. It has SIMD-functionality, with its own instruction set. The SPU is specially geared towards computing with single precision floating point numbers, and can achieve a potential 25.6 Gflop performance. A defining attribute of the SPE is how it accesses memory. Instead of having direct access to the main storage, it has a local storage of 256 kbyte. Access to the main storage is handled with DMA commands via the MFC. The reason for this change from common CPUs is to minimize memory latency. CPU-cycles going to waste because of memory latency has increased hundredfolds these last 20 years, which has made memory latency into a main bottleneck in performance. In addition to the local storage, the SPU has a large register file, 128 128-bit registers are available.

The SPU's architecture is simplified in other ways too. It does not have a cache, instead relying on the programmer to make efficient use of the local storage. This does away with the

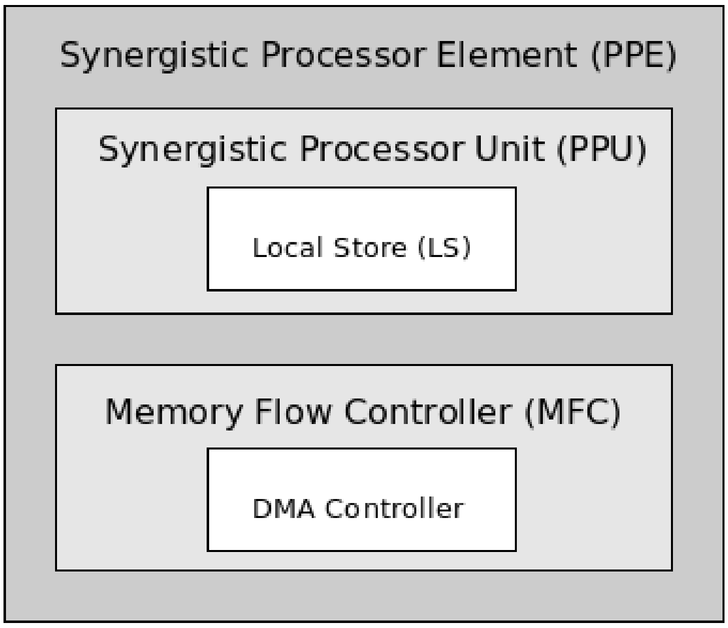

Synergistic Processor Element (PPE)
Synergistic Processor Unit (PPU)
Local Store (LS)
Memory Flow Controller (MFC)
DMA Controller

Figure 2.3. The SPE and its parts.

problem of cache-misses. Furthermore, it lacks branch-prediction, and other typical features common in CPUs nowadays. Since the SPU is simplified and focused, one has to be mindful of its differences compared to conventional CPUs in order to maximize performance. For example, because of the lack of branch-prediction, reaching the full potential of the Cell will be difficult if heavy branching is used.

2.1.3 The memory flow controller

As shown in figure 2.3, the MFC is also part of the SPE. The MFC handles the data transfers in the CPU. This encompasses transfers between SPE's local stores, and main memory, and transfers between local stores themselves. The MFC is designed to handle these transfers with speed and security, to maximize the overall computing performance of the processor. The MFC can handle multiple transfer commands (called MFC DMA commands) at the same time.

Each DMA transfer can be up to 16 KB in size. However, through DMA-list commands, up to 2048 DMA transfers can be issued with one command.

2.1.4 The reasoning behind the design of the CBEA

In theory, the CBEA is designed to be able to overcome three big limiting factors in contemporary microprocessors: power use, memory use, and processor frequency.

The dissipation of power use, and cooling of the CPU, has become an increasingly limiting factor as processor performance has gone up. To be able to increase processor performance, one must also increase the power efficiency. The CBEA differentiates between processors optimized to handle computationally intensive code, and processors optimized to handle control-intensive code. This increases the power efficiency with regards to performance per watt, and lowers the power-consumption overall. Compared to conventional CPUs, the power efficiency of the CBEA is significantly higher.

The CBEA deals with the issue of memory as a limiting factor in two ways:

- The SPEs have a 3-level memory structure, main storage, local stores, and a large register file.

- Asynchronous DMA transfers between local stores and main storage.

The programmer is able to send 128 simultaneous transfers between the eight SPEs and main storage, allowing for scheduling of the data-transfers and minimizing of latency because of memory access.

The third limiting factor lies in operating frequency. Conventional processors achieve high operating frequency by having a deep instruction pipeline. However, diminishing returns from this technique has reached its peak. If one takes power use into consideration, it has even reached negative returns. The separation of control- and computationally intensive tasks into the PPE and SPE respectively, allows both the PPE and the SPE to to be designed for high frequency, without excessive overhead. Instead of optimizing single-thread execution, the PPE achieves efficiency by executing two threads simultaneously. The SPE achieves high efficiency by its large register file, which supports many simultaneous in-process instructions. The use of *asynchronous* DMA transfers may increase the SPE's efficiency by doing away with the overhead of speculation.

2.2 Programming the CBEA

The instruction set for the PPE is an extended version of the PowerPC instruction set. The SPE carries a similar instruction set, but the sets are different, and must be compiled by different compilers. Pervasive through both the PPE and the SPEs is SIMD vector-instructions. To able to use the full potential of the CBEA, the data-level parallelism of SIMD operations should be used.

A software development kit (SDK), developed by IBM, is available for the CBEA. It includes useful, or even essential tools for developing code for the CBEA. Part of this is a set of C-language extensions called intrinsics. These intrinsics substitute for one or more in-line assemble-language instructions for the PPU and SPU respectively. There are a wide variety of these intrinsics, reaching from specific instructions, substituting for one or few assembly commands, to more composit intrinsics that handle more complex tasks.

The SDK also includes:

- The IBM Full System Simulator for the Cell Broadband Engine.

- System root image containing Linux execution environment for use within the system simulator.

- GNU tools including C and C++ compilers, linkers, assemblers and binary utilities for both PPU and SPU.

- IBM xlc (C and C++) compilers for both PPU and SPU.

- IBM xlf (FORTRAN) compiler for both PPU and SPU.

- newlib for the SPU, which is a C standard library designed for use on embedded systems.

- gdb debuggers for both PPU and SPU with support for remote gdbserver debugging. The PPU debugger also provides combined, PPU and SPU, debugging.
- PPC64 Linux with CBE enhancements.
- SPE Run-time Management Library providing a standardized, low-level application programming interface for application access to the SPEs.
- Libraries to assist in the development and execution of parallel applications, including the:
    - Accelerated Library Framework (ALF).
    - Data Communication and Synchronization (DaCS) library.
- Performance tools, including:
    - oprofile – a system-wide profiler for Linux,
    - CellPerfCount – a low level tool to configure and access hardware performance counters.
    - FDR-Pro – a tool that gathers information, and performs feedback directed optimization,
    - CodeAnalyzer – examines executable files and displays detailed information about functions, basic blocks, and assembly instructions, and
    - Visual Performance Analyzer (VPA) – an Eclipse-based performance visualization toolkit.
    - `spu_timing` – a timing analysis tool that instruments assembly source (either compiler or programmer generated) with expected, linear, instruction timing details.
    - PDT – a performance debugging tool which provides a tracing infrastructure for application timing analysis.
- An Eclipse-based Integrated Development Environment (IDE) to improve programmer productivity and integration of development tools.
- Standardized SIMD math libraries for the PPU's Vector/SIMD Multimedia Extension and the SPU.
- Mathematical Acceleration Subsystem (MASS) libraries supporting both long and short (SIMD) vectors.
- Cell optimized domain-specific application libraries, including Basic Linear Algebra Subprograms (BLAS) library, Fast Fourier Transform (FFT) library, and Monte Carlo Random Number Generator library.
- Example source code containing programming examples, example libraries, benchmarks, and demos.

2.3 Spiking Bayesian Confidence Propagation Neural Network (BCPNN) with hypercolumns

A short introduction to auto-associative artificial neural networks will first be given. The particular artificial neural network that has been implemented in this work, the BCPNN, will then be described.

2.3.1 Auto-associative artificial neural networks

An artificial neural network is a mathematical model of biological neural networks. It consists of a number of interconnected artificial neurons, whose relations in their connections are used for information processing. This can be used for many different things. A fairly intuitive example is that of an associative network.

In a recurrent network, the neurons are recurrently connected. This stands as a contrast to the feed-forward type of net, where the neurons are connected in a more step-wise manner, feeding forward the information to following layers of neurons. The feed-forward network's first layer of neurons are activated with some input, and they fire forward their output to the next layer, or to the designated output of the network.

In the recurrent network on the other hand, the neurons "simply" connect to each other. For example, each neuron could correspond to a pixel in a picture, and different values of input could correspond to color. By first training the network to remember a set of pictures (more on training later), they would then be able to recall trained pictures from partly distorted versions of the trained pictures. This is exactly what the associative network does by definition, it associates input with data it was been trained with. The output is called a fix-point. Ideally, a fix-point would always be a valid picture it has been trained with.

A problem with associative networks is that they often suffer from catastrophic forgetting, and that fix-points sometimes stabilize on non-trained pictures (patterns). Catastrophic forgetting means that a network can store patterns in its memory up to a point, beyond which everything suddenly breaks down, and it becomes unable to remember anything. For example, network A can remember 100 patterns, if it is trained with 100 patterns it may recall them all without problem (there are other problems that can manifest themselves, see below). If, however, it is trained with 101 patterns, it can only recall 50 of them, and if trained with 102 patterns, none are recalled correctly, not even if the net is given the undistorted patterns it was trained with.

The problem of catastrophic forgetting is a variant of the stability-plasticity dilemma, the general problem in learning systems, which ideally are sensitive both to new input, but not disrupted by it. The issue of faulty fix-points is another problem. You would think that a network only stabilizes (recalls) on patterns it has learned, but that is not the case. Simple associative networks such as the naive implementation of a hopfield network also stabilize on the inverse of trained patterns, and on patterns consisting of portions of different patterns. (for example, half of one stored picture, and half of another).

2.3.2 The BCPNN

The BCPNN is a neural network architecture and learning rule derived from Bayes' rule. With a Hebbian learning rule, it reinforces connections between simultaneously active units, and weakens anti-correlated units. Applied to a recurrent attractor network, it generates a symmetric weight-matrix, with fix-point attractor dynamics (Johansson and Lansner, 2006).

In this work we organized the structure of the net into hypercolumns, each consisting of equal numbers of units. The pattern-information was coded by having each hypercolumn represent an integer value. A single active unit in a hypercolumn would represent a value in a pattern. The number of values in a pattern would correspond to the number of hypercolumns. See figure 2.4 for a visualization. In this work a number of different network configurations, with regard to the number of hypercolumns, and units per hypercolumn, have been simulated.

Returning to the functionality of the BCPNN. The network operates by initializing the activity vector with a pattern. It then, through a so called relaxation-process, updates the activity until stability (a fix-point) has been reached. The relaxation has two steps: First the potential m is updated (2.2) with the current support (4.1). Then a soft-max function is applied to compute the new activity from the potential (2.3).

$$s_j = \log(\beta_j) + \sum_{h=1}^{H} \log(\sum_{k \in Q_h} w_{kj} o_k) \tag{2.1}$$

$$\tau_m \frac{dm_j}{dt} = s_j - m_j \tag{2.2}$$

$$o_j \leftarrow \frac{e^{Gm_j}}{\sum k \in Q_h e^{Gm_j}} : j \in Q_h \ for\ each\ h = 1, \ldots, H \tag{2.3}$$

Taking the values of the computed activity into consideration, a random spike is then generated, setting the winning unit to one, while the rest are set to zero. The variable “G” above determines how steeply the soft-max function will normalize. High values on G exaggerates the values of the probabilities in a degree which makes it easier for a single winner to get set. Too high values on G effectively makes the net non-random, while a too low value on G evens out the probability, effectively making the activity completely random, never stabilizing. Different values for G and τ was used thoughout the work. G is especially relevant in order for the network to recall the patterns correctly, since the function of the soft-max-process affects the results of the random spiking to a high degree (Johansson and Lansner, 2001).

To initialize the weights, probability estimates are first computed: Post-synaptic units are indexed with i, and pre-synaptic with j.

$$p_i = \frac{1}{P} \sum_{\mu=1}^{P} \xi_i^{\mu} \tag{2.4}$$

$$p_{ij} = \frac{1}{P} \sum_{\mu=1}^{P} \xi_i^{\mu} \xi_j^{\mu} \tag{2.5}$$

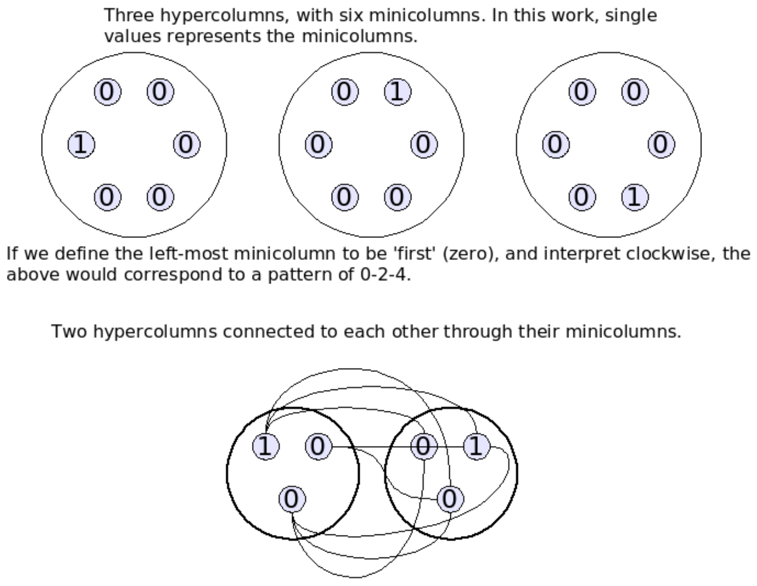

Three hypercolumns, with six minicolumns. In this work, single values represents the minicolumns.
0 0 1 0 0 0
0 1 0 0 0 0
0 0 0 0 0 1
If we define the left-most minicolumn to be 'first' (zero), and interpret clockwise, the above would correspond to a pattern of 0-2-4.
Two hypercolumns connected to each other through their minicolumns.
1 0 0
0 1 0

Figure 2.4. An example of artificial neurons organized into hypercolumns.

In this work, ξ is a unary-coded pattern, P is the number of patterns, and μ is the index of a pattern. The weights and biases are then computed as:

$$w_{ij} = \begin{cases} 1 & \text{if } p_i = 0 \lor p_j = 0 \\ \frac{1}{P} & \text{else if } p_{ij} = 0 \\ \frac{p_{ij}}{p_i p_j} & \text{otherwise} \end{cases} \tag{2.6}$$

$$\beta_i = \begin{cases} \frac{1}{P^2} & \text{if } p_i = 0 \\ p_i & \text{otherwise} \end{cases} \tag{2.7}$$

Chapter 3

Implementation

The two different implementations of the BCPNN will be explained here. First the tools of development will be described. Then the the development begins with an initial test-version of the BCPNN. The two different final versions are then described, and issues of performance-testing are explained.

3.1 Development-tools

Development tools both reragrind of hardware and software are described in this section.

3.1.1 Hardware

The only Hardware required for this work was a Cell processor, which was procured by CBN in the form of a Playstation 3 console. Remote access to it was set up, in order to work with it more conveniently. Later on in the project a Blade-server model Q21 was made available, for further testing with the dual Cell processor which it includes. A laptop was used to run the systemsim, and an Intel Pentium 4 2.60GHz desktop computer was used for comparisons of the implementations with the Cell.

3.1.2 Software

To install a Linux operating system on the Playstation 3 is relatively easy. From the default operating system the hard drive was first formatted into two partitions, one for the gaming system, and one other for the Linux operating system. A so called live-disc can then be used at boot, to launch the Linux operating system. The ubuntu-distribution of Linux was installed, and the console was set up to a network, which made it easy to access and work with the Cell processor.

To be able to actually program and run code on the Playstation 3, the Software Development Kit was then installed. The SDK is distributed by IBM in RPM-files, which first had to be converted into debian packages with the “alien”-command. Since the Ubuntu-installation was configured for a normal PowerPC architecture, some of the ppc64-dependent tools had to be installed using force-architecture options with dpkg.

A matrix multiplication-program, which achieves 99 percent of peak-performance from the SPUs was successfully tested (Hackenberg, 2007). A test-program with simple, trivial, calculations from an SPU, without using assembly-optimizations was also successfully implemented. It achieved 22 Gflops.

3.1.3 Lessons learned from other CBEA-applications

A premier example of good performance on the CBEA is Hackenberg's matrix multiplication (Hackenberg, 2007) It achieves 99.43% performance on the SPUs. It accomplishes this by multibuffered data-transfers and partitioning of the matrices into smaller matrices of 64x64 size, and a well optimized matrix-multiplier, written in assembly, that handles the partitioned matrices.

3.2 Test-implementation of a non-spiking BCPNN-net

The work began with a test-implementation of a non-spiking BCPNN-net. The work of implementing the test-version is described in this section.

3.2.1 Initial scalar version

A test-implementation of a non-spiking BCPNN was made. Given the formulas for the BCPNN-net, this was made in a simple, straightforward manner, without utilizing SIMD-functionality. Since the PPE is designed to be able to handle any general PowerPC-compatible code, special considerations did not have to be made in the PPU-part of the code. The patterns, biases, activity, and potential were all represented as arrays of floats. Single-precision floats would be used throughout the project, since the Cell performs poorly with double-precision in comparison to single-precision.

```
#define NODES_IN_COLUMN 10
#define COLUMNS 10
#define TOTAL_NODES COLUMNS * NODES_IN_COLUMN
#define NUMBER_OF_PATTERNS 200

float trainingPatterns[NUMBER_OF_PATTERNS][TOTAL_NODES];

...
```

The summations in the formulas were implemented using basic for-loops, for example:

$$s_j = \log(\beta_j) + \sum_{h=1}^{H} \log(\sum_{k \in Q_h} w_{kj} o_k)$$

was implemented as:

```
for (i=0; i<TOTAL_NODES; i++) {
    potentialRHS = 0;
    for (j=0; j<COLUMNS; j++) {
        summation = 0;
        for (k=j*NODES_IN_COLUMN; k<j*NODES_IN_COLUMN+NODES_IN_COLUMN; k++) {
            potentialRHS += log10f(summation);
        }
    potential[i] = log10f(biases[i]) + potentialRHS;
}
```

Given this scalar implementation of a non-spiking BCPNN-net, the task then came to:

1. Transforming the scalar code to vectorized, in order to SIMDize the code (see section 3.2.2). SIMD-functions are available on the PPU though the so called altivec-functions.
2. Porting the code for execution on the SPE.
3. Parallelizing the code for execution on multiple SPEs.

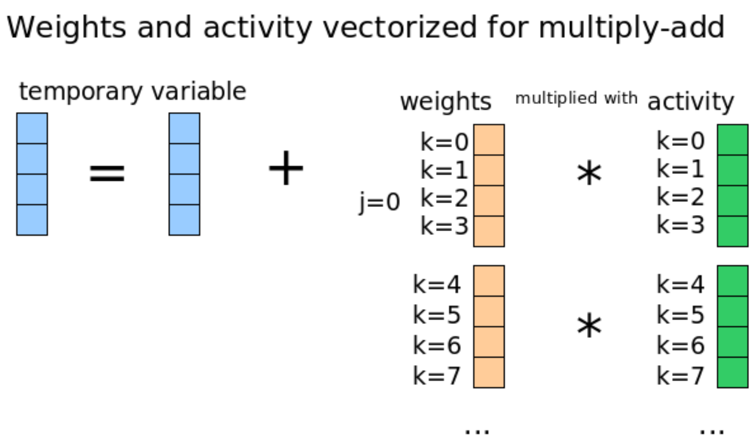


Figure 3.1. Multiply and add is performed on weights and activity, and the result is stored in a temporary variable.

