Università degli Studi di Ferrara

DOTTORATO DI RICERCA IN MATEMATICA E INFORMATICA

Ciclo XXI

Coordinatrice: Zanghirati Luisa

Monte Carlo Simulations of Spin Glasses on Cell Broadband Engine

Settore Scientifico Disciplinare: INF/01

Dottorando:

Dott. Belletti Francesco

Tutori:

Dott. Schifano Sebastiano Fabio

Prof. Tripiccione Raffaele

Anni 2006/2008

Abstract

Several large-scale computational scientific problems require high-end computing systems to be solved. In the recent years, design of multi-core architectures delivers on a single chip tens or hundreds Gflops of peak computing performance, with high power dissipation efficiency, and it makes available computational power previously available only on high-end multi-processor systems.

The aim of this Ph.D. thesis is to study the capability of multi-core processors for scientific programming, analyzing sustained performance, issues related to multi-core programming, data distribution, synchronization, in order to define a set of guideline rules to optimize scientific applications for this class of architectures.

As an example of multi-core processor, we consider the Cell Broadband Engine (CBE), developed by Sony, IBM and Toshiba. The CBE is one of the most powerful multi-core CPU current available, integrating eight cores and delivering a peak performance of 200 Gflops in single precision and 100 Gflops in double precision. As case of study, we analyze the performances of CBE for Monte Carlo simulations of the Edwards-Anderson Spin Glass model, a paradigm in theoretical and condensed matter physics, used to describe complex systems characterized by phase transitions (such as the para-ferro transition in magnets) or model "frustrated" dynamics.

We descrive several strategies for the distribution of data set among on-chip and off-chip memories and propose analytic models to find out the balance between computational and memory access time as a function of both algorithmic and architectural parameters. We use the analytic models to set the parameters of the algorithm, like for example size of data structures and scheduling of operations, to optimize execution of Monte Carlo spin glass simulations on the CBE architecture.

Contents

Introduction

Several large-scale computational scientific problems require high-end computing systems to be solved, or in some cases the development of "application-driven" machines, custom-designed computing systems optimized for the resolution of a specific problem.

The huge amount of computational requirements and data to be processed have made until the recent past years very hard, and sometimes impossible, the use of commercial High Performance Computer (HPC) based on available standard computing technology.

Typical examples of such applications comes from several scientific fields like Earth Sciences (weather forecast, oceanography), Atrophysics (study of large-scale structures), Quantum Physics, Fluidodynamics, Biology, Molecular Dynamics [1, 2]. The request of an high computational power raises from the large scale of the events that have to be studied, and by the accuracy of the results to be achieved. Some of such applications are defined as *Grand Challenge Problems*, which means that with the actual technologies they may require *years* to be solved [3].

In the past, to solve this kind of problems two main categoires of computing systems have been used: general-purpose HPC commercial systems and special-purpose machines usually developed by research university groups. For example, for applications like weather forecast or biology, computing systems like the Cray machines and the IBM Power parallel systems have been extensively used with reasonably performance. General purpose systems are designed to be used for a wide range of applications, even for irregular applications where computing and communication patterns can change dynamically.

For other interesting scientific application this approach has not been sufficient and it required the design of specific computing architectures optimized for a particular application. A typical example of such applications is the Lattice Quantum Chromodynamics (LQCD) which has triggered the development of several generations of massively-parallel machines like the APE family in Europe [4] and the Columbia University systems, like QCDOC [5] or QCDSP [6], in the United States. The case of LQCD is particular interesting because the experience of the Columbia University has triggered the development of the IBM BlueGene systems, currently the most used system from the LQCD community. Other interesting examples are

the GRAPE [7] system, developed in Japan for studying large gravitational systems, and Spin Glass simulations for which two generations of special-pourpose computing systems have been developed: SUE [8] and Janus [9].

The need of developing special-pourpose parallel machines which required the design of the processor, was mainly due to the lack of commercial CPUs in terms of performance/Watt and performance/cost ratios. Moreover, to allow the system to scale in the strong regime, that is to scale among several processors keeping the volume of the problem constant, it was also necessary to develop the interconnection network. In most cases, the design of special purpose systems has been tuned carefully to meet the computing requirements of the algorithms [10].

In the recent years, thanks to the raise of multi-cores architectures, a single chip has tens or thousand of Gflops of peak performance, with an high efficiency in power dissipation. This kind of processors are composed by cores strongly optimized for SIMD floating-point intensive applications. They also have on-chip memory, as well as efficient (in terms of latency and bandwidth) inter-core communication mechanisms, that reduce the data-access bottlenecks that may be expected when a very large number of operations is performed concurrently. The efficiency of inter-core communications makes a multi-core processor faster than a multi-processor system with an equivalent number of computing elements, as the low latency and the high bandwith allows a strict coupling between the cores embedded in the same chip. In this way it is possible to exploit a finer grain parallelism.

Nowadays there are essentially two different types of multi-core machine: processors with *few* (2 or 4) general purpose cores and a large amount of on-chip memory (4 or 8 megabytes), and graphics processing units (GPU) that have hundreds of special pourpose cores (although they are not independent unit and the naming is quite ambiguous) and a hierarchy of fast on-chip memory optimized for graphical applications. Multi-core systems make available in a single chip computational powers that previously were achieved only by HPC or special pourpose system, and offer high-speed connection channels that promise to allows an efficient use of the available parallelism. In effect, inter-core communications are more efficient in a multi-core processor than in a multi-processor system. Although parallelism were not largely available or well exploited in commodity processors, it is well known in HPC field, so the experience matured in the last decades can be applied to multi-core processors. For all these reasons multi-core processors are very interesting for scientific computing.

However, the multi-core systems require a non trivial effort in the development of applications, because an increase of performance can be expected only if the parallelism is explicitly exploited. Until few years ago it was sufficient to wait for a faster single-core processor to obtain a significant speedup. Nowdays compiler technology not able to generate efficient parallel core, and the parallelization of the algorithm and the distribution of data among the cores, ha to be done carefully

tuned by the user.

The Cell Broadband Engine, developed by IBM, Sony and Toshiba, is one of the first available multi-core processor, and it is a rare example of an heterogeneous multi-core: it has a general purpose core based on PowerPC architecture and eighth SIMD special purpose cores designed to achieve in computational-heavy workloads. All the cores are connected together by an high-speed bus, and data transfers are performed independently of computation by dedicated DMA engines. A peculiarity of Cell is that the on-chip memory is software controlled and in general there are no hardware for out-of-order execution or branch prediction: the computational core have been kept as simple as possible, in order to be able to put an high number of cores in a single chip and to achieve an high clock speed (3.2 GHz). As a consequence, to achieve high performance it is necessary to develop software that (i) take advantage of parallelism (due to the nine SIMD cores) (ii) is well optimized (because hardware do not help) and (iii) explicitly manage data transfers. The peak performance of Cell processor is greater than 200 Gflops in SP (AND 100 in DP), that are an order of magnitude greater than those achievable few year ago. Moreover, it has a relatively low power dissipation (80W), so in the optic of scientific programming it is interesting as a basic block for both small cluster of 16-64 processor and for the development of massively-parallel computers.

One of the most challenging scientific problems from the the computational point of view is the simulation of *spin models*. Spin models are relevant in several areas of condensed matter and high-energy physics. They describe systems characterized by phase transitions (such as the para-ferro transition in magnets) or model "frustrated" dynamics, which appears when the complex structure of the energy landscape of the system makes the approach to the equilibrium state very slow. These systems were extensively studied in the last two decades and are considered paradigmatic examples of complexity. They have been applied to a large set of problems, such as quantitative biology, optimization, economics modeling, social studies [11, 12, 13].

Spin glass models are defined on discrete, finite-connectivity regular grids (e.g., 3-D lattices, when modeling experiments on real magnetic samples) and are usually studied via Monte Carlo simulations. The dynamical variables of the systems are "spins", that are located at the edges of the lattice and assume a discrete and finite (usually small) set of possible values. State-of-the-art simulations require at least 10^{10} Monte Carlo updates of the full lattice, and have to be repeated on hundreds or thousands of different instantiations, called *samples*, and each sample must be simulated more than once (*replicas*), as most properties of the model are encoded in the correlation between independent histories of the same system [14]. For example, a 3-D system of linear size 80 requires at least $10^{17} \cdots 10^{18}$ spin updates, a major computational challenge.

Several features of the relevant algorithms can be exploited to handle the computational problem effectively. First of all, computational kernels have a large and

easily-identified degree of available parallelism, associated to the large number of samples and replicas that have to be independently simulated (referred to in spin glass jargon as asynchronous parallelism). However, this kind of parallelism can be managed running concurrently different instances of the problem on a set of machines, so it is not very interesting. For each given sample or replica the update of up to one half of all spins can be executed in parallel (synchronous parallelism), if enough computational resources are available. While asynchronous parallelism can be obviously exploited by farming out the overall computation to independent processors, the large available synchronous parallelism is poorly exploited by traditional processor architectures, due to the difficulty to process single bit variables.

On of the major issue is the data representation. Spin variables can be coded on short words, using a small number of bits (in the simplest models, spins are two-valued, so they can be encoded by just one bit). The typical Monte Carlo simulation updates all spins in the lattice in a regular sequence. Each update in turn involves in most cases a sequence of logical operations on bit-valued variables, as opposed to the long, integer or floating-point variables for which virtually each processor is optimized. For this reasons, in the last two decades several application-driven machines have been developed, strongly focused for spin glass simulations [8]. In recent years, this approach has been based on FPGAs, on which a very large number of processing cores each core being a spin-update engine can be easily implemented [15].

However, spin glass simulations have many other features that make an efficient implementation on parallel machines relatively easy to achieve:

- the computation can be easily partitioned among many computing cores: the whole lattice can be divided in smaller equally-sized sub-lattices, and each one can be assigned to a different core
- *Single Instruction Multiple Data* (SIMD) parallelism can be exploited to update in parallel spins of a system, and in particular for computing random numbers necessary by the Monte Carlo procedure
- the large data-words (16 bytes) available in modern processors can be used efficiently thanks to multispin coding techniques, that allow to use all the available bit to represent useful data
- data access patterns are regular and predictable, allowing data-prefetching to avoid stalls of the processor
- only nearest-neighbor are required

While until few year ago the above underlined features would be exploited only on HPC system or on clusters of commodity processors, recent developments in

multi-core or many-core processor architectures make an high level of parallelism available on commodity processors. Spin glass simulations can take advantage of multi-core processors because it is a intrinsically parallel application, and as long as the spins are two-valued, a large amount of them can be embedded in a single data word, and there is a high chance that relevant lattice (e.g. lattice of side L=64) can be completely stored in local memories. With previous architectures, where on-chip memories were small and inter-core communications were slow, this was not possible.

The level of exploitable parallelisms on multi-core processors will not probably stretch to the level of custom engines, where more then thousands spins are updated concurrently, however the much higher clock frequency of a state-of-the-art processor, as compared to an FPGA-based engine, may substantially close the performance gap.

The aim of my Ph.D. thesis is to study the capability of multi-core processor in the context of scientific programming and in the perspective for the development of massively parallel systems able to solve Grand Challenges Problems. For this purpose, I've studied one specific multi-core processor, the Cell Broadband Engine. It is one of the first multi-core and today it is still the commercial processor with the highest number of cores. To evaluate its capability I've used as a test bench a scientific problem, the spin glass simulations, that it is very demanding from the computational point of view but that is also intrinsically parallel. The step required to evaluate processor are (i) to determine which absolute performance are achievable for spin glass with the CBE processor, (ii) to evaluate the limits of the architecture and the amount of required programming effort, and (iii) to collect a set of techniques and strategies that allows to efficiently exploit the computational power of CBE. The experience with this processor should allow to extrapolate a more general view on the multi-core world: which are the benefits that they can grant but also which challenge they offer to scientists and developers in order to exploit all their capabilities.

The thesis is organized as follows. The first chapter describe the multi-core world, analyzing the reasons behind the advent of multi-core architectures, proposing a survey of the actual situation, and illustrating the problems that have to be resolved in the next years. The second chapter introduces the Edward-Anderson spin glass model and the Metropolis algorithm that is used to study the model. A technique for the efficient use of "large" data-words in Metropolis algorithm, called multispin coding, is then described in details, as it represent the computational kernel executed on each core. Chapter 3 discuss the implementation of spin glass simulations on an abstract multi-core architecture (that is a simplification of Cell), emphasizing the issues the distribution of data set among main and local memories. In particular balance equations predict the behavior of an ideal algorithm as a function of parameters: the number of cores and the lattice linear size. The bal-

ance equations depends on both algorithmic parameters (the size of the problem, the SIMD-granularity) and on architectural parparametershe number of available cores and the bandwidth of interconnection bus and memory). They are used to determine the optimal sizes of the problem for which the computational time is not dominated by the data access time, so that computational resources are efficiently exploited. Chapter 4 describe the implementation of spin glass simulation on Cell, analyzing the data layout issue, describing an efficient implementation of random number generation and spin update code, with the support of many SIMD-granularity. Moreover, various estimations of the performance of the computational code are proposed, bases both on static analysis of assembler code and on run-time measurements. The estimations are then applied to the balance equations determine the sizes of the problems and the number of cores for which the global time is not dominated by data transfers. Finally, chapter 5 shows the performance of the implementations of the programs. In particular, the theoric estimations are compared with the real performance, analyzing the overhead associated to data transfers and synchronization between cores.

Chapter 1

Multi-core Architectures

The computer world is living an epochal change, as multi-core architectures have become widely available. In this chapter the reasons that have caused the world to turn multi-core will be described and analyzed. Later, we will introdue the currently available multi-core architectures that promise to bring benefits in the fields of High Performance Computing and more in general scientific programming. We will emphasize their differences and similarities . Finally, the expected future evolution will be discussed.

1.1 The Raise of Multi-Cores

According to Moore's Law, in the last 30 years the density of microelectronic devices in a chip has doubled every 18 months (Figure 1.1).

Thanks to Moore's Law, for many years technological improvements were enough to obtain better performance, due to the the increase of frequency associated to the reduction of the size of transistors. *Technological* improvements were not the only way to go faster, as there was also an *architectural* development. The *instruction level parallelism* (ILP) in the past has taken advantage of the incrementing number of circuits that was possible to put in a single chip. Successfull techniques as pipelining, superscalar processors, out-of-order execution, register renaming, speculative execution and VLIW architectures were largely adopted. At the same time, cache hierarchies have been used to mitigate the difference of speed between external memories and the processor (the Memory Wall [16]), that was becoming larger and larger. Both technological and architectural developments lead to the the improvements of computing power shown Figure 1.2.

In the past, parallelization an exception : to increase the performance of a program it was fairly more convenient to wait for a new faster processor than trying to parallelize the code. When the main concern was the throughput (in terms of number of task completed for unit of time), a solution was to adopt systems with

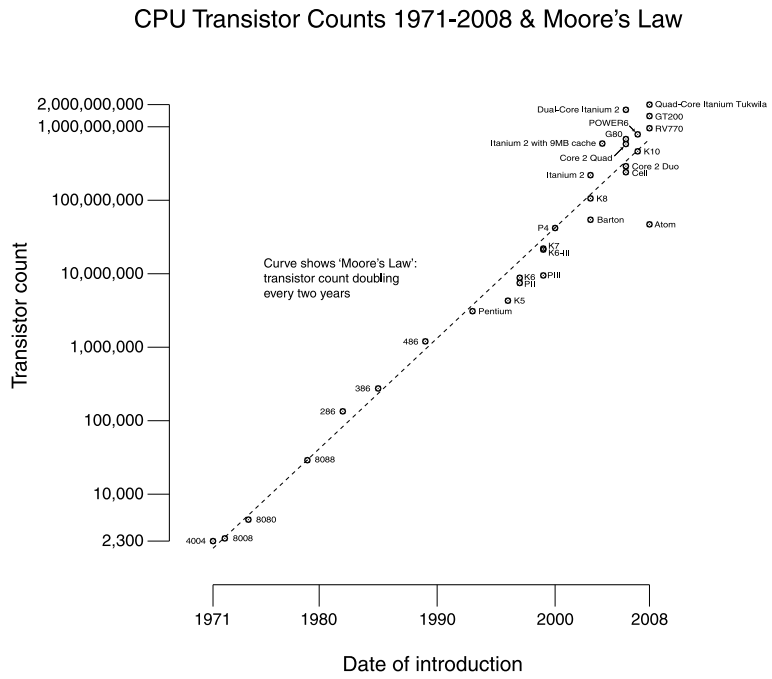

Figure 1.1: The trend of the density of microelectronic devices inside a chip in the last decades..

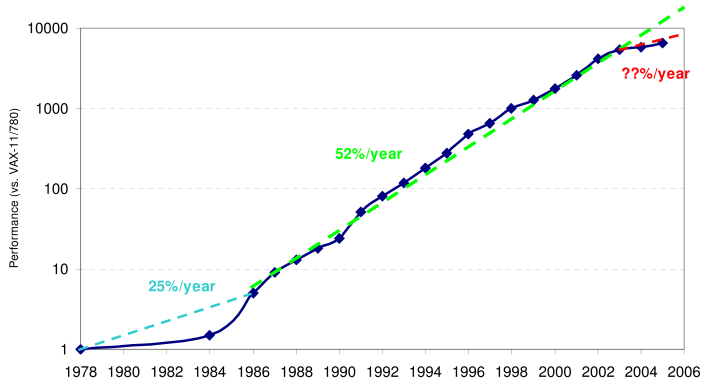

Figure 1.2: The improvements of computing power of processors in the last 30 years (This figure is Figure 1.1 in [17].

multiple CPUs, to increase the number of tasks that could be processed concurrently.

Nonetheless, in HPC parallelism has always been widely exploited. During the Eighties the HPC world was dominated by systems composed by few (four to sixteen) vector processors (although the first vector processors appeared in the Seventies) while the early Nineties saw the rise of Multiprocessors (MPP). However , until it was easy to get more performance with a single processor, data and processor parallelization was a concern only to specific fields, like high performance scientific programming.

Both technological and architectural developments were aimed to get a better performance with a single thread of execution, and both have now reached their limits, or it is too expensive to further develop them. As a consequence parallelization at each level seems the only viable way to continue to improve performance, as very little improvements in serial performance of general purpose processors are expected. To increase the performance of a processor has become harder and harder for several causes:

- the *Power Wall*
- the *Memory Wall*
- the *ILP Wall*

Physical limits of semiconductor-based microelectronics have become quite critical under various aspects. Power dissipation is proportional to the clock frequency, and as a consequence there is a natural limit to clock speed. More precisely, power consumption typically rises as the square or cube of the frequency. With current cooling systems and material technology, a significative increment of frequency is not to be expected.

The increase of speed of processors was not matched by an equal increase of the speed of memory. As a consequence, due to latency and bandwidth, it is difficult to give the processor enough data to exploit its computational power. Usually this kind of problem is mitigated with the use a hierarchy of caches, but actually the cache occupies the higher fraction of a die and is responsible of a large amount of heat dissipation. So, the Memory Wall is also one of the major responsible of the huge power dissipation of modern processors.

The techniques for ILP exploitation are effective if the code is predictable and if enough instructions are submitted to the processor. Both code and data suffer from the Memory Wall bottleneck. Moreover, ILP require a super-linear increase of CPU complexity and associated heat dissipation without granting linear speedup.

To summarize, power consumption rises as the square of cube of the frequency, while the expected speedup is less than linear, so increasing the frequency is no longer convenient. A more aggressive exploitation of ILP would require more complex

hardware and as a consequence the power consumption is too heavy with respect to the to achievable benefits. Moreover, feeding the processor with data is difficult due to memory latency, that can be mitigated with caches, that also requires more circuits and improve heat dissipation.

Given that the increase of the frequency or complicated ILP logic are no longer a viable solution, possible solutions to obtain more performance are: production of smaller chips, increase the amount of cache, addition of more cores or improvement of memory bandwidth [18]. To balance these different options is not a trivial problem and each CPU developer is proposing its own solution. Generally, the number of cores have been incremented, and at the end of 2008 quad-core processors are the standard and architectures with more cores are also available.

Typically the clock speed increase has been stopped, and in many cases it is slower than those of the last uni-core CPUs. Commodity processors also have adopted power saving technologies that in the past was exclusive of mobile and embedded processor. Also the ILP hardware has been subject to minor refinements and in most cases has been heavily simplified or removed (as in CBE and GPUs). In general, there is more interest in data-level parallelism (the rise of SIMD ISAs) and, as long as there are more cores, it can be exploited at a higher level of abstraction. The memory wall is still a problem: memory is slowly becoming faster in terms of bandwidth but, as there are more processor to feed, the problem can be worse than before. Luckily, it is possible to put large on-chip caches that can help to mitigate this problem. However, as have been observed in [19, 18, 20, 21] this problem is expected to become worse and wors in next years, when there will be hundreds or thousands of cores inside a single chip.

Multi-core processors have to be appropriately programmed to guarantee performance improvements, and as a amtter of facts many applications have to be reengineered and restructured as parallel programs. First of all, all the available parallelism has to be exposed. Some applications are intrinsically parallel, while other are less likely to take advantage of concurrency. The critical point is the relevance of the parallelizable fractions of the program into respect of global execution time [22]. The availability of many computational resources cannot be exploited if an application is memory bound. In this case the bottleneck is represented by data transfer channels, between computational cores and main memory of between cores. From this point of view, multi-core processor have an interconnection infrastructure that is more efficient than those available in many multi-processor machines, so is more probable that computational resources can be efficiently used.

1.2 Toward Many-Cores

It is expected that the number of cores inside a chip will double at each generation and that in few years processors with 128 or more cores will be available. Processor with hundreds or thousands of cores are classified as "many-core" [19]. Some special purpose processors have already tents of cores, like the Cisco QuantumFlow 40-cores newtwork processor (able to support up to 160 threads [23]) and, although this interpretation is ambiguous, GPUs have already hundreds of independent computational units.

Although many-cores do not still exists, it is possible to foresee that Memory Wall will be the primary concern [24, 25, 18, 26, 27]. As reported in [28], traditionally the CPU speed increases by 70 percent per year, while the bandwidth increases by 25 percent per year, and the memory latency shows the lowest improvement with a modest decrease of the 5 percent. With the actual "few-core" machine data access is a very important issues, and in many cases the computational resources are not used efficiently. It the next year will be very important to find new strategies to face the Memory Wall issue.

1.3 Multi-Core Architectures in 2008

At the end of 2008, multi-core architectures have become the standard. The two extremes of the spectrum are the commodity processors from Intel and AMD on one side, and the GPUs of NVIDIA and AMD (that has acquired ATI) on the other side. In the middle, there are processors like the Cell Broadband Engine and many other processors aimed at more specialized targets.

Commodity CPUs are largely based on the old single-core processor, which are replicated two of four times inside a single chip. A major feature is the presence of large shared caches between the cores, that are also connected by high-speed on-chip buses. The cores inside a processors are *homogeneous* and, although their ISA supports SIMD extensions, in principle they are multi-pourpose cores. While in HPC context it is vital to be able to use all the cores in parallel to gain a speed-up for a single program, theis chips are largely used as desktop or server processor, where multiple tasks are run in parallel, so distributing processes among the cores is sufficient to assure a performance gain.

At the other extreme, GPUs are based on arrays of simpler cores. Although the definition of "core" in this context is quite ambiguous, the amount of exploitable parallelism inside a GPU is much greater than those found in a multi-core CPU. To be used efficiently, a GPU has to run a program that has been subdivided into many subtasks. A basic concept of modern GPU is to run more threads than available cores, in order to mitigate the inevitable stalls intrinsic in each sub-task. To succeed in this purpose, in a GPU the context switch is very inexpensive. However, GPUs

are highly specialized for compute-intensive and highly parallel applications so, while they can deliver impressive performance in their specific area, they are not general purpose processors.