3.2.2 SIMDizing the scalar version

The task of SIMDizing the scalar version was undertaken first. This meant that scalar values would be transformed into vector values, on which multiple operations could then be performed with a single instruction. The SIMDification could take different forms. Looking at the function:

$$s_j = \log(\beta_j) + \sum_{h=1}^{H} \log(\sum_{k \in Q_h} w_{kj} o_k)$$

the inner summation consist of a multiplication. Use of the Cell's multiply and add-function would be ideal. The multiply and add performs both a multiply, and addition, in 6 cycles. Performing an addition or a multiplication by itself also takes 6 cycles, which means that one should use the multiply-add where possible. The weights and activity was vectorized for use of the multiply-add. See figure 3.1.

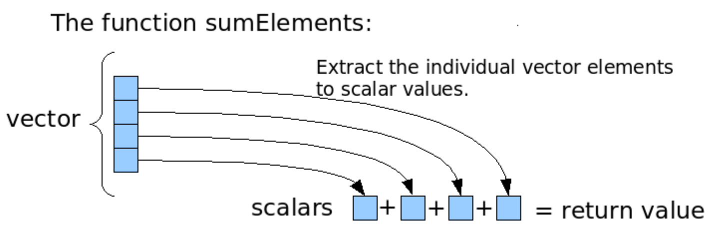

The function sumElements:
Extract the individual vector elements to scalar values.
vector
scalars
+
+
+
= return value

Figure 3.2. Adding the elements in a vector with each other means extracting the values, and adding them using regular scalar operations.

This transformed the previous code snippet into:

```
for (i=0; i<totalVectors; i++) {
    potentialRHS = spu_splats(0.0f);

    for (j=0; j<columns; j++) {

        summation[0] = spu_splats(0.0f);
        summation[1] = spu_splats(0.0f);
        summation[2] = spu_splats(0.0f);
        summation[3] = spu_splats(0.0f);

        for (k=j*vectorsInColumn; k<j*vectorsInColumn+vectorsInColumn; k++) {
            summation[0] = spu_madd(weights[0][k], activity[k], summation[0]);
            summation[1] = spu_madd(weights[1][k], activity[k], summation[1]);
            summation[2] = spu_madd(weights[2][k], activity[k], summation[2]);
            summation[3] = spu_madd(weights[3][k], activity[k], summation[3]);
        }
        vector float tempQ = {sumElements(summation[0]),
                              sumElements(summation[1]),
                              sumElements(summation[2]),
                              sumElements(summation[3])};
        potentialRHS = spu_add(log10f4(tempQ), potentialRHS);
    }
    potential[i] = spu_add(theBias[i], potentialRHS);
}
```

The number of times the outer loop would run would now be the total number of vectors, instead of nodes. The total number of vectors is 1/4 of the number of nodes. Assuming that the inner workings could be efficiently implemented, the code should be four times as fast as the scalar version. However, as you can see, the inner loop still had some scalar operations in the form of the function sumElements, which sums the elements in a vector as seen in figure 3.2. The SPU has limited hardware support for scalar operations. Scalar operations entail extra loading and unloading of values from vectors, which means that extra work is performed, in addition to the utilization of SIMD-functionality being lost.

To solve the problem of scalar operations, the way the values were laid out in memory was changed. Instead of storing the matrix in the memory as k by j, it was stored as j by k. The vector types were still distributed along the j-“axis”, as shown in figure 3.3.

This change also made it possible for the elements in the weight-matrix to be sequentially accessed in the loop, as the values were organized in order of how they were accessed in the algorithm. This is further explained in section 3.3.1.

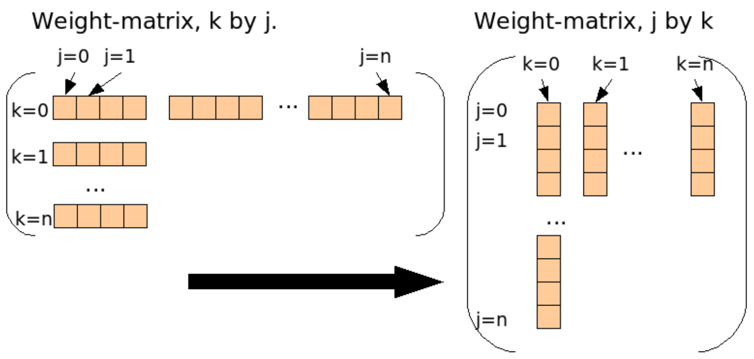


Figure 3.3. In order to minimize scalar operations in the main loop of the program, the weight-matrix was reorganized.

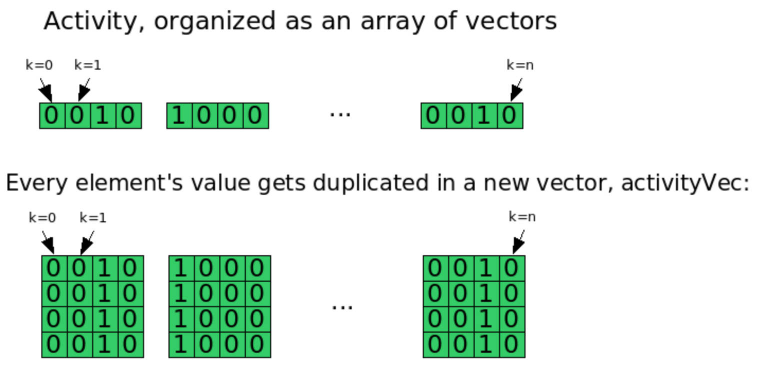

Activity, organized as an array of vectors
k=0
k=1
k=n
0 0 1 0
1 0 0 0
0 0 1 0
Every element's value gets duplicated in a new vector, activityVec:
k=0
k=1
k=n

Figure 3.4. The reorganization of the weight-matrix forced a reorganization of the activity as well.

The new code looked like:

```
for (i=0; i<totalVectors; i++) {
    potentialRHS = spu_splats(0.0f);

    for (j=0; j<columns; j++) {
        summation = spu_splats(0.0f);

        for (k=j*nodesInColumn; k<j*nodesInColumn+nodesInColumn; k++) {
            summation = spu_madd(weights[k], activityVec[k], summation);
        }
        potentialRHS = spu_add(log10f4(summation), potentialRHS);
    }
    potential[i] = spu_add(theBias[i], potentialRHS);
}
```

For this to work the activity-array had to be transformed so that the vectors in the weight-matrix was multiplied with the relevant activity. See figure 3.10.

This was to become a significant drawback later on, but at the moment it resulted in a faster program. Note that the above code is the ported SPU-version. The common PPE-version does not differ significantly. Only the prefix "spu_" is changed from "vec_", which is the corresponding altivec-version of the SPU's vector-functions. More about porting code to the SPU follows.

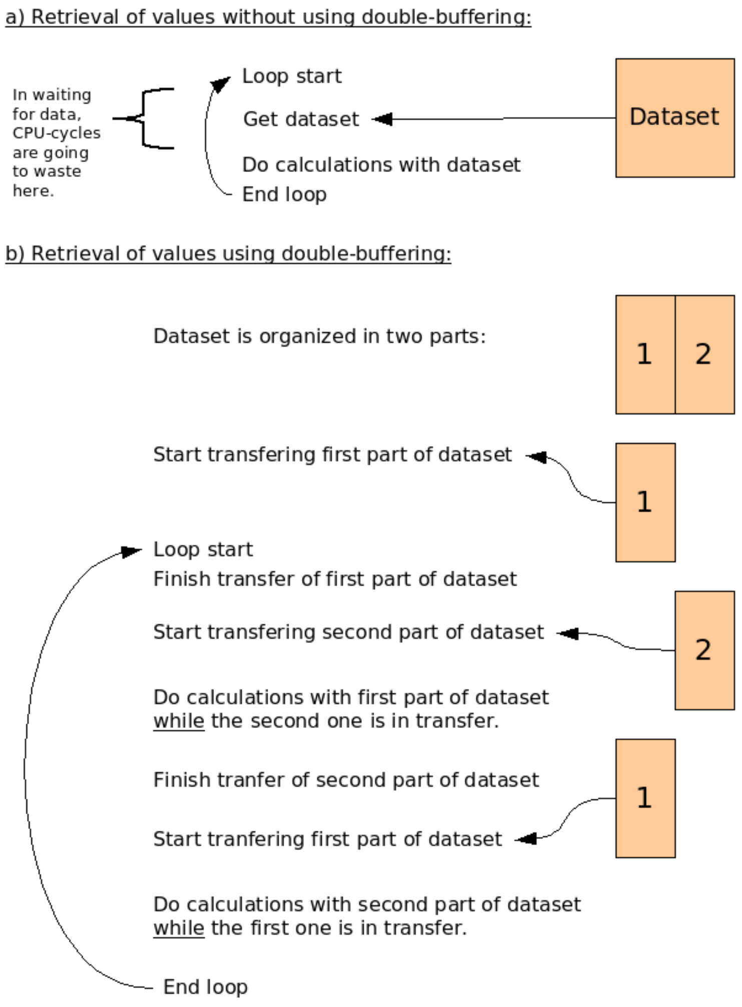

a) Retrieval of values without using double-buffering:
In waiting for data, CPU-cycles are going to waste here.
Loop start
Get dataset
Do calculations with dataset
End loop
Dataset
b) Retrieval of values using double-buffering:
Dataset is organized in two parts:
1
2
Start transfering first part of dataset
Loop start
Finish transfer of first part of dataset
Start transfering second part of dataset
Do calculations with first part of dataset while the second one is in transfer.
Finish tranfer of second part of dataset
Start tranfering first part of dataset
Do calculations with second part of dataset while the first one is in transfer.
End loop

Figure 3.5. An example-scheme of double-buffered memory-transfers, compared to non-buffered transfers, is shown here. Alternative a) does not use double-buffring, and every time data needs to be transfered, the CPU is stalled in waiting for data. The alternative b) is doing computations on half the dataset, while the other is being transfered.

3.2.3 DMA-transfer considerations

In porting PPU-code to the SPU, one has to take DMA-transfers into consideration. Since the SPU only has its local store of 256kB available, efficient use of data-transfers becomes a fundamental part of the program. For a BCPNN-network trained with non-sparse patterns, the size of the weight-matrix' increases as the square of the network-size, only for very small networks is it possible to store all relevant data in the local storage.

In the test-implementation of a non-spiking BCPNN, the decision was to keep the weight-matrix in the main store, and to stream it through double-buffered DMA-transfers. Double-buffered memory-transfers is shown in figure 3.5.

3.2.4 Distributing the workload to multiple SPEs

The distribution of workload among the SPEs was made in a simple manner: the test-patterns were equally distributed to the SPEs, which in turn performed all calculations independent of each other. For example: SPE 1 would test patterns 1 to 10, SPE 2 would test patterns 11-20, and so on. With this scheme the SPEs did not have to be synchronized with each other, since the datasets were independent of each other. The PPE would initiate SPE-threads using POSIX threads (pthreads), and the SPEs would get all the information they needed, network size, location of data sets in main memory, etc.

During this work, a new version of the SDK was released (3.0), and the SDK was upgraded from 2.1 to 3.0. The 3.0 version was used for the rest of the project.

3.3 Description of two different implementations of spiking BCPNN

In order to test different aspects of the CBEA, two different implementations of the BCPNN would be made. One would test the memory interface with the main storage, and the other would test memory interface concerning the ring which connects the SPEs with each other. The difference would lie in how the weight-matrix was stored. The first version, Net 1, would store the weight-matrix in the main memory. The other, Net 2, would store the weight-matrix distributed in the SPEs' local stores.

The tests were also to be made with "large-scale", networks consisting of numbers of units $\gg 1000$.

In the test-implementation of a BCPNN-net, the weight-matrix had been stored in the main memory, which meant that the method of transfering data in Net 1 would be similar to the data-transfers in the test-implementation. The network in Net 1 would be larger than in the test-implementation though, which would force a larger amount of DMA-transfers per test-pattern.

For the second version, with the weights in the local stores, it had to be decided how calculations would be organized with regards to data-transfers. In the first version, the DMA-transfers handled transfers of the weights. It was decided that the data-transfers of Net 2 would be of the nodes themselves.

This meant that the implementation of the algorithm would be re-thought. The weights would be stored in the local stores, forcing a much smaller net than in Net 1 above. We felt, however, that this approach to the problem was sufficiently interesting to warrant the resulting smaller network. The nodes would then be cycled around the SPEs, and be updated in a streaming manner, going from SPE to SPE, synchronized with each other. Figure 3.6 illustrates the idea.

In order to make this efficient, partly processed patterns would be passed around the ring of SPEs, as seen in figure 3.7.

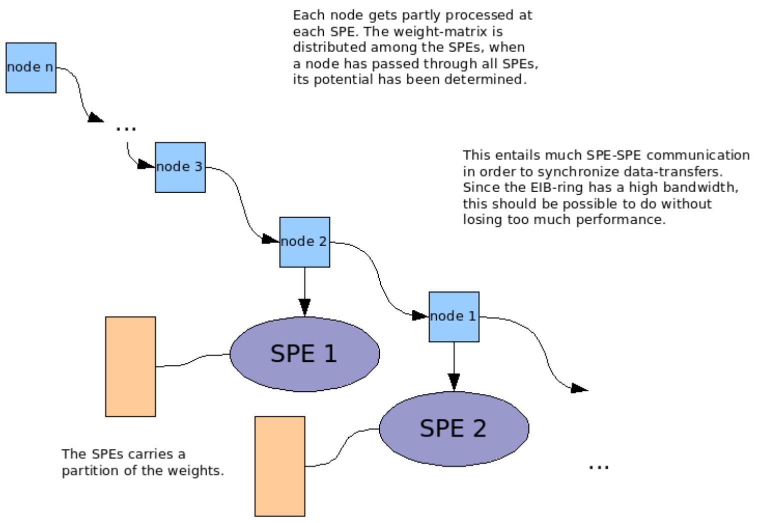

Each node gets partly processed at each SPE. The weight-matrix is distributed among the SPEs, when a node has passed through all SPEs, its potential has been determined.
node n
...
node 3
node 2
node 1
This entails much SPE-SPE communication in order to synchronize data-transfers. Since the EIB-ring has a high bandwidth, this should be possible to do without losing too much performance.
SPE 1
SPE 2
...
The SPEs carries a partition of the weights.

Figure 3.6. The idea of Net 2, where the SPEs store the weight-matrix in their local store, is shown here.

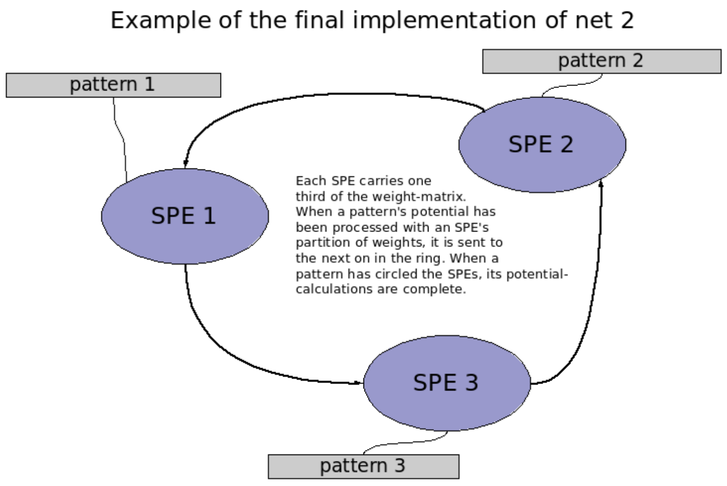

Example of the final implementation of net 2
pattern 1
pattern 2
SPE 2
SPE 1
Each SPE carries one third of the weight-matrix. When a pattern's potential has been processed with an SPE's partition of weights, it is sent to the next on in the ring. When a pattern has circled the SPEs, its potential-calculations are complete.
SPE 3
pattern 3

Figure 3.7. Outline on how Net 2 would eventually work.

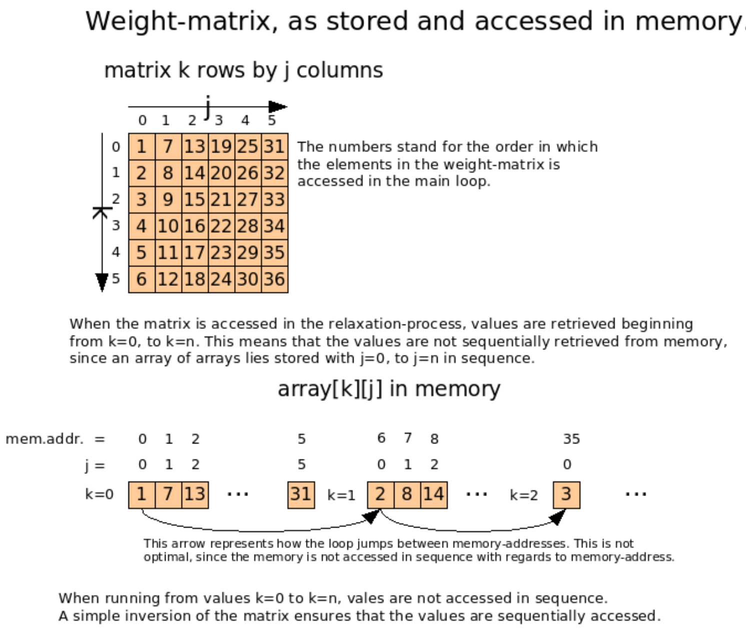

Weight-matrix, as stored and accessed in memory.
matrix k rows by j columns
j
0 1 2 3 4 5
1 7 13 19 25 31
2 8 14 20 26 32
3 9 15 21 27 33
4 10 16 22 28 34
5 11 17 23 29 35
6 12 18 24 30 36
The numbers stand for the order in which the elements in the weight-matrix is accessed in the main loop.
When the matrix is accessed in the relaxation-process, values are retrieved beginning from k=0, to k=n. This means that the values are not sequentially retrieved from memory, since an array of arrays lies stored with j=0, to j=n in sequence.
array[k][j] in memory
mem.addr. =
j =
k=0
k=1
k=2
This arrow represents how the loop jumps between memory-addresses. This is not optimal, since the memory is not accessed in sequence with regards to memory-address.
When running from values k=0 to k=n, vales are not accessed in sequence.
A simple inversion of the matrix ensures that the values are sequentially accessed.

Figure 3.8. As the weight-matrix first was stored in the main memory, it was not sequentially accessed by the main loop.

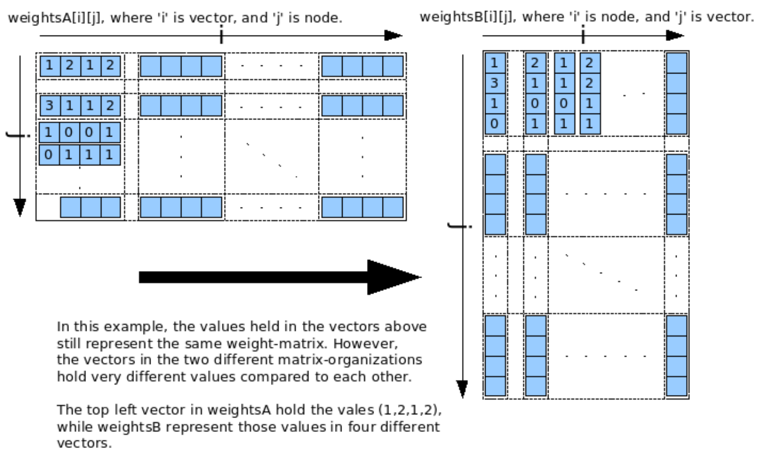
3.3.1 Implementation of Net 1

The first version would have the weight-matrix stored in the main memory. This was a further development of the non-spiking test-version. This meant that the relaxation-process was completely done in each SPE, independent of each other. The partitioning of the workload was made by dividing the number of test-patterns among the SPEs. If there, for example, were 60 test-patterns and 6 SPE-threads, each SPE would run ten patterns each. The weight-matrix would be streamed from the main storage memory, everything else, bias, patterns, and so on, would lie in the SPE's local storage. When updating the activity, the soft-max-function was modified to randomly spike according to the generated potential.

Reforming the weight-matrix

The weight-matrix was to be transformed, in order to be able to access it more efficiently. In the non-spiking implementation, concern with how the weight-matrix was stored in the memory had not been taken.

This meant that the weight-matrix was not sequentially accessed when considering memory addresses, see figure 3.8. To make memory access easier, and more efficient, the weight-matrix was transposed so that the matrix would be sequentially accessed in the relaxation. A result of this was that the data vector-types that the weight-matrix consisted of would hold different values, so further modifications in generating the weight-matrix had to be made, as seen in figure 3.9.


weightsA[i][j], where 'i' is vector, and 'j' is node.
weightsB[i][j], where 'i' is node, and 'j' is vector.
1 2 1 2
3 1 1 2
1 0 0 1
0 1 1 1
1 3 1 0
2 1 0 1
1 1 0 1
2 2 1 1

In this example, the values held in the vectors above still represent the same weight-matrix. However, the vectors in the two different matrix-organizations hold very different values compared to each other.

The top left vector in weightsA hold the vales (1,2,1,2), while weightsB represent those values in four different vectors.

Figure 3.9. The weight-matrix, before and after reorganization.

Issues of the larger data-sets

If data is not aligned on a 128-bit border, you will get bus error when trying to use the data. The function malloc does not automatically align data on a 128-bit border, which means that you will get bus errors when using the default malloc function. The function malloc_align included in the SDK (IBM, 2007e) can be used to align data. To accomodate the larger data-sets used in a larger network, malloc_align was used. The larger data-sets also required more DMA-transfers to be changed to DMA-list-transfers, the double-buffered transfers of the weight-matrix had to be made in lists.

Initial results and possible bottle-necks

The first results was far from optimal, and attempts to optimize were made. However, the optimized code did not give satisfactory results. There could be many reasons for this. To begin with, the assembly code was investigated for possible improvements. The code showed poor dual-issue rate, which meant that the dual-pipeline of the SPU was poorly utilized. To deal with the poor dual-issuing of instructions, the highly optimized matmul-algorithm developed Daniel Hackenberg was investigated to see how it dealt with it (Hackenberg, 2007). The matmul had a very optimized 64x64 matrix-multiplication entirely coded in assembly. The timing tool showed that the algorithm in a very clear and nice way issued floating point-calculations and load/store-instructions beside each other, so full dual-issue was achieved. However, since the relaxation-process included numeral summations and logarithms interwoven with the multiplications, an equivalent solution was not obvious. The matmul-program also worked on multiplication of similarly sized matrices, whereas the BCPNN-relaxation involved a calculation between one vastly bigger (proportionally) data-set with another smaller. This meant that the double-buffered data-traffic between main store and local stores would meet different challenges compared to the matmul-solution.

Final version and final optimizations

To use the program for a net-size of 64 hypercolumns & 64 minicolumns, the size of the local storage became a limiting factor. As mentioned earlier (figure 3.10), in order to efficiently run

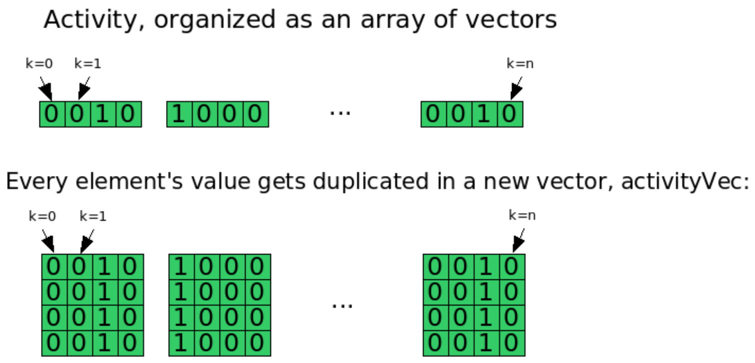

Activity, organized as an array of vectors
k=0
k=1
k=n
0 0 1 0
1 0 0 0
...
0 0 1 0
Every element's value gets duplicated in a new vector, activityVec:
k=0
k=1
k=n
...

Figure 3.10. The activity was also reorganized, in order to make it work with the new state of the weight-matrix.

through the main loop, the activity had to be transformed.

The size of this transformed activity became 65KB in the 64x64-version of the net, too large to fit in the local store (LS) among all the other data. Instead of pre-calculating the transformation, it was moved to be handled in the main loop.

```
    summations0 = spu_madd(weights[k+0], activityVec[k+0], summations0);
    summations1 = spu_madd(weights[k+1], activityVec[k+1], summations1);
    ...
```

To:

```
    summations0 = spu_madd(weights[k+0], spu_splats(spu_extract(activity[(k+0)/4],
                                                     (k+0)%4)),
                                                     summations0);
    summations1 = spu_madd(weights[k+1], spu_splats(spu_extract(activity[(k+1)/4],
                                                      (k+1)%4)),
                                                      summations1);
    ...
```

As expected, this hurt performance, lowering it to 12.7 Gflops. The same changes to a net-size of 32x32 gave the performance of 15.6 Gflops.

For every extra SPE in use, a performance-degradation was found. The final optimizations was made with only one SPE engaged for easier comparisons.

Enabling huge translation look-aside buffers (Kurzak and Buttari, 2007) raised the performance from 3.5 Gflops to 5.1.

Only the weight-matrix was allocated on the hugetlbf. The hugetlb-functions in cp_hugemem.h provided by cellperformance.com was used for allocation and deallocation.

```
#include "cp_hugemem.h"

    ...

  const size_t  hmem_size = 6 * 16 * 1024 * 1024;
  cp_hugemem    hmem;
  int was_hugemem_allocated = cp_hugemem_alloc( &hmem, hmem_size );
  if ( !was_hugemem_allocated )
    {
      fprintf(stderr,"Error:␣Could␣not␣allocate␣hugemem\n");
      return (-1);
    }
  else
    {
      printf("hugemem␣allocated\n");
    }

  vector float *weights = (vector float*)hmem.addr;

    ...

  cp_hugemem_free( &hmem );
```

A fully unrolled main loop increased the performance from 5.1 Gflops to 5.3 Gflops. The “for” command was still there in the code, and since the loop only was run through once, it was removed all-together. This, surprisingly, made the performance jump up to 6.4 Gflops. Furthermore, removal of the mainloop function gave 6.55 Gflops performance.

Before:

```
 for (i=0;  i<totalVectors; i+=linesOfWinDB) {

    ...

    for (j=0; j<linesOfWinDB; j++) {
      mainLoop(columns,
               nodesInColumn,
               &weights[linesOfWinDB*buffer*totalNodes+j*totalNodes],
               activityVec,
               &supportRHS);

      support = spu_add(theBias[i+j], supportRHS);

    ...
```

After:

```
 for (i=0;  i<totalVectors; i+=linesOfWinDB) {

    ...

  supportRHS = spu_splats(0.0f);
      for (h=0; h<columns; h++) {
        summations0 = spu_splats(0.0f);
          ...
        summations8 = spu_splats(0.0f);
        k=h*nodesInColumn;
        woff=linesOfWinDB*buffer*totalNodes+j*totalNodes+k;

        summations0 = spu_madd(weights[woff+0], activityVec[k+0], summations0);
        ...
        summations8 = spu_madd(weights[woff+31], activityVec[k+31], summations7);


        supportRHS = spu_add(_log10f4(summations0), supportRHS);

    ...
```

Compiling with the option -funroll-all-loops increased the performance to 6.66 Gflops. This was running a single SPE, and testing with a netsize of 32x32. Increasing the number of SPEs gave the following results.

Results of net 1 running a network of size 32x32. For figure, see figure 4.3.

1 SPE:	6.66 Gflops	total 6.66 Gflops of max 25 (26.6%)
2 SPEs:	6.65 Gflops	total 13.3 Gflops of max 50 (26.6%)
3 SPEs:	5.82 Gflops	total 17.5 Gflops of max 75 (23.3%)
4 SPEs:	4.58 Gflops	total 18.3 Gflops of max 100 (18.3%)
5 SPEs:	3.71 Gflops	total 18.6 Gflops of max 125 (14.7%)
6 SPEs:	3.10 Gflops	total 18.6 Gflops of max 150 (12.4%)

The problem of bandwidth to the main memory

Increasing the number of SPEs in the program severely lowered performance per SPE. Since the performance degradation was so great for increased number of SPEs, further investigation was called for.

To further investigate the causes of the sinking performance, measurement of the size of the data-flow between the main storage, and the SPEs, was added to the program. The results suggested an answer to the problem with the sinking performance.

Data-flow of net 1 running a network of size 52x52. For illustration, see figure 4.4.

1 SPE:	6.66 Gflops	9.15 GB/s data flow
2 SPE:	6.65 Gflops	18.3 GB/s data-flow
3 SPE:	5.81 Gflops	24.0 GB/s data-flow
4 SPE:	4.59 Gflops	25.2 GB/s data-flow
5 SPE:	3.73 Gflops	25.6 GB/s data-flow
6 SPE:	3.14 Gflops	25.9 GB/s data-flow

Since the bandwidth of the flow to the XDR memory is 25.6 GB/s, it seemed like the problem was solved. Earlier in the work, optimization attempts did not have big effect. These attempts had been made with 6 SPEs running at all time. These results explained the difficulties in optimization that had been met earlier.

Net 1 was tested with a network-size of 24 hypercolumns, 16 minicolumns, in order to compare results with net 2. The results of this test were:

Results of net 1 running a network of size 24x16

1 SPE	5.16 Gflops	total 5.16 Gflops of 25 (20.7%)	5.37 GB/s dataflow
2 SPE	5.15 Gflops	total 10.3 Gflops of 50 (20.6%)	10.7 GB/s dataflow
3 SPE	5.10 Gflops	total 15.3 Gflops of 75 (20.4%)	15.9 GB/s dataflow
4 SPE	5.02 Gflops	total 20.1 Gflops of 100 (20.1%)	20.9 GB/s dataflow
5 SPE	4.45 Gflops	total 22.3 Gflops of 125 (17.8%)	23.2 GB/s dataflow
6 SPE	3.83 Gflops	total 23.0 Gflops of 150 (15.3%)	23.9GB/s dataflow

3.3.2 Implementation of Net 2

In Net 2 the weight-matrix was to be split among the SPEs, and the data cycled around them in the computation. This would imply a high degree of SPE to SPE-communication to be present. SPE-SPE communication can happen in two ways – first data can be directly written or read from/to another local storage. Second, short data-messages (single values) called mails, or signals, can be sent using the channel registers. Mailboxes or signals are used for synchronization of the SPEs, to ensure control and safety of the regular data-transfers.

Initial test-version of Net 2

The purpose of the first version of Net 2 was to test the mailboxes in a simple manner. The only calculations delegated to the SPEs in the relaxation was the summation in the main loop.

$$\sum_{h=1}^{H} \log(\sum_{k \in Q_h} w_{kj} o_k)$$

which in code looked like:

```
for (i=0; i<totalVectors; i++) {
    potentialRHS = spu_splats(0.0f);

    for (j=0; j<columns; j++) {
        summation = spu_splats(0.0f);

        for (k=j*nodesInColumn; k<j*nodesInColumn+nodesInColumn; k++) {
            summation = spu_madd(weights[k], activityVec[k], summation);
        }
        potentialRHS = spu_add(log10f4(summation), potentialRHS);
    }
    potential[i] = spu_add(theBias[i], potentialRHS);
}
```

Using mailboxes for communication, every SPE was to wait for its value, calculate, and send to the next SPE. The last SPE would then send the final sum to the PPE. Calculations were not made in parallel, since every SPE would simply wait, and only one SPE at a time would actually do calculations. Figure 3.11 shows an example of this.

A couple of issues came up during the development of this version of the code. The program always froze after a certain number of iterations. It was found out that the problem was that many threads was created without being destroyed, which froze the SPU, and forced a reboot. Another problem was with addressing. When sending and receiving addresses using the default int-types built in in C, the addresses becomes erroneuos when casting. The uint32-types were used in order to ensure that addresses held correct values at all times.

Enabling parallel calculations in Net 2

When mailboxes had been successfully implemented, a version that performed the calculations in parallel was to be made. To increase parallel processing of the nodes, the ring-algorithm was redesigned to process several nodes per SPE at a time. As it was, the program circled the data several turns around the SPEs, this would now be changed into a circling of a single time. The data was passed around the SPEs in terms of bundles of hypercolumns. This meant that the workload partitioning depended on number of SPEs. For example (figure 3.12), six hypercolumns running on three SPEs, would have two hypercolumns processed per SPE-step in the algorithm.

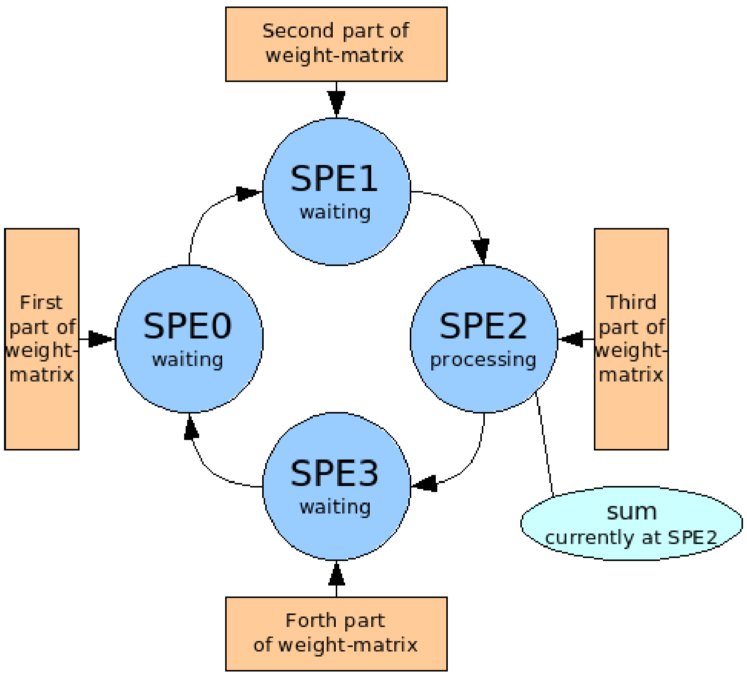
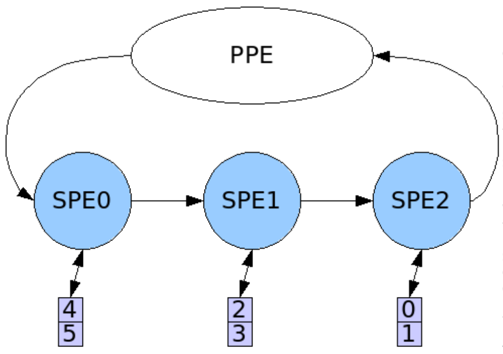

Second part of weight-matrix
SPE1 waiting
First part of weight-matrix
SPE0 waiting
SPE2 processing
Third part of weight-matrix
SPE3 waiting
sum currently at SPE2
Forth part of weight-matrix

Figure 3.11. An example of the first implementation of Net 2. In this example, each SPE stores ¼ of the weight-matrix in their local storages. Each SPE take their turn in doing the main loop calculations.



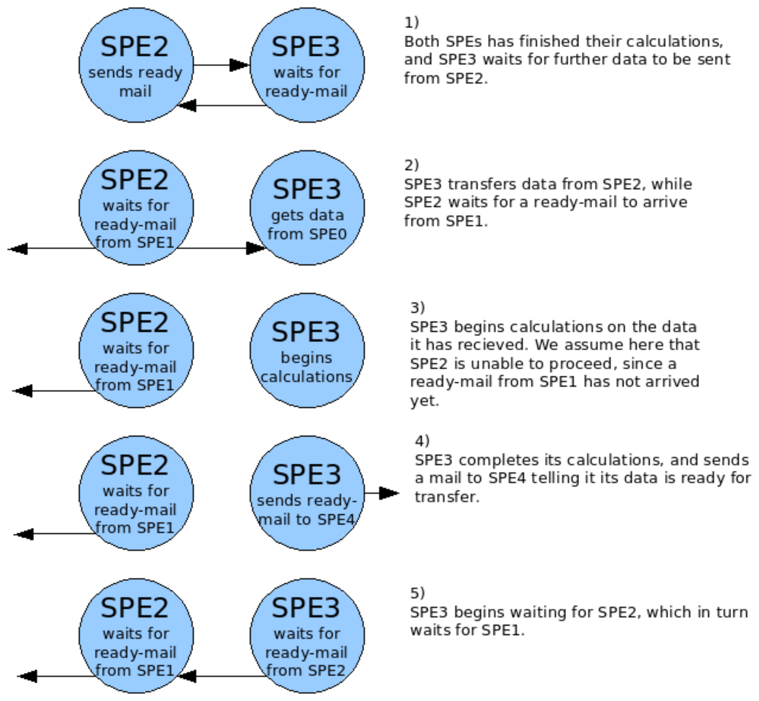
In this picture, SPE2 is currently processing hyper-columns 0 & 1, which already has been processed by the previous SPEs. When SPE2 is done, it will pass the final sum to the PPE.

SPE1 is currently processing hyper-columns 2 & 3, which it previously received from SPE0. When it is done it will send the sum to SPE2, for the final calculations.

SPE0 has begun processing hyper-columns 4 & 5, which are the last ones in the network. When it is done it will pass on the sum to SPE1 for further calculations.

There calculations are performed simultaneously.

Figure 3.12. Further development of Net 2 used more parallel calculations.



SPE2
sends ready mail
SPE3
waits for ready-mail
1)
Both SPEs has finished their calculations, and SPE3 waits for further data to be sent from SPE2.
SPE2
waits for ready-mail from SPE1
SPE3
gets data from SPE0
2)
SPE3 transfers data from SPE2, while SPE2 waits for a ready-mail to arrive from SPE1.
SPE2
waits for ready-mail from SPE1
SPE3
begins calculations
3)
SPE3 begins calculations on the data it has recieved. We assume here that SPE2 is unable to proceed, since a ready-mail from SPE1 has not arrived yet.
SPE2
waits for ready-mail from SPE1
SPE3
sends ready-mail to SPE4
4)
SPE3 completes its calculations, and sends a mail to SPE4 telling it its data is ready for transfer.
SPE2
waits for ready-mail from SPE1
SPE3
waits for ready-mail from SPE2
5)
SPE3 begins waiting for SPE2, which in turn waits for SPE1.

Figure 3.13. A situation where many SPEs are idle, waiting for data.

The bundle-size would correspond to the partitioning of the weight-matrix-distribution in the SPEs. From SPE one to six, every SPE would:

1. Wait for a mail indicating that data was ready to be sent from the preceeding SPE in the ring.

2. Get data from preceeding SPE (or just begin calculations, for the case of SPE0, which lacks a “preceeding” SPE) when ready-mail was received.