GPU are aimed at a *fine-grain* parallelism, as the single core is very simple and it is not flexible as a general pourpose processor. Multi-core CPU are instead oriented to *coarse-grain* parallelism, as a single core is capable to run complete applications (that it is not possible with the core of a GPU) and in some fields the presence of multiple cores is simply exploited to improve the throughput of the system, intended as the number of jobs completed for unit of time. At the other extreme, to obtain high performance in a GPU it is necessary to subdivide a single job into many threads that can are executed concurrently.

These two different philosophies are converging: in recent years specific-purpose extensions were added to general purpose processors, while GPUs are trying to become more general purpose in order to be used in scientific and multimedia programming, and not only in 3-D graphics.

CPUs and GPUs also differs from the point of view of memory hierarchies. While CPU are still based on caches to try to mitigate the Memory Wall, GPUs are based on customized an special pourpose on-chip memories. Moreover, GPUs usually have a dedicated Video RAM that is faster, although smaller and more expensive, than the memory used by general pourpose processor (for example the latest Intel CPUs can uses various GBytes of DDR3 memory with a bandwith of 32 GB/s, while the last NVIDIA GPU uses GDDR3 memory with a bandwidth of 141.7 GB/s, but it is limited to 4GB).

Both actual commodity CPUs and GPUs are homogeneous multi-cores, although GPU in the past were composed by two types of specialized cores (vertex and pixel shaders). Despite that, it is not obvious if homogeneous are better than heterogeneous multi-cores.

A notable exception is the Cell Broadband Engine, that was designed to run the tasks of both a CPU and a GPU. As will be explained in details later, it has a general purpose core that runs the operating system and a set of eight specialized cores that are used for compute-intensive tasks. Originally the CBE was intended as the only computational core of PlayStation 3, although later a GPU was added. For many other aspects (the complexity of cores, the interconnection, the instruction set and so on) Cell Broadband Engine stands in an hypothetical middle line between CPUs and GPUs.

Another notable exception is the core used for Cray XD1 [29], that embeds an FPGA, although this is more a co-processor than a true additional core.

1.3.1 Cell Broadband Engine

The Cell Broadband Engine (CBE) is the first implementation of the Cell Broadband Engine Architecture (CBEA), developed by a collaboration of Sony Computer Entertainment, Toshiba, and IBM (usually referred as "STI"). CBE is a rare case of an heterogenous multi-core: a PowerPC (PPE) core and eight Synergistic Processor Elements (SPE) are embedded into the same die.

Development began in late 2000 and the first processor were widely available at the end of 2006, mainly as the PlayStation 3 processor. The CBEA has been initially designed for game consoles and high-definition multimedia device, but it has been extended to support a wider range of applications, like blade servers and High Performance Computing [30].

The 64-bit PowerPC core is a two-way simultaneous multithreaded Power ISA v.2.03 compliant core, and it is used to run the operating system and is primarily intended as a dispatcher of workloads to SPEs and for top-level control of applications.

The Synergistic Processor Elements are the main source of computational power of the processor. Their instruction set is optimized for SIMD operations and at 3.2 GHz they have a theoretical peak performance of 25.6 Gflops. Each SPE can directly access a private high-performance memory of 256KB (Local Store) with a constant latency, and cannot directly use the main memory. Data transfers between local store and main memory are performed via DMA transfers.

The nine cores (one PPE and eight SPE) and the main memory are connected together with the Element Interconnect Bus (EIB). The EIB is composed by eight rings that allows parallel data transfers. Each ring provides a bandwidth of 25.6 GB/s. To allow simultaneous transfers each SPE has its own Memory Flow Controller (MFC), a device that can perform DMA transfers autonomously, so that the SPE can concurrently perform computations.

The project started with the main goal of obtaining a performance enhancement of two order of magnitudes in respect to current game systems (in 2001, the PlayStation 2 with its Emotion Engine processor), and to develop a processor well suited for multimedia an floating-point intensive applications. The main target of the new processor was clearly the PlayStation 3, but also Digital Entertainment Center (as such devices and game console tend to converge in the last years), multimedia devices in general, and finally High Performance Computing, because the high computational power required by modern 3-D games is not too far from that required by scientific application.

Usually, in the past, performance improvements were obtained with faster clock frequencies, wider superscalar architectures, deeper pipelines and caches. In 2001 it was clear that this path was paying diminishing returns. First of all, due to technological limitations incrementing clock frequencies lead to power consumption and

dissipation problems. Moreover, as memory technology did not show comparable speed enhancement in the last years, increasing the performance of a single core leads to higher data access latency, that can make useless the speed enhancement of a single core. In effect, much of the bandwidth of a system can only be used speculatively, due to latency. Latency-hiding requires expensive and sophisticate hardware, and makes nearly impossible for the programmers to perform low-level optimizations.

The basic concepts that drove the development of the CBE where essentially to use powerful but simple computational cores and to connect all the elements of the system (including the memory) with a high-bandwidth interconnection bus. To allow an effective use of the available bandwith, simultaneous direct memory access can be performed without the assistance of computational cores.

More precisely, a general pourpose core (the PPE) is used to run the operating system an all the normal activities. Computational load is managed by multiple specialized cores (the SPEs), that are very fast and efficient for specific applications (single-precision floating point intensive computations, with predictable branches) but are not suited for general applications. Independent entities (the MFCs) perform DMA data transfers, so that techniques as prefetching can be effectively used and the impact of the latency is hidden.

Notably, each SPE can directly use only its 256KB local memory, and data transfers between main and local memory have to be explicitly requested to the MFCs. SPE have not cache hierarchies and do not support out-of order execution. This simplifications allowed to multiply easily the number of cores embedded into a single die.

Power Processor Element

PPE is a dual-issue, in order implementation of the IBM PowerPC Architecture, with vector multimedia extensions (VMX). It includes a 32KB L1 cache for both data and instructions and a 512KB L2 cache. The register file is composed by 32 x 64-bit general pourpose registers and 32 x 64-bit floating point registers. Moreover, there are 32 VMX 128-bit registers. The PPE is able to interleave instructions from two separated hardware threads. This means that state register are duplicated, but functional units, caches and queues are shared. The pipeline depth is 23 stages, considerably less than the previous implementation of PowerPC processors. Integer arithmetic and load instructions complete in two cycles, while double-precision floating-point instruction requires ten cycles. [30] Although the primary target of PPE is not computation, VMX extension allow a peak performance of 25.6 Gflops in single-precision when running at 3.2 GHz.

The PPE has not an associated MFC, so to perform DMA transfers it has to remotely access the MFC of the SPE involved in the data transfer.

One of the major benefits of PPE is its backward compatibilyty with the PowerPC software. In this way it was not needed to write the operating systems and the toolchain from scratch, thus shortening the development of the processor and allowing the CBE to run existing software once it was ready.

Synergistic Processor Element

A Synergistic Processor Element (SPE) contains a Synergistic Processor Unit (SPU) and a Memory Flow Controller (MFC). The SPU is composed by the 256KB Local Store and the SIMD functional units. The MFC is a DMA controller with a MMU (to help address translations) and an atomic unit (to allow synchronizations). The SPU and the MFC operate concurrently and independently.

As said earlier, a SPU does not need to run the operating system, so it does not support mechanisms like address translation and protection (that are delegated to the MFC). A SPU can operate only on data and instructions memorized into the local store. It can communicate with the other elements of the system (including main memory) only via DMA transfers performed by its own MFC or by one of the other MFCs. A SPU sumbits DMA transfer requests to MFC through a channel interface.

The instruction set of the SPEs is not the same of that of the PPE, although it is quite similar to VMX extensions. In any case, binaries are not compatible. In particular, the SPE ISA has specific instructions to issue commands to the MFC.

A SPU is essentially a SIMD architecture and it is organized around a 128bit dataflow. It has a large unified register file composed by 128 SIMD registers 128-bit wide, that allows deep unrolling to cover functional units latencies. It does not supports out-of-order execution nor branch prediction but, because the local store has a constant latency, its behavior is predictable and optimizations can be performed at compile time.

A SPU supports various levels of SIMD-granularities:

- 16×8-bit integers (bytes)
- 8×16-bit integers (half-words)
- 4×32-bit integers (words)
- 2×32-bit single-precision floats

Each SPU has two separate pipelines, so it's able to issue two SIMD instructions for each clock cycle if they are of different types. The "even" pipeline is able to perform float and integer computation, while the "odd" pipeline is dedicated to loads, store, branches and other operations that involve bits manipulations. A

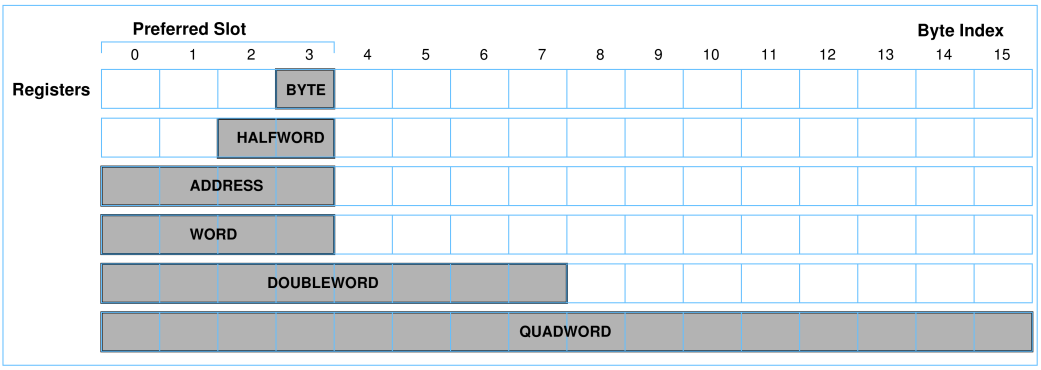

Figure 1.3: The SIMD-granularities supported by CBE. Note that each type of instructions supports only specific types of variables. For example, single-precision floating-point instructions supports only the "word" (four 32-bit scalar variables) granularity. (Figure is from [31])

somple but very important implication is that load/stores instructions can in principle be executed in parallel with computation, thus allowing a higher efficiency in single-precision floating-point pipeline exploitation. Single-precision floating-point instructions are executed in 6 cycles, while integer and logic instructions are performed in only two cycles. In the first release double precision instructions are not pipelined and require 7 cycles. When using single-precision floating point, if a multiply-add is issued at each clock cycle, at 3.2GHz a peak performance of 25.6 Gflops can be achieved.

The register file has six 128-bit read-only and two 128-bit write only ports, that are enough two feed two instruction that takes each one four operand (a multiply-add in the even pipe and a shuffle in the odd pipe, for example). A 128-bit bidirectional port connects the register file to the local store. A load/store instruction has a fixed latency of six clock cycles. As said earlier, the fixed latency allows various optimizations, but in principle local store access can be a bottleneck. However, the size of the register files allows to store a large amount of variables, thus reducing the local store access. Local store has also a 128-bytes read-only port used for instruction fetching an to send data to the MFC and a 128-bytes write-only port to get data from the MFC. However, the EIB can take only 8 bytes for cycle (which means 25.5 GB/s for each direction), and in this way the local store can be accessed only once every 8 cycles.

Unlike traditional processors, a SPE is not a scalar core with SIMD extension, but is completely SIMD. Scalar programs simply use a single slot of the SIMD registers. A SPE support only a single program context at any time. A context can be a user application ("problem-state") or operating system extension ("supervisor"). Moreover, a "hypervisor" state allows multiple operating systems to run concurrently.

We have said that the MFCs move data across the system. In particular, there are three different types of information exchange: signals, mails and regular DMA transfers. Signals and mails are 32-bit messages that are stored in special-pourpose queues, while regular DMA transfers are transfers take play between main and local memory or between two local memories, and can range from 1 byte to 16 KBytes. Although signals and mails are actually implemented as DMA transfers, in future CBEA compliant processors they will have a dedicated data-path.

Element Interconnect Bus

The Element Interconnect Bus (EIB) consists of one address bus and four 16-bye data rings, two of of them running clockwise and the other two counter-clockwise. Each ring support up to three concurrent transfers if the paths do not overlap. The EIB operates at half the speed of the cores, so at the reference clock speed of 3.2 GHz it can sustain a bandwidth B' of

$$B' = 3 \times 4 \text{ rings } \times 16 \text{ bytes } \times 1.6 \text{ GHz } = 307.2 \text{ GB/s} \tag{1.1}$$

Each element connected to the EIB can send and receive 16 bytes of data every bus cycle, so the bandwidth at which each element can send or receive data is 25.6 GB/s in both direction, for a total bandwidth of 51.2 GB/s.

The maximum bandwidth of the EIB is limited by the address snooping. Only one 16 byte long data transfer can be snooped per bus cycle. Each snooped address request can potentially transfer up to 128 bytes, so the theoretical peek is:

$$B = 128 \text{ bytes } \times 1.6 \text{ GHz } = 204.8 \text{ GB/s} \tag{1.2}$$

To send data to another unit, each element makes a request to the data ring arbiter on the the EIB. The arbiter decides which ring is granted and to which requester in which time slot, trying to optimize the EIB usage. The higher priority is given to the memory controller, because memory is a precious shared resource. All other elements are treated with a round-robin priority.

Any requester can use any of the four rings, given that it does not interfere with another transfer already in progress. Moreover, a ring is not assigned to a requester if the transfer would cross more than halfway around the ring on its way to destination.

Sustained bandwidth can be lower than the peak bandwidth because of several factors: the relative positions of the sender and of the receiver (for example, a six hops transfer inhibits the unit on the way to use the ring), the number of requests to the same destination and the number of requests per direction. Moreover, the bus efficiency is lowered when there are a large number of partial cache line transfers.

The EIB is coherent to allow a single address space shared by all the elements connected to the bus.

Memory

The internal memory controller (MIC) of the first implementation of CBE is connected to the external Rambus XDR DRAM through two XDR controller I/O (XIO) at 3.2 GHz. Each channel can support eight banks, for a maximum size of 512 MB.

The MIC manages independently the read and write requests queues for each channel. Writes in the range 16-128 bytes can be directly written to memory using a masked-write operation, while writes less than 16 bytes require a read-modify-write operation. The peak raw memory bandwidth is 25.6 GB/s, although management operations reduce it to 24.6 GB/s when all requests are of the same type. If reads and writes are intermingled, the effective bandwidth is reduced to 21 GB/s, due to the need to turning around the MIC-to-XIO bus.

The PowerXCell 8i implementation supports DDR2 memory instead of XDR DRAM, supporting up to 32GB of memory with ECC and a bandwidth of 25.6 GB/s.

Flexible I/O Interface

The CBE has seven transmit and five receive 5GHz, byte-wide Rambus FlexIO links, that are configured as two logical interfaces. When two processor are connected together, data and commands are transmitted as packets using a coherent protocol called "broadband interface" (BIF). BIF Operates over IOIF0.Typically the two FlexIO links are configured is such a way that an IOF0 has 30GB/s outbound and 20 GB/s inbound bandwiths and IOF1 has 5 GB/s outbound and 5 GB/s inbound and works in noncoherent mode. [32, 33] However, the flexibility of FlexIO allows to support different system configurations, from a single-chip configuration with dual I/O interfaces to a dual-processor configuration that does not require any extra chip to connect the processors.

Addressing

A peculiarity of CBEA is the definition of a main and a local storage domains [34]. The main one contains the address space for main memory and memory-mapped I/O registers and devices and is common to all the system. Each SPE has its own local domain that refers to its local store, used for both data and instructions. Each local domain is also mapped in main storage domain, in an address range called *local storage alias*.

The instructions that compose a CBE program use the *effective-address* space, while the processor itself supports the *real-address* space. A combination of hardware and operating system support allows to also support a *virtual-address* (VA) space. Both PPE and MFCs have dedicated hardware for address translation. The Cell

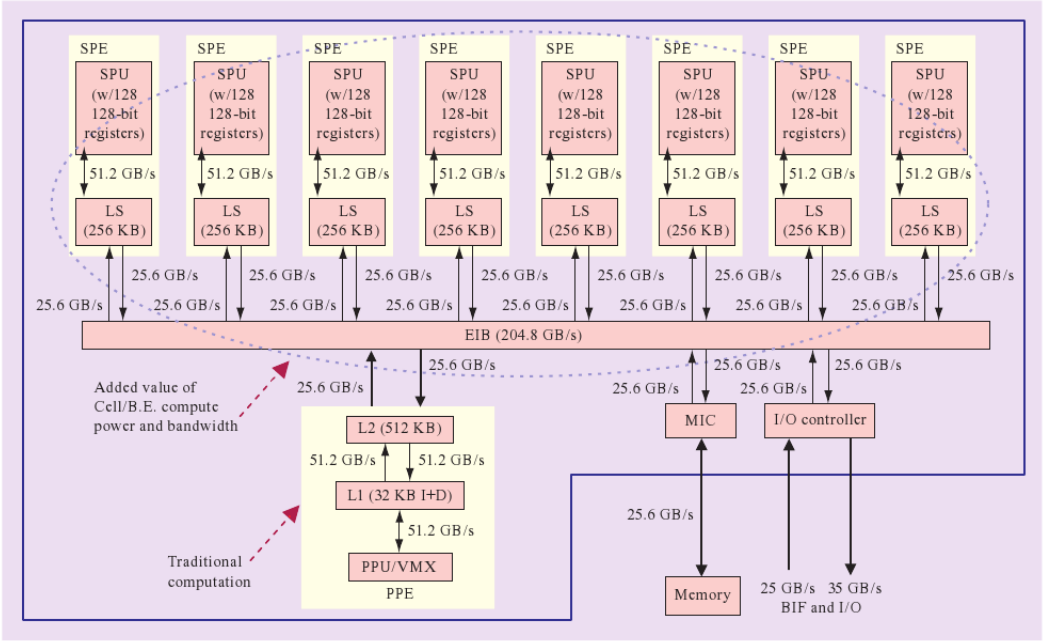

Figure 1.4: Block diagram of the CBE processor. Figure is from [32].

Broadband Engine Architecture (CBEA) processors is compatible with the PowerPc virtual-storage environment that is compatible with that defined by [35].

The real-address (RA) space is the set of all addressable bytes in physical memory and on devices whose physical addresses have been mapped to the RA space (as the SPEs local stores and MMIO registers).

The PPE uses effective-address (EA) address. Each SPE can access the EA space through MFCs and use local-storage (LS) addresses to access its own local store. Each MFC command has a pair of EA and LS addresses.

Virtual-address translation can be activated independently for instructions and data. Each PPE program can access 2^{64} bytes of EA space, that are a subset of 2^{65} bytes of virtual-address (VA) space. A EA address is translated into a VA address, and then in to a RA address. RA space is composed by 2^{42} bytes. An overview of address translation is shown in figure 1.5.

VA space is subdivided in protected and nonoverlapping *segments* of 256MB of contiguous address, while RA space is subdivided in protected and nonoverlapping *pages* that contain a number of consecutive address between 4KB and 16MB.

The conversion from EA to RA is performed in two steps: firstly, the EA is converted to VA using the SLB (segmentation) and then the VA is converted to RA using the page table (pagination).

The virtual-address mechanism allow to associate different restriction and different attributes (like cacheability and coherence) to each page or segment.

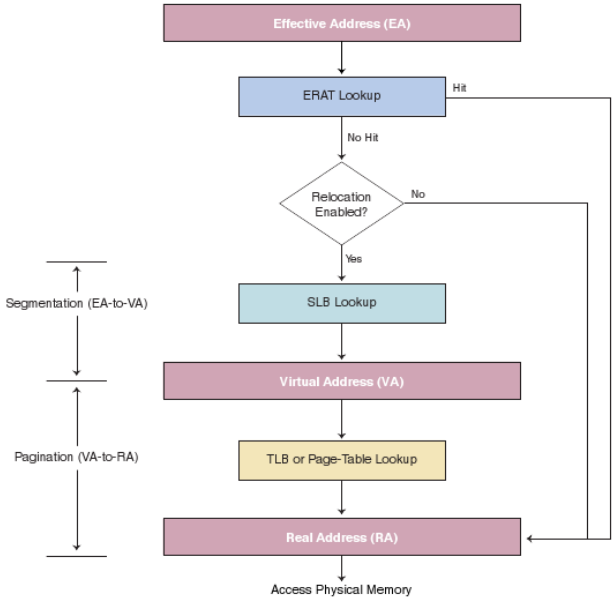

Figure 1.5: PPE Address-Translation Overview. Figure is from [36].

In general CBE architecture makes all SPE resource available through MMIO. There are three main classes of resources:

- *Local Storage.* The local stores of all SPEs. PPE can access to SPU's local storages using load and stores, but this process is fairly inefficient and is not synchronized with SPU execution
- *Problem State Memory Map.* These resources are intended for use by the application and includes MFC MMIO, mailbox channels and signals notification channels
- *Privilege 1 Memory Map.* Resources for monitoring and control by the operating system of the execution of SPU programs
- *Privilege 2 Memory Map.* Resources for control of SPE by operating system

The local store of each SPE is mapped into main memory, for programming convenience, but they are not coherent in the system. Figure 1.6 shows an overview of the mapping of SPEs resources into effective address space.

MFC and DMA Transfers

DMA requests can be issued to MFC in various ways. The SPU of the same SPE can use specific instructions to insert commands in the queues, or it can prepare a

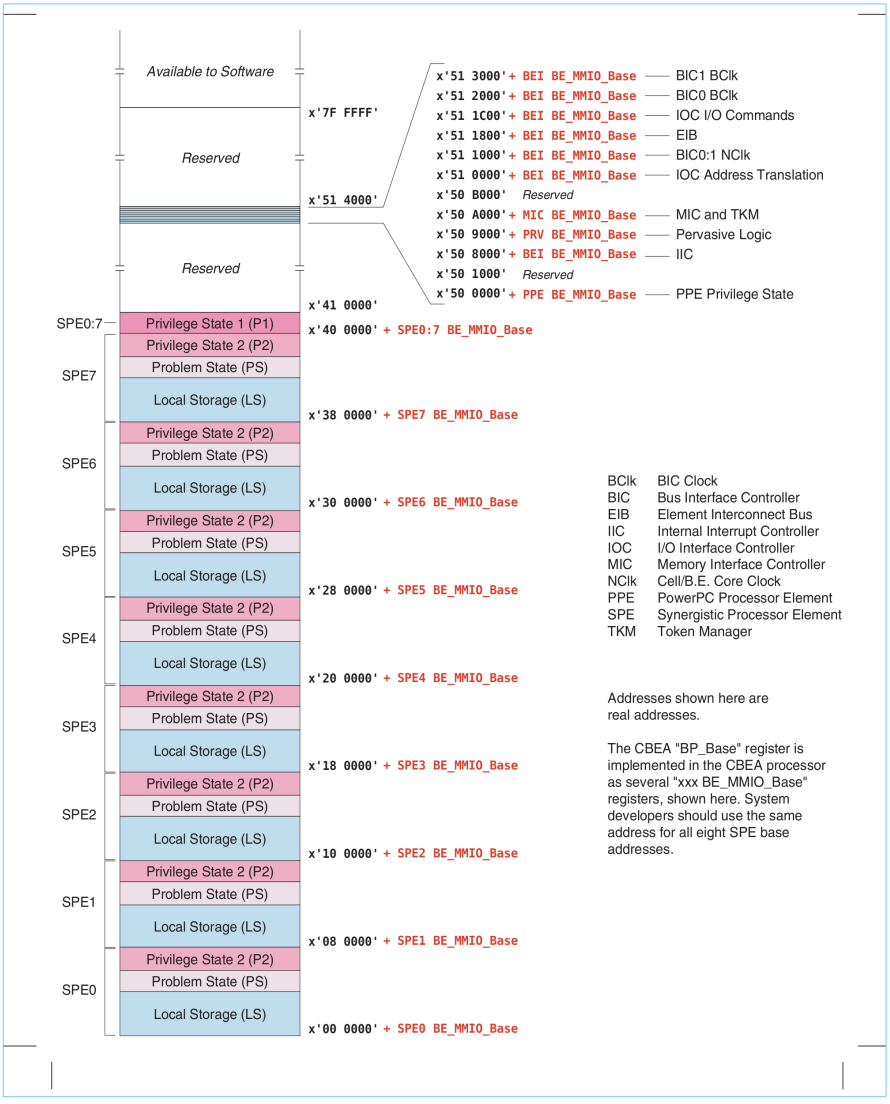

Figure 1.6: An overview of the mapping of SPEs resources into effective address space. Figure is from [36].

list of commands in the local store and issue a single "DMA list" command to the MFC. These commands are issued using the channel interface.