3. Do calculations.

4. Send ready-mail to next SPE. (or send sums to the PPE if SPE5)

5. Repeat from 1.

The first SPE would process the first batch of nodes, and when finished, send the result to the next SPE. While SPE 2 was processing its data-set, the previous SPE 1 would start on a new one.

As it was now, much work was still performed by the PPE, the soft-max and updating of activity for example. Much SPU idling time was still wasting processing cycles too. Every SPE made few computations, before sending forth the data to the next SPE. This lead to the prevalent SPU idling. All work should be put to the SPEs, and minimizing idling should also be of priority. Net 2 was redesigned with these issues at mind.

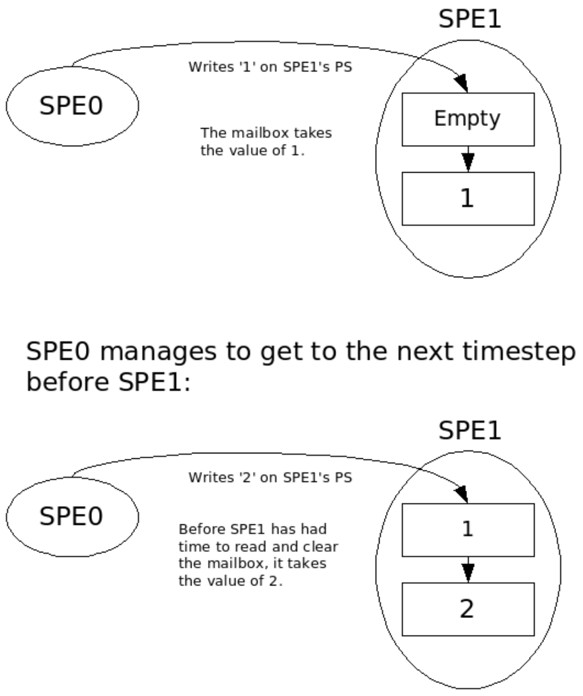

SPE1
Writes '1' on SPE1's PS
SPE0
The mailbox takes the value of 1.
Empty
1
SPE0 manages to get to the next timestep before SPE1:
SPE1
Writes '2' on SPE1's PS
SPE0
Before SPE1 has had time to read and clear the mailbox, it takes the value of 2.
1
2

Figure 3.14. The problem of the mailbox getting overwritten is shown here.

Problems with synchronization

There were problems in the way that synchronization was performed. A significant problem was found with how mailboxes were used for synchronization. Simply polling the channel-registries' number-count using spu_stat_in_mbox() was not a reliable way of checking for mail when communicating with SPEs. When sending mails between the SPEs one did not use special functions, one had to map the channel registries to a memory map called problem state. When sending a mail from SPE to SPE, one would write directly to the channel using the problem state mapping. The problem with this was that the channel count, that kept track of number of mails in the mailbox, did not update quickly enough to be reliable in the program. Furthermore, there was no way of checking the mailbox state of the receiver when sending mail. This meant that if the mailbox of the receiver was full, the mailbox would simply be overwritten when sending mails to it. This is illustrated in figure 3.14.

To gain control over channel-count, one has to run the program in privileged mode, which did not seem like a sensible course of action at this point. It was decided to test the use of signals for synchronization instead.

An improved version of Net 2

Net 2 was redesigned to allow for better parallelization. To minimize idling SPEs, and minimize computations on the PPE, it was decided that each SPE would process a whole pattern at a time, a different pattern for each SPE. When a pattern had been processed by all SPEs,

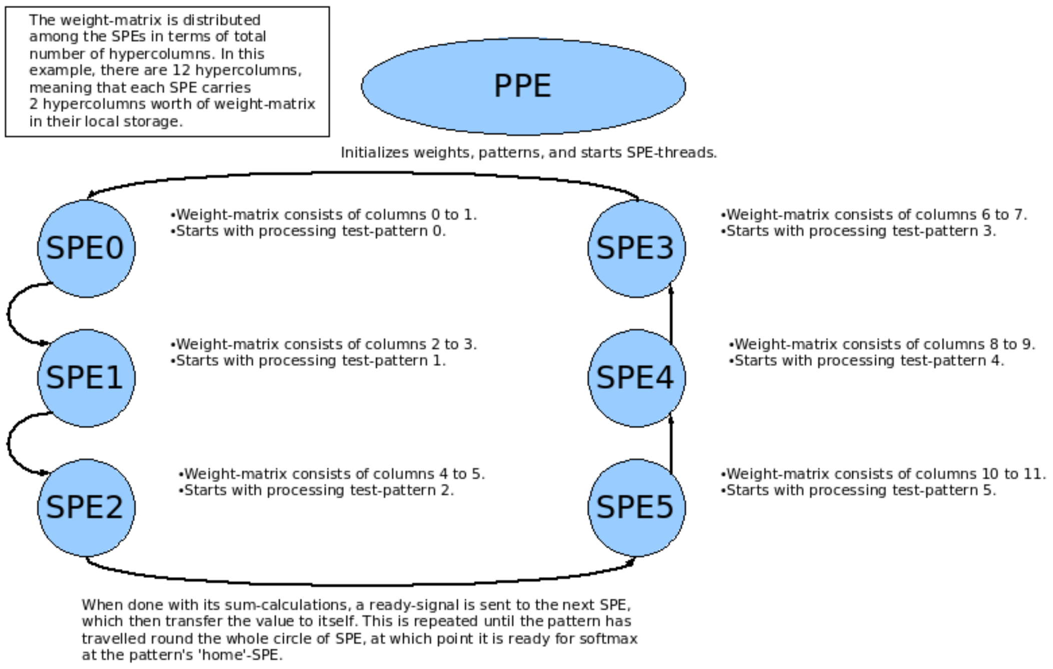

The weight-matrix is distributed among the SPEs in terms of total number of hypercolumns. In this example, there are 12 hypercolumns, meaning that each SPE carries 2 hypercolumns worth of weight-matrix in their local storage.
PPE
Initializes weights, patterns, and starts SPE-threads.
SPE0
•Weight-matrix consists of columns 0 to 1. •Starts with processing test-pattern 0.
SPE1
•Weight-matrix consists of columns 2 to 3. •Starts with processing test-pattern 1.
SPE2
•Weight-matrix consists of columns 4 to 5. •Starts with processing test-pattern 2.
SPE3
•Weight-matrix consists of columns 6 to 7. •Starts with processing test-pattern 3.
SPE4
•Weight-matrix consists of columns 8 to 9. •Starts with processing test-pattern 4.
SPE5
•Weight-matrix consists of columns 10 to 11. •Starts with processing test-pattern 5.
When done with its sum-calculations, a ready-signal is sent to the next SPE, which then transfer the value to itself. This is repeated until the pattern has travelled round the whole circle of SPE, at which point it is ready for softmax at the pattern's 'home'-SPE.

Figure 3.15. The final version of Net 2 is shown here. The ring of SPEs circle whole patterns in this version, allowing for all calculations to be performed in the SPEs.

and the pattern had come to the SPE from where it started, the soft-max function would be applied. This is illustrated in figure 3.15.

This relied even more on good synchronization. Before starting the programming of this, a scheme of the relaxation-process, and what each SPE would do at a given time-step, was written up, shown in figure 3.16.

The synchronization was made by sending bits to the signal registry of the SPEs, in OR-mode. At every synchronization, each SPE would send their synchronization-bit to all SPE's signal registries, and then wait until their own was filled up with all the bits required. An example is shown in figure 3.17.

An inescapable problem in synchronization on the Playstation 3

A problem in synchronization, that made perfect synchronization practically impossible, was found. The idea of Net 2 was that the fast Element Interconnect Bus (EIB) would be utilized for fast data-transfer. The thought was that every SPE would send data to the SPE that lay physically beside it on the chip, to allow for the fastest possible data-transfer. On the PS3 however, the affinity-functions that makes this possible is not available for use. These functions *are* available on the Cell Blade-center, but other problems was found when attempting to use them there. The kernel that was installed on the Cell Blade-center that was used did not allow for use of affinity. Several attempts to enable the affinity-functions were made, but the program persisted in giving the “error in the underlying operating system”-message.

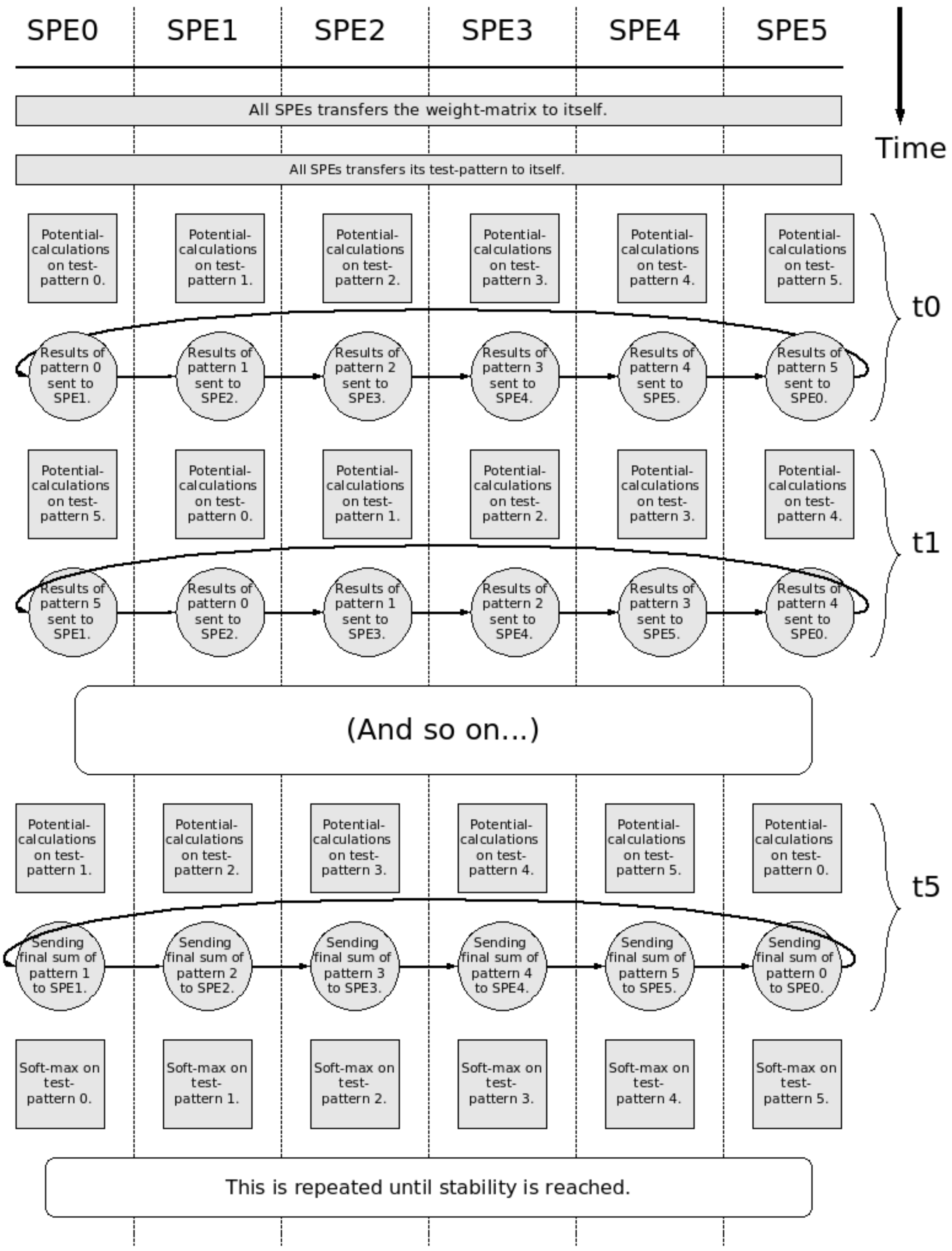

SPE0
SPE1
SPE2
SPE3
SPE4
SPE5
Time
All SPEs transfers the weight-matrix to itself.
All SPEs transfers its test-pattern to itself.
t0
Potential-calculations on test-pattern 0.
Potential-calculations on test-pattern 1.
Potential-calculations on test-pattern 2.
Potential-calculations on test-pattern 3.
Potential-calculations on test-pattern 4.
Potential-calculations on test-pattern 5.
Results of pattern 0 sent to SPE1.
Results of pattern 1 sent to SPE2.
Results of pattern 2 sent to SPE3.
Results of pattern 3 sent to SPE4.
Results of pattern 4 sent to SPE5.
Results of pattern 5 sent to SPE0.
t1
Potential-calculations on test-pattern 5.
Potential-calculations on test-pattern 0.
Potential-calculations on test-pattern 1.
Potential-calculations on test-pattern 2.
Potential-calculations on test-pattern 3.
Potential-calculations on test-pattern 4.
Results of pattern 5 sent to SPE1.
Results of pattern 0 sent to SPE2.
Results of pattern 1 sent to SPE3.
Results of pattern 2 sent to SPE4.
Results of pattern 3 sent to SPE5.
Results of pattern 4 sent to SPE0.
(And so on...)
t5
Potential-calculations on test-pattern 1.
Potential-calculations on test-pattern 2.
Potential-calculations on test-pattern 3.
Potential-calculations on test-pattern 4.
Potential-calculations on test-pattern 5.
Potential-calculations on test-pattern 0.
Sending final sum of pattern 1 to SPE1.
Sending final sum of pattern 2 to SPE2.
Sending final sum of pattern 3 to SPE3.
Sending final sum of pattern 4 to SPE4.
Sending final sum of pattern 5 to SPE5.
Sending final sum of pattern 0 to SPE0.
Soft-max on test-pattern 0.
Soft-max on test-pattern 1.
Soft-max on test-pattern 2.
Soft-max on test-pattern 3.
Soft-max on test-pattern 4.
Soft-max on test-pattern 5.
This is repeated until stability is reached.

Figure 3.16. The operation-schedule is shown here. In order to achieve high efficiency, all SPEs should perform the soft-max-step simultaneously.

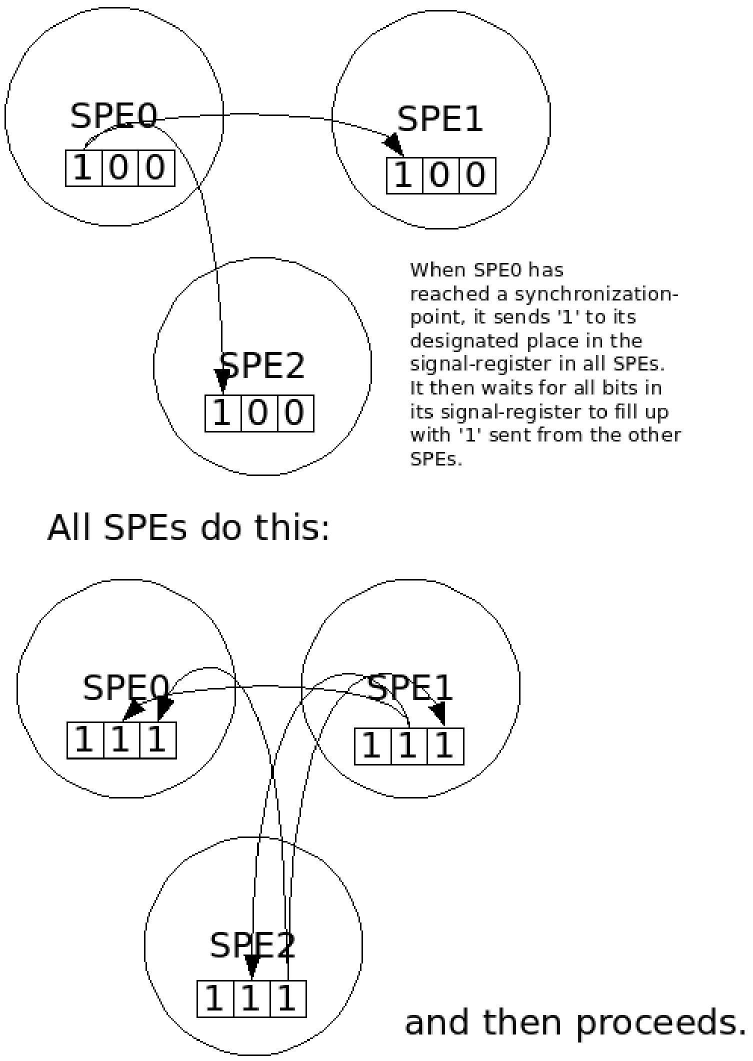

SPE0
1 0 0
SPE1
1 0 0
SPE2
1 0 0
When SPE0 has reached a synchronization-point, it sends '1' to its designated place in the signal-register in all SPEs. It then waits for all bits in its signal-register to fill up with '1' sent from the other SPEs.
All SPEs do this:
SPE0
1 1 1
SPE1
1 1 1
SPE2
1 1 1
and then proceeds.

Figure 3.17. The method of synchronizing the SPEs was accomplished through the “signal”-functionality.

Optimizations

A number of optimizations were performed on the code, these included:

- Substituting the inlined version _log10f4 for log10f4.
- Removing some operations on variables for loop-incrementation from the loop, lowering the number of scalar operations in the main loop.
- Explicitly telling the compiler to use the registers for some common variables.

Special care was now also taken to ensure that DMA-transfers was performed in an optimal manner. When the command mfc_read_tag_status_all() is called, the program waits for a selected group of DMAs to complete. In order to minimize this waiting-time, only the tags of the DMAs that brought the needed data was selected at a given call to mfc_read_tag_status_all(). This is of high importance in an implementation that uses double-buffering.

The performance of this version was 23 Gflops.

Switching between use of unsigned short and unsigned int did not have a noticeable effect. The function "calculateSupportRHS", which handles to inner loop of the BCPNN, calls the function "runThroughColumns". Removing the function "runThroughColumns", and putting the code in the "calculateSupportRHS"-function had a detrimental effect on performance, lowering it to 21.5 Gflops. Inlining it also lowered performance in similar numbers. Since the loop over columns in "runThroughColumns" only made four iterations, further unrolling of it was sensible, and it turned out to have a big positive effect on performance.

Before:

```
  for (; node < end; node += nodesInColumn) {

    summations0 = spu_splats(0.0f);
...
    summations0 = spu_madd(weights[woffset+node], activityVec[node], summations0);
    summations1 = spu_madd(weights[woffset+node+1], activityVec[node+1], summations1);
...
   passedSum = spu_add(_log10f4(summations0), passedSum);
   }
return passedSum;
```

After:

```
    summations0 = summations1 = spu_splats (0.0f);
...
    summations0 = spu_madd(weights[woffset+node], activityVec[node], summations0);
...
   passedSum = spu_add(_log10f4(summations0), passedSum);

    //The second iteration in the loop
    node += nodesInColumn;
    summations0 = summations1 = spu_splats (0.0f);
 ...
    summations0 = spu_madd(weights[woffset+node], activityVec[node], summations0);
...
    //The end of the forth iteration
    passedSum = spu_add(_log10f4(summations0), passedSum);
return passedSum;
```

This increased the performance to 33 Gflops. Putting "woffset+node" in the array-index above into a single variable did not have a noticeable effect. Using a pointer to the passedSum-variable, instead of passing it as a value, lowered the performance to 23 Gflops. Rolling up the loop that calls runThroughColumn increased the performance to 34.5 Gflops.

Before:

```
 for (currentVector = 0; currentVector < totalVectors; currentVector++) {
     data[currentVector] = runThroughColumns( ... );
 }
```

After:

```
for (currentVector = 0; currentVector < totalVectors; currentVector+=8) {
     data[currentVector] = runThroughColumns( ... );
     data[currentVector+1] = runThroughColumns( ... );
     ...
 }
```

Compiling with the option -funroll-all-loops further increased the performance to 35.5Gflops.