Channels are unidirectional communication paths that work like fixed-size FIFO queues. Each channel is either read-only or write-only, and can be blocking or non-blocking. Each channel has an associated *count* that indicates the number of elements in the queue. Three instructions allow the SPU to interact with the channel interface: read channel (rdch), write channel (wrch) and read channel count (rchcnt). The CBEA specifies the depth of some channels, while for the others it is an implementation specific parameter (for further details see [37]).

All other elements connected to the EIB can insert commands in specific "proxy" queues using memory-mapped (MMIO) registers (see section 1.3.1).

Each MFC processes two separate DMA command queues:

- *MFC SPU command queue*, that contains the command issued by the local SPU
- *MFC proxy command queue*, for the commands issued using MMIO registers by other elements connected to the EIB

The CBE architecture does not specify the size of the queues, but when they are full the result is a performance degradation, as the core issuing commands stalls until the queue is free. Alternatively, non blocking instruction are available to access the queues. In the first implementation, the CBE, the MFC SPU queue contains 16 entries and the MFC proxy command queue eight entries.

A single DMA command can request the transfer of 1,2,4 or 8 bytes or a multiple of 16 bytes, up to 16KB. A DMA command list, that is accepted only by MFC SPU queue, can request up to 2048 DMA transfers. A DMA list is an array of source/destination address and length tuples that is stored in the local memory. Both source and destination addresses have to be 16 byte aligned. The hardware can transfer only 128-byte aligned blocks of 128 bytes, so the peak performance is achievable only if both addresses are 128-byte aligned and the length of the transfer is a multiple of 128 bytes.

The MFC supports also two signaling channels. The SPU access the channels using the channel interface, while the PPE has to use the MMIO registers. It is possible to perfrom a logical OR of the signals of all the SPU of the system.

Each MFC has also three 32-bit mailbox queues: a four-entry, read-blocking inbound mailbox and two single-entry, write-blocking outbound mailboxes. One of the outbound mailboxes is able to generate interrupts to PPE when it is written by the SPU. The PPE has to use MMIO registers to write the SPUs inbound mailboxes or to read from SPUs outbound mailboxes. Mailboxes are well suited for one-to-one communications, and the roundtrip time between two SPUs is approximately 300 ns ([33], or 960 clock cycles (at 3.2 GHz).

PowerXcell i8

PowerXcell i8 is based on a 65nm process and the main difference with is predecessor is the support of DDR2 memory and better performance for double-precision floating-point operation.

DDR2 memory has been chosen because it allows the use of a greater amount of memory per board. CBE was limited to 1GB per board, clearly insufficient for HPC applications. Now a single processor supports up to 4 DIMM slots and up to 16GB of memory.

The peak bandwidth has been preserved by expanding the memory buses to 128-bit, although this implies an increase of pin count and pin incompatibility with previous generations.

While in the first implementation of CBE double-precision floating-point performance where fairly poor, now a single SPE is capable of 12.8 Gflops, which lead to a peak performance of the whole processor of 102.4 Gflops [38].

While double-precision functional units are fully IEEE 754 compatible, single-precision is not fully compliant due to truncations.

The next generation of CBE processor should be released in 2010. The number of SPE should be expand to 16 or 32 using a 45nm process [38].

CBE blades

In IBM QS211 blades, two CBEs are configured as a two-way, symmetric multiprocessor (SMP). Each processor has its own directly connected Rambus XDR memory and it is connected to a separate South Bridge.

The bandwidth of the interface between the CBE processor is 20 GB/s in each direction, and it is realized through FlexIO.

The bandwidth of the South Bridge interface is (CIRCA) 1.0 GB/s in each direction. The South Bridge 0 chip provide all the required features: a PCI-E channel, a PCI bus interface, an IDE channel, a Gigabit Ethernet interface, USB 2.0 ports, a UART and an external bus controller. More details can be found in [39].

An interesting feature of the dual-processor configuration is the possibility to perform DMA transfers between the local stores of the SPEs belonging to different processors. Moreover, a SPE can access both memories, although accessing the local memory is faster.

QS22 is an updated version of QS21 with PowerXcell 8i processors and DDR2 memory. In each board there are two 3.2 GHz PowerXcell 8i connected via BIF interface. Each processor can be equipped with up to 16GB of DDR2 memory.

An IBM BladeCenter H chassis can host up to 14 QS21 or QS22 blades.

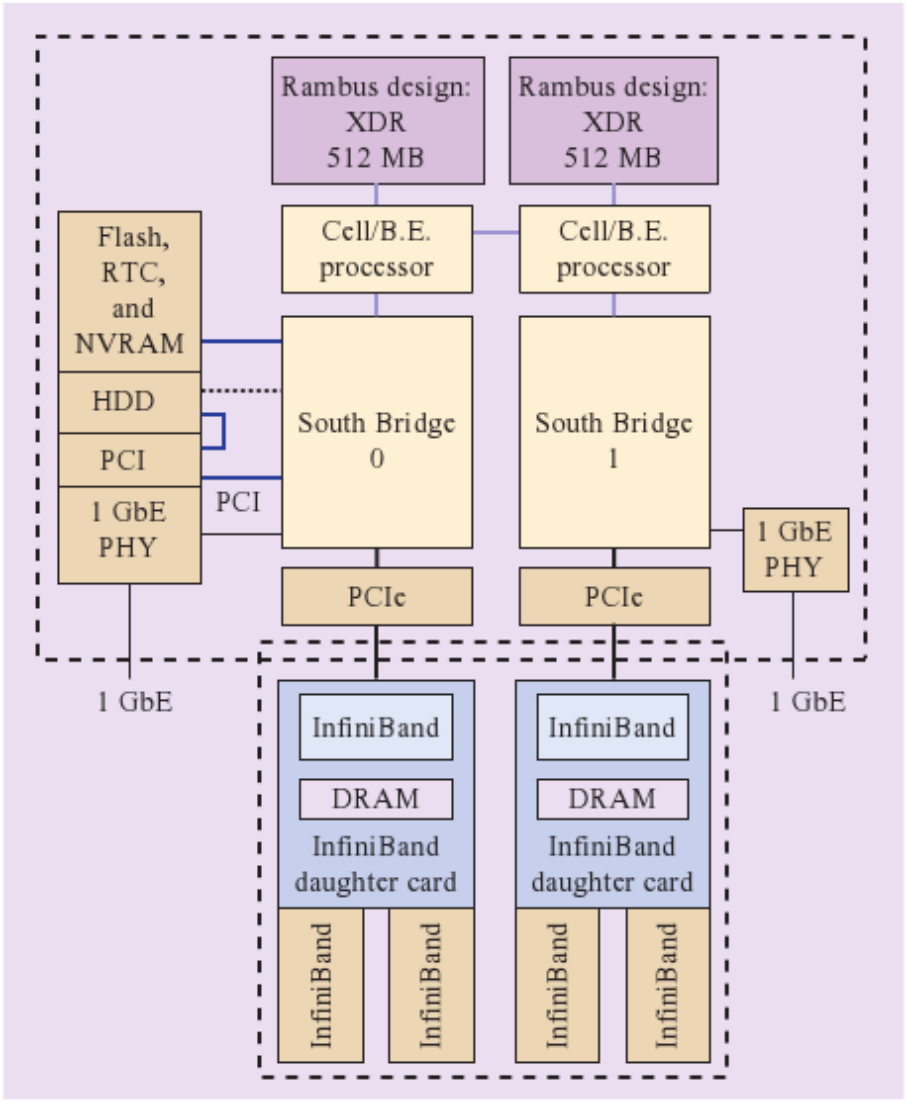

Figure 1.7: The organization of a QS21 Blade Server. Note that the two processors are directly connected and that, although some resources are shared (as the PCI interface), each processor has its own XDR memory, Gbit PHY and PCIe interfaces. Figure is from [39].

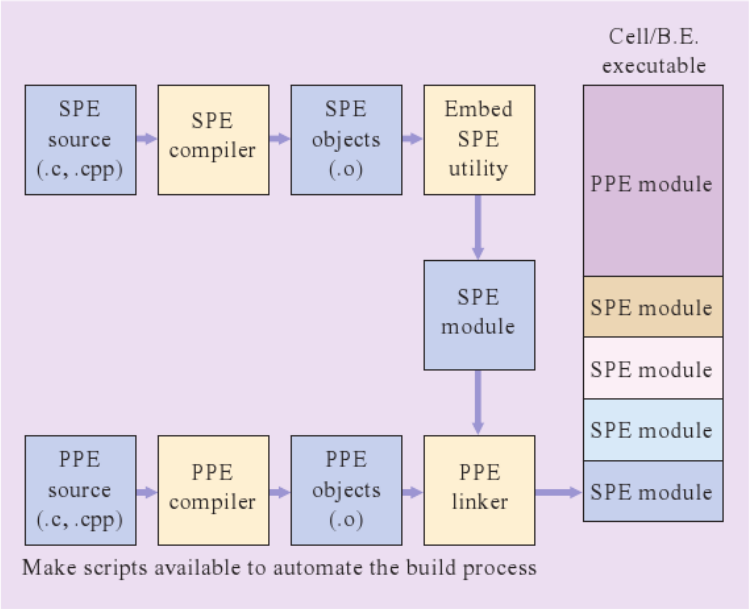

Figure 1.8: The compile chain of CBE. Note that the executable of PPE and SPEs are compiled independently, and that only in the linker phase are united into a single executable file. Figure is from [39].

Application Development

The Cell Broadband Engine is supported by Linux, and the *Cell Broadband Engine (Cell/B.E.) Software Development Kit (SDK)* provides all the tools needed to develop an application for CBE. The two main alternative are the GNU Toolchain or the IBM XLC/C++ compiler, although Fortran and ADA compilers are also available.

The SDK include various libraries, including the SPE Runtime Management Library that allow to run executables on the SPEs.

The CBE is an heterogeneous processor that supports two different instruction sets. Two distinct toolchains are used for the two types of core, although both toolchains produce executables in ELF format. An interesting feature is the *CBE Embedded SPE Object Format* (CESOF), an application of the ELF of the PPE that allows PPE executable objects to contain SPE executables [36, 40].

CBE Programming

The issue of efficiently programming the CBE has been widely analyzed in literature. OpenMP [41] and MPI [42] have been ported to CBE, so it is possible to port old programs with a relatively small effort (although efficiency is not guarantee) and it is possible to continue to adopt these two well-known programming paradigms.

As long as data-transfers and local memories usage is completely software controlled, it is important to understand which is the best way to manage them [43]. There are also many researches in the field of compilers and automatic code generation [44, 45, 46, 47, 48], as CBE processor in-order hardware requires a non trivial programming effort to produce optimized code. Many scientific application have already been ported to CBE [49, 50, 51, 43, 52, 53, 54, 55, 56, 57, 58].

1.3.2 Intel Nehalem

Nehalem is the first Intel "real" quad-core processor and it is availabel since November 2008. In a single die are embedded four cores. Each core has 32 KBytes of L1 program cache and 32 KBytes of L1 data cache. Each core has also its own 256 KBytes L2 cache. The four cores also share 8MB of L3 cache. L3 cache is inclusive: if a line is in L1 or L2, then it's also in L3. This prevents the need to snoop L1 and L2 of each core. The die also integrates the memory controller. Nehalem supports DDR3 RAM with a bandwidth of 32 GB/s. In multiprocessor configurations, CPUs are connected through the Quick Path Interconnect, that provides a bandwidth of 25.6 GB/s in both directions.

Each core supports hyper-threading and SSE4.2. It has also 4-wide dynamic execution engine and has hardware dedicated to loop detection.

1.3.3 AMD Shangai and Istanbul

Shangai chips are the first AMD's 45 nm processors and features 4 cores in a single die. Three level of cache are present: 6 MBytes of L3cache are shared between the cores, while each core has 512 KBytes of L2 cache and 64 KBytes of L1 data and instruction cache.

DDR2 RAM is supported at up to 800MHz, while DDR3 RAM is not supported. Only HyperTransport 2.0 (with a bandwidth of 8 GB/s) communication bus is supported.

Shangai processors supports SSE and SSE3 SIMD instructions, and add the support to AMD's SSE4a, that are incompatible with Intel's SSE4.1

In the second half of 2009 Istanbul processors will be available. They will feature 6 cores and will add the support for DDR3 RAM and HyperTransport 3.0 communication bus, with a bandwidth of 17 GB/s. The cache hierarchy will be unchanged, and the die size will be of 300mm.

1.3.4 NVIDIA GT200

At the end of 2008 NVIDIA high-end GPU offer is the GT200 Series, that are used across the GeForce, Tesla and Quadro product lines.

As long as we are interested in computational peak performance, CUDA programming model and implications will not be discussed.

A GT200 GPU is structured wtih many levels of hierarchy and a direct comparison with "standard" multi-core architectures is not always possible. A GT200 GPU is composed of 10 Thread Processing Clusters (TPC), each of one is made of 3 Streaming Multiprocessors (SM) and one memory pipeline. Each SM is composed by 8 Streaming Processors (SP). An SP has a private program counter and a set of registers, but it is not an independent core, as it lacks may features, like a fetch unit, and it is equivalent to a scalar 32bit pipeline in a multi-threaded CPU.

The Streaming Multiprocessor is the smallest independent unit of the GPU, and its similar to a single-issue processor with 8-way SIMD support.

NVIDIA defines its computational model as Single Instruction, Multiple Thread (SIMT). The main difference with SIMD is that while SIMD exposes architectural width and data must be packed into vectors and the same operations are applied to all the scalars packed into a vector, in SIMT each thread works on it's own data and it's possible for each thread to take independent branches (with a loss of performance if the threads diverges).

Each SM has 64 KBytes register file (16 KBytes $\times$ 32bit) partitioned between SPs, and a 16 KBytes shared memory with the same latency of the register file for communication between threads. Data cannot be directly loaded from main to shared memory. Data can be exchanged between the main memory and the register file and between the shared memory and the register file, so a transfer between shared and main memory requires an intermediate step into register file.

Each SM has a shared fetch unit and a set of eight 32bit ALU that can perform a multiply-add between IEEE single-precision floating point in 4 clock cycles. Moreover, each SM includes a double-precision execution unit and a Special Function Unit (SFU) for reciprocal, interpolation and other special functions. The SFU can also be used to perform two extra 32bit madd.

Load/Store units are decoupled from the computational units. Each Thread Processing Cluster (TPC), as said earlier, has three SMs and a texture pipeline that is used to access the render output (ROP) units. Load/store instructions are generated by SMs, but they are issued and executed by the texture pipeline. The texture pipeline has two texture caches that, unlikely traditional caches, have 3-dimensional locality. Moreover, they are read-only and have no cache-coherency. Texture caches are intended to save bandwith and power, and do not impact latency. Each TCP embed 24 KBytes (8 KB for each SM) L1 caches, while a 32KB L2 cache resides with the memory controller.

Main memory load/stores must be 4 bytes aligned. Load instructions of different threads can be coalesced to reduce the total amount of transactions.

A GT200 chip requires 1.4 billions of transistors and is fabricated with 65nm process. The die size is $583.2mm^2$. There are three independent clock domains:

- graphics clock, used for texture pipeline, ROPs and SM (except SP functional units)
- processor clock, the double of graphics clock, used for SPs
- memory clock, used for the GDDR3 memory controller

Typically the processor frequency is 1296MHz, while memory frequency is 1107MHz.

The peak performance is 933.1 Gflops in single precision and 77.8 Gflops in double precision. The GT200 supports GDDR3 memory that allows a bandwidth of 141.7GB/s. Typically 1GB is mounted on each board, but up to 64GB are supported. The TDP [1] is of 236W under load, but it goes down to 25W when idle.

1.3.5 AMD ATI RV770

At the end of 2008 AMD high-end is represented by the RV700 GPU. An RV700 contains 10 SIMD cores. Each SIMD core is composed by 80 Stream Processors Units grouped into 16 Stream Processors. A stream processor can execute up to 5 instructions in parallel if they belong to the same thread. Each SIMD core has also 4 Texture Address Units (TAU), 16KB L1 cache and 16KB of Global Data Share memory. L1 cache bandwidth is 480 GB/s

In the chip are embedded 4 x 64bit memory controllers, that also integrate a L2 cache. The bandwith of transfers between L1 and L2 cache is 384 GB/s. Memory controllers supports GDRR3/4/5. Hi-end boards mount 1GB of 900MHz GDRR5, for a total bandwidth of 115.2 GB/s.

RV770 contains a PCI-e 2.0 bridge that allows a bandwidth of 10 GB/s for both communications with the main memory and a second GPU in multi-chip configuration. Moreover, two RV770 are also connected with a Sideport bus that allows an extra bandwidth of 10.2 GB/s.

The die size is $256mm^2$ and there are 956M transistors. The TDP is 250 W. In high end configuration the clock speed is 750 Mhz. If all Stream Processor Unit are concurrently performing a multiply-add, its possible to achieve a single-precision floating-point peak performance of 1.2 Tflops with a single GPU. Double-precision performance is 240 Gflops.

1.3.6 Many-core: Intel Larrabee

Intel Larrabee is a project for a discrete GPU based on x86 cores. Presumably it will be available at the end of 2009. The basic idea is to embed 32 or more cores into a single die. While NVIDIA and ATI GPU are based on programmable but

[1] TDP is the Thermal Design Power and represents the maximum amount of power the cooling system in a computer is required to dissipate.

highly-specialized cores (stream processors), Larrabee will have very little graphic specialized hardware, and each core supports the well-know x86 ISA, with the addition of a 512bit vector processor unit that can perform the same operation onto 16 independent floating point variables. The consequence is that Larrabee should be more flexible than a traditional discrete GPU.

Concerning data model, while a traditional GPU highly-specialized memory, Larrabee supports cache coherence between cores and it provides instructions for cache manipulation. The cores are interconnected with a 1024-bit wide interconnect bus.

1.4 Conclusions

Table 1.1 shows a comparison between the multi-core architectures available at the beginning of 2009.

GPUs are very impressive, as they feature the highest number of cores (they are the first example of many-core architectures), although they are designed for a narrow range of applications. Howeverer, today there is a great interesting in the General Purpose GPU programming (GPGPU). Both NVIDIA and AMD are trying to introduce GPUs into to the world of high performance computing, proposing two software infrastructures that allow to use GPUs for general purpose computation (NVIDIA CUDA [59] and AMD ATI [60]). The OpenCL (Open Computing Language) framework has been proposed as a effective way to write programs that executes across heterogeneous platforms as CPUs and GPUs.

Multi-core processors are more general purpose than GPUs, but they also suffer from the relatively small bandwidth of main memory. Moreover, the relatively small number of cores now available does not allow to fully take advantage of parallelism. However, the future trend should for commodity processors should see a doubling of the number of cores every 18 months, so that in 2015 general purpose many-cores processor with 1024 cores should be available. For that time, it is not clear which amount of parallelism will be achieved by GPUS, and if CPUs and GPUs will have converged into a single type of device. In the next few years the first chips including both CPU and GPU in a single die are expected.

The CBE processor seems to take advantage from the best of the two world, because it has more cores than a commodity processor and, although they are not well suited for every type of applications, they are more general purpose than the streaming processors included in GPUs. Moreover, the presence of on-chip memory completely under software control should allow to refine compiler technology in order to be able to automatically adapt the programs to the available resources. However, the balance of computational power and data transfer channels is algorithm dependent, so while CBE grant high computational power in some application, each

	CBE	PowerXcell i8	Nehalem	Shangai	GT200	RV770
Cores	1+8	1+8	4	4	10	10
32bit-ways x core	4	4	4	4	24	80
32bit-ways x chip	32	32	16	16	240	800
SP Fops x chip	64	64	16	16	720	800
Clock (GHz)	3.2	3.2	3.2	2.7	1.3	1.5
Peak SP Gflops	230.4	230.4	51.2	43.2	933.1	1200
Peak DP Gflops	13+20	13+100			77.8	240
Process (nm)	90	65	45	45	65	55
Transistors (M)	241	250	731	758	1400	956
Die size (mm^2)	235	212	246	243	583.2	256
TDP (W)	80	100	130	95	236	250
Memory	XDR	DDR2	DDR3	DDR2	GDDR3	GDDR5
Memory BW (GB/s)	25.6	25.6	32	25.6	141.7	115.2
L1 cache x core	256KB	256KB	32KB+32KB	128KB	24KB	
L2 cache	-	-	4 x 256KB	4 x 512KB	256KB	
L3	-	-	8MB	6MB	-	
IEEE SP	no	yes	yes	yes	yes	no
IEEE DP	no	yes	yes	yes	yes	no
Speculative	no	no	yes	yes	no	no
Model	SIMD	SIMD	SIMD	SIMD	SIMT	

Table 1.1: A comparison of the main multi-core architectures available at the beginning if 2009. The highest computational power are achieved by AMD ATI RV770 and NVIDIA GT200 GPU architectures, although an high efficiency can be achieved only in specific applications. Note also that the power dissipation of these devices is very high (at least the double of a processor) and that they use very fast memories. Typically a graphic board has at most 1GB of memory, that it is sufficient for video applications but much less than the amount of memory supported by a traditional processors. Intel Nehalem and AMD Shanghai architectures have both four cores and comparable amounts of on-chip memories. The cores of these devices are general-purpose processor, that grant an high flexibility but that also are an order of magnitude slower than GPUs in single-precision floating-point computations. Finally, CBE processors not only have 9 cores instead of 4, but also have a peak floating-point performance that is four times higher than those of Intel and AMD processors, although the computational cores are efficient only in computing-intensive applications.

case should be evaluated independently.

Chapter 2

Spin Glasses

Spin models describe systems characterized by phase transitions (such as the para-ferro transition in magnets) or model "frustrated" dynamics, which appears when the complex structure of the energy landscape of the system makes the approach to the equilibrium state very slow. They are relevant in several areas of condensed-matter and high-energy physics and have been successfully applied to a large set of problems.

A *spin glass* is a disordered magnetic system in which the interaction between the magnetic moments associated to the atoms of the system are mutually *conflicting*, due to some frozen-in structural disorder [61]. The macroscopic consequence is that it is *difficult* for the system to relax to its equilibrium state: glassy materials never truly reach equilibrium on laboratory-accessible times, since relaxation times may be, for macroscopic samples, of the order of centuries.

Short-ranged spin glass models are defined on discrete, finite-connectivity regular grids (e.g., 3-D lattices, when modeling experiments on real magnetic samples) and are usually studied via Monte Carlo simulation techniques. The dynamical variables of the systems are "spins", that sit at the vertices of the lattice and assume a discrete and finite (and usually small) set of possible values. The nature of different couplings and different allowed spins values give birth to different models of spin glasses (see for example [62] or [63]).

State-of-the-art simulations may stretch for well over 10^{10} Monte Carlo updates of the full lattice, and have to be repeated on hundreds or thousands of different instantiations (*samples*). In addition, each sample must be simulated more than once (*replicas*), as most properties of the model are encoded in the correlation between independent histories of the same system. The physical events that are simulated have a magnitude of 10^{-12} seconds, and it is interesting to analyze time periods that go from 1 second to 100 seconds. As a consequence, $10^{12} \cdots 10^{14}$ updates have to be executed for each spin. For a 3-D system of linear size 80 this translates into $10^{17} \cdots 10^{18}$ spin updates, a major computational challenge.

The computational kernels have a large and easily-identified degree of available parallelism, thanks to the fact that, for each given sample or replica, the update of up to one half of all spins of a system can be performed concurrently, if computational resources allow to do so. In spin glass jargon this is referred to as *synchronous* parallelism. Yet more parallelism can be exploited thanks to the large number of samples and replicas that have to be independently simulated. This is usually referred to as *asynchronous* parallelism.

While asynchronous parallelism can be obviously exploited by farming out the overall computation to independent processors, the large available synchronous parallelism is poorly exploited by traditional processor architectures. These problems, and the fact that the computation is based on simple bit-wise logic operations, have triggered, in the last two decades, the development of several application-driven machines, strongly focused for spin glass simulations [8]. In recent years, this approach has been based on FPGAs, on which a very large number of processing cores – each core being a spin-update engine – can be easily implemented [15].

The emerging multi-core processor architectures, like the IBM Cell Broadband Engine, promise to offer a new opportunity for fast spin glass simulations. The large and increasing number of cores, combined with the availability of SIMD instruction set in each core, allows to exploit synchronous parallelism. Multi-core processors also have a non negligible amount of on-chip memory (typically from 2 to 8 MBytes), as well as efficient core-to-core communication mechanisms, that can reduce the data-access bottlenecks that may be expected when a very large number of operations is performed concurrently. The level of exploitable parallelisms on these processors will not probably reach those of dedicated engines, where $\simeq 1000$ spins are updated concurrently. For example, in chapter 4 we show that with a single CBE processor it is possible to update at least $\simeq 100$ spins of the same system in parallel. However, the clock frequency of a state-of-the-art processor is an order of magnitude faster than those of a dedicated system, so it may substantially close the performance gap. Moreover, price-performance ratio may also become very interesting, as multi-core processors are already widely available and their are less expensive than a dedicated system.

The following discussion will be focused on arithmetic performance on a single core, assuming that all the data required to update a fraction of the lattice is available in local memory, a realistic assumption in the case of Cell Broadband Engine. One of the features of Metropolis algorithm is data locality, because the update procedure of a spins requires only its neighbors. The local memory of a SPE is completely under programmer control, so it is possible to assure that it contain all the required data. Moreover, local store accesses have a fixed latency so, as long as the data is locally available, data access pattern do not affect performance, although it can influence the addressing.

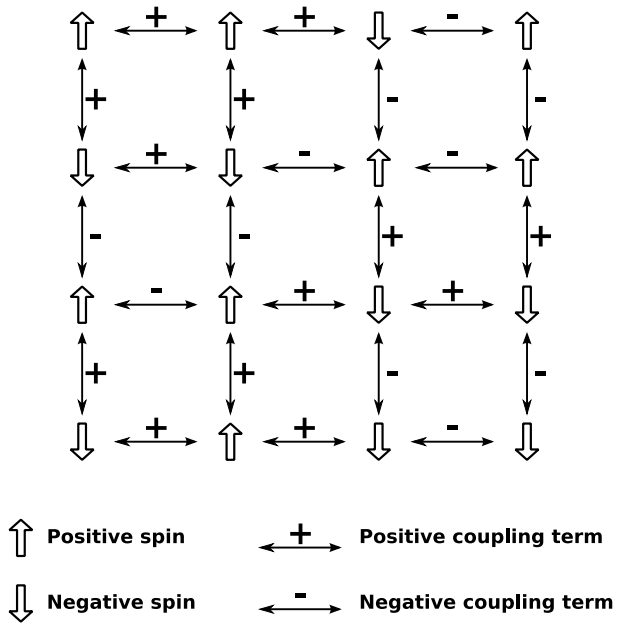

Figure 2.1: An example of a 2-D lattice. Periodic boundary conditions are no reported.

2.1 The Edward-Anderson Model

The *Edwards-Anderson (EA)* model is a widely studied theoretical model of a typical spin glass, namely a disordered magnet. The model is defined on a D-dimensional square lattice of linear size L. Atoms sit at the vertexes of the lattice and their magnetic moments are modeled by spins σ, discrete variables that take just two values, $\sigma = \pm 1$ (in other similar models σ takes a larger set of values; for instance, in the Potts model [64] spins are n-valued, with n typically ranging in $3 \cdots 6$). The magnetic interaction between spins is modeled by an energy function that is a sum of terms associated to all pairs of nearest-neighbor spins in the lattice. Interaction terms between atoms that are not nearest-neighbor are neglected. Periodic boundary conditions are usually applied, so the lattice takes the topology of a D-dimensional discrete torus (this not only has physical relevance, making surface effects negligible, but also simplifies computer coding, with a strong impact on performances).

For each pair of neighboring spins i and j an interaction term (*coupling*) J_{ij} is defined. The J_{ij} are randomly assigned to all pairs in the system, mimicking the structural disorder of the real system, and kept fixed during Monte Carlo evolution. Figure 2.1 shows the example of a 2-D lattice.

Physically relevant results are obtained by averaging over a large number of different assignments for the J_{ij}. The probability distribution from which the coupling are extracted determines the properties of the system: in glassy systems one usually takes $J_{ij} = \pm 1$ with 50% probability (*binary EA* model) or samples a Gaussian

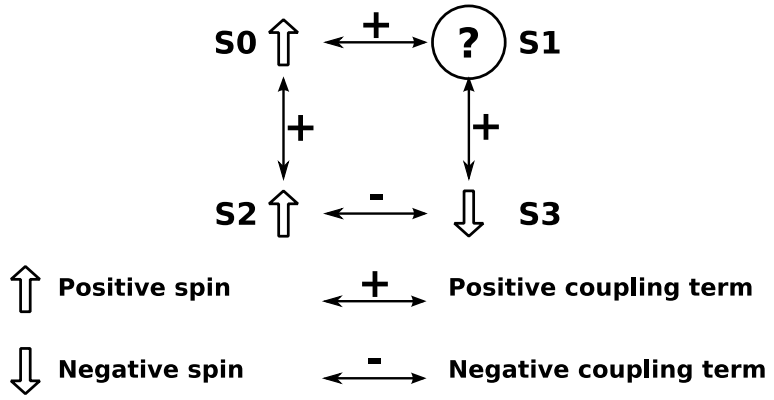

Figure 2.2: Example of frustration: spin $S0$ and $S2$ satisfy their mutual coupling; spins $S2$ and $S3$ satisfy their mutual coupling; spin $S1$ cannot simultaneously satisfy the couplings with both spins $S0$ and $S2$ because of concurrent values of coupling terms.

distribution with zero mean and unit variance (*Gaussian* EA model). Each given assignment of all couplings defines a *sample* of the system. For each sample, we will need to consider several *replicas* (that is several copies of the system that have the same set of couplings and evolve independently of one another).

The energy function of the system is given by

$$E = -\sum_{\langle ij \rangle} J_{ij}\sigma_i\sigma_j. \tag{2.1}$$

Notation $\sum_{\langle ij \rangle}$ denotes the sum on all nearest-neighbor spins. A positive coupling will favor alignment of the corresponding coupled spins, while a negative one will push for misalignment.

The randomness of the values of J_{ij} has dramatic consequences on the dynamics of the system, associated to the so-called "frustration". If we consider the simple spin system of Figure 2.2 it is easy to verify that no configuration of the four spins can simultaneously satisfy all energy constraints of the system: in these cases the system is said to be "frustrated".

This structure means that the system has a very sluggish drift toward equilibrium, that computationally translates into the fact that finding the state of lowest energy of the full system is an NP-hard problem.

In particular, for three-dimensional lattices, it has been showed that a graph-theoretic problem known to be NP-complete [1], the task of finding a maximum set

[1]NP-complete is a subset of NP, the set of all decision problems whose solutions can be verified in polynomial time; NP may be equivalently defined as the set of decision problems that can be solved in polynomial time on a nondeterministic Turing machine. A problem p in NP is also in NPC if and only if every other problem in NP can be transformed into p in polynomial time. A problem H is NP-hard if and only if there is an NP-complete problem L that is polynomial time reducible to H.

of independent edges in a graph for which each vertex has degree 3, can be reduced to the spin glass problem [65]. Further details about the NP-completeness of the problem can be found in [66, 67, 68, 69].

The EA model is usually studied via canonical (i.e. fixed-temperature) Monte Carlo simulations. The computational kernel of a Monte Carlo procedure, based on the standard single spin-flip Metropolis algorithm [70], will be described in section 2.2 and then it will be described as it can be implemented in a processor architecture that supports w-bit data-words, with $w > 1$.

2.2 Metropolis Algorithm

The basic computational kernel of a Monte Carlo procedure based on the standard single spin-flip Metropolis algorithm consists in the following steps, to be repeated many times. Note that a Monte Carlo step is defined as the update attempt of a number of spins equal to the number of lattice spins.

1. pick one site at random
2. compute the local energy E, summing all contribution from nearest neighbor spins and relative couplings
3. flips the value of the spin $\sigma' = -\sigma$
4. compute the new local energy E'
5. compute the energy change: $\Delta E = E' - E$
6. if $\Delta < 0$ the new value of the spin σ' is accepted
7. if $\Delta \geq 0$ then the new state is accepted if $\rho < e^{-\beta \Delta E}$, where ρ is a random number ($\rho \in [0, 1]$) and β is defined as the inverse of the temperature

The actual order in which sites are visited is not important, as long as all sites are visited an equal number of times on average, so any lexicographic order can be followed. This brings to an important simplification: the steps from 2 to 7 outlined above can be applied in parallel to all spins that do not share a coupling term in the energy function. The cubic lattice is bipartite in a checkerboard scheme and the algorithm can be applied first to all black sites and then to all white ones.

When studying the out-of-equilibrium dynamics of the model, one usually simulates very large systems (order 10^6 lattice sites) so that the equilibrium is never reached, on relatively few samples(10^2-10^3). When instead one is interested in equilibrium properties, small sizes (order 10^2-10^4 lattice sites) are mandatory in order to keep equilibration times short. In this case it is necessary to simulate many

more samples (10^4-10^5) to obtain good statistics, as usually interesting equilibrium properties have strong sample-to-sample fluctuations.

The former task would require an intensive exploitation of all *synchronous* parallelism, as it is important to speed up the simulation of very few samples, and asynchronous techniques would be of little use or degrade the performance; in the latter one, parallelizing among different samples may be preferable, as the *asynchronous* approach might be in some cases much more efficient in term of single spin-flip time.

2.3 The Binary Model

In the binary Edwards-Anderson model both spins and couplings are two-valued:

- $\sigma_i \in \{-1, 1\}$
- $J_{ij} \in \{-1, 1\}$ foreach pair of nearest neighbors spins i and j

The local energy of interaction E_i of the spin σ_i is defined as:

$$E_i = \sum_{<j>} \sigma_i J_{ij} \sigma_j \tag{2.2}$$

The energy difference ΔE_i is the difference between energy E_i associated to σ_i and the energy E_i' associated to the spin $\sigma_i' = -\sigma_i$:

$$\Delta E = E_i' - E_i \tag{2.3}$$

The energy E_i can be rewritten as:

$$\begin{aligned} E_i = & \sum_{<j>} \sigma_i J_{ij} \sigma_j \\ = & \ \sigma_i \sum_{<j>} J_{ij} \sigma_j \end{aligned} \tag{2.4}$$

while E_i' can be re-defined as:

$$\begin{aligned} E_i' = & \sum_{<j>} \sigma_i' J_{ij} \sigma_j \\ = & \ \sigma_i' \sum_{<j>} J_{ij} \sigma_j \\ = & -\sigma_i \sum_{<j>} J_{ij} \sigma_j \\ = & -E_i \end{aligned} \tag{2.5}$$

# -1	# 1	E_i
0	6	6
1	5	4
2	4	2
3	3	0
4	2	-2
5	1	-4
6	0	-6

Table 2.1: The values that can be assumed by energy E_i. The first two columns indicate the number of equation (2.5) whose value is -1 or 1.

This implies that the energy difference ΔE_i is equal to:

$$\begin{aligned} \Delta E_i &= (-E_i) - E_i \\ &= -2E_i \end{aligned} \tag{2.6}$$

Because E_i is a sum of six terms (one for each neighbor of the spin i) taken from the set $\{-1, 1\}$, it can assume seven values, as shown in Table 2.1. As a consequence, the energy difference ΔE_i can assume seven different integer values:

$$\Delta E_i \in \{-12, -8, -4, 0, +4, +8, +12\} \tag{2.7}$$

As we will describe, this property allows to ..?

Data representation is a major issue. Spins and couplings can be represented by a single bit, but actual architectures support word sizes of several bits, that in this case would be wasted. Quite obviously, if more bits than necessary are used, more data has to be transferred in the systems and less useful information is stored into precious fast local memories. A trivial optimization would be data compression, so that more than one spin or coupling is packed into registers, and computations are performed on binary variables that have been extracted from data structures.

An interesting property of the Metropolis algorithm applied to the binary model is that computation are performed between integer values that are representable with very few bits. Moreover, operations between integer values such as sums, multiplications and comparisons can be rewritten as logical bitwise instructions that allow to concurrently apply the same operation on all the bits of the variables. In this way bits that otherwise would have been wasted can be used to update in parallel a set of spins, thus increasing parallelism.

The spins embedded in the same register can be corresponding spins of different samples (asynchronous parallelism), or they can be different spins of the same system, thus improving the spin update time of a single system (synchronous parallelism).

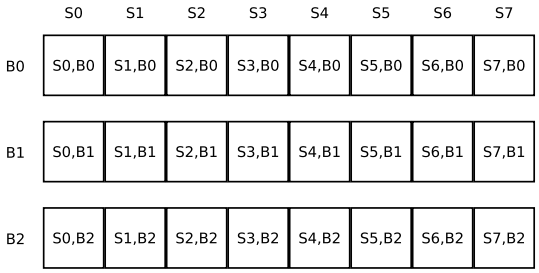

Figure 2.3: An example of the representation of eight variables that assume values representable with three bits. Each 8-bit data-word represent the corresponding bits of the eight variables

.

2.3.1 Data and Operations Remapping

A spin or a coupling can be represented by a single bit of a w-bit word. The transformation we apply on spin and couplings is:

$$\begin{aligned} \sigma_i \in \{-1,1\} \rightarrow \quad & S_i = (\sigma_i + 1)/2 \in \{0,1\} \\ J_{ij} \in \{-1,1\} \rightarrow \quad & I_{ij} = (J_{ij} + 1)/2 \in \{0,1\} \\ E(\sigma_i) \rightarrow \quad & X(S_i) = \sum_j I_{ij} \oplus S_i \oplus S_j = \\ & = (6 - E(\sigma_i))/2 \end{aligned} \tag{2.8}$$

where $\oplus$ denotes the multiplication of corresponding bits inside the data words and as will be soon clear, it is implemented as a logical XOR instruction. Please note that we always have $\Delta E = -2E$, so the quantity X is a positive integer representation of ΔE that is the quantity of interest in the Metropolis algorithm. Values as X can be stored using one w-bit word for each bit that is needed to represent it. So, it is sufficient to define three words X0, X1, X2, one per bit of the representation, from the least to the most significant bit respectively. Figures 2.3 and 2.4 show an example of this type of representation.

This type of representation allows to perform operations on the w bits of a variable concurrently. To obtain such results, operations between integers have to be remapped using bitwise logical operations.

A multiplication between two bits is still represented by one bit, and it is equivalent to the logical XOR:

$$S = A \times B \quad \rightarrow \quad \text{S = A XOR B} \tag{2.9}$$

The result of a sum between two bits requires two bits to be represented. So, if $A+B = S[2]$, where A and B are two w-bit data-words that embed w different spins or couplings, and S is an array of two w-bit data-words, the result is computed as:

$$\begin{aligned} \text{S[0] = A XOR B} \\ \text{S[1] = A AND B} \end{aligned} \tag{2.10}$$

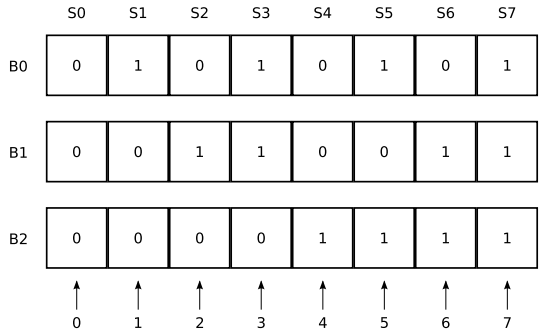

Figure 2.4: The example shows the different values that can assume each of the eight 3-bit variables with this type of representation.

When adding two variables that have to be represented by more than one bit, carry has also to be added to the most significant bit. For example, to add A and B, that are represented with three data-words (because each embedded variable needs three bits to be represented), the following operations have to be performed. The result S requires four data-words to be represented, due to the carry.

$$\begin{aligned}
\text{S[0]} &= \text{A[0]} \quad \text{XOR B[0]} \\
\text{Carry[1]} &= \text{A[0]} \quad \text{AND B[0]} \\
\text{S[1]} &= \text{A[1]} \quad \text{XOR B[1]} \\
\text{S[1]} &= \text{S[1]} \quad \text{XOR Carry[1]} \\
\text{Carry[2]} &= \text{S[1]} \quad \text{AND Carry[1]} \\
\text{S[2]} &= \text{A[2]} \quad \text{XOR B[2]} \\
\text{S[2]} &= \text{S[2]} \quad \text{XOR Carry[2]} \\
\text{S[3]} &= \text{S[2]} \quad \text{AND Carry[2]}
\end{aligned} \tag{2.11}$$

In general, each bit of the results of the sum of two words A and B is given by:

$$\text{S[i]} = \text{A[i] XOR B[i] XOR Carry[i]} \tag{2.12}$$

where Carry[i] is the carry propagated from position $i-1$ and is defined as:

$$\begin{aligned}
\text{Carry[i]} =&(\text{A[i-1] AND B[i-1]}) \text{ OR} \\
&(\text{Carry[i-1] AND (A[i-1] OR B[i-1])})
\end{aligned} \tag{2.13}$$

In Metropolis algorithm often a single-bit variable is added to a multi-bit variable. In this case the sum is simpler. For example, if each variable embedded in A is

represented by a single bit and B by three bits:

$$\begin{aligned} \text{S[0]} &= \text{A[0]} \quad \text{XOR B[0]} \\ \text{Carry[1]} &= \text{A[0]} \quad \text{AND B[0]} \\ \text{S[1]} &= \text{S[1]} \quad \text{XOR Carry[1]} \\ \text{Carry[2]} &= \text{S[1]} \quad \text{AND Carry[1]} \\ \text{S[2]} &= \text{S[2]} \quad \text{XOR Carry[2]} \\ \text{S[3]} &= \text{S[2]} \quad \text{AND Carry[2]} \end{aligned} \tag{2.14}$$

Note that the previous representation does not make assumptions about the variables represented by the bits that compose each data-word, as they are completely independent. Because of that, in the following discussion, talking of a 3-bit variable is equivalent to say that three $w-bit$ data-words are used to represent corresponding bits of w different and independent variables.

2.3.2 Asynchronous Multispin Coding

Assuming that w corresponding spins of w different samples are embedded in the same w-bit data-word, and that spins and couplings values have been remapped to be represented with a single bit, then :

$$\begin{aligned} X(S_i) &= \sum_j I_{ij} \oplus S_i \oplus S_j \\ X &\in \{0,1,2,3,4,5,6\} \end{aligned} \tag{2.15}$$

that is a remapping of equation 2.2 using logical bitwise operations instead of integer operations, as described in section 2.3.1.

The variable X requires three bits to be represented, which means that there are three w-bit data-words to memorize the three bits of X for each spin embedded in a single data-word. The original energy difference ΔE_i is obtained by the following translation function:

$$\Delta E_i = 4 \times X - 12 \tag{2.16}$$

Table 2.3.2 shows the values of X associated to each of the seven possible values of ΔE_i and E_i. Note that X is always positive.

Metropolis algorithm states that a spin-flip is triggered if one of the two following conditions is met:

1. $\Delta E_i < 0$
2. $\rho < e^{-\beta \Delta E_i}$, where ρ is a random number such that $\rho \in [0,1]$

X	0	1	2	3	4	5	6
E_i	-6	-4	-2	0	+2	+4	+6
ΔE_i	-12	-8	-4	0	+4	+8	+12

Table 2.2: The table shows the equivalence between ΔE_i and X.

Condition 1 is verified when $X < 3$, that is equal to say that the most significant bit is equal to zero and in the two least significant bits there is at most only one bit equal to one, as can be seen in Table 2.3. In practice, the spin has to be flipped if the following equation is true:

$$\text{(NOT X[2]) AND (NOT (X[1] AND X[0])) = 1} \tag{2.17}$$

In the case that condition 1 is not true, then condition 2 has to be checked. Taking X into account it is rewritten as:

$$\begin{aligned} \rho &\leq e^{-\beta \Delta E_i} \\ \rho &\leq e^{-\beta(4X-12)} \\ \rho &\leq e^{-4\beta X+12\beta} \end{aligned} \tag{2.18}$$

The inequality can be further manipulated, in order to obtain some convenient properties:

- the computation of exponential function , that is expensive, is not required
- it is easy to check if the spin has to be flipped
- there are not negative values, because we can easily represent only positive integer numbers

These convenient properties emerge if the inequality 2.18 is manipulated in the following way:

$$\begin{aligned} log(\rho) &\leq -4X\beta + 12\beta \\ log(\rho) + 4\beta X &\leq 12\beta \end{aligned} \tag{2.19}$$

We know that $log(\rho)$ is negative, because $\rho \in [0, 1]$. To make it positive we divide both terms by -4β:

$$-\frac{log(\rho)}{4\beta} - X \geq -3 \tag{2.20}$$

However in the inequality there are still negative terms (-3 and $-X$). To make them positive $+7$ is added to both sides:

$$-\frac{log(\rho)}{4\beta} + 7 - X \geq 4 \tag{2.21}$$

ΔE	X		**not** X	$7-X$	
-12	0	000	111	7	111
-8	1	001	110	6	110
-4	2	010	101	5	101
0	3	011	100	4	100
+4	4	100	011	3	011
+8	5	101	010	2	010
+12	6	110	001	1	001

Table 2.3: The relation between ΔE and X and shows the binary representation of X, that is convenient to simplify the update procedure.

Note that the difference $(7 - X)$ can be computed as the logic negation of the value of X, as can be seen in Table 2.3. The inequality 2.18 can finally be rewritten as:

$$-\frac{log(\rho)}{4\beta} + \overline{X} \geq 4 \tag{2.22}$$

It can be easily seen that $\Delta E_i \leq 0$ only if the most significant bit of $\overline{X}$ is equal to one. Note that this implies that to verify condition 1 ($\Delta E_i < 0$) it is no necessary to check all the bits that represent X, but only the negation of its most significant bit. If it is equal to one, the spin flips, otherwise the test with the random number (condition 2) has to be performed.

Let us define R as the first term of inequality 2.22:

$$R = \left[-\frac{log(\rho)}{4\beta} \right] \tag{2.23}$$

It is sufficient to take R and verify that the result of the sum $(R + \overline{X})$ is greater or equal than 4:

$$S = R + \overline{X} \geq 4 \tag{2.24}$$

Because this test is performed only if $\overline{X} \geq 3$, to decide if the spin have to be flipped, it is sufficient to perform the sum $(R + \overline{X})$ and to check if the result is greater or equal than 4. The result is greater of equal than 4 if its most significant bit is equal to one.