3.4 Implementation-, and measurement-tools

Issues of performance tests are dealt with in this section. The different tools of testing, and tuning, performance are described, and tests with a "normal" CPU architecture is performed.

3.4.1 How to measure performance

Performance was investigated in different ways. The primary measurement was that of floating point operations per second, flops. This number gives us something to view in regards to how well a program is performing, compared to its peak potential. This number however, can be of limited use in the process of improving a program. The full system simulator (systemsim) gives a number of statistics of a program-run that are helpful in diagnosing your code, among these an exact number of floating point operations performed. It also gives: cycles per instruction, dual-issue rate, stalled cycle types, etc - a clearer view of what kinds of deficiencies that the program is suffering of. The spu timing device gives further means of analyzing the code, where you view the assembly code itself, and can easily detect stalls due to dependencies etc.

The SDK also includes several profiling, tracing, and other tools. These are described in detail in section 3.4.6.

3.4.2 Tests with the full system simulator

The full system simulator is a simulation-tool that simulates a Cell architecture. It can be run under a x86-CPU. The IBM Full System Simulator was installed on a laptop running the Fedora Core distribution of Linux, which is the one recommended for the SDK.

The simulator was started using "Linux"-mode, which means that it gets a virtual file-system. To this file-system the program to run must be copied, and the executables set to +x status.

After the program has run its course, you stop the simulation, and get access to the statistics. An example run gives the following statistics:

```
SPU DD3.0
***
Total Cycle count                353115243
Total Instruction count          119692700
Total CPI                        2.95
***
Performance Cycle count          353115243
Performance Instruction count    119692700 (114387372)
Performance CPI                  2.95 (3.09)

Branch instructions              13854021
Branch taken                     12082965
Branch not taken                 1771056

Hint instructions                1335671
Pipeline flushes                 1681894
SP operations (MADDs=2)          119193120
DP operations (MADDs=2)          18

Contention at LS between Load/Store and Prefetch 1333261

Single cycle                                               45601938 ( 12.9%)
Dual cycle                                                 34392717 (  9.7%)
Nop cycle                                                   1396853 (  0.4%)
Stall due to branch miss                                   29297224 (  8.3%)
Stall due to prefetch miss                                        0 (  0.0%)
Stall due to dependency                                    80737263 ( 22.9%)
Stall due to fp resource conflict                                 0 (  0.0%)
Stall due to waiting for hint target                        1064592 (  0.3%)
Issue stalls due to pipe hazards                                 42 (  0.0%)
Channel stall cycle                                       160624605 ( 45.5%)
SPU Initialization cycle                                          9 (  0.0%)
-----------------------------------------------------------------------
Total cycle                                               353115243 (100.0%)

Stall cycles due to dependency on each instruction class
 FX2        4997722 (  6.2% of all dependency stalls)
 SHUF       3911705 (  4.8% of all dependency stalls)
 FX3        2249644 (  2.8% of all dependency stalls)
 LS         45535812 ( 56.4% of all dependency stalls)
 BR         4704 (  0.0% of all dependency stalls)
 SPR        7761 (  0.0% of all dependency stalls)
 LNOP       0 (  0.0% of all dependency stalls)
 NOP        0 (  0.0% of all dependency stalls)
 FXB        0 (  0.0% of all dependency stalls)
 FP6        16728480 ( 20.7% of all dependency stalls)
 FP7        7301411 (  9.0% of all dependency stalls)
 FPD        24 (  0.0% of all dependency stalls)

The number of used registers are 128, the used ratio is 100.00

Instruction Class                                 Cycles/Inst
---------------------------------------------   -----------
FX2 (EVEN): Logical and integer arithmetic              2.06
SHUF (ODD): Shuffle, quad rotate/shift, mask            2.82
FX3 (EVEN): Element rotate/shift                        3.99
LS   (ODD): Load/store, hint                            3.41
BR   (ODD): Branch                                      4.32
SPR  (ODD): Channel and SPR moves                       2.74
LNOP (ODD): NOP                                         0.00
NOP (EVEN): NOP                                         0.00
FXB (EVEN): Special byte ops                            0.00
FP6 (EVEN): SP floating point                           6.07
FP7 (EVEN): Integer mult, float conversion              3.56
FPD (EVEN): DP floating point                           7.00

dumped pipeline stats
```

The last section, for instruction class, has been modified to fit the page.

At the top you can see the cycle per instruction-ratio, CPI: 2.95, This is not a good value, a value of at most 0.7-0.9 is desirable. (IBM, 2007d) The next value of interest is the count of single-precision operations:

```
SP operations (MADDs=2)          119193120
```

This value was used to determine the performance in Gflops for the programs. Of note here is that this value does not include scalar operations. It only includes vector-operations. This means that the value is slightly lower than the true amount of floating point operations performed. However, since the simulator's value for operations performed otherwise is accurate and precise, it was found to be more useful to use that, instead of trying to calculate operations performed manually.

The next values of interest is the ones showing cycle-stalls, dual issue rate, etc.

```
Single cycle                                          45601938 ( 12.9%)
Dual cycle                                            34392717 (  9.7%)
Nop cycle                                              1396853 (  0.4%)
Stall due to branch miss                              29297224 (  8.3%)
Stall due to prefetch miss                                   0 (  0.0%)
Stall due to dependency                               80737263 ( 22.9%)
Stall due to fp resource conflict                            0 (  0.0%)
Stall due to waiting for hint target                   1064592 (  0.3%)
Issue stalls due to pipe hazards                            42 (  0.0%)
Channel stall cycle                                  160624605 ( 45.5%)
SPU Initialization cycle                                     9 (  0.0%)
-----------------------------------------------------------------------
Total cycle                                          353115243 (100.0%)
```

Here you can see that the program suffers heavily from channel stalls, whose cause is that of channel-registry-operations, mail and signals. The dual issue versus single issue-ratio does not appear to be poor. The heavy channel stalls, and dependency-stalls (which are further explained below) cause the program performs poorly though. The stalls due to dependencies is further analyzed in the next section of the statistics report:

```
Stall cycles due to dependency on each instruction class
 FX2        4997722 (  6.2% of all dependency stalls)
 SHUF       3911705 (  4.8% of all dependency stalls)
 FX3        2249644 (  2.8% of all dependency stalls)
 LS         45535812 ( 56.4% of all dependency stalls)
 BR         4704 (  0.0% of all dependency stalls)
 SPR        7761 (  0.0% of all dependency stalls)
 LNOP       0 (  0.0% of all dependency stalls)
 NOP        0 (  0.0% of all dependency stalls)
 FXB        0 (  0.0% of all dependency stalls)
 FP6        16728480 ( 20.7% of all dependency stalls)
 FP7        7301411 (  9.0% of all dependency stalls)
 FPD        24 (  0.0% of all dependency stalls)
```

LS is instructions of type load/store or branch hints, and FP6 is stalls due to single precision float-point operations. From this you could infer that load/store operation-stalls was the biggest cause of dependency stalled cycles.

The registry use of all version of both programs were always 100%. This meant that the 128 registries of the SPU was fully used. If the number had been lower than 100%, the code could probably be improved by utilizing the fast registry-files of the SPU.

For both Net 1 and 2, the systemsim was used to determine number of floating point operations performed. There did exist an uncertainty though. Did the counter in the systemsim include operations performed in library-calls, such as the log10f4-function used from the simdmath library?

A simple program was made to count the number of operations performed by log10f4, expf4 and divf4. It was found that log10f4 made 60 floating point operations, expf4 and divf4 made

36. Another significant finding was that operations performed to increment loops, and other scalar operations was not counted by the systemsim. This meant that operations performed by library-calls did get included in the systemsim's operation count, as mentioned earlier, it also became clear that the actual number of floating point operations performed was higher than what the systemsim reported. It was decided to use the systemsim's number as a guide anyway, since it gave reliable and exact numbers when it came to the vector-operations.

Using the systemsim for timing could not be done. In order to receive reliable timing-information, you have to set the systemsim to cycle mode. This is extremely slow, and was practically unusable. It took a whole day just to boot the system. This meant that the performance statistics earlier received from the systemsim was wrong. Using the systemsim for timing outside full cycle-mode can be done. Using pipeline-mode on the SPEs gives reliable information, provided that the SPE-code don't use any DMAs to or from the PPE. In using the systemsim to time Net 1 & 2, very different results had been produced. Since Net 1 use transfers with the main memory to a large degree, the timing results of the simulator had been very erroneous. Net 2 on the other hand, which use only small amounts of transfers with the main memory, had not given as erroneous timing results. However, timing with the simulator was still ultimately unreliable.

The systemsim still had its use though, since the statistics on float-operations, and cycle-counts were useful for diagnostics on the programs.

3.4.3 Code analysis with the spu-timing static timing analyzer

The spu-timing tool gives you an analysis of dependencies and dual issue-rate in the assembly code. A typical view of a spu-timing printout is:

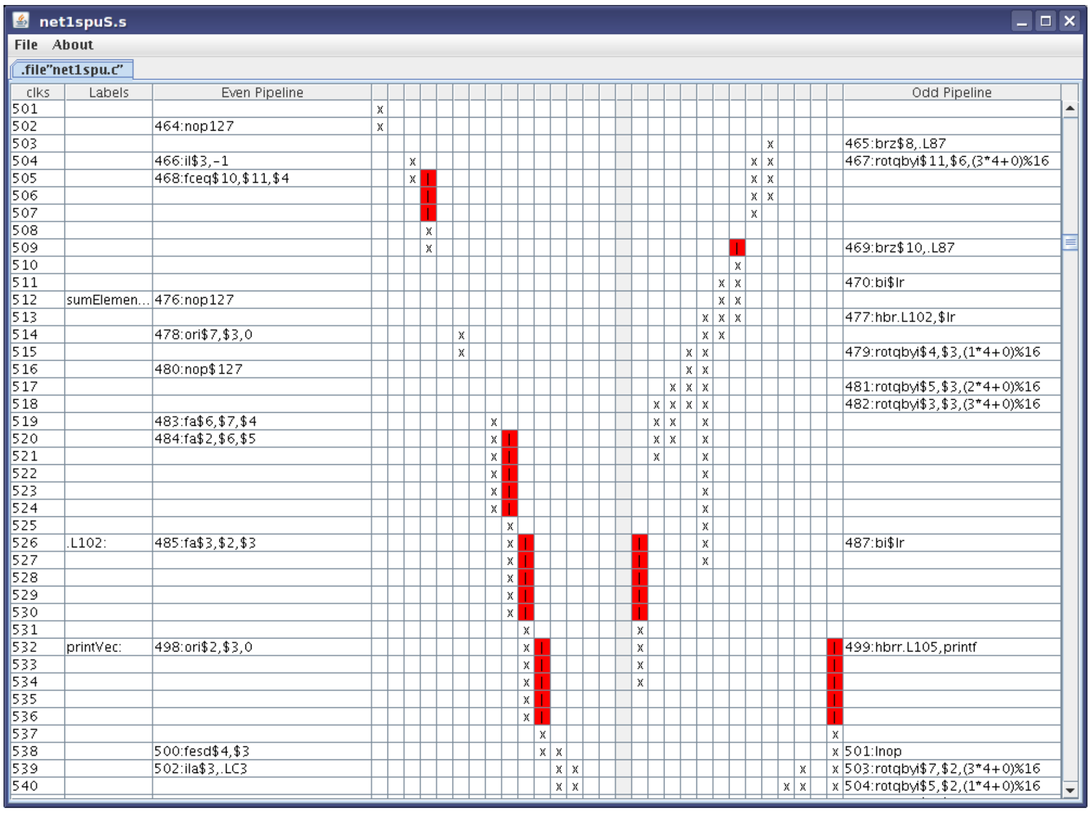
```
                                                                         sumElements:
000493 0d                     3                      nop     127
000496 1d 01                  ---6789                hbr     .L102,$lr
000497 0D                         78                 ori     $7,$3,0
000497 1D 0                       789                rotqbyi     $4,$3,(1*4+0)%16
000498 0D                          8                 nop     $127
000498 1D 01                       89                rotqbyi     $5,$3,(2*4+0)%16
000499 1  012                       9                rotqbyi     $3,$3,(3*4+0)%16
000501 0  -123456                                    fa     $6,$7,$4
000507 0      -----789012                            fa     $2,$6,$5
000513 0           -----345678                       fa     $3,$2,$3
                                                 .L102:
000514 1                4567                         bi     $lr

                                                 .size     sumElements, .-sumElementsx
```

The number at the left-most side is clock cycle. The next number is pipeline number. The character following is either D, which shows that dual issue was done, a d which shows that dual issue was possible, but failed due to dependencies, or nothing, which marks a single issue cycle. The strings of numbers in the next section represents clock cycles, the number of cycles here is corresponds to the number of cycles that the instruction being carried out takes. For example, a float add (assembly-call: "fa") takes six cycles, as can be seen above. A dash "-" represents stalled cycles. The right-most section of the printout shows the assembly instructions.

From the above one sees that the floating point operations are issued in a manner that fail to utilize the dual-pipeline of the SPU.

The assembly visualizer tool available from IBM is a tool that gives visualization of assembly, which makes it easier to spot dependencies. Using the tool on the same section as above, the picture in figure 3.18 is given.



clks	Labels	Even Pipeline	Odd Pipeline
501			
502		464:nop127	
503			465:brz$8,.L87
504		466:il$3,-1	467:rotqbyi$11,$6,(3*4+0)%16
505		468:fceq$10,$11,$4	
506			
507			
508			
509			469:brz$10,.L87
510			
511			470:bi$lr
512	sumElemen...	476:nop127	
513			477:hbr.L102,$lr
514		478:ori$7,$3,0	
515			479:rotqbyi$4,$3,(1*4+0)%16
516		480:nop$127	
517			481:rotqbyi$5,$3,(2*4+0)%16
518			482:rotqbyi$3,$3,(3*4+0)%16
519		483:fa$6,$7,$4	
520		484:fa$2,$6,$5	
521			
522			
523			
524			
525			
526	.L102:	485:fa$3,$2,$3	487:bi$lr
527			
528			
529			
530			
531			
532	printVec:	498:ori$2,$3,0	499:hbrr.L105,printf
533			
534			
535			
536			
537			
538		500:fesd$4,$3	501:lnop
539		502:ila$3,.LC3	503:rotqbyi$7,$2,(3*4+0)%16
540			504:rotqbyi$5,$2,(1*4+0)%16

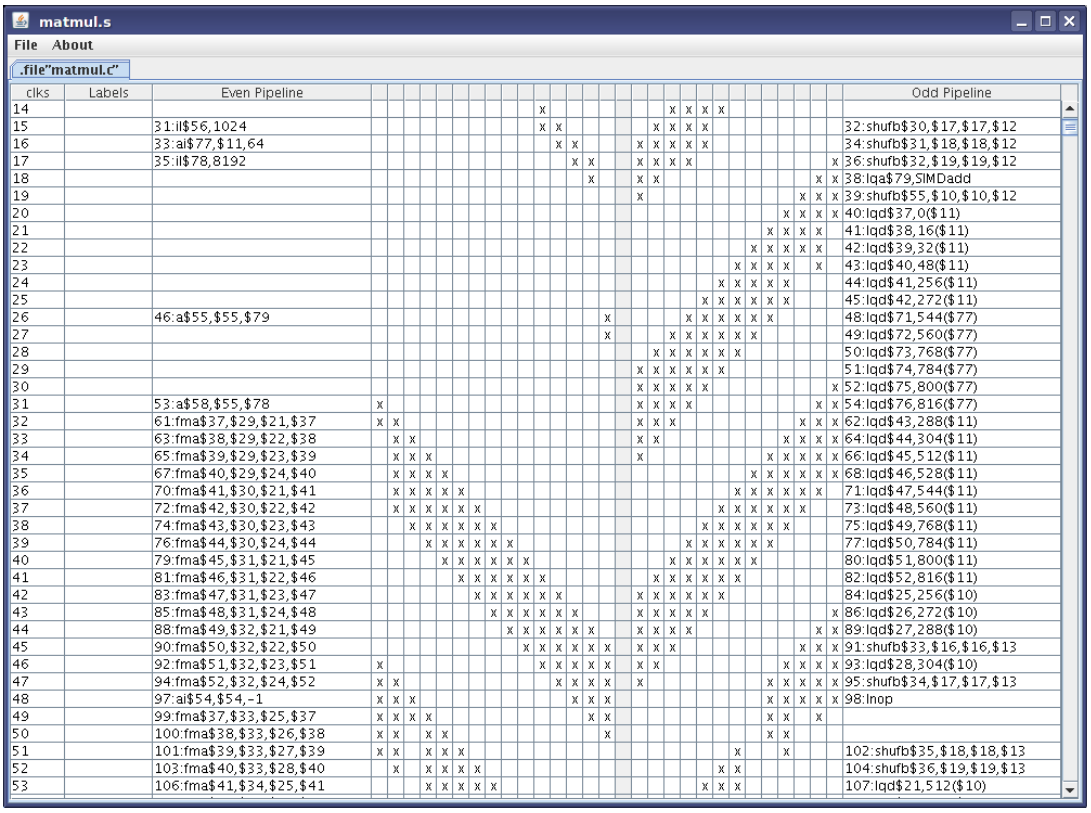
Figure 3.18. The function "sumElements" illustrated with the assembly visualizer.

The information given by the assembly visualizer tool is presented in a slightly different manner than by the spu-timing tool. The instruction calls to the SPU is separated into the dual-pipeline, and the cycle-flow of the instructions is shown falling vertically downwards. In figure 3.18, the same dependencies in the float add-instructions can be easily spotted by the red boxes containing a single vertical line, which stands for stalls.

To give a picture of how a good use of the pipeline looks like, the matmul-program can be used. Matmul is the name of a highly optimized implementation of matrix multiplication, created by Daniel Hackenberg. It achieves near peak performance. He uses a hand-coded assembly function that performs multiplication on 64x64-matrices. Figure 3.19 shows a snippet of his program using the spu-timing tool.

After an initial stream of instructions where values are being loaded, the program achieves almost constant dual issue-rate, where the float multiply add (fma) instructions are performed, and upcoming values are being loaded concurrently in a most efficient manner. The program is in practice completely unrolled, showing the value of unrolling loops. Its organization in regards of dual-issue also gives a good hint of how to organize load-instructions and calculations.