To summarize, the spin-flip is realized performing a logical XOR between its value and the one-bit variable C defined as:

$$\text{C} = (\text{NOT X[3]}) \text{ OR S[3]}) \tag{2.25}$$

that by definition (it is the OR of the two bits that are equal to one if the first and the second spin-flip conditions are true) is equal to 1 only when the spin has to be flipped.

To obtain R it is not necessary to calculate a logarithm. It is sufficient to check if R is in the set $\{0, 1, 2, 3\}$, because the case of $R > 3$ was already covered checking `S[3]`. The three following inequalities allows to determine the value of R performing comparison with β-dependent constant values:

$$\begin{aligned}
-\frac{log(\rho)}{4\beta} \geq 3 \quad &\Rightarrow \qquad\qquad\quad \rho \leq e^{-12\beta} \quad \Rightarrow R = 3 \\
3 > -\frac{log(\rho)}{4\beta} \geq 2 \quad &\Rightarrow e^{-12\beta} \leq \rho \leq e^{-8\beta} \quad \Rightarrow R = 2 \\
2 > -\frac{log(\rho)}{4\beta} \geq 1 \quad &\Rightarrow e^{-8\beta} \leq \rho \leq e^{-4\beta} \quad \Rightarrow R = 1 \\
1 > -\frac{log(\rho)}{4\beta} \quad &\Rightarrow e^{-4\beta} < \rho \quad \Rightarrow R = 0
\end{aligned} \tag{2.26}$$

Note that the exponential comparison terms do not depend on spin value, so that can be computed once for the whole lattice (until β remains constant). In practice it is sufficient to execute a sequence of three comparisons, starting from the smallest value $e^{-12\beta}$ to the largest $e^{-4\beta}$:

1. R = ($\rho \leq e^{-12\beta}$) ? 3 : R;
2. R = ($\rho \leq e^{-8\beta}$) ? 2 : R;
3. R = ($\rho \leq e^{-4\beta}$) ? 1 : 0;

or alternatively

1. R = ($\rho > e^{-4\beta}$) ? 0 : 1;
2. R = ($\rho > e^{-8\beta}$) ? R : 2;
3. R = ($\rho > e^{-12\beta}$) ? R : 3;

that is useful in those ISA that include only instructions for the "greater-than" comparison.

Asynchronous multispin coding is convenient because a single random number can be used to update in parallel all the spin embedded in a word. Using a trivial integer coding, only 5 sums and 6 multiplications are needed to calculated ΔE, while with asynchronous multispin coding 27 XORs and 9 ANDs are required. In most modern architecture the cost of integer and logical operations is the same. However, the generation of random numbers is heavier than spin update, and of we can expect that additional instruction will be required in real code to calculate addresses and load data.

2.3.3 Synchronous Multispin Coding

The same type of representation can be used for synchronous multispin coding. In this case neighbor spins are embedded into the same word, so to get their values rotations inside the vector data-words are required. Moreover, a different random number has to be computed for each spin of the same system. The energy differences ΔE_i can still be computed as in the asynchronous case and also the operation that decide if the spins have to be flipped are unchanged. This means that if we put V spins of the same systems are embedded in a vector data word, we obtain the parallel update of V spins. However, random number generation is not so simple. Because 32-bit random numbers are required, the concurrency of their generation is fixed, and as a consequence for high SIMD-granularities V we can expect that the random number generation to be the dominant fraction of the computational kernel execution time.

2.3.4 Mixed Multispin Coding

Asynchronous multispin coding can assure the best asynchronous spin update time, because a random number is shared between replicas and because data structures allow to easily access to neighbors' values. However, to make dynamic analysis asynchronous parallelism is of a little use, and is better to exploit the synchronous parallelism.

Actual SIMD architectures allow to perform operations on various scalar variables in parallel. It's possible to use a mixed method that exploit SIMD features to update spins of the same system in parallel (synchronous parallelism) and at the same time embed more than one system in each scalar variable in order to update several systems concurrently (asynchronous parallelism).

In addition, the same random number may be used to updated all the spins of independent samples of one site at once, as the small correlation introduced is compensated by being the sample coupling configurations completely uncorrelated. This is a crucial point, as if the w bits represented on the same lattice or in different replicas, using the same random number, that permits a considerable improvement in performance, would violate the algorithm.

2.4 Gaussian Model

The main difference between the Binary and the Gaussian model is that in the Gaussian model the couplings are no longer binary values, but are floating point numbers. This implies that it is no longer possible to use the multispin coding technique to update spins in parallel, because computations are done between floating-point variables. Because coupling are floating point number, ΔE is also a float variable, so it

has to be compared with the logarithm of the random number.

In conclusion, no obvious optimizations are possible, but datatypes are well mapped into actual words. Floating point instructions have to be used, and they are typically slower than logical bitwise instructions. We expect a limited drop in performance when considering a single system (synchronous parallelism), because the pattern of memory access is unchanged and the latency of instructions should be mitigated by pipelining.

2.5 Conclusions

In this chapter we have introduced the Edward-Anderson model for spin glasses and the Metropolis algorithm that is used to study the problem. In principle the algorithm is not efficient in current computer architecture, because its basic data type, the spin, require much smaller memory than those available in the smaller data-word available in a processor (usually a byte). We have presented a technique, the multispin coding [71], that allows to embed more than one spin in a single data-word and to update them in parallel. This technique can be used to implement programs that update concurrently spins of the same lattice and also of different samples, thus achieving a high degree of parallelism.

I'd like to express my gratitude to Andrea Maiorano for his work on multispin coding, that was of inestimable value for the development of my Ph.D. thesis.

Chapter 3

Spin Glasses on Multi-Core Architectures

Spin glasses simulations realized with the Metropolis algorithm are intrinsically parallel, as $L^3/2$ spins can be concurrently updated. If enough resources, in terms of memory and processing units, are available a very high degree of concurrency can be exploited. For example, a cubic lattice of linear size $L = 64$ is composed by 262144 spins and 131072 of them can in theory be concurrently updated. None of the currently available computer architecture allows such a large degree of parallelism when using a single processor. However, as currently available multi-core systems have high computing power and large embedded memories, there is nonetheless the chance to exploit a high degree of parallelism.

This chapter is organized in the following way. In the first section, an abstract multi-core architecture will be defined, followed by the description of the data structures that are used to describe the data set of a generic cubic lattice. Three different implementations of Monte Carlo spin glass simulations on the abstract multi-core architecture will be discussed and analyzed, in particular emphasizing the balance between computational power and the amount of data that has to be exchanged. In particular, we will propose balance equations that, taking into account both algorithmic (the lattice size, the SIMD-granularity) and architectural (the number of available cores, the bandwidth of inter-core and core-to-memory communications) parameters describe the behaviour of and ideal case in which details concerning syncronization and latency are not taken into account. The target of this model is to determine in which cases the computational time is greater of data transfer time, so that it is possible to efficiently use all the available cores. Finally, the difference and the similarities of the strategies will be discussed.

3.1 An Abstract Multi-core Architecture

To analyze the implementation of Metropolis algorithm and its expected performance we define an abstract multi-core processor described by the following properties:

- C cores are integrated on a single processor
- each core support SIMD-like vector instructions on W-bit vector data-words. Each vector data-word can be partitioned in two or more scalar data-words of different sizes. We call *SIMD-granularity* V the number of scalar data words allowed within one vector data-word. For example, if a vector data-word of 128-bit can be partitioned in sixteen 8-bit scalar data-words, the processor has SIMD-granularity $V = 16$
- each core has access to a local private memory (also referred to as *onchip memory*)
- each core can access data stored on the local memories of the other cores
- all core shares an (arbitrarily) large main memory external to the chip (also referred to as *offchip memory*)
- memory accesses and data-transfers can be performed concurrently with computation
- memory access performance has a strong dependence on "distance": latency to the local memory is constant and small. Access to the local memories of the other cores has a significantly longer latency, while access to the shared memory is still much slower

The mapping on CBE is clear: ignoring the PPE, a chip has $C = 8$ SPEs with their own 256 KBytes of local memory. Each core can access the local store of other cores through the EIB, while data transfer are performed independently by MFC. The mapping of the abstract architecture on CBE will be discussed in more details is chapter 4. In any case, this abstract architecture is well suited for Cell Broadband Engine, that does not have a complex hierarchy of caches and that allow a complete software control of local memories. However, it is enough generic to describe, with a sufficient grade of approximation, the main multi-core architectures currently available, although much more accurate models have been proposed [72].

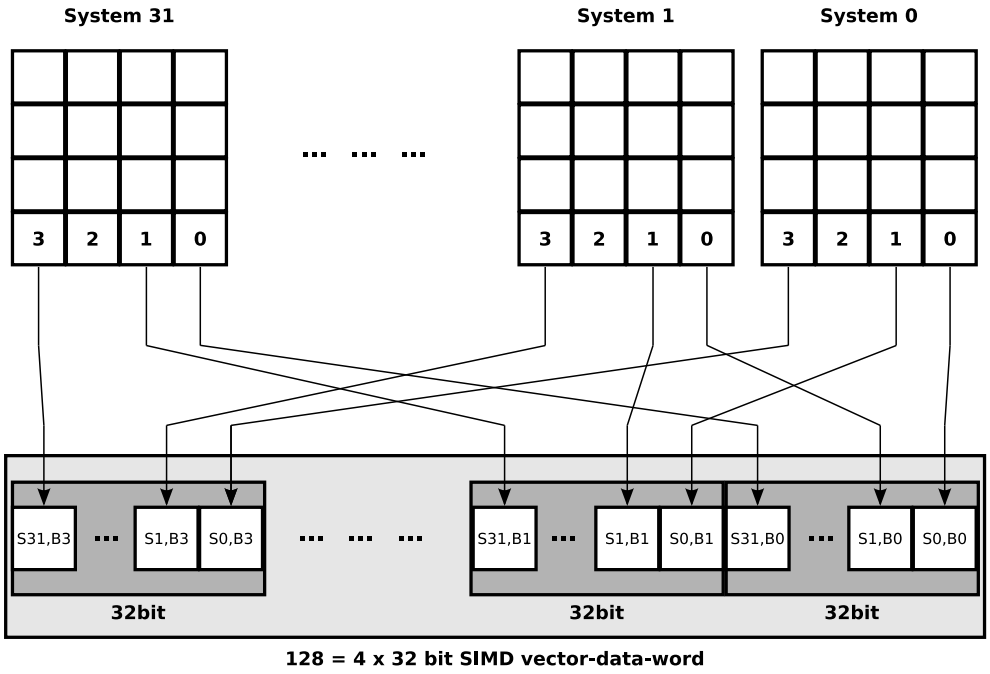

Figure 3.1: An example of the mapping of $w = 32$ different systems into one $W = 128$-bit SIMD vector data-word. Each vector represent $V = 4$ spins of each system.

3.2 Data Structures

For the Binary model, spins and couplings can be easily represented by a one-bit variable. Commodity architectures do not support one-bit words, so spins and couplings have to be allocated on scalar words of W/V bits, with $V = 2, 4, 8, 16, \ldots$ strictly depending on the architecture, with a large waste of memory resources. This naive representation can be improved if spin-values of different systems are allocated in the same variable. For example, w corresponding spins of w different lattices can be allocated on a single word of $w = W/V$ bits. This allows to save space and to update in parallel spins of different lattices as will be later described.

Following this approach, we allocate V spins of W/V different lattices on a single W-bit vector data-word of a system with SIMD-granularity V, as shown in Figure 3.1. We then use SIMD instructions to update in parallel V spins of a single lattice and to simulate concurrently $w = W/V$ different lattices. In this approach – using again spin glass jargon – we asynchronously handle w lattices and, at the same time, we synchronously consider V spins of each lattice.

When considering the Gaussian model, it is no longer possible to embed corresponding spins of different systems in the same scalar data-word, but we can still use a vector data-word to represent V different spins or couplings of the same system. Both spins and coupling have to be represented as 32-bit floating-point numbers, so

the allowed SIMD-granularity is $V = W/32$. Given the fixed SIMD-granularity and the absence of different samples, the considerations about balance between computations and data transfers are the same for both Binary and Gaussian model.

A cubic lattice of linear size L is represented by four data structures:

- `S`: L^3 spins values
- `Jx`: L^3 couplings between each spin and the first-neighbor in positive X direction (Jx)
- `Jy`: L^3 couplings between each spin and the first-neighbor in positive Y direction (Jy)
- `Jz`: L^3 couplings between each spin and the first-neighbor in positive Z direction (Jz)

Given a SIMD-granularity V and the consequent number of samples w, a lattice requires M bits to be represented:

$$M = 4 \times w \times L^3 \tag{3.1}$$

For example, when $L = 64$ and $w = 8$, a very common case, 1 Mbyte of data is required. There are essentially two correlated problems regarding data structures: where to store them and how consequently distribute workload between cores.

One of the simplest way to distribute the workload among the cores is to assign to each core a sub-lattice of L/C adjacent XY-planes (see Figure 3.2. A core is responsible of the update of all the spins of is sub-lattice. The data set can be allocated in main or local memories. Basically, if the data set is small enough, it is allocated in local memories. Otherwise, the local memory is used to store only the data required to update a fraction of the sublattice assigned to the core. We will describe and analyze three main cases of allocation of data set on the storage resources of the abstract multicore architecture:

1. data structures are small enough to be distributed between the local stores of the used cores. Main memory access is not required
2. data structures are too big to fit in local memories, so the lattice has to be stored in main memory. However, lattice is small enough to allow each core to load into its own local memory enough data to update a whole plane. Moreover, a large fraction of data can be reused to subsequently update an adjacent plane, thus reducing data exchange between main memory and local memories

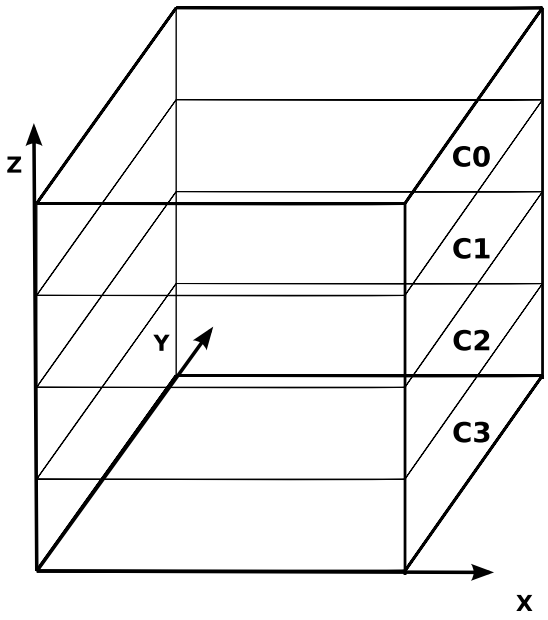

Figure 3.2: The cubic lattice is subdivided in sub-lattices of L/C XY planes, and each sub-lattice is assigned to a different core. Note that in this figure each element of the lattice represents both spins and couplings.

3. data is stored in main memory, but it is not possible to load into each local memory enough data to update a full plane of the lattice. Each plane is splitted into several smaller *slices* that are small enough to allow each core to load into its local store enough data to update a slice and then reuse some of that data to update an adjacent slice

The amount of data that has to be exchanged is different in the three cases. However, when data is in local memories only core-to-core communication is required, and in the case of CBE the EIB has a large aggregate bandwidth of more than 200 GB/s. In our implementations we extensively use latency-hiding techniques and in particular data prefetching, so latency should not be an issue and the available bandwidth is probably sufficient to avoid that data exchange is a bottleneck. When data is stored in main memory, the available bandwidth is only 25.6 GB/s, so it is more likely that it can be a severe bottleneck.

The distribution of data set to the cores is done along only one dimension in order to minimize the interaction between cores: in this way each core has to exchange data or to synchronize with only two neighbors. This it is not a limiting factor, because our target linear lattice size ($L \in [16, 128]$), as will be later shown, can be managed by a single CBE or by two-processors configurations in terms of local memories. Systems with more than two processors have not be taken into consideration.

In the ideal case the lattice is stored in local memories, because they are fast and because inter-core data exchanges are fast, but local memories are relatively small, so they could not be able to store the whole lattice. If only a small fraction of the data structures can be held in local memories, it is important to define a strategy that minimizes data exchange.

Spins and couplings can be represented by the same primitive data type. In this document the term *element* will be used to refer to a data-word that contains w corresponding spins or couplings of w different samples. Because their memory requirements are the same, it can be simpler to analyze memory usage and requirements in terms of elements rather than separate spins and couplings.

The spins and the couplings that form the lattice are divided in a checkerboard style in two disjoint sets *black* and *white*. An important implication is that the nearest neighbors of a "white spin" $S(x, y, z)$ are all in the black set.

Checkerboard subdivision is useful because all the spin of the same color can be concurrently updated, because spins of the same color do not share coupling terms. Note that two spins cannot be updated in parallel only if they are nearest neighbors, because they share couplings. The number of spins of a given color ($L^3/2$) is an upper bound of the achievable parallelism.

In our case, the original lattice of L^3 spins is subdivided in two half-lattices of $L^3/2$ spins and the same subdivision is applied to all the couplings, as shown in Figure 3.3.

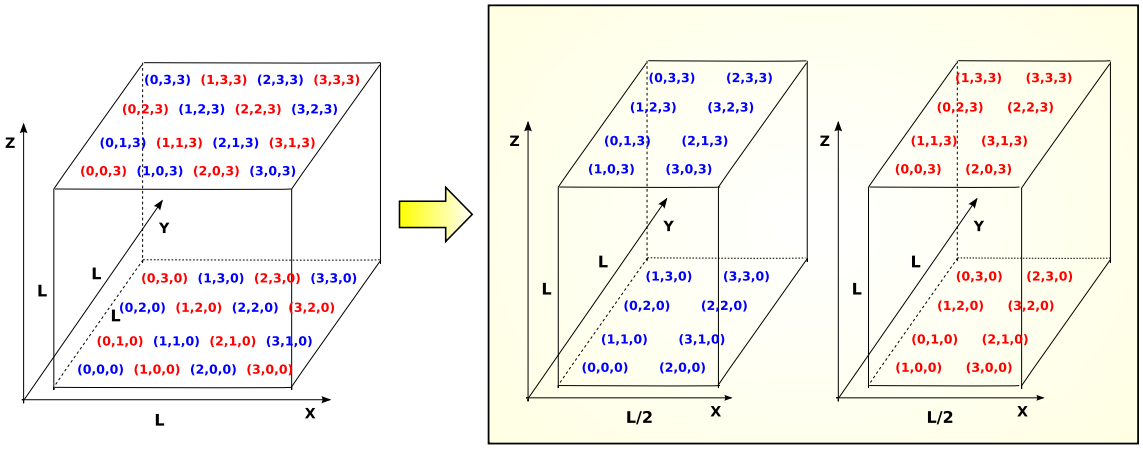

Figure 3.3: The checkerboard subdivision of spin and coupling data structures. Red and blue are used in this and in the following figures instead of white and black.

We assume that the checkboard subdivision is done along the X axis. Due to this subdivision, the concept of *half-plane* has to be introduced. Each half-plane is the set of all the spins of a plane with the same color, so it contains $L^2/2$ spins. In terms of storage requirements, it needs $2 \times w \times L^2$ bits of memory to store all the spins of the half-plane and the three couplings terms associated to each spin.

To preserve the semantics of the spin glass simulation, in a single Monte Carlo iteration all the half-planes of a given color have to be updated before the half-planes of the opposite color.

As said before, V spins of the same system are embedded in the same vector data-word. We assume that in a vector data-word are embedded spins that are in the same X-line in the half-plane. In other words, the same direction subject to checkerboard subdivision is also subject to vectorization. As a consequence, a half-plane is composed by $(L/(4 \times 2) \times L)$ vector data-words. Note that black and white subdivision implies that the spins embedded in a vector data-words are not nearest neighbors, and later will be shown how this property can be exploited to reduce storage requirements.

We can assume that the spins and the couplings of a lattice are basically stored in main memory in eight different arrays:

- two arrays of $L^3/2$ scalar elements that are used to represent the spins
- six arrays of $V = L^3/2$ scalar elements that are needed to represent Jx, Jy and Jz couplings

In the ideal case all the data required to update the lattice is available in local memory, but the size of on-chip memory makes this assumption not realistic. Each of the three proposed strategies of allocation of the data set assumes a fraction of the

lattice as the basic updatable unit. To update a spin, that is the smallest updatable unit, the following values are required:

- the value of the spin
- the values of its six first-neighbors spins
- the values of the six couplings that it shares with its first-neighbors

In total 13 elements are needed to generate a new spin value. The black and white subdivision ensures that, for each spin:

- its value is not needed for the (potentially concurrent) update of other spins of the same color
- its six first-neighbors are all of the opposite color

An important consequence is that the original value of a spin being updated can be overwritten, because it is not required for the update of other spins of the same color. Similarly, the values of its neighbors spins will be update only in the next step, because they are all of the opposite color, so they are not overwritten in thi step. This implies that separate output data structures are not needed, thus reducing the storage requirements.

In terms of black and white half-planes, to update the z-th white XY half-plane, the following half-planes are required:

- the white S half-plane (z)
- the black S half-planes $(z-1)$, (z) and $(z+1)$
- the black and white Jx half-planes (z)
- the black and white Jy half-planes (z)
- the white Jz half-plane (z) and the black Jz half-plane $(z-1)$

In total 10 half-planes are required to update an single half-plane. Given that when considering a single spin as the minimum updatable unit 13 different elements are required, it is convenient to assume that a half-plane as the minimum updatable unit. It is not possible take a whole plane as basic updatable-unit, due to algorithmic constraints (it is not possible to concurrently update spins that shares a couplings, that would mean that they have the same color). Conveniently, the update of a half-plane requires all data relative to the same (z) plane, plus three extra half-planes. Working with half-planes also assures that all neighbors in the X and Y directions are available.

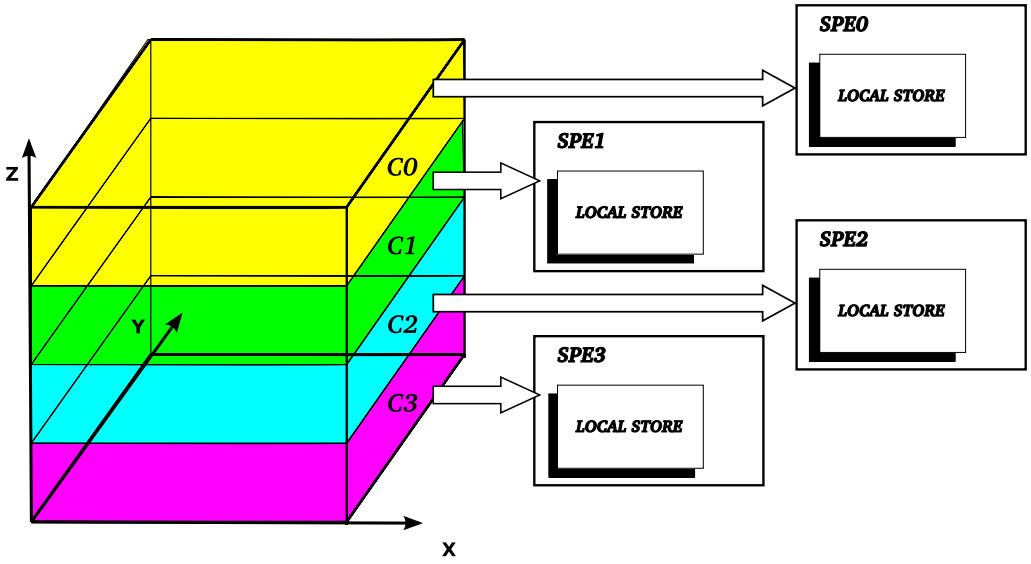

Figure 3.4: Each core loads into its local memory a sub-lattice of L/C XY-planes and some additional border data. The sums of all the local memories allows to store the representation of the whole lattice.