Using the spu-timing-tool on the final version of Net 1 & 2's main loop showed that the inner loop performed well, and did not suffer from heavy dependencies.



clks	Labels	Even Pipeline																														Odd Pipeline
14												x								x	x	x	x								
15		31:il$56,1024										x	x						x	x	x	x									32:shufb$30,$17,$17,$12
16		33:ai$77,$11,64											x	x				x	x	x	x	x									34:shufb$31,$18,$18,$12
17		35:il$78,8192												x	x			x	x	x	x									x	36:shufb$32,$19,$19,$12
18															x			x	x										x	x	38:lqa$79,SIMDadd
19																		x										x	x	x	39:shufb$55,$10,$10,$12
20																											x	x	x	x	40:lqd$37,0($11)
21																										x	x	x	x		41:lqd$38,16($11)
22																									x	x	x	x	x		42:lqd$39,32($11)
23																								x	x	x	x		x		43:lqd$40,48($11)
24																							x	x	x	x	x				44:lqd$41,256($11)
25																						x	x	x	x	x	x				45:lqd$42,272($11)
26		46:a$55,$55,$79														x					x	x	x	x	x	x					48:lqd$71,544($77)
27																x				x	x	x	x	x	x						49:lqd$72,560($77)
28																			x	x	x	x	x	x							50:lqd$73,768($77)
29																		x	x	x	x	x	x								51:lqd$74,784($77)
30																		x	x	x	x	x								x	52:lqd$75,800($77)
31		53:a$58,$55,$78	x																x	x	x	x								x	x	54:lqd$76,816($77)
32		61:fma$37,$29,$21,$37	x	x															x	x	x								x	x	x	62:lqd$43,288($11)
33		63:fma$38,$29,$22,$38		x	x														x	x								x	x	x	x	64:lqd$44,304($11)
34		65:fma$39,$29,$23,$39		x	x	x													x								x	x	x	x	x	66:lqd$45,512($11)
35		67:fma$40,$29,$24,$40		x	x	x	x																			x	x	x	x	x	x	68:lqd$46,528($11)
36		70:fma$41,$30,$21,$41		x	x	x	x	x																	x	x	x	x	x	x		71:lqd$47,544($11)
37		72:fma$42,$30,$22,$42		x	x	x	x	x	x															x	x	x	x	x	x			73:lqd$48,560($11)
38		74:fma$43,$30,$23,$43			x	x	x	x	x	x													x	x	x	x	x	x				75:lqd$49,768($11)
39		76:fma$44,$30,$24,$44				x	x	x	x	x	x											x	x	x	x	x	x					77:lqd$50,784($11)
40		79:fma$45,$31,$21,$45					x	x	x	x	x	x									x	x	x	x	x	x						80:lqd$51,800($11)
41		81:fma$46,$31,$22,$46						x	x	x	x	x	x							x	x	x	x	x	x							82:lqd$52,816($11)
42		83:fma$47,$31,$23,$47							x	x	x	x	x	x					x	x	x	x	x	x								84:lqd$25,256($10)
43		85:fma$48,$31,$24,$48								x	x	x	x	x	x				x	x	x	x	x								x	86:lqd$26,272($10)
44		88:fma$49,$32,$21,$49									x	x	x	x	x	x			x	x	x	x								x	x	89:lqd$27,288($10)
45		90:fma$50,$32,$22,$50										x	x	x	x	x	x		x	x	x								x	x	x	91:shufb$33,$16,$16,$13
46		92:fma$51,$32,$23,$51	x										x	x	x	x	x		x	x								x	x	x	x	93:lqd$28,304($10)
47		94:fma$52,$32,$24,$52	x	x										x	x	x	x		x								x	x	x	x	x	95:shufb$34,$17,$17,$13
48		97:ai$54,$54,-1	x	x	x										x	x	x										x	x	x	x	x	98:lnop
49		99:fma$37,$33,$25,$37	x	x	x	x										x	x										x	x		x		
50		100:fma$38,$33,$26,$38	x	x		x	x										x										x	x				
51		101:fma$39,$33,$27,$39	x	x		x	x	x																	x			x				102:shufb$35,$18,$18,$13
52		103:fma$40,$33,$28,$40		x		x	x	x	x															x	x							104:shufb$36,$19,$19,$13
53		106:fma$41,$34,$25,$41				x	x	x	x	x													x	x	x							107:lqd$21,512($10)

Figure 3.19. A snippet of Daniel Hackenberg's hand-coded assembly for matrix-multiplication is shown here.

3.4.4 The benefits from using huge translation look aside buffers with the nets

To use huge translation look aside buffer file system (hugeTLBfs), the kernel must have support for it. The PS3 kernel did not have support for hugeTLBfs by default, which meant that the kernel had to be re-compiled with the proper settings for hugeTLBfs.

The use of hugeTLBfs is not beneficial for all Cell-applications. For Net 2, its use was found to have negliable impact. Net 1 on the other hand, with its heave dependence on fast transfers with the main memory, increased performance to a significant degree. Using a netsize of 32x32 showed in increase in performance with 8% (running a single SPE). With a netsize of 52x52, the speed-increase was 45%. More memory in use increased the performance-benefit from using hugeTLBfs.

3.4.5 Tests on a "normal" CPU

For comparison, tests on a normal computer was to be made. These tests were performed to give a point of reference in comparison with a common modern CPU.

The computer used for the test was a Intel(R) Pentium(R) 4 CPU 2.60 GHz. The clock frequency is lower than the 3.19 GHz of the Cell. Attempts were made to make the implementation on for the normal computer as close to the Cell-implementations as possible. With regard to compiler options, the -funroll-all-loops option used for the Cell, made the normal implementation slower, so it was not used.

3.4.6 Performance tools

The SDK includes a number of tools for tuning performance, and profiling your program.

CPC

The cell-perf-counter tool is a tool for counting hardware events. Events from all the logical units of the Cell are counted; the PPE, the SPEs, the interface bus, memory, and I/O controllers. The CPC failed to start, and it was hard to find support given the error-message given.

OProfile

OProfile is a system-level profiler for Cell, which collects profiling information with low overhead. This tool is not supported with the systemsim kernel, nor the Playstation 3 kernel, and had to be tested on the Blade. Since it caused the system to crash, tests were abandoned.

Performance Debugging Tool (PDT)

PDT provides tracing capabilities of events during program execution. It is used ti trace events of interest in real time, and to record such data.

FDPR-Pro

The Post-link Optimization for Linux on POWER tool (fdprpro), is a performance tuning utility. It collects behavior-information of the program during execution, and creates a new version of the program that typically runs faster, and uses less memory. This tool was tested, but it failed to improve performance.

Visual Performance Analyzer

This is an Eclipse based tool set that works with plug-ins of the Profile Analyzer, Code Analyzer, Pipeline Analyzer, Counter Analyzer, Trace Analyzer, and Control Flow Analyzer. A subjective view of the visualizer gave a favorable impression, but the problems encountered with the plug-ins themselves made testing difficult, and was abandoned.

Cmpware SPU Scheduling Tool

The Cmpware SPU Scheduling Tool is not part of the standard SDK, but its potential use warrants mention. The purpose of the program is to make pipeline management easier. One of the challenges in programming efficient code for the SPU is that it is hard to get an easy overview of a particular passage of assembly code. For good use of the pipeline, dependencies must sometimes be managed by hand, and handling of that part is the purpose of the Cmpware SPU Scheduling Tool. Because of lack of time, the program was not tested in this work.

Chapter 4

Results

The results of both timing, and simulator statistics are given for both nets here. Different network-configurations are tested and explained.

4.1 Net 1

Results from runs on the Blade-center server were almost identical as runs on the Playstation 3. Running a single SPE on the Blade-center gave the same results as on the Playstation 3. Running six SPEs on the Blade-center gave results varying in the range of 18.2-18.9 Gflops. The Blade-center has two Cells, and the random allocation on SPE-threads is the reason for the variation.

Further results from running the program on the Playstation 3 follows.

4.1.1 Timed results for Net 1

These are the results from timing the program for measurement of floating point operations performed per second, and the data-flow with the main memory. The data-flow is measured by counting how many times the weight-matrix is transfered over the course of the run.

Tests using, and not using, the huge translation look-aside buffer filesystem (hugetlbfs) is presented. The program can be structured so that the logarithm in function 4.1 is pre-computed.

$$s_j = \log(\beta_j) + \sum_{h=1}^{H} \log(\sum_{k \in Q_h} w_{kj} o_k) \tag{4.1}$$

The log10f4-function that performs the logarithm was found to give rise to stalled cycles. Results when performing the logarithm, and pre-computing it, are presented.

Figure 4.1 shows the result from running the basic version, huge translation look-aside buffers enabled, and performing logarithms in the main loop. The net-size is 32 hypercolumns, with 32 single-node minicolumns.

Figure 4.2 shows the result from running a big net, size 52 hypercolumns, with 52 single-node minicolumns. Hugetlbf is enabled, and log is used in the main loop.

Results using different solutions is showed here. With, or without using logarithms in the main loop, and with, or without using hugetlb. The tests are running with a single SPE, with a net-size of 32 hypercolumns, 32 minicolumns (figure 4.3), and 52 hypercolumns, 52 minicolumns (figure 4.4).

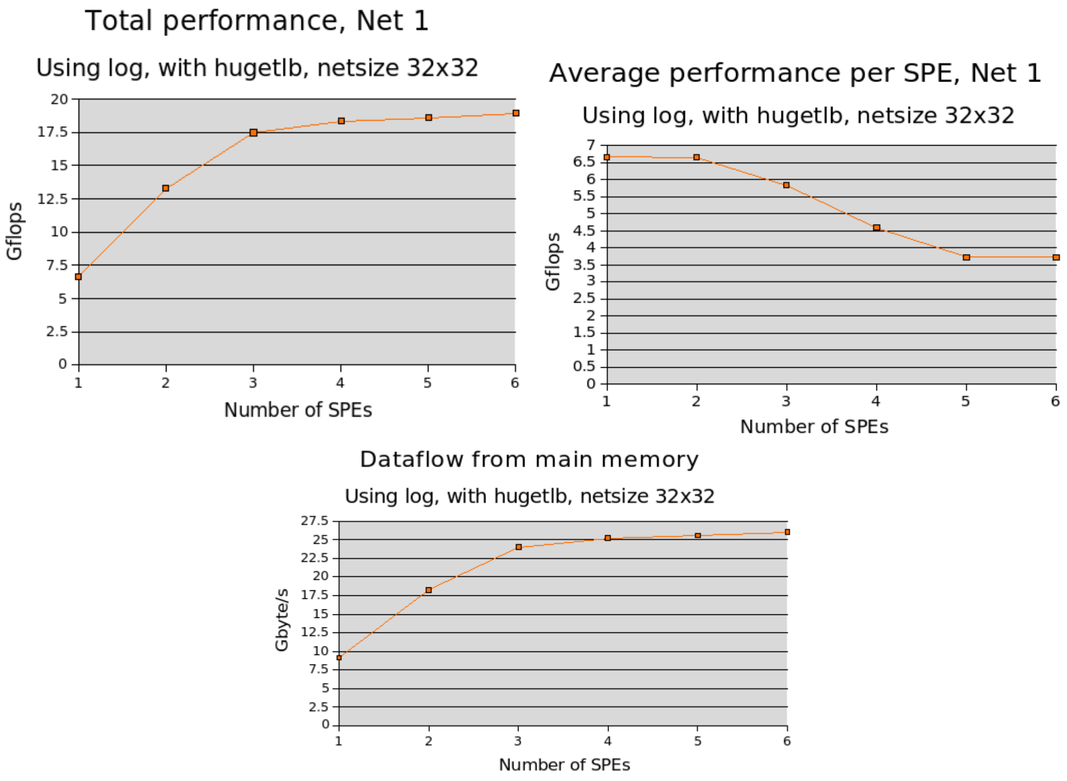
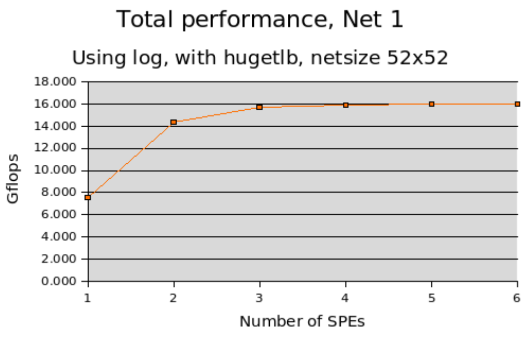
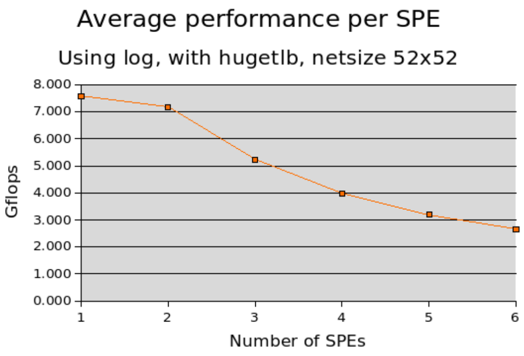


Figure 4.1. The results of Net 1 (storing the weights in main storage), running a net-size of 32x32, is shown here.

Results with smaller net-sizes, including a configuration matching the typical configuration of Net 2 is shown in figure 4.5.

Results were generally stable, and performance varied only very slightly in a range of 0.01 Gflops. This variation could be due to the physical distance from main storage for the SPEs, and differences in access-times to main storage due to the random allocation of SPE-threads.





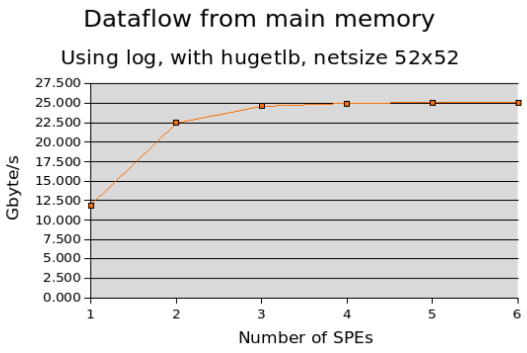


Figure 4.2. The results of Net 1 (storing the weights in main storage), running a net-size of 52x52, is shown here.

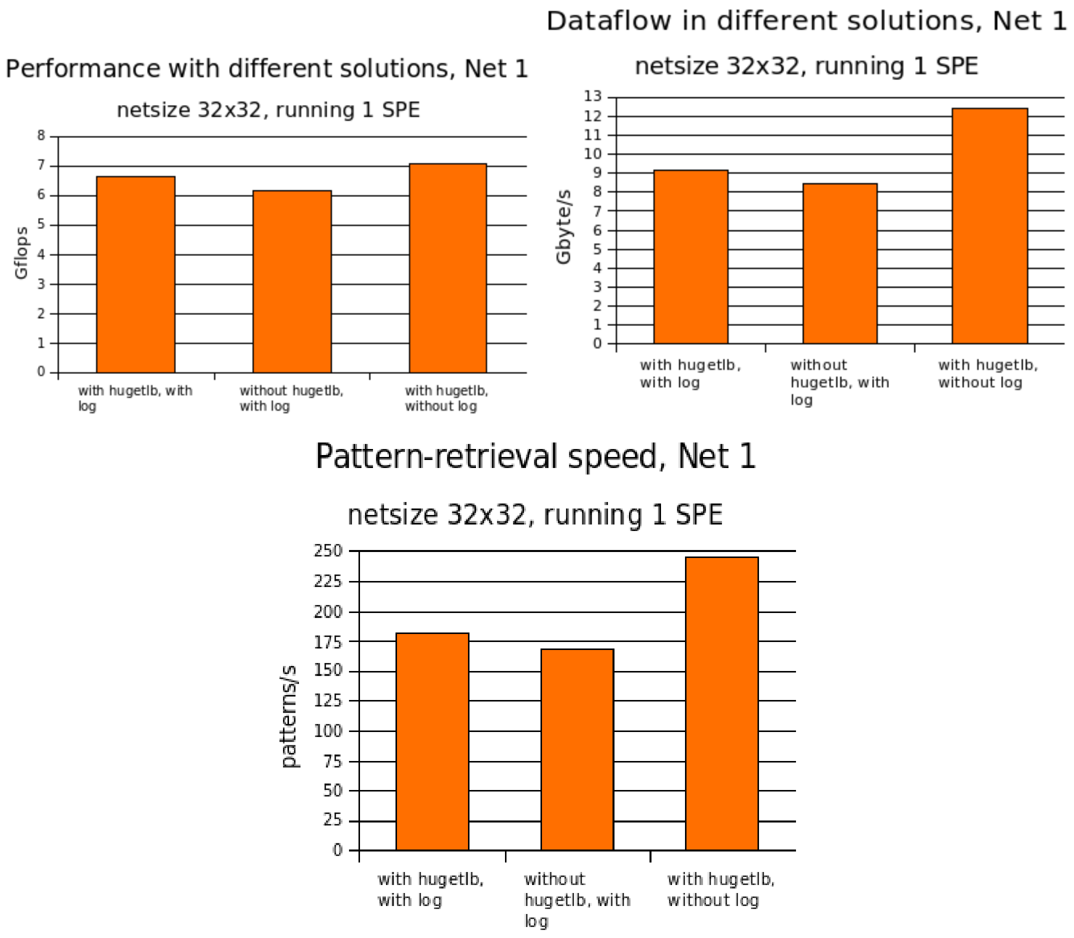


Figure 4.3. The results of Net 1 (storing the weights in main storage), running a net-size of 32x32, with different methods tested is shown here.

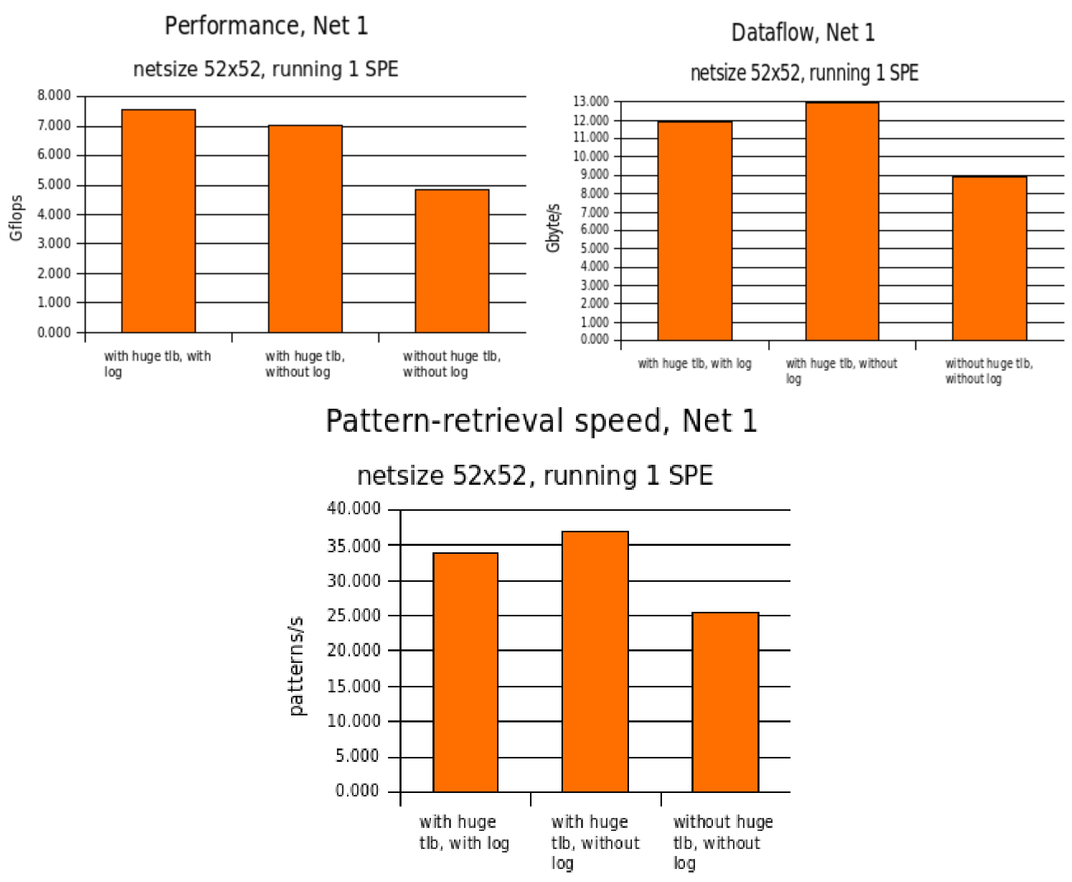


Figure 4.4. The results of Net 1 (storing the weights in main storage), running a net-size of 52x52, with different methods tested is shown here.