3.3 Local Memory Version

In this section will be described how to map the algorithm on the abstract multi-core in the case in which data structures are small enough to be distributed among local memories and main memory access is not required (or, more precisely, it is required only at the start and at the end of a run to load/save data from/to the mass storage).

As described earlier, the cubic lattice is subdivided in sub-lattices of L/C planes and each sub-lattice is assigned and loaded into the local memory of a different core (Figure 3.4).

As long as all the L/C planes are in local memory, it is possible to update the whole sub-lattice without accessing main memory, given that:

- required border data can also be stored in each local memory
- after the update of all the same spins of a given color border data is exchanged between cores

The half-planes that compose each sub-lattice are in the *local range* $z \in [0, ((L/C)-$

1)]. As a consequence, the coordinates of each spin or coupling are in the following ranges:

- $x \in [0, L-1]$
- $y \in [0, L-1]$
- $z \in [0, (L/C)-1]$

Not all required data is included in this local coordinates system. The spin values and Jz couplings relative to the plane ($z = -1$) are required to update the spins of the plane ($z = 0$) and the spin values relative to the plane ($z = L/C$) are required to update plane ($z = (L/C) - 1$). As a consequence, the following additional planes are required to update the whole sub-lattice assigned to a core:

- spin plane $z = (-1)$
- spin plane $z = (L/C)$
- Jz coupling plane $z = (-1)$

From a high level of abstraction, the update procedure can be described as a loop in which, for each Monte Carlo step, each core executes the following tasks:

1. Update the white spins of its own sub-lattice.
2. Exchange border data with the other cores.
3. Update the black spins of its own sub-lattice.
4. Exchange border data with the other cores.

In phases 1 and 3 each core updates $((L^3/2C)$ spins of the same color.

Couplings are constants, so they can be loaded at the very start of the simulation, and in phases 2 and 4 only spin values have to be exchanged between cores.

In particular, the just updated values of the spins on half-planes ($z = 0$) and ($z = (L/C) - 1$) have to be sent to the *previous core* and the *next core*. The previous and the next core are defined as the cores housing the two adjacent sub-lattices. Using GETs instead of SENDs, half-planes ($z = -1$) and ($z = L/C$) have to be get from the *previous core* and the *next core*, without changes in the sematic of the scheme.

The previous core needs spin values of ($z = 0$) to update the spins of the opposite color that lies on its plane ($z = (L/C) - 1$), while the next core needs the spin values of ($z = (L/C) - 1$) to update the spins of the opposite color that lies on its plane ($z = 0$).

Note that although it is necessary to wait the end of data transfers to update the first and the last half-planes of a sub-lattice, the half-planes in the range $z \in [1, (L/C) - 2]$ can be immediately updated. This implies that the transfer of border half-planes and the update of internal half-planes can be performed concurrently.

The procedure for updating the spins of the same color of a sub-lattice can be described in more details as the following sequence of steps:

1. Update the half-plane $z = (0)$.
2. Send the half-plane $z = (0)$ to the previous core.
3. Update the half-plane $z = ((L/C) - 1)$.
4. Send the half-plane $z = ((L/C) - 1)$ to the next core.
5. Update the half-planes in the range $z \in [1, (L/C) - 2]$.
6. Wait the end of data transfers.

This scheme hides synchronization details and shows how steps 2 and 4 can be overlapped with step 5. The time required to execute the algorithm (for the spins of a given color) can be estimated:

$$T = T_1 + T_3 + \max\{(T2 + T4), T5\} \tag{3.2}$$

So a good balancing criteria requires that $T2 + T4 = T5$. Let us analyze in details the time required to execute each step. The time required to update a half-plane is:

$$T_1 = T_3 = \tau(V) \times \frac{L^2}{2} \tag{3.3}$$

where $\tau(V)$ is the time needed by each core to update one spin of a single system. Note that, because all cores are independently and concurrently updating a different half-plane, $\tau(V)$ does not depend on the number of cores. However, the update of the first and the last half-planes is not overlapped with data transfer, so it does not influence the balance but has to be taken in account when calculating the total execution time.

The time required to exchange all border data between cores is:

$$T_2 = T_4 = \frac{w \times C \ L^2}{B} \tag{3.4}$$

where B is the aggregate bandwidth (in bits) of core-to-core communications. The time to update all half-planes in the range $[1, (L/C) - 2]$ is:

$$\tau(V) \times \left(\frac{L^3}{2 \times C} - L^2 \right) \tag{3.5}$$

Now is is possible to define the equation $(T_2 + T_4) = T_5$ that balance the system:

$$\tau(V) \times \left(\frac{L^3}{2 \times C} - L^2 \right) = \frac{w \times C \times L^2}{B} \tag{3.6}$$

If $T_5 > (T_2 + T_4)$ processing is the bottleneck while, in the opposite case, data transfers among cores limit the performance.

The procedure described above is realizable only if the data is small enough to be stored into the local memories. Each local memory must be large enough to host the following data structures:

1. the spin array S, that requires M_S bits:

$$M_C = w \times \left(\frac{L^3}{C} + 2 \times L^2 \right) \tag{3.7}$$

2. the coupling array Jx, that contains Jx_n elements:

$$M_{Jx} = w \times \frac{L^3}{C} \tag{3.8}$$

3. the coupling array Jy, that contains Jy_n elements:

$$M_{Jy} = w \times \frac{L^3}{C} \tag{3.9}$$

4. the coupling array Jz, that contains Jz_n elements:

$$M_{Jz} = w \times \frac{L^3}{C} + L^2 \tag{3.10}$$

Globally, M bits of local memory are required for each core:

$$M = w \times \left(\frac{4L^3}{C} + 3L^2 \right) \tag{3.11}$$

Note that this number depends on the number of cores C, so the number of used cores directly influences the size of the maximum manageable lattice. For example, for a lineare lattice size $L = 16$, a number of cores $C = 8$ and a SIMD-granularity $V = 4$, in total $M = 11264$ bytes of local store space are required in each SPE, that is the 23% of the available space. Extending lattice size to $L = 40$ would increase the memory requirement to $M = 147200$ bytes, that is more than the 70% of the available space.

3.3.1 Synchronization

Data transfers between cores have to be synchronized, because each core can be seen as both a producer and consumer of data. Data transfers can be realized as GET or as RECEIVE data transfer operations that, except for some implementation details, are equivalent for our purposes. If each core gets the half-planes from its neighbor it must be sure that the half-planes have been already updated and it cannot overwrite its own data structures if it isn't sure that its neighbors have already got them. Basically, the following operations have to be performed, assuming to use GETs to perform data transfers:

1. update the white half-plane $z = (0)$
2. update the white half-plane $z = ((L/C) - 1)$
3. grant the previous core the authorization to get the white half-plane $z = (0)$
4. grant the next core the authorization to get the white half-plane $z = ((L/C) - 1)$
5. wait the authorization to read white half-plane $z = (-1)$ from previous core
6. start to get the white half-plane $z = (-1)$ from the previous core
7. grant the previous core the authorization to overwrite the white half-plane $z = (-1)$
8. wait the authorization to read the white half-plane $z = (L/C)$ from next core
9. start to get the white half-plane $z = (L/C)$ from the next core
10. grant the next core the authorization to overwrite the white half-plane $z = (L/C)$
11. update the white half-planes in the range $z \in [1, (L/C) - 2]$
12. wait the end of the transfer of the white half-plane $z = (-1)$
13. wait the end of the transfer of the white half-plane $z = (L/C)$
14. wait the authorization to overwrite the white half-plane $z = (0)$ from previous core
15. wait the authorization to overwrite the white half-plane $z = ((L/C) - 1)$ from next core.
16. swap colors

Steps 12 and 13 are needed to assure that all data needed to update black spins is available. Steps 14 and 15 are needed to be sure that neighbors has already got border data so that, for example, after the update of the black spins it is possible to proceed with another Monte Carlo step to change the values of the white border spins.

To begin the update of the spins of a given color, the following conditions have to be met:

- half-planes $z = (-1)$ and $z = (L/C)$ of the opposite colors have to be available
- neighbors cores have already got the current color half-planes $z = (0)$ and $z = ((L/C) - 1)$

After the update of the current color spins, there is no need to wait now for the overwrite authorizations, because the current color spins will be overwritten only after the update of the spins of the opposite color. Overwrite authorizations are really needed only at the beginning of the next update of the spins of the same color. Waiting overwrite authorizations so early implies a strong synchronization between cores, because it assures that all the cores are always concurrently updating spins of the same color.

3.4 Main Memory

In this section we assume that using C cores it is not possible to use only the local memory to store a lattice of a given linear size L. In this case we assume that each local memory is however large enough to store all the data needed to update a half-plane and that, moreover, a large fraction of these data can be reused to update another adjacent half-plane.

As seen in section 3.2, ten half-planes are required to update a single half-plane, which implies that at least ten half-planes have to be stored in local memory. This is obviously an upper bound to the size of L. Please note that, while in the local memory version of the program each core had to store all its own sub-lattice in local-memory, now the amount of data loaded into each local memory is independent by the number of cores. The local memory usage for a lattice of linear size L is at least:

$$M = 5 \times w \times L^2 \tag{3.12}$$

Although this is an upper bound to the size of L, ten half-planes are not sufficient, because data structures have to be dedicated to the data exchange between main and local memory, so that computation and data transfers can be overlapped.

The main data structure, containing the values of all spins and all couplings, resides in main memory and occupies $4 \times w \times L^3$ bits. A sub-lattice of L/C planes is assigned to each core.

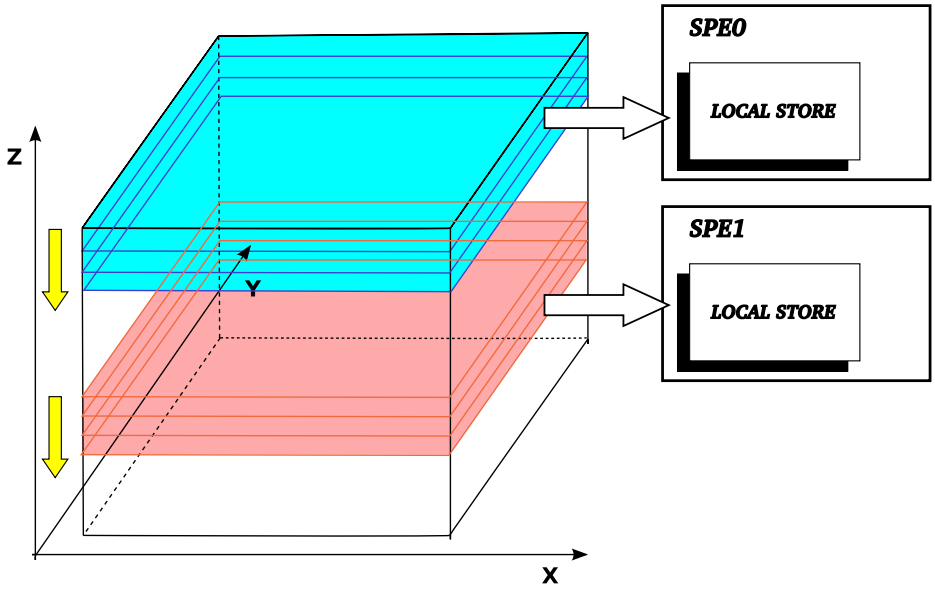

Figure 3.5: Each cores has to update a sublattice of L/C XY-planes that is too large to be stored in local memory. Instead, a set of half-planes is loaded from main to local memory to update a single half-plane.

The algorithm can be described as a loop where each core first updates all its white half-planes in the range $[0..(L/C)-1]$, synchronizes with the other cores and then repeat the same procedure to update the black half-planes. The update of each half-plane (i) in the range $[0..(L/C)-1]$ is performed executing the following steps:

1. if $i > 0$, store half-plane $(i-1)$ into main memory
2. if $i < (L/C)-1$, load plane $(i+2)$ from main to local memory
3. update half-plane (i) and proceed to half-plane $(i) = (i+1)$

Computation and data transfers are overlapped, because while the half-plane (i) is updated, half-planes $(i-1)$ and $(i+2)$ are respectively stored to and loaded from main memory.

When the half-planes of both colors have been updated, then the current Monte Carlo iteration is over, and the procedure can be repeated for the next iteration.

Each core loads two planes $z = (-1)$ and $z = (L/C)$ that belong to its neighbors, so synchronization is required: each core cannot update its border half-planes if the neighbors have not read them yet. In practice it is sufficient to perform a global barrier when swapping from a color to its opposite. This type synchronization assures that cores are not concurrently updating spins of different colors. To update a spin of a given color the values of its neighbors are required but, because of the checkerboard subdivision, all neighbors are of the opposite color. So, if all cores are updating spins of the same color, we are sure that the required data is not concurrently overwritten.

The program is balanced if $T_1 + T_2 = T_3$. The time required to store back a half-plane to main memory is:

$$T_1 = \frac{w \times C \times L^2}{2 \times B} \tag{3.13}$$

where B is the bandwidth of main memory. The time required to load the next eight half-planes from main memory is:

$$T_2 = \frac{8 \times w \times C \times L^2}{2 \times B} \tag{3.14}$$

and time required to update a half-plane is:

$$T_3 = \tau(V) \times \frac{L^2}{2} \tag{3.15}$$

Because all cores perform the update in parallel, in equation 3.15 the number of cores C does not appear. Equations 3.13 and 3.14 assumes a main memory bandwidth B (in bits), that is a resource shared between all the cores. As a consequence, the

time required to transfer all the data depends on the number of cores C. The global time required for data transfers is:

$$T_1 + T_2 = \frac{9 \times w \times C \times L^2}{2 \times B} \tag{3.16}$$

In a balanced system $T_1 + T_2 = T_3$:

$$\frac{9 \times w \times C \times L^2}{2 \times B} = \tau(V) \times \frac{L^2}{2} \tag{3.17}$$

which implies:

$$\tau(V) = \frac{9 \times w \times C}{B} \tag{3.18}$$

This formula shows how the balance does not depend on the lattice linear size L and, given the spin update time $\tau(V)$ and the bandwidth B of the main memory, there is a number of cores C that keeps the system balanced.

To ensure the concurrency of computation and data transfers it is necessary to allocate in local memory enough data structures to preload the half-planes that are needed at the next updated step and to store the results, as shown in Figure 3.6.

More precisely, while updating the white half-plane (z) the following half-planes have to be loaded from main to local memory:

- white spins half-plane ($z + 1$)
- black spins half-plane ($z + 2$)
- white and black Jx couplings half-planes ($z + 1$)
- white and black Jy couplings half-planes ($z + 1$)
- white couplings Jz half-plane ($z + 1$)
- black coupling Jz half-plane (z)

The same data structure can be used to store back to main memory the half-plane ($z-1$) (that has been updated in the previous step) and then to load half-plane ($z + 2$). In total eight half-planes are needed for data transfers, and consequently eighteen half-planes have to be allocated into local memory. This implies that each local memory has to be large enough to store M bits of data:

$$M = 9 \times w \times L^2 \tag{3.19}$$

In equation 3.19 the number of cores C does not appear, because the maximum size of L does not depend on the number of cores, but only by the size of the local memory. This implies that with this implementation the size of the largest manageable lattice does not depend on the number of cores (as in the program that use only local memories), but the number of cores that can efficiently used depends on the linear size L and on the bandwidth B of the main memory.

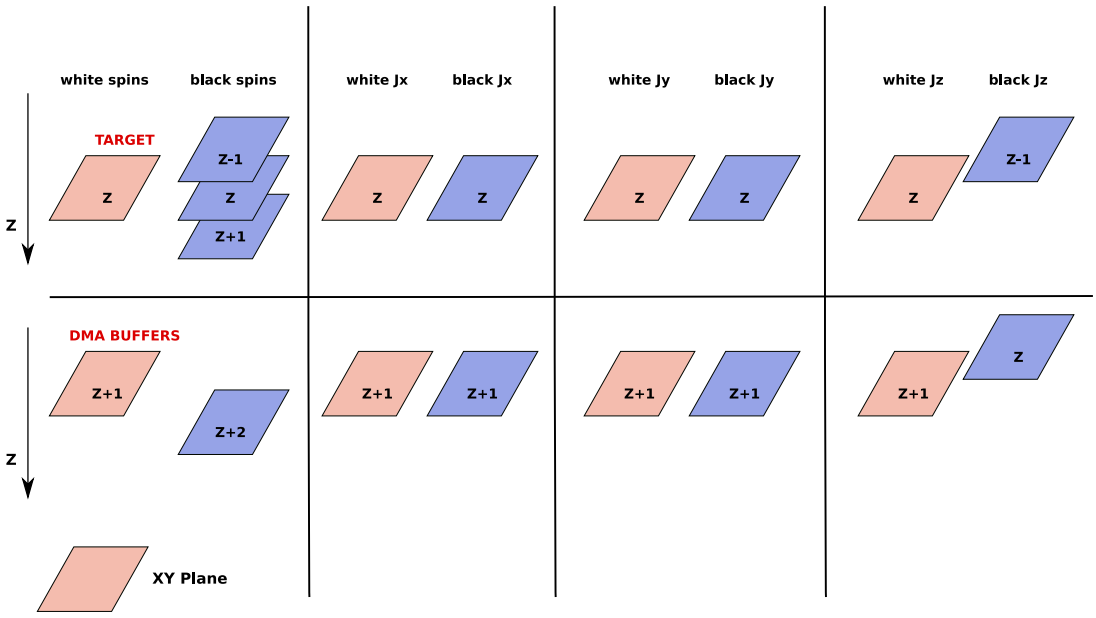

Figure 3.6: The buffers that have to be used to update a half plane while storing back a previously update half-plane and loading from main memory the data required to update the next half plane

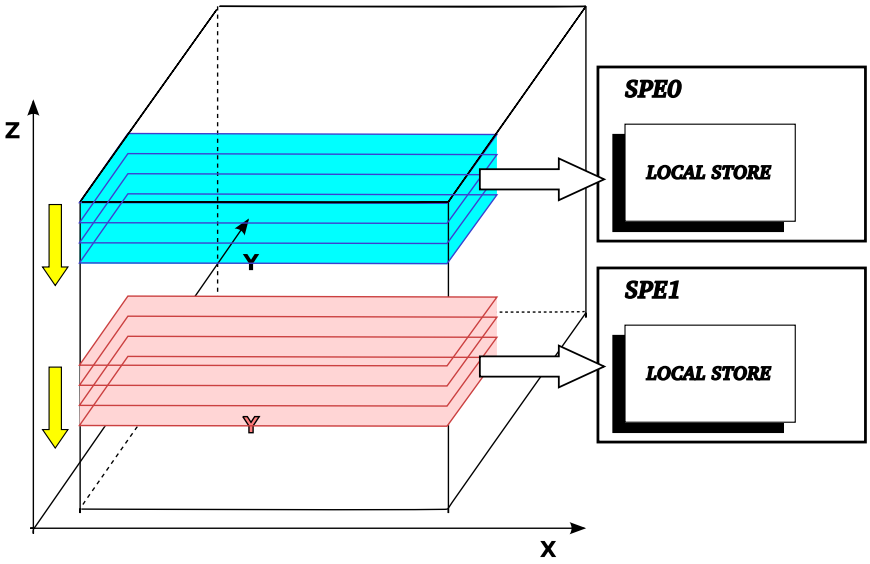

Figure 3.7: A sub-lattice of L/C XY-planes is assigned to each cores. Half-planes are subdivided in slices that are sequentially updated by the core in charge for that half-plane.

3.5 Main Memory and Slices

The largest lattice that can be updated with the strategy described in the previous section is determined by the size of the local store, because enough planes have to be available in local memory. The basic idea of this new strategy is that, although the lattice the we want to update is so large that it cannot be managed by that program, it is possible to define a smaller chunk of data, referred to as *slice*, that can be stored and updated using the available local memory.

The update procedure is the same of the previous case, but it has to be repeated p times, where p is the number of slices that compose a half-plane. Note that, because the slice subdivision is limited to single planes, there is no need for extra synchronization between cores, and a barrier is sufficient.

An ideal slice is composed by $p \times q$ spins, where p is a fraction of $L/2$ and q is a fraction of L, but for our purposes it is sufficient for a slice to have $L/2$ spins in the X direction (the same of the half-plane) and $S = L/p$ spins in the Y direction. In this way the complexity of the procedure is reduced while the size of the largest manageable lattice is still large enough for our purposes.

An half-plane can be seen as splitted into p slices along the Y axis. To update the first and the last X lines the first neighbors in the Y direction are required,

so two border lines have to be available to update the slice. It is very important that the X dimension of a slice be equal to the X dimension of the corresponding half-plane, because a direct consequence is that border columns are not required and less data have to be transferred.

A slice *without borders* is composed by $(L/2) \times S$ spins and requires $4 \times w \times (L/2) \times S$ bits of memory when considering both spins and couplings. A slice *with borders* is composed by $(L/2) \times (S+2)$ spins and with the couplings requires $4 \times w \times (L/2) \times (S+2)$ bits of memory.

To update the z-th slice of white spins, the following slices are needed:

- the white spins slice (z) composed by $\frac{S \times L}{2}$ spins
- the black spins slices $(z-1)$, (z) and $(z+1)$, each of one is composed by $\frac{(S+2) \times L}{2}$ elements
- the black and white Jx slices (z) ($\frac{S \times L}{2}$ elements each slice)
- the white Jy slice (z) ($\frac{(S+1) \times L}{2}$ elements)
- the black Jy slice (z) ($\frac{(S+1) \times L}{2}$ elements)
- the white Jz slice (z) ($\frac{S \times L}{2}$ elements)
- the black Jz slice $(z+1)$ ($\frac{S \times L}{2}$ elements)

As a consequence, at least M bits have to be allocated in each local store:

$$M = \left(\frac{9 \times S}{2} + 4 \right) \times w \times L \tag{3.20}$$

Eight more slices have to be added to be allow the concurrent execution of data transfer and computation, so in total M bits of local memory are required:

$$M = (9 \times S + 6) \times w \times L \tag{3.21}$$

The time required to update a slices is:

$$T = \tau(V) \times \frac{S \times L}{2} \tag{3.22}$$

While a core updates the white slices (z), the following slices are concurrently loaded from main memory:

- the white spins slice $(z+1)$ composed by $\frac{S \times L}{2}$ spins

- the black spins slice $(z+2)$ composed by $\frac{(S+2)\times L}{2}$ elements
- the black and white Jx slices $(z+1)$ ($\frac{S\times L}{2}$ elements each slice)
- the white Jy slice $(z+1)$ ($\frac{(S+1)\times L}{2}$ elements)
- the black Jy slice $(z+1)$ ($\frac{(S+1)\times L}{2}$ elements)
- the white Jz slice $(z+1)$ ($\frac{S\times L}{2}$ elements)
- the black Jz slice $(z+2)$ ($\frac{S\times L}{2}$ elements)

Moreover, the white slice $(z-1)$ is stored back to main memory, so globally M_{dma} bits of memory are globally transfered between main and local memories:

$$M_{\text{dma}} = (5 \times S + 2) \times w \times C \times L \tag{3.23}$$

The system execution time is balanced if the following equation is true

$$\tau(V) \times \frac{L \times S}{2} = \frac{(5 \times S + 2) \times w \times C \times L}{B} \tag{3.24}$$

The number of slices that compose a half-plane is directly proportional to the overhead of data transfer.

3.6 Conclusions

In this chapter we have analyzed three different strategies to map the Metropolis Algorithm for spin glasses to an abstract multi-core architecture. As data exchange is a major issue when dealing with multi-core architectures, emphasis has been put on the problem of data allocation and distribution. The computational request has been represented with a variable that will be analyzed more in detail in chapter 4, where we discuss the practical implementation of the algorithm on CBE. The distribution of the data-set among the cores is done along only one dimension, that allows to minimize the interaction between cores and it is not a limitation as we expect that one or two CBE processors are enough to manage the lattices we are interested to ($L \in [16, 129]$).

The equations that describe the balance between computations and data transfers and show that the size of the problem can has a significant impact on performance, because if data structures are not small enough or if border data is too large, transfer time can be larger than computation time. However, the balance equations have to be filled with the bandwidths of the real architectures and with the spin update time $\tau(V)$ that is achievable by the computational kernel.

Chapter 4

Spin Glasses on CBE

The global performance of spin glasses on CBE is given by the combination of three factors: (i) the performance achievable with a single core, (ii) the interaction between cores and, in case it is used, (iii) the interaction with main memory. In chapter 3 we have analyzed the second and the third issues, proposing a theoric analysis of the balance between data transfer and computation. The balance equations proposed in that chapter depends on the spin update time $\tau(V)$, determined by the implementation of the actual code.

The value of $\tau(V)$ is essentially determined by two fractions of code: the random number generation and the effective spin update. With the term *spin update procedure* we refer to the set of instructions that update a vector data-word, including the generation of the required random numbers, while with *spin update* we mean the fraction of instructions that update the vector data-word assuming that random numbers are already available.

The real $\tau(V)$ differs from the theoric estimate due to overheads imputable to data accesses, that will be discussed in detail in the first section of the chapter, putting in evidence the difference between the ideal case and the real one, and describing the behavior of the code in relation to the SIMD-granularity and the size of the lattice.

The first section of this chapter is focused on the analysis of the performance of single a core, proposing a static and a run-time estimate of the system spin update time $\tau(V)$. In the second section we will determine the cases in which computation and data transfers are balanced, proposing equations that describe the behavior of the program. Applying the estimates of $\tau(V)$ to these equations allows to determine which performance should be expected when updating a lattice of a given size L when using C cores. The issues that can lead to a sub-optimal usage of the bandwidth will be briefly described.

4.1 Core implementation

Programming the CBE is not a trivial task. In order to achieve an efficient use of a SPE, many levels of parallelism have to be exploited:

- *Instruction Level Parallelism*, to effectively hide the latencies of the instructions
- *Dual-issue Parallelism*, to overlap calculations and data manipulations
- *Data Parallelism*, to exploit the SIMD capabilities of the architecture

Instruction level parallelism, that in CBE is obtained putting the the instructions into pipelines, is exploited not only avoiding all the false dependences between accesses to variables (that of course should always be avoided), but more importantly keeping as much data as possible in the (large) register file. This type of parallelism is available in virtually all the processors designed in the last twenty years. However, the peculiarity of CBE is to provide a large amount of general purpose registers (128 x 128-bit vector register), so that many instructions can be executed concurrently. However, if the code contains too much dependences that force the compiler to strictly serialize the order of local memory accesses (and as a consequence the order of the instructions), it is not possible to take advantage of the presence of such a large register file.

The existence of two heterogeneous pipelines (that *dual issue parallelism*) in each SPE allows basically to hide the latency of load/store instructions and in general of data permutations. In Table 4.1 the types of instructions that can be executed by the two pipelines is shown. Note that the even (0) pipe is able to perform float and integer arithmetic, while the "odd" (1) pipe executes load and stores. Obviously, besides load and stores there is a number of operations not needed by the algorithm but required by the real code, that cannot be completely hidden by the presence of two pipelines. For example, the instructions used to calculate memory addresses and to pack scalar words into vector data-words are typically instructions executed by the even pipeline, that also performs integer and floating point arithmetic. As a consequence, a certain amount of cycles have to be dedicated to the execution of these instruction and the computations of the algorithm have to be postponed. Although the two pipelines do not allow a complete overlap of computations and data access, they are nonetheless a key issue to optimize a program for the CBE processor.

Data parallelism is granted by the multispin coding technique that, as seen in Chapter 2, allows to update V different spins of the same system and w systems concurrently. Exploiting data parallelism is fundamental to properly exploit the available computational resources, and an appropriate value of V has to be chosen

Instruction Class	Pipe	Execution timing
Double-precision floating-point	0	7+13 cycles
Integer multiply	0	7 cycles
Integer interpolate	0	7 cycles
Integer/float conversion	0	7 cycles
Single-precision floating-point	0	6 cycles
Element rotate/shift, special byte operations	0	4 cycles
Integer add/subtract	0	2 cycles
Load immediate	0	2 cycles
Logical operations	0	2 cycles
Sign extend	0	2 cycles
Count leading zeros, select bits, carry/borrow generate	0	2 cycles
Loads/stores	1	6 cycles
Branch hints	1	6 cycles
Channel operations	1	6 cycles
Move to/from SPR	1	6 cycles
Branch	1	4 cycles
Shuffle bytes, quadword rotate/shift	1	4 cycles
Estimate	1	4 cycles
Gather, form select mask	1	4 cycles
Generate insertion control	1	4 cycles

Table 4.1: SPU instructions classification based on pipeline and latency.

to benefit of synchronous data parallelism, according to the result that we want to achieve.

4.1.1 Data Parallelism and SIMD-Granularity

Asynchronous parallelism is a type of data parallelism that does not directly take advantage of the SIMD nature of the ISA [1], as it simply allows to update a given number of spins in parallel using scalar variables. In effect, in the context of multispin coding, the SIMD instructions improve the performance of the production of random numbers and provide an efficient way to compose the vector data-words that are compared to the energy difference vectors. The spin update fraction of the procedure, however, is performed independently for each bit of the 128-bit vector data-words of CBE, including the final comparison with the two vectors derived from the 32-bit random numbers, and from a computational point of view does not take any advantage of SIMD instructions. However, if the w-bit scalar word used to represent corresponding spins of w different systems matches one of the scalar sizes supported by the architecture, then data permutations are more efficient, as we will see later.

Although CBE supports only SIMD-granularities $V = 2, 4, 8, 16$, as long as they do not have an impact on the efficiency of the spin update fraction of code, in principle it is possible to use every SIMD-granularity in the range $V = 1, \ldots, 128$. Smaller granularities grant a better asynchronous parallelism, due to the high parallelism of the spin update code and to the small amount of random numbers that have to be generated. High granularities, instead, allow to reach a better synchronous parallelism, according with our target to go as fast as we can for the update of a single system.

Let us discuss briefly the implications of a small granularity, to explain why it will no longer be taken into account. With $V = 1$ only one 32-bit random number is needed to update a vector data-word: the equivalence of vector and scalar data-words makes the retrieval of neighbor spins and coupling terms relatively easy, and the code is presumably very efficient. However, such a scenario it is not very interesting, not only because this strategy does not allow to obtain the best possible synchronous parallelism (that is our target), but also because its main benefit is only to produce a high amount of statistics. The same result can be simply achieved running many instances of the problem on a large set of commodity processors. In this case we are not really exploiting the capabilities of CBE, in particular the relatively high amount of cores CBE by an high-speed interconnect bus, that is a key difference between the CBE and a simple collection of single or "few-core" machines.

There also some drawbacks: first of all, the presently largest number of samples ($w = 128$) implies that the "local memory" strategy can be adopted in very few cases,

[1] Instruction Set Architecture

as the local store space required is directly proportional to w. If main memory is used and only a small fraction of the lattices can be held in local stores, the amount of data that have to be exchanged between memories would saturate the bandwith with a few cores, because the sizes of data transfers also directly proportional to w. For all these reasons, in our implementations we set the lower boundary of SIMD-granularity to $V = 4$, that is the smaller granularity supported by the architecture for integer arithmetic, and incidentally it is also the same granularity of random number generation code (so in this case it has to execute very few permutations of data).

The highest theoric SIMD-granularity of our implementation (but not of the processor) is $V = 128$. Since the random number generation procedure is bound to $V_r = 4$, it has to be repeated $R = V/4 = 32$ times in sequence to produce enough random numbers. We can expect some improvement in the time needed to generate a scalar 32-bit random number due to pipelining, but probably it is possible to find a practical $R' = V'/4$ that saturates the pipelines (or at least one of them). For an higher SIMD-granularity $V'' > V'$ the procedure has to be repeated $R'' = V''/4$ times, with $R'' > R'$, which implies that $\Delta V = V'' - V' = ((R'' - R')/4)$ additional random numbers have to be produced after the first V' random numbers.

Because the instructions generating the first V' random numbers saturate at least one pipeline, the execution of the instructions that generate the ΔV additional random numbers has to be postponed. However, the additional instructions can overlap each other. As a consequence, producing more than $V' = R' * 4$ random numbers should not be a significant advantage nor a disavantage for the performance. In general, it would be interesting to determine if the speed-up of the spin update procedure granted by a SIMD-granularity V higher than V_r can notably improve the performance, and also if global time wold be dominated by random number generation time (that cannot be reduced, because of the definition of R'). From this point of view, the spin update code should be performed with a SIMD-granularity of at least $V' = (4 \times R')$ and the convenience of even higher granularities has to be evaluated.

To allow the comparison between random numbers and spin vector data-words with with SIMD-granularity $V > 4$, data permutations are required. The `shuffle` instruction is very efficient and convenient for this pourpose, but it operates only at byte level, that correspond to maximum SIMD-granularity $V = 16$. For higher granularities, we need to use additional instructions to operate at bit level that, in turn, could make the generation of random numbers less efficient.

Another important issue is that both memory occupation and data transfers size, as shown by the balance equations, are inversely proportional to V, so a high SIMD-granularity allows to store larger lattices into local memories, and even when the lattices are so large that have to been stored into main memory, the amount of data that has to be transferred between cores and main memory is proportionally reduced,

thus decreasing the pressure over main memory, that is a potential bottleneck.

In conclusion, since our target is to achieve the best possible system spin update time, we need to adopt the highest possible SIMD-granularity. Using a granularity not directly supported by the architecture could introduce overheads that need to be analyzed and evaluated.

4.1.2 Data Layout

Data layout has a heavy impact on performance in a SIMD program because, if the operands are not located in corresponding scalar slots, a number of data move instructions have to be executed before and after the effective computation.

In the case of a spin glass program there are various possible data layouts, but we start from the assumption that the basic elements are scalar values containing w corresponding spins or coupling terms of w different system. In principle, it is possible to define as basic data type a structure composed by four scalar variables used to represent a spin and its three associated coupling terms. In a trivial implementation, four scalar variables can be embedded in a single vector data-word, but this implies a fixed value of $w = W/4$ and an effective granularity $V = 1$, so we discard this strategy.

A possible improvement is to define the structure as composed by four vector data-words that represent spins and couplings, so that there are no limitations on SIMD-granularity. In this way the lattice is composed as a pair of white and black half-lattices that are defined as *arrays of structures*. However, this strategy has two major drawbacks. First of all, as we have seen in chapter 3, when the lattice is small enough to be kept in local memories the cores need to exchange the values of the spins that lie in the border planes. If spins and coupling terms are stored in adjacent locations, then it is not possible to perform efficient DMA transfers, because the couplings are transferred with the spin values, and as a consequence the volume of data exchange is four times greater than the optimal one. If instead only the spins are transferred, as they are small variables in non adjacent position, they do not match the properties of EIB channels, so we can expect poor performances (it is likely that a number of tricks are needed if spins are not 16-byte aligned).

The second drawback is that, even in the case of the lattice stored in main memory, although both spins and coupling terms need to be loaded into local memory, only the spins need to be stored back to main memory. Again, if spins and couplings are embedded in the same struct, DMA transfers are not efficient as they can be.

In general, an constraint for the data layout is the 16 byte alignment for both data transfers and computations. The start address of a DMA transfer must be 16 bytes aligned and the length of the transfer must be a multiple of 16 bytes (if it is longer than 16 bytes). 16 bytes is also the size of a vector data-word, so it is important the use a layout that allow to use a vector data-word as a basic element,

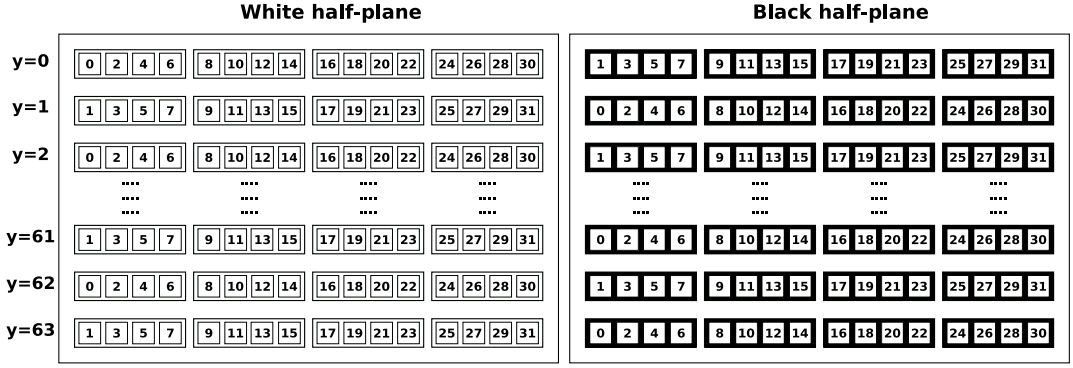

Figure 4.1: Distribution of the spins of a plane among vector data-words. The numbers inside the square indicate the X coordinates of the w spins embedded in that scalar element.

avoiding separate accesses to its components.

For all these reasons, we use instead a *structure of arrays*, keeping eight separate arrays for black and white spins and coupling terms to represent a lattice or a sublattice. Each array is composed by vector elements that contains V scalar variables representing w spins or coupling terms. In this way we can separately transfer spins or coupling terms without overheads concerning the addressing, as a spins and its associated couplings are located at the same offset inside the arrays. Moreover, each vector data word can be used as a primitive unit containing only spins or couplings.

Note that in chapter 3 the same data structure was implicitly assumed, but in that chapter the details of communication channels were ignored, as an abstract multi-core architecture was taken in consideration. Moreover, issues as the addressing and the location of scalar elements were not taken in consideration: it was only necessary to allow separated accesses to the data structures containing spins and coupling terms of a given half-plane of a given color. As we describe now, the proposed data layout matches those requirements.

In the case of our spin glass implementation, all the X-lines of the lattice are splitted into half-lines of $L/2$ spins due to the checkerboard subdivision, with the consequence that in corresponding positions of an half-line there can be spins with even or odd X coordinates. Each half-line is composed by $L_V = L/(2 \times V)$ vector data-words (for sake of simplicity we do not consider line lengths that are not a multiple of V). The access to the neighbors of the V scalar elements of a vector data-word can introduce a non negligible overheaded if they are stored in multiple vectors data-words or they are not located in the same scalar slots.

The checkerboard subdivision implies that the L half-lines that compose an half-plane contain alternatively spins with even X coordinates and spins with odd X coordinates. For example, assuming a trivial distribution of scalar elements into vectors, a half-plane with 4 vector data-words in the X direction and 64 spins in the Y direction is subdivided in the way shown in Figure 4.1.

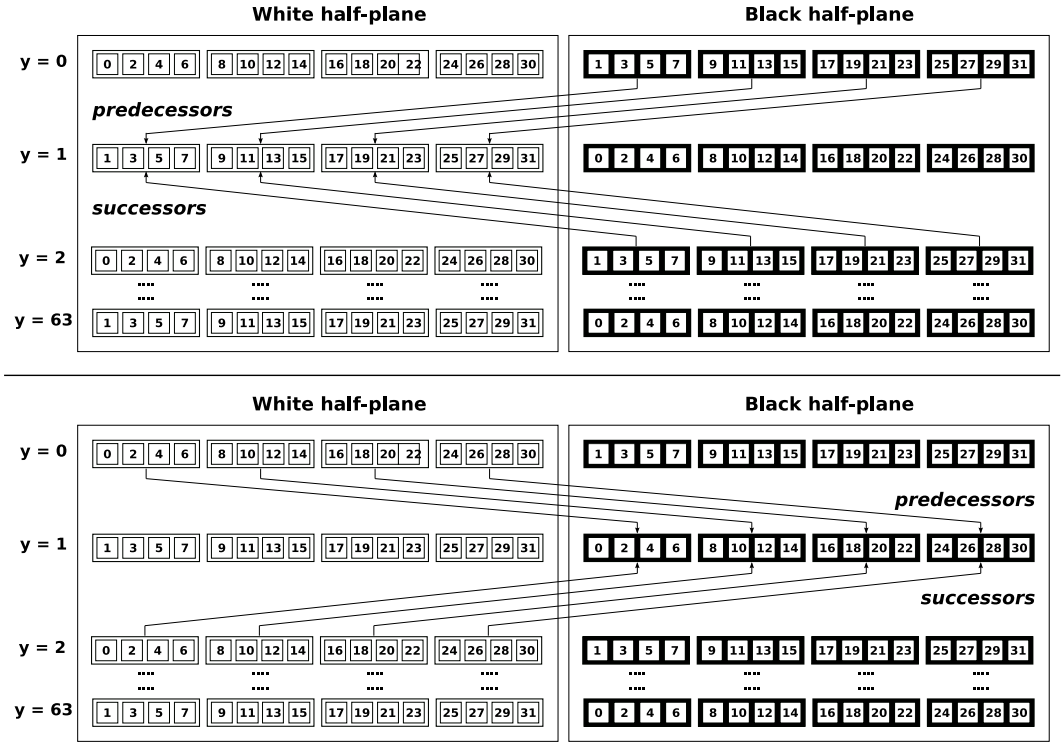

Figure 4.2: The arrows indicate the location of the neighbors in the Y direction of the pair of half-lines ($y = 1$).

Note that neighbor spins and coupling terms in Y and Z directions are conveniently located in corresponding scalar slots, so the vector data-words can be directly used for computation, without the need to rearrange scalar elements. More precisely, for all the V scalar elements of a vector data word (x, y, z) that contains spin values, the neighbor spins in Y and Z directions are located in same scalar position inside the vector data-words with coordinates $(x, y-1, z)$, $(x, y+1, z)$, $(x, y, z-1)$ and $(x, y, z+1)$. In the same way, coupling terms for the positive Jy and Jz directions are located in the same scalar slots of the vector-data words with coordinates (x, y, z), while coupling terms for negative Jy and Jz directions are in the vector data-words with coordinates $(x, y-1, z)$ and $(x, y, z-1)$ respectively. Figure 4.2 shows an example of the location of neighbors spins in Y direction.

The invariance of scalar slot position is not preserved along X direction due to the packing of scalar elements into vector data-words. The location of neighbors along X direction essentially depends on:

- the oddness of the X coordinate of the spins and coupling terms embedded in the vector data-word
- the length of the line

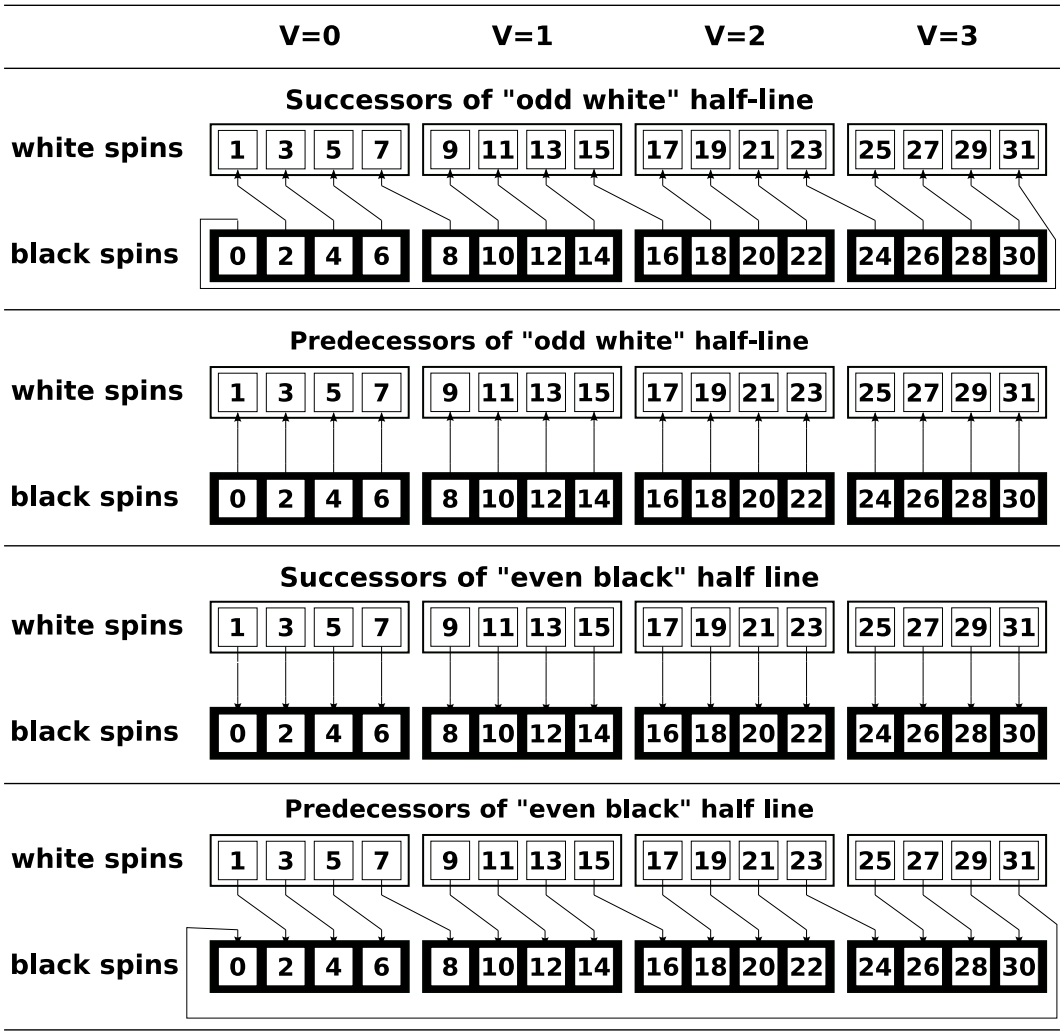

Figure 4.3: The location of the neighbor spins in X direction of a pair of white and black half-lines.

- the position of the vector data-word inside the line

The location of predecessors and successors of the spins of a "white odd" X-line is shown if Figure 4.3. If a line contains odd spins, then their predecessors are located in the same scalar slots, while their successors are in two different vector data-words and the scalar elements have to be rotated or shuffled. In the opposite case, when the line contains even spins, rotations are not needed to get the successors, while are required to get the predecessors. The `shuffle` instruction is able to produce an output vector in which each byte is got from an arbitrary position of two input data words, so a single `shuffle` instruction is able to collect the neighbors of a vector data-words as illustrated in Figure 4.3.