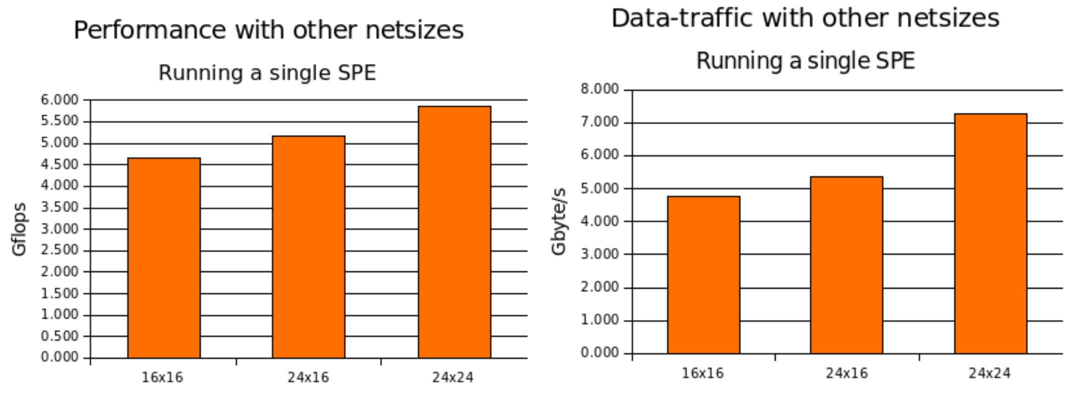


Figure 4.5. The results of Net 1 (storing the weights in main storage), running smaller nets, including a configuration similar to the typical configuration of Net 2, is shown here.

4.1.2 Systemsim statistics for Net 1

Systemsim statistics from Net 1 is shown here. The configuration is a netsize of 32x32, performing logarithm in the main loop, and having hugetlb enabled.

First the resut from running 6 SPEs, 36 test-patterns. The section for instruction class counts has been edited to fit the page.

```
Performance Cycle count          102786725
Performance Instruction count    89709391 (79062726)
Performance CPI                  1.15 (1.30)

Branch instructions              914804
Branch taken                     758202
Branch not taken                 156602

Hint instructions                145434
Pipeline flushes                 69083
SP operations (MADDs=2)          219657520
DP operations (MADDs=2)          6

Contention at LS between Load/Store and Prefetch 13575664

Single cycle                                              46834940 ( 45.6%)
Dual cycle                                                16113893 ( 15.7%)
Nop cycle                                                  3140335 (  3.1%)
Stall due to branch miss                                   1195810 (  1.2%)
Stall due to prefetch miss                                       0 (  0.0%)
Stall due to dependency                                   34961768 ( 34.0%)
Stall due to fp resource conflict                                0 (  0.0%)
Stall due to waiting for hint target                        212918 (  0.2%)
Issue stalls due to pipe hazards                                12 (  0.0%)
Channel stall cycle                                         327040 (  0.3%)
SPU Initialization cycle                                         9 (  0.0%)
-----------------------------------------------------------------------------
Total cycle                                              102786725 (100.0%)

Stall cycles due to dependency on each instruction class
 FX2         1943512 (  5.6% of all dependency stalls)
 SHUF        671694 (  1.9% of all dependency stalls)
 FX3         3103287 (  8.9% of all dependency stalls)
 LS          583234 (  1.7% of all dependency stalls)
 BR          34658 (  0.1% of all dependency stalls)
 SPR         25 (  0.0% of all dependency stalls)
 LNOP        0 (  0.0% of all dependency stalls)
 NOP         0 (  0.0% of all dependency stalls)
 FXB         0 (  0.0% of all dependency stalls)
 FP6         28303527 ( 81.0% of all dependency stalls)
 FP7         321825 (  0.9% of all dependency stalls)
 FPD         6 (  0.0% of all dependency stalls)

The number of used registers are 128, the used ratio is 100.00

Instruction Class                                  Cycles/Inst
-------------------------------------------------  -----------
FX2 (EVEN): Logical and integer arithmetic                1.83
SHUF (ODD): Shuffle, quad rotate/shift, mask              3.29
FX3 (EVEN): Element rotate/shift                          3.60
LS   (ODD): Load/store, hint                              1.14
BR   (ODD): Branch                                        3.98
SPR  (ODD): Channel and SPR moves                         2.52
LNOP (ODD): NOP                                           0.00
NOP (EVEN): NOP                                           0.00
FXB (EVEN): Special byte ops                              0.00
FP6 (EVEN): SP floating point                             2.44
FP7 (EVEN): Integer mult, float conversion                6.78
FPD (EVEN): DP floating point                             7.00

dumped pipeline stats
```

The statistics were identical for runs with different numbers of SPEs.

Net 1 running an identical net-configuration as net 2 - 48 test-patterns, size 24x16:

```
Performance Cycle count         48237122
Performance Instruction count   33842542 (29409946)
Performance CPI                 1.43 (1.64)

Branch instructions             593588
Branch taken                    466501
Branch not taken                127087

Hint instructions               110669
Pipeline flushes                58118
SP operations (MADDs=2)         81614460
DP operations (MADDs=2)         6

Contention at LS between Load/Store and Prefetch 1733146

Single cycle                                         17683974 ( 36.7%)
Dual cycle                                            5862986 ( 12.2%)
Nop cycle                                             1142209 (  2.4%)
Stall due to branch miss                              1010625 (  2.1%)
Stall due to prefetch miss                                  0 (  0.0%)
Stall due to dependency                              21672915 ( 44.9%)
Stall due to fp resource conflict                           0 (  0.0%)
Stall due to waiting for hint target                   171962 (  0.4%)
Issue stalls due to pipe hazards                           18 (  0.0%)
Channel stall cycle                                    692433 (  1.4%)
SPU Initialization cycle                                    0 (  0.0%)
-----------------------------------------------------------------------
Total cycle                                          48237122 (100.0%)

Stall cycles due to dependency on each instruction class
 FX2        1135756 (  5.2% of all dependency stalls)
 SHUF       471414 (  2.2% of all dependency stalls)
 FX3        1752870 (  8.1% of all dependency stalls)
 LS         735283 (  3.4% of all dependency stalls)
 BR         27820 (  0.1% of all dependency stalls)
 SPR        24 (  0.0% of all dependency stalls)
 LNOP       0 (  0.0% of all dependency stalls)
 NOP        0 (  0.0% of all dependency stalls)
 FXB        0 (  0.0% of all dependency stalls)
 FP6        17310853 ( 79.9% of all dependency stalls)
 FP7        238889 (  1.1% of all dependency stalls)
 FPD        6 (  0.0% of all dependency stalls)

The number of used registers are 128, the used ratio is 100.00

Instruction Class                                    Cycles/Inst
---------------------------------------------       ----------
FX2 (EVEN): Logical and integer arithmetic                1.80
SHUF (ODD): Shuffle, quad rotate/shift, mask              3.29
FX3 (EVEN): Element rotate/shift                          3.55
LS   (ODD): Load/store, hint                              1.30
BR   (ODD): Branch                                        3.96
SPR  (ODD): Channel and SPR moves                         2.63
LNOP (ODD): NOP                                           0.00
NOP (EVEN): NOP                                           0.00
FXB (EVEN): Special byte ops                              0.00
FP6 (EVEN): SP floating point                             3.01
FP7 (EVEN): Integer mult, float conversion                6.62
FPD (EVEN): DP floating point                             7.00

dumped pipeline stats
```

Net 1 running a large 52x52 network, without using logarithms in the main loop. This version gave the best simulator statistics (running a single SPE):

```
Performance Cycle count         398731839
Performance Instruction count   523330006 (440310869)
Performance CPI                 0.76 (0.91)

Branch instructions             3491717
Branch taken                    3093285
Branch not taken                398432

Hint instructions               2905620
Pipeline flushes                299404
SP operations (MADDs=2)         1140615960
DP operations (MADDs=2)         6

Contention at LS between Load/Store and Prefetch 61774968

Single cycle                                        202610329 ( 50.8%)
Dual cycle                                          118850270 ( 29.8%)
Nop cycle                                            13420584 (  3.4%)
Stall due to branch miss                              5259652 (  1.3%)
Stall due to prefetch miss                                  0 (  0.0%)
Stall due to dependency                              55468990 ( 13.9%)
Stall due to fp resource conflict                           0 (  0.0%)
Stall due to waiting for hint target                  2894541 (  0.7%)
Issue stalls due to pipe hazards                           12 (  0.0%)
Channel stall cycle                                    227461 (  0.1%)
SPU Initialization cycle                                    0 (  0.0%)
-----------------------------------------------------------------------
Total cycle                                         398731839 (100.0%)

Stall cycles due to dependency on each instruction class
 FX2        645281 (  1.2% of all dependency stalls)
 SHUF       2355098 (  4.2% of all dependency stalls)
 FX3        770284 (  1.4% of all dependency stalls)
 LS         3051216 (  5.5% of all dependency stalls)
 BR         89956 (  0.2% of all dependency stalls)
 SPR        25 (  0.0% of all dependency stalls)
 LNOP       0 (  0.0% of all dependency stalls)
 NOP        0 (  0.0% of all dependency stalls)
 FXB        0 (  0.0% of all dependency stalls)
 FP6        47629825 ( 85.9% of all dependency stalls)
 FP7        927297 (  1.7% of all dependency stalls)
 FPD        8 (  0.0% of all dependency stalls)

The number of used registers are 128, the used ratio is 100.00

Instruction Class                                  Cycles/Inst
-------------------------------------------------- ------------
FX2 (EVEN): Logical and integer arithmetic                1.88
SHUF (ODD): Shuffle, quad rotate/shift, mask              3.55
FX3 (EVEN): Element rotate/shift                          2.94
LS   (ODD): Load/store, hint                              1.08
BR   (ODD): Branch                                        4.04
SPR  (ODD): Channel and SPR moves                         3.13
LNOP (ODD): NOP                                           0.00
NOP (EVEN): NOP                                           0.00
FXB (EVEN): Special byte ops                              0.00
FP6 (EVEN): SP floating point                             2.20
FP7 (EVEN): Integer mult, float conversion                5.44
FPD (EVEN): DP floating point                             7.67

dumped pipeline stats
```

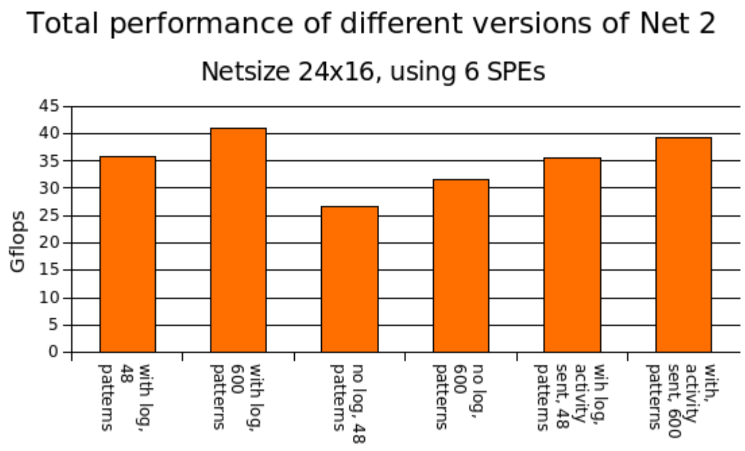


Figure 4.6. The results of Net 2 (storing the weight-matrix in the SPEs' local stores), net-size 24x16, is shown here.

4.2 Net 2

The results from running Net 2 on the PS3 and Blade, with no affinity enabled, were identical: The data-flow in the EIB-bus in all tests varied between 1–3 GB/s, far from peak. The data traffic to the main memory only consisted of pattern-transfers, which only amounted to 48 patterns of 24 hypercolumns (ca. 4.6 Kbyte) read and written — practically negligible.

The results for Net 2 varied slightly. This was probably due to the SPE-threads random allocation, suboptimal placement of jobs would result in a longer transport-path for the data from SPE to SPE, whereas the random allocation sometimes would result in an optimal placement. The variation in the result was slight, running 600 patterns with the default setting would lead to a variation in performance between 40.9–41.0 Gflops. While this is small, Net 1 did not show this variation.

4.2.1 Timed results of Net 2

The performance of Net 2 varied depending on the number of test-patterns. Because of this variation, the results in figure 4.6 are showed with two different numbers of test-patterns. The performance get diminishing returns with increased number of test-patterns, and the higher number is representative of a typical maximum performance.

As explained in section 4.1.1, tests with, or without using logarithms in the main loop are performed. In addition to this, tests when cycling the activities, or not cycling the activities are also performed. Not cycling the activity means that all SPEs store all the other SPEs' pattern-activity in their local store. This allows for smaller amounts of memory-transfers in the main-loop-part of the program. Cycling the activity on the other hand, means that in additional to sending forth the computation-results in the main loop, the activity of the pattern being tested is also sent. This allows for larger patterns to be tested, since more room in the local stores are freed up. However, the main loop contain more memory-transfers.

These results are shown in figure 4.6. All six SPEs are used in all tests.

The results of a larger net of size 24 hypercolumns, 20 minicolumns is shown in figure 4.7, in order to fit the weights of such a net in the local stores, the activity has to be sent around along with the main-loop results.

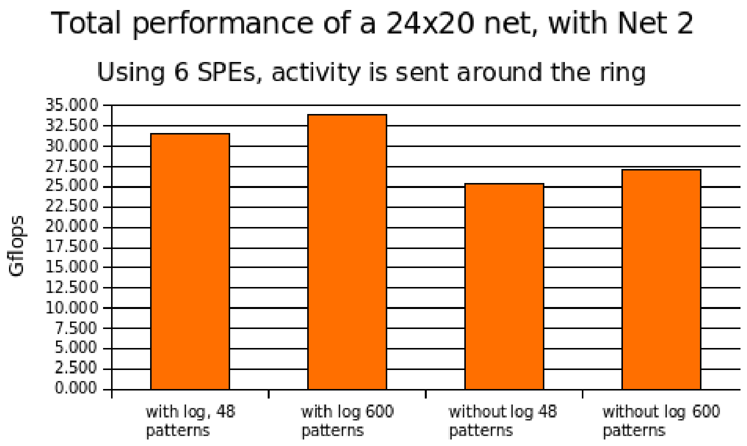


Figure 4.7. The results of Net 2 (storing the weight-matrix in the SPEs' local stores), net-size 24x20, is shown here.

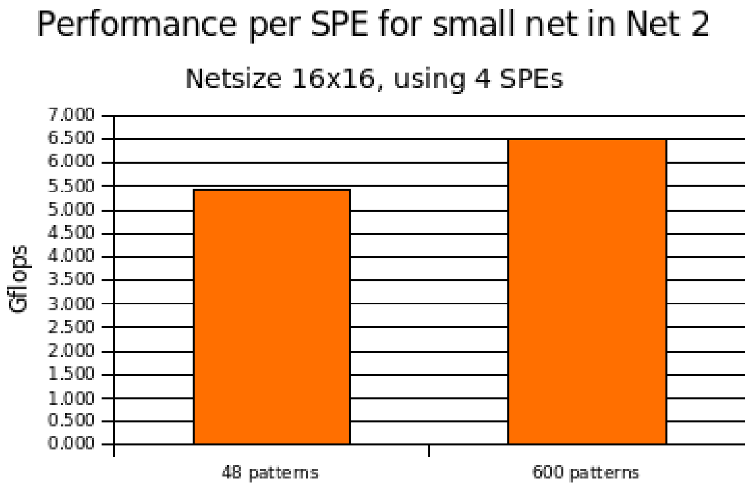


Figure 4.8. The results of Net 2 (storing the weight-matrix in the SPEs' local stores), smaller net-sizes, individual SPE-performance, is shown here.

The performance of the individual SPEs, running a smaller net of 16 hypercolumns, 16 mini-columns is shown in figure 4.8. Running 4 SPEs.

4.2.2 Systemsim statistics for Net 2

The section for instruction class counts has been edited to fit the page.

```
SPU DD3.0
***
Total Cycle count               35635665
Total Instruction count         43372995
Total CPI                       0.82
***
Performance Cycle count         35635665
Performance Instruction count   43372995 (40381603)
Performance CPI                 0.82 (0.88)

Branch instructions             444698
Branch taken                    343991
Branch not taken                100707

Hint instructions               292192
Pipeline flushes                48063
SP operations (MADDs=2)         81612112
DP operations (MADDs=2)         4

Contention at LS between Load/Store and Prefetch 4243635

Single cycle                                               16902069 ( 47.4%)
Dual cycle                                                 11739767 ( 32.9%)
Nop cycle                                                    396691 (  1.1%)
Stall due to branch miss                                     846415 (  2.4%)
Stall due to prefetch miss                                        0 (  0.0%)
Stall due to dependency                                     4582779 ( 12.9%)
Stall due to fp resource conflict                             82944 (  0.2%)
Stall due to waiting for hint target                         494216 (  1.4%)
Issue stalls due to pipe hazards                                 12 (  0.0%)
Channel stall cycle                                          590763 (  1.7%)
SPU Initialization cycle                                          9 (  0.0%)
-----------------------------------------------------------------------------
Total cycle                                                35635665 (100.0%)

Stall cycles due to dependency on each instruction class
 FX2        161395 (  3.5% of all dependency stalls)
 SHUF       380888 (  8.3% of all dependency stalls)
 FX3        152341 (  3.3% of all dependency stalls)
 LS         840385 ( 18.3% of all dependency stalls)
 BR         109982 (  2.4% of all dependency stalls)
 SPR        5758 (  0.1% of all dependency stalls)
 LNOP       0 (  0.0% of all dependency stalls)
 NOP        0 (  0.0% of all dependency stalls)
 FXB        0 (  0.0% of all dependency stalls)
 FP6        2615847 ( 57.1% of all dependency stalls)
 FP7        316177 (  6.9% of all dependency stalls)
 FPD        6 (  0.0% of all dependency stalls)

The number of used registers are 128, the used ratio is 100.00

Instruction Class                                 Cycles/Inst
-----------------------------------------------   ------------
FX2 (EVEN): Logical and integer arithmetic                1.51
SHUF (ODD): Shuffle, quad rotate/shift, mask              3.29
FX3 (EVEN): Element rotate/shift                          2.87
LS   (ODD): Load/store, hint                              1.30
BR   (ODD): Branch                                        3.90
SPR  (ODD): Channel and SPR moves                         2.86
LNOP (ODD): NOP                                           0.00
NOP (EVEN): NOP                                           0.00
FXB (EVEN): Special byte ops                              0.00
FP6 (EVEN): SP floating point                             1.69
FP7 (EVEN): Integer mult, float conversion                3.66
FPD (EVEN): DP floating point                             7.00

dumped pipeline stats
```

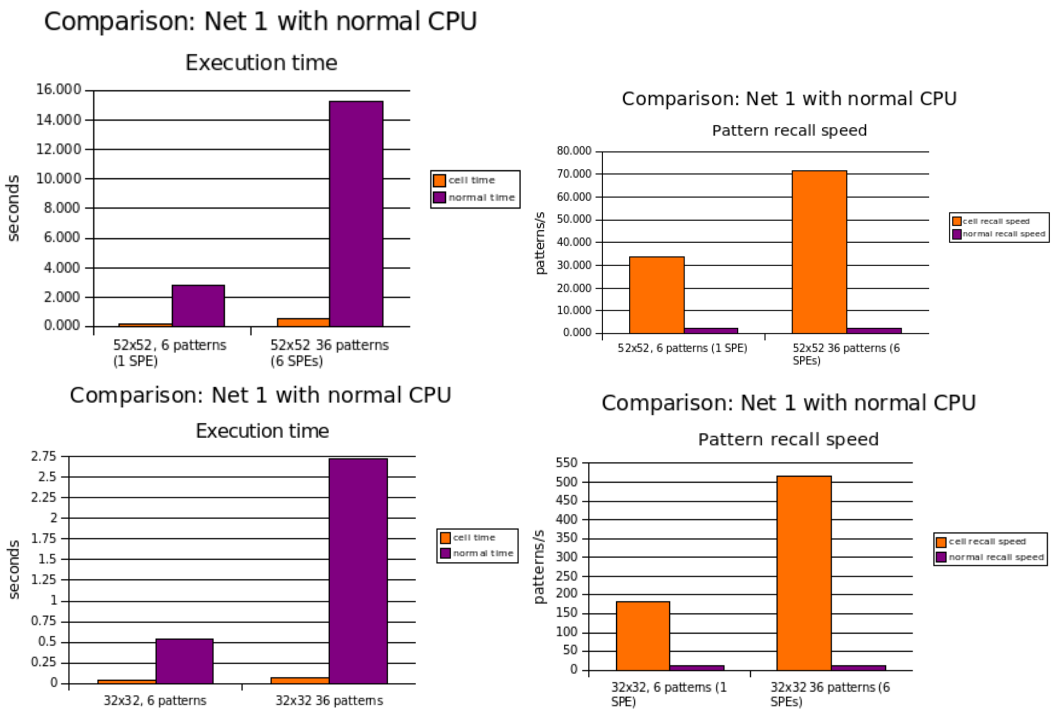


Figure 4.9. The result of Net 1, comparing the Playstation 3 to a normal CPU, is shown here.

4.3 Results from a conventional computer, and comparisons

The computer used as a "conventional" CPU was an Intel Pentium 4, clockfrequency 2.60 GHz For the comparison between conventional CPU and the Cell, the execution time and pattern-recall-speed were used.

Results for Net 1 is shown in figure 4.9. Using a single SPE and a netsize of 52x52, the Cell performed 15 times faster than a normal computer. Using all six SPEs, with the significant main memory bottle-neck that it entail, the Cell executed 30 times faster than the conventional CPU.

With a net-size of 32x32, the ratio was 18 times faster for a single SPE, and 39 times faster when running all SPEs.

The results against Net 2 is shown in figure 4.10. When running 48 patterns, the ratio was 79 times faster, and 113 times faster when recalling 600 patterns.

The results on Net 2 is as expected in comparison with the performance of Net 1. Net 1 fails to use all SPEs fully, but if it could do it, it should reach the the performance that Net 2 actually achieves, which is around 100 times faster than a conventinal CPU of 2.60 GHz.

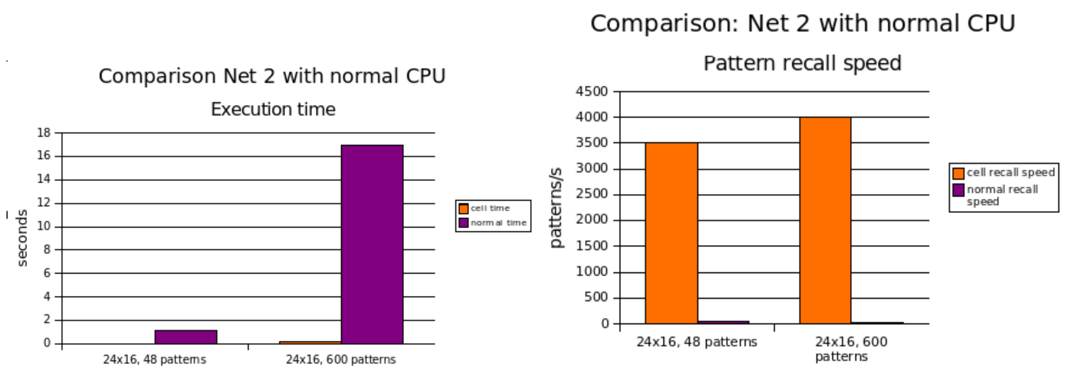


Figure 4.10. The result of Net 2, comparing the Playstation 3 to a normal CPU, is shown here.

Chapter 5

Conclusions and discussion

The two different implementations illustrated important matters of considerations when using the Cell-processor for neural networks, specifically the BCPNN-net.

Net 1 stored the weight-matrix in the main memory, allowing for the largest network possible. However, the limit of the bandwidth between the CPU and the main memory was quickly reached when using multiple SPEs, since the SPEs needed to stream more data than the XDR-memory could handle. Double-buffering data-transfers is a fundamental technique when developing for the Cell. It was not enough for this implementation though. Alternative workarounds for this could be considered: Increasing the numbers of calculations performed with a given chunk of the weight-matrix would probably be of use. With a piece of double-duffered data-set, you could for example process several pattern at a time, instead of one. This would lower the bandwidth-load to the main storage, since more calculations per memory-transfer would be performed. With large nets however, the patterns would have a hard time fitting in the local storage, which would entail more data conversions in order to fit everything. This would lower the individual performance of the SPEs, but it might imrove the overall performance with all SPEs able to perform relatively well.

Net 2 stored the weight-matrix distributed among the SPEs local stores. The performance was higher than that of Net 1. The reason for this was that the bandwidth of the element interconnect bus (the EIB connects the SPEs) is much higher than the bandwidth to the main memory – 200GB/s compared to 25GB/s. A limiting factor here was the structure of communication between the SPEs, something that could not be handled optimally with the PS3. The reason for this was that affinity, the functions that allows you to allocate workloads after the SPE's physical locations, is disabled on the PS3. This meant that the data-transfers did not take optimal routes. The threads automatically gets allocated an SPE. You can only hope that an optimal configuration gets allocated, where the circling of data in the "ring" is performed with minimal distance between each SPE. This is highly unlikely though. We were not able to test the program with affinity on the Cell Blade, since there was an error with the underlying operating system. Another drawback of this second implementation is in the small maximal size of the network. The differences in netsizes between net 1 & 2 is huge. Tests on net 1 running on a net of the same size as net 2 showed that despite the difference in data-transfer-sizes, net 2 fared better.

Why did Net 2's performance increase with larger numbers of test-patterns? Larger networks lead to the longer sequences of calculations in the main loop, which give opportunities for loop-unrolling that increases performance. An increased number of test-patterns, however, does not give more opportunities for loop-unrolling. The performance of Net 1, as expected, did not increase with increased number of test-patterns. Why then, did Net 2 get improved performance with more test-patterns? A simulator-run was made in order to see if the generated statistics would give the answer. It did not – the statistics were practically identical: same cycle-per-instruction-ratio, same dual-issue rate, and same dependency stall-amount. However, since data-traffic with the main storage is something that does not give change to the SPE-statistics, it could mean that data-traffic with the main storage flowed in a more efficient manner when there were more patterns to test. Why that would lead to better performance is still unclear though.

In order to use larger networks with the Net 2-implementation, one could possibly use sparse-matrix techniques or something similar, but the resulting increase of data-transformations, and other types of managing code, would probably lower individual SPE performance, and lower overall relative performance.

Results from Net 1 running the same network size as Net 2 showed that the memory bandwidth again severely limited performance. Performance-increase per added SPE is almost negligible when the bandwidth-limit is reached.

The maximum achieved performance for an individual SPE was 7.6 Gflops, achieved in Net 1, using a single SPE to run a 52x52-size network (see figure 4.2. If the case was that of comparing individual performance while running a *single* SPE, other configurations of Net 1 performed in the range of 5–7 Gflops, which was similar to the individual performance of the SPEs in Net 2. The cycle per instruction-count for individual comparison was in the 0.7-0.9 range for both versions. The method of memory-transfers was the factor that contrasted the two implementations. Net 2 was able to run all SPEs at maximum individual speed, while Net 1 became severely bogged down by the bottleneck of bandwidth to main storage, even when running small nets.

There are different levels of parallelization in the Cell, and all must be taken in consideration to achieve efficient code.

A good SIMDification strategy is important, as it in many ways determines your options for achieving high dual-issue rate, in loop unrolling, and data dependencies in load/store and float operations.

Distributing the workload in a good way on the SPEs is also of fundamental importance. BCPNN and other artificial neural networks often deal with large data sets, and can therefore be very intensive for the memory interface. In Net 1 it became apparent how easily the memory bandwidth with the main storage could be exceeded. Being aware of this limitations is important for efficient programming of artificial neural networks. The distribution of the weight-matrix to the SPEs in Net 2 proved to be a good way of dealing with the problem of overloading the memory bus with weight-transfers.

Insomniac Games had this to say about developing video games for the Playstation 3 (Almond et al., 2008):

> Do not hide the Cell BE architecture but exploit it instead. For a successful port to Cell BE:
>
> - Understand the architecture.
> - Understand the data: movement, dependencies, generation, usage (read, write, or read-write)
> - Do the hard work.
>
> Put more work on the SPE, less on the PPE. Do not view the SPE as a co-processor but rather view the PPE as a service provider for the SPE. Ban Scalar code on the SPE. Less PPE/SPE synchronization, use deferred updates, lock-free

synchronization and perform data-flow management as much as possible from the SPE.

In the case of our implementations of the BCPNN artificial neural network alghorithm, all work in the relaxation process was performed on the SPEs. Net 1 was a simple port of a scalar implementation, where the big weight-matrix was stored in the main memory. This turned out to be the less efficient solution. Net 2 used the Cell's architecture to minimize data traffic to the main storage. This was the faster solution, with the caveat of crippling the network size. Putting the two different implementations in contrast to each other, the points of the above become clear. Even though Net 1 certainly was easier to implement, and even though it seemed obvious as the better performer, the memory bandwidth became a severe bottleneck. The more Cell-specific implementation of Net 2 illustrated some of the processor's strengths, though the more complex parallelization required more work from the programmer.

The reasons peak performance was not achieved on net 1:

- The bottleneck of 25 GB/s to the main storage effectively became a wall.
- Non-trivial calculations such as logarithms and exponential functions do not optimally use the Cell hardware.
- Some scalar operations.
- Some performance lost due to branching.
- Data transformations.
- Data-dependencies that were hard to avoid due to the nature of the BCPNN-formulas.

The reasons peak performance was not achieved on net 2:

- Need for synchronizing the SPEs with each other, resulting in some waiting.
- Sub-optimal placing of the workloads, due to the lack of affinity-functionality in the PS3.
- Non-trivial calculations such as logarithms and exponential functions do not optimally use the Cell hardware.
- Some scalar operations.
- Some performance lost due to branching.
- Data transformations.
- Data-dependencies that were hard to avoid due to the nature of the BCPNN-formulas.

Given time, these issues could probably be further dealt with.

In comparing the performance with a conventional Intel Pentium 4 CPU 2.60 GHz, the Cell fared faster by around 100 times if all SPEs in the Playstation 3 was able to perform in the 6-7 Gflop range. These tests were unfair considering the slower clock frequency, and the existence of dual/quad-core CPUs available for common desktops today. But even if we would simply scale up the results according to clock frequency, and scale up to "Quad"-core, (this is naive, since the issues of memory-access would be relevant in the implementation, and multicore-processors also suffer from the bottleneck with main memory), the results of these tests would show a performance difference in favor of the Cell by a ratio of around 23:1. This is also assuming that the platform for the Cell used for comparison is the Playstation 3.

The two most important points for achieving good performance for the Cell was found to be efficient management of memory, and of the SPU's dual-pipeline. Avoiding the memory bottlenecks, and assuring good flow of data is of fundamental importance. The dual-pipeline of the SPU can very easily perform poorly, which makes it important to make an effort to assure high dual-issue rate. Avoiding the different sorts of stalls, branch-misses, data-dependencies, etc, is also of fundamental importance for achieving good performance.

A tool for tweaking your code for efficient use of the SPU pipeline is available in "Cmpware SPU Scheduling Tool". This tool was not tested in this work, but it might make it easier to deal with the difficulty of achieving optimal SPU-pipeline usage.

5.1 Future work

Future work on this project could include modularization of the implementations, which would put the whole thing to the test in a more general, and practical perspective. The performance of the specialized code here would probably be compromised somewhat, and it would be interesting to see what kinds of restrictions that would crop up. Even though hand-crafted assembly was not used in this work, implementing optimized assembly-functions in the main loops might lead to increased performance. The usefulness of other tools that were not used in this work is also worth investigating, tools for workload partitioning such as DaCS, tools for easier porting of existing code such as the xlc single source compiler, and the various other profiling tools that was left unused here.

Bibliography

Almond, C., Arevalo, A., Matinata, R. M., Pandian, M., Peri, E., Ruby, K., and Thomas, F. (2008). *Programming the Cell Broadband Engine Examples and Best Practices*. IBM Corp., draft edition.

Bartlett, J. (2007). *Programming high-performance applications on the Cell BE processor series*. IBM Corp.

Brokenshire, D. A. (2006). Maximizing the power of the cell broadband engine processor: 25 tips to optimal application performance. url.

Buttari, A., Luszczek, P., Kurzak, J., Dongarra, J., and Bosilca, G. (2007). *A rough Guide to Scientific Computing On the Playstation 3*. Technical report, Innovative Computing Laboratory, University of Knoxville.

Chow, A. C., Fossum, G. C., and Brokenshire, D. A. (2005). *A Programming Example: Large FFT on the Cell Broadband Engine*. IBM.

Djurfeldt, M., Johansson, C., Örjan Ekeberg, Rehn, M., Lundqvist, M., and Lansner, A. (2005). *Massively parallel simulation of brain-scale neuronal network models*. Technical report, Computational Biology and Neurocomputing, School of Computer Science and Communication Stockholm.

Easton, J., Meents, I., Stephan, O., Zisgen, H., and Kato, S. (2007). *Porting Financial Markets Applications to the Cell Broadband Engine Architecture*. Technical report, IBM Systems and Technology Group.

Eichenberger, A. E., O'Brien, J. K., O'Brien, K. M., Wu, P., Chen, T., Oden, P. H., Prener, D. A., Shepherd, J. C., So, B., Sura, Z., Wang, A., Zhang, T., Zhao, P., Gschwind, M. K., Archambault, R., Gao, Y., and Koo, R. (2006). Using advanced compiler technology to exploit the performance of the cell broadband enginetm architecture. *IBM Syst. J.*, 45(1):59–84.

Fabritiis, G. D. (2007). Performance of the cell processor for biomolecular simulations. *Computer Physics Communications*, 176(11-12):660–664.

Gschwind, M. (2007). The cell broadband engine: exploiting multiple levels of parallelism in a chip multiprocessor. *Int. J. Parallel Program.*, 35(3):233–262.

Hackenberg, D. (2007). Fast matrix multiplication on cell (smp) systems. `http://tu-dresden.de/die_tu_dresden/zentrale_einrichtungen/zih/forschung/architektur_und_leistungsanalyse_von_hochleistungsrechnern/cell/`. Last visited June 26, 2008.

Hammarlund, P. and Örjan Ekeberg (1998). Large neural network simulations on multiple hardware platforms. *Journal of Computational Neuroscience*, 5(4):443–459.

Holst, A. (1997). *The Use of a Bayesian Neural Network Model for Classification Tasks*. PhD thesis, Stockholm University.

IBM (2007a). *C/C++ Language Extensions for Cell Broadband Engine Architecture Version 2.4*. IBM Corp. and Sony Computer Entertainment Inc. and Toshiba corp.

IBM (2007b). *Cell Broadband Engine Programming Handbook*. IBM Corp. and Sony Computer Entertainment Inc. and Toshiba corp.

IBM (2007c). *Cell Broadband Engine Programming Tutorial Version 2.1*. IBM Corp. and Sony Computer Entertainment Inc. and Toshiba corp.

IBM (2007d). *Cell Broadband Engine Programming Tutorial Version 3.0*. IBM Corp. and Sony Computer Entertainment Inc. and Toshiba corp.

IBM (2007e). *Example Library API Reference Version 3.0*. IBM Corp. and Sony Computer Entertainment Inc. and Toshiba corp.

IBM (2007f). *Installation Guide for the SDK for Multicore Acceleration v3.0*. IBM Corp. and Sony Computer Entertainment Inc. and Toshiba corp.

IBM (2007g). *Software Development Kit 2.1 Installation Guide Version 2.1*. IBM Corp. and Sony Computer Entertainment Inc. and Toshiba corp.

IBM (2007h). *SPU Assembly Language Specification version 1.6*. IBM Corp. and Sony Computer Entertainment Inc. and Toshiba corp.

IBM (2007i). *SPU Timer Library Programmer's Guide and API Reference version 3.0*. IBM Corp. and Sony Computer Entertainment Inc. and Toshiba corp.

Johansson, C. and Lansner, A. (2001). *A Parallel Implementation of a Bayesian Neural Network with Hypercolumns*. Technical report, Department of Numerical Analysis and Computer Science, Royal Institute of Technology.

Johansson, C. and Lansner, A. (2006). *BioADIT, LNCS*, volume 3853, chapter Attractor Memory with Self-organizing Input, pages 265–280. Springer-Verlag, Berlin Heidelberg.

Kistler, M., Perrone, M., and Petrini, F. (2006). Cell multiprocessor communication network: Built for speed. *IEEE Micro*, 26(3):10–23.

Knobe, K., Lukas, J. D., and Guy L. Stelle, J. (1990). Data optimization: allocation of arrays to reduce communication on simd machines. *J. Parallel Distrib. Comput.*, 8(2):102–118.

Kurzak, J. and Buttari, A. (2007). Howto: Huge tlb pages on ps3 linux. url.

Liu, W. and Schmidt, B. (2003). Parallel design pattern for computational biology and scientific computing applications. *cluster*, 00:456.

Maass, W. (1999). *Pulsed neural networks*, chapter Computing with spiking neurons, pages 55–85. MIT Press, Cambridge, MA, USA.

Minor, B., Fossum, G., and To, V. (2005). *Terrain Rendering Engine (TRE): Cell Broadband Engine Optimized Real-time Ray-caster*. IBM.

Misra, M. (1997). Parallel environments for implementing neural networks. *Neural Computing Surveys*, 1:48–60.

Petrini, F., Fossum, G., Fernández, J., Varbanescu, A. L., Kistler, M., and Perrone, M. (2007). Multicore surprises: Lessons learned from optimizing sweep3d on the cell broadband engine. In *IPDPS2007*, volume CDROM, pages 1–10. IEEE Society.

Ruckert, U. (2002). Ulsi architectures for artificial neural networks. *IEEE Micro*, 22(3):10–19.

Sachdeva, V., Kistler, M., Speight, W. E., and Tzeng, T.-H. K. (2007). Exploring the viability of the cell broadband engine for bioinformatics applications. In *IPDPS*, pages 1–8. IEEE.

Sandberg, A. (2003). *Bayesian Attractor Neural Network Models of Memory*, chapter Bayesian Confidence Propagation Neural Networks (BCPNN), pages 39–58. Universitetsservice US AB.

Seiffert, U. (2004). Artificial neural networks on massively parallel computer hardware. *Neurocomputing*, 57:135–150.

Strey, A. (25-27 Oct 1993). Implementation of large neural associative memories by massively parallel array processors. *Application-Specific Array Processors, 1993. Proceedings., International Conference on*, pages 357–368.

Vreeken, J. (2002). *Spiking Neural Networks, an Introduction.* Technical report, Institute for Information and Computing Sciences, Utrecht University.

Williams, S., Shalf, J., Oliker, L., Kamil, S., Husbands, P., and Yelick, K. (2006). The potential of the cell processor for scientific computing. In *CF '06: Proceedings of the 3rd conference on Computing frontiers*, pages 9–20, New York, NY, USA. ACM.

TRITA-CSC-E 2008:079
ISRN-KTH/CSC/E--08/079--SE
ISSN-1653-5715