The location of the coupling terms required to update a pair of white and black

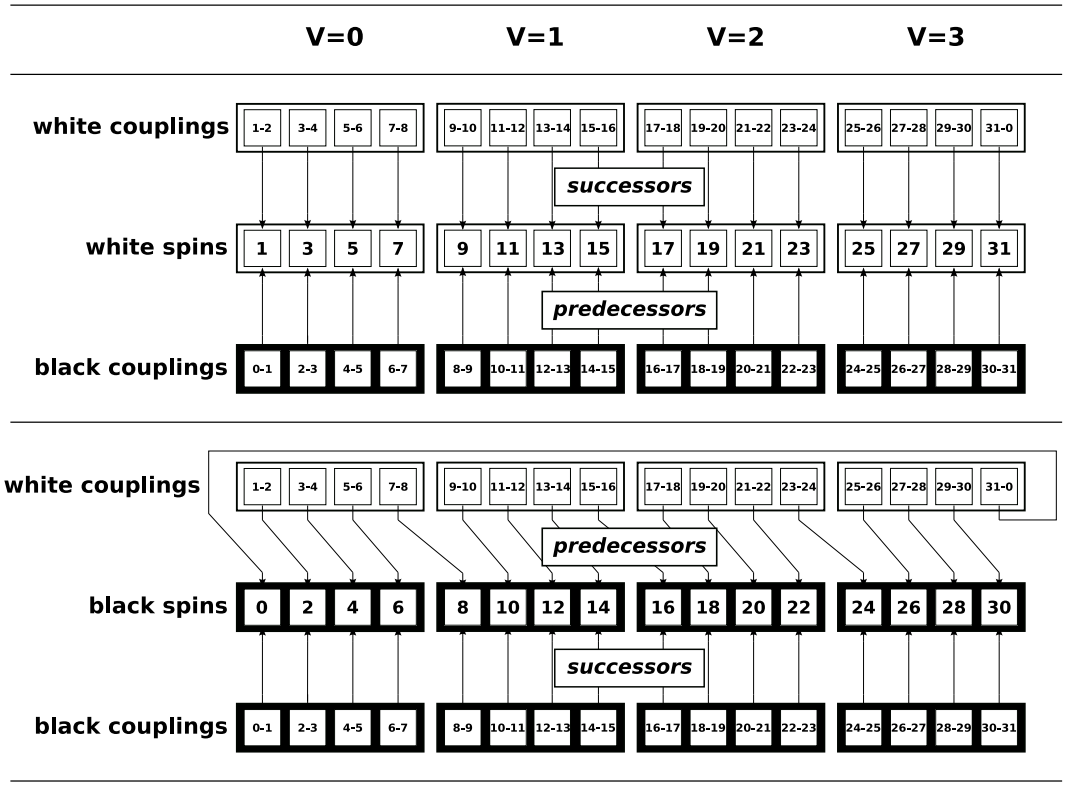

Figure 4.4: The location of the coupling terms required to update an X-line.

half-lines is shown in Figure 4.4. Both the predecessors and successors of the even half-line are in corresponding scalar slots, while the predecessors of the odd half-line are distributed among two different vector data-words. In this case it is necessary to reshuffle data, thus incrementing the overhead associated to the load of input data. In conclusion, specific vector data-words require more than one `shuffle` instructions to collect all the data required for their update.

To make the access to neighbors in the X direction more efficient, an alternative way to embed scalar data into vector-data words in shown in Figure 4.5. Vectorization is made along Y direction instead that along the X direction.

Unfortunately, in this case the predecessors (or successors) in the X direction are always located in two different vector data-words, so the access to neighbors is not more efficient than in the previous case. Moreover, to access the neighbors along Y direction now permutations are required, while previously it was not necessary. So, we can discard this strategy and assume that vector data words always contain elements that are located in the same X line.

In order to reduce the number of instructions and the latency associated to the load of the neighbor data, we propose a new distribution of spin and coupling terms

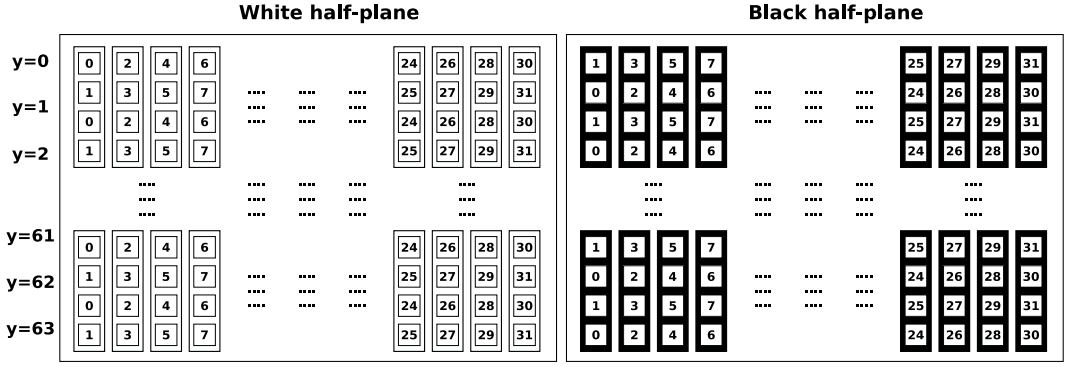

Figure 4.5: A distribution of the spins of a plane among vector data words. In this case the scalar elements of a vector are adjacent in the Y directions instead that the X direction.

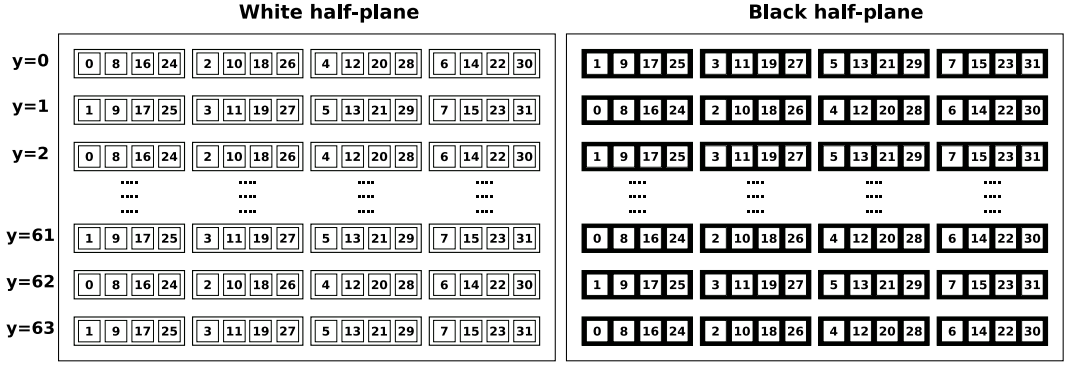

Figure 4.6: New distribution of the spins of a plane among vector data-words. In the same vector data-word are embedded spins which X distance is $(2 \times (L/(2 \times V))$.

that lies on the same line. The basic idea is to store in the scalar slots of each vector data-word spins with a distance of $(2 \times N_V)$ along X direction, where $N_V = L/(2 \times V)$ is the number of vector data-words that compose a half-line (see Figure 4.6). In the previously described trivial case, the distance of the elements embedded in the same vector was only 2, which means that in the context of the associated color they were adjacent.

The new ordering assures that in the most common case, that is a vector data word that is not at the end nor at the beginning of a half-line, predecessors and successors are located in corresponding scalar slots, so that shuffling is avoided. An example of the location of neighbor spins is shown in Figure 4.7, while the distribution of coupling terms is illustrated in Figure 4.8.

The removal of the shuffle instruction not only reduces the total instruction count, but also allows to use a data-word for calculation as soon as it is loaded into local store. Otherwise, before using a data-word that contains neighbors, it was necessary to wait for the load of two different words and their shuffling.

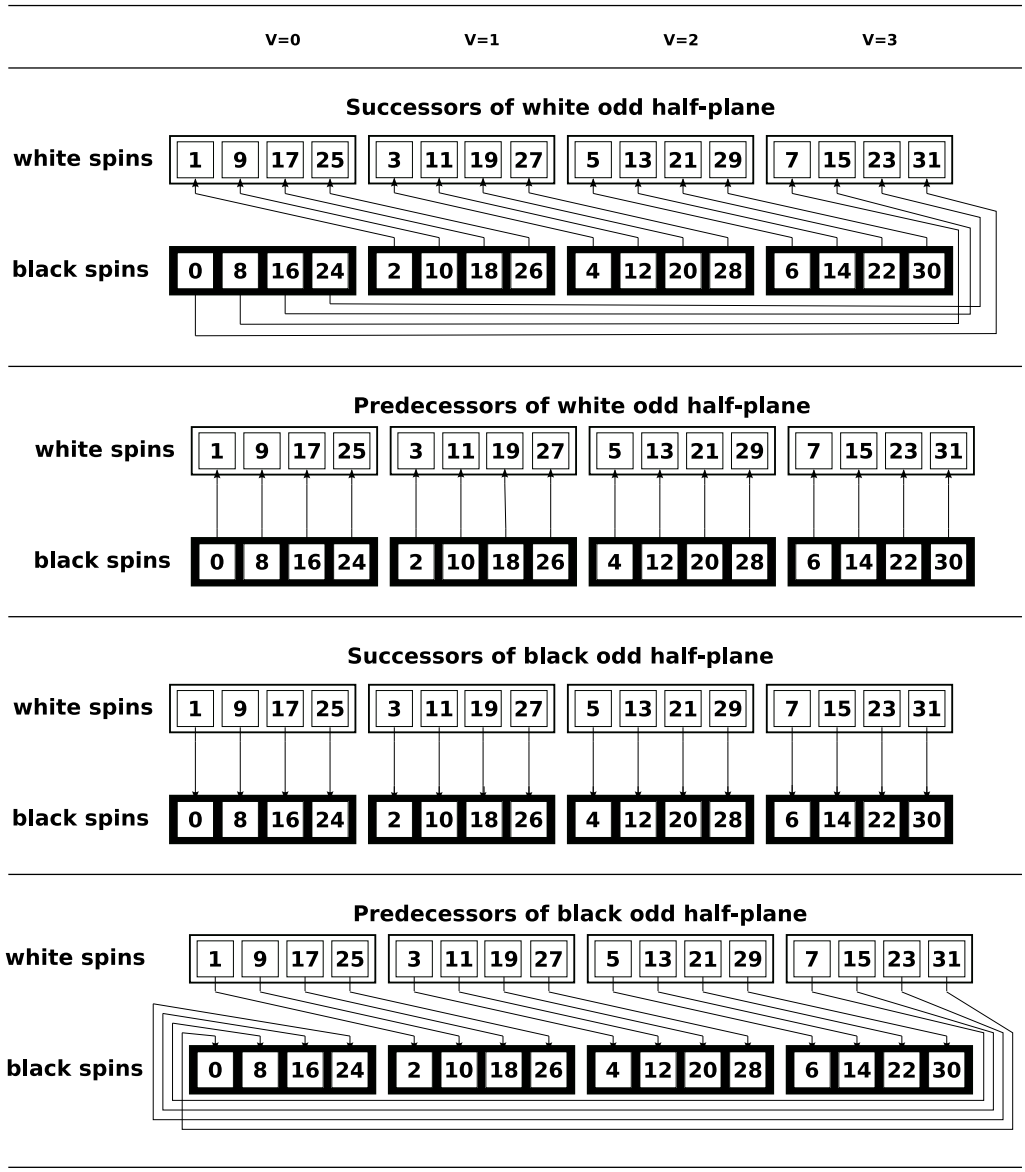

Figure 4.7: The location of neighbor spins when using a data displacement that reduces the permutations. In contrast with 4.3, it is never necessary to compose scalar elements of two different vector data-words and a single rotation is required.

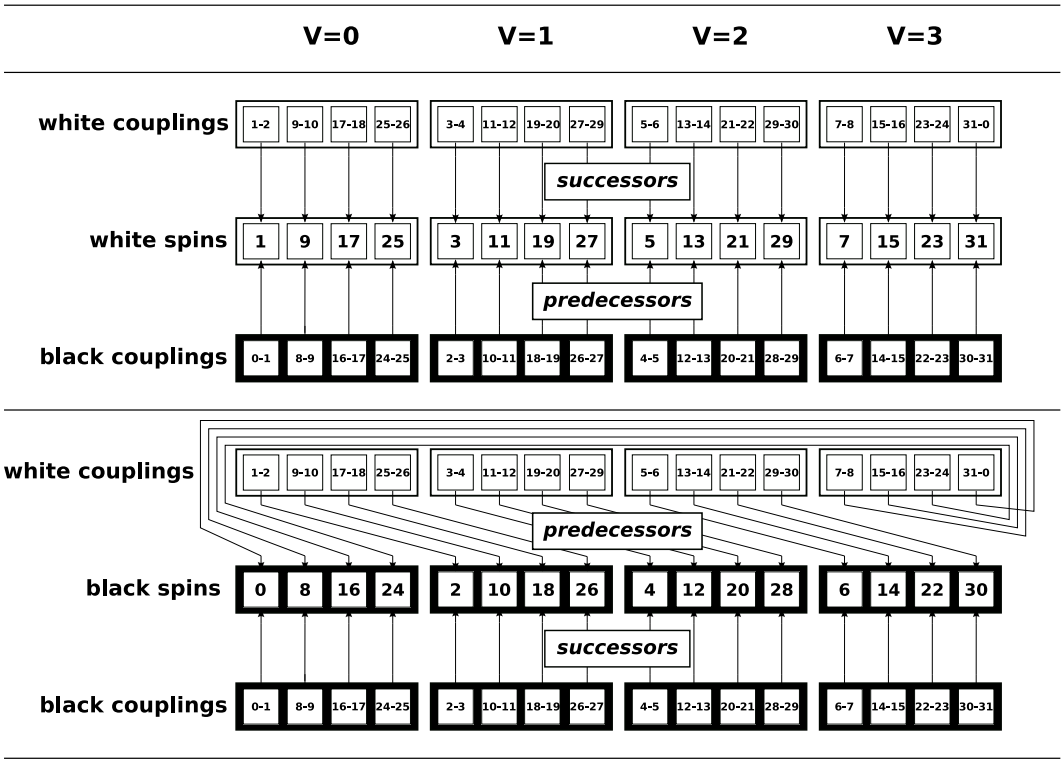

Figure 4.8: The location of required coupling terms when using a data displacement that reduces the reordering. In contrast with 4.4, it is never necessary to compose scalar elements of two different vector data-words and only a rotation is needed.

.

A vector data-word of a X-line can only be in one of the following four condition:

1. it is the unique vector of the half-line
2. it is the first vector of the half-line
3. it is the last vector of the half line
4. it is not the first nor the last nor the unique vector of the half-line

In each case the position of the neighbor spins and of coupling terms is different. Moreover, the oddness of the x coordinate of the spins also makes a difference. Let us define a *even half-line* as a line that contains spins with even x coordinates, and in the same way an *odd half-line* as a line of spins with odd x coordinates. The *predecessor* of a vector data-word A is a vector data-word that contains in each scalar slot a spin which, in terms of x coordinates, is the predecessor of the spin contained in the corresponding scalar slot of A. The *successor* of a vector data-word A as a vector data-word that contains in each scalar slot a spin which, in terms of x coordinates, is the successor of the spin contained in the corresponding scalar slot of A.

Figure 4.9 shows the example of the special case of half-lines composed by only one vector data-word. The location of neighbor spins is illustrated in Figure 4.10, while Figure 4.11 shows the position of coupling terms. In this peculiar case the location of data depends only on the oddness of the half-line:

- if the line is even, the vector data word with the same Y and Z coordinates and the opposite color is the successor, while the same vector right rotated by one scalar position is the predecessor. The Jx couplings with the predecessors are obtained left rotating by one scalar position the corresponding vector for the opposite color
- if the line is odd, the vector with the same X and Z coordinates is the predecessor, while the same vector has to be left rotated by one scalar position to become the successor. The Jx couplings with the predecessors are int the corresponding vector for the opposite color

In a more general case, each half-line is composed by at least two vector data-words. A vector data-word can be the first or the last vector of the half line, or it can be an internal vector. It is however possible to individuate general rules concerning the position of neighbor spins and of coupling terms that do not depend on the position of the vector data word inside the half-line:

- both predecessor and successor spins are in the half-line with the opposite color

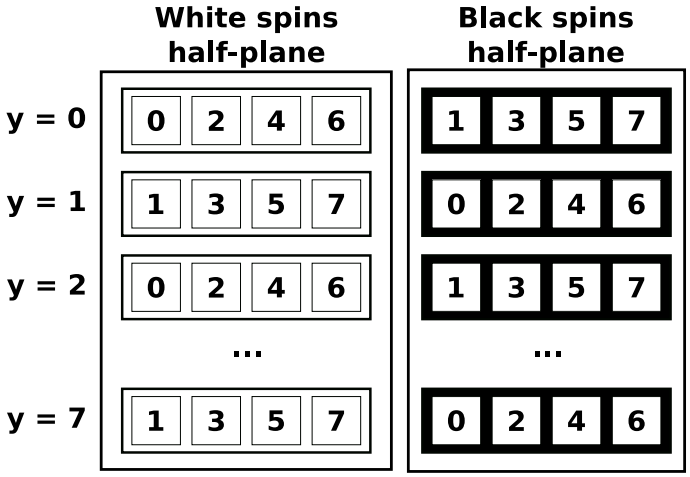

Figure 4.9: The composition of a pair of white and black half planes when $L = 8$.

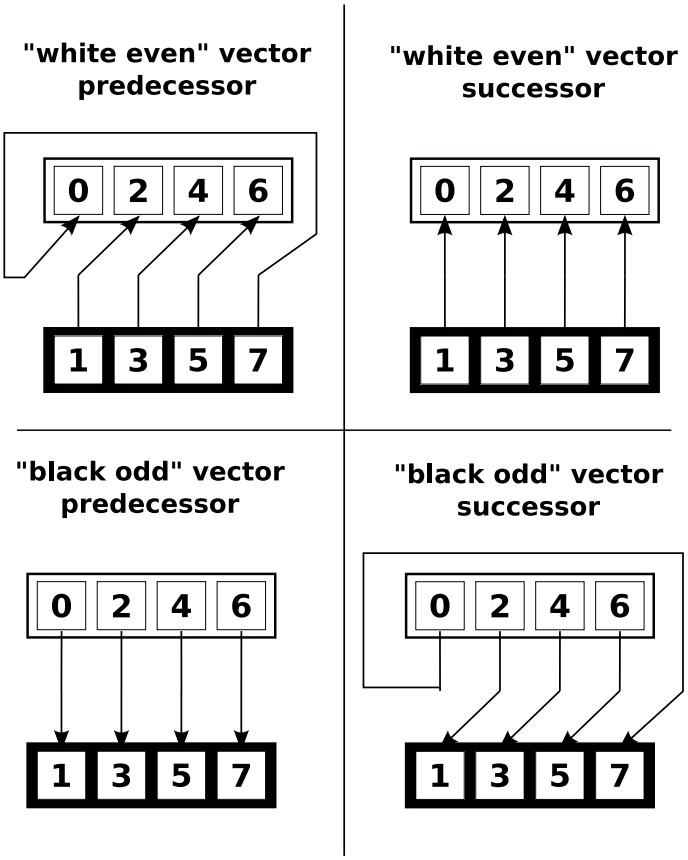

Figure 4.10: The location of neighbor spins in the case of a pair of black and white half-planes when $L = 8$.

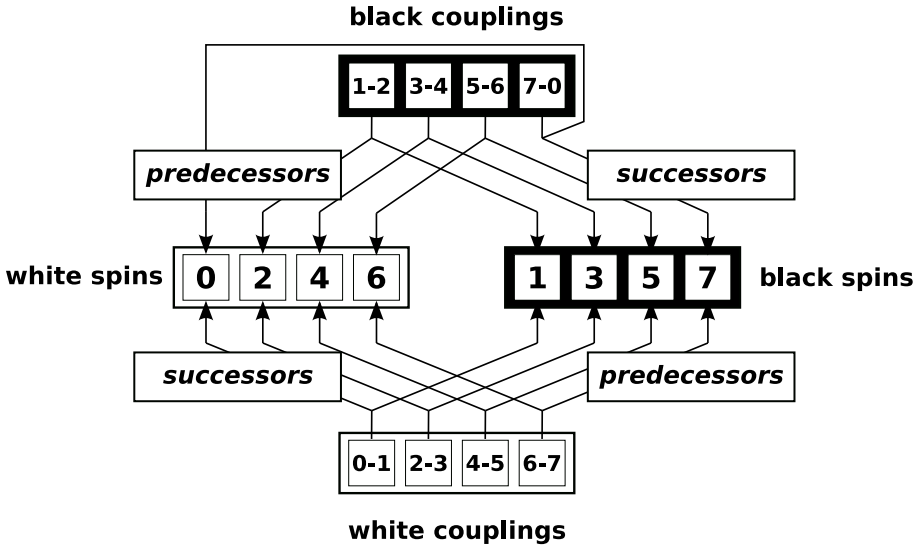

Figure 4.11: The location of the required coupling terms in the case of a pair of black and white half-planes when $L = 8$.

- the coupling terms shared between the current spins and their predecessors are in the coupling half-line with the opposite color
- the coupling terms shared between the current spins and their predecessors are in the coupling half-line of the same color

The previous general rules tell us in which data structures the required data is located, but then it is necessary to make distinctions based on the position of the vector and its oddness. For internal vector data-word (V_x) of a half-line, with $V_x \in [1, L/(2 \times V) - 2]$:

- if (V_x) contains even spins, the predecessor and successor spins are located in the the vectors (V_{x-1}) and (V_x) of the half-line of the opposite color, respectively. The coupling terms shared with predecessors are stored in the vector data-word (V_{x-1}) in the half-line of opposite color
- if (V_x) contains odd spins, the predecessor and the successor are stored into the vectors (V_x) and (V_{x+1}) of the half-line of the opposite color. The coupling terms shared with predecessors are stored in the vector data-word (V_x) in the half-line of opposite color

In any case the coupling terms shared with the successors are always in the vector data-word (V_x) of the coupling half-line of the same color.

The previous rules are still applicable in the case of the first and the last vector data-word of a half-line, given that if $(V_{x-1}) < 0$, then the last vector of the half-line, right rotated by one scalar slot, is used instead. In the same way, if $(V_{x+1}) > 0L/(2 \times V)$, then the first vector of the half-line is taken instead, rotated by one scalar position to the left.

In conclusion, the proposed data layout minimizes the permutations associated to the access of input data, and keeps spin and coupling vector data-words separated, so that the minimal amount of data has to be exchanged between the cores and the main memory.

4.1.3 Random Number Generation

Multispin coding allows to use the same random value to update corresponding spins of independent systems, while independent random numbers are mandatory when updating different spins belonging to the same lattice. This allows to reduce the impact of random number generation on the global system spin update time, in particular when using a high number of samples w. As long as our goal is to exploit synchronous parallelism, we aim to use the lowest possible sample number. In turn this implies that, as long as a large quantity of random numbers has to be produced (up to 128), their generation may have a heavy impact on performances.

As said earlier, random number generation and spin update fractions of the code do not need to adopt the same SIMD-granularities. In particular, the spin update code is invariant with to respect of V and in principle it is not required that V matches one of the SIMD-granularities supported by the architecture. The random numbers must have an adequate size to guarantee the correctness of the algorithm so, given a fixed size W of vector data-words, there is a limit to the amount of random numbers that can be concurrently generated. In the following we will call V_r the SIMD-granularity of random number generation and V the SIMD-granularity of spin update fraction of code.

In this work we have used the Parisi-Rapuano [73] generator, which is a popular choice for spin glass simulations. It is defined by the following equations:

$$\begin{aligned} I(k) &= I(k-24) + I(k-55) \\ R(k) &= I(k) \oplus I(k-61) \end{aligned} \tag{4.1}$$

where $I(k)$ is a circular array of random unsigned integers and $\oplus$ indicates the logical XOR. In most cases, 32-bit random numbers are used, and we also adopt this choice, which implies a SIMD-granularity $V_r = 4$ in CBE. Though, as long as the comparison between the random number and the energy differences ΔE of the spins embedded in a vector data-word with $V \neq 4$ can be done in parallel thanks to multispin coding, the spin update procedure can be executed with any SIMD-granularity. Sets of V_r random numbers are sequentially generated, until V random

numbers are available, and then they are rearranged in order to be compared with the energy difference vector data-word.

The Parisi-Rapuano generator requires a mix of integer, floating-point and bit-wise logical instructions. It is computationally quite inexpensive, because only an integer sum, a XOR and a floating-point multiplication are required, but it also needs to write and read the circular array (hereafter referred as to `IRA`), so three LOADs, one STORE and at least four couples of integer sums and modulus (to update the four pointer to the array) are unavoidable.

Note that the sequence of the random numbers that are produced depends on the initialization values of the array, so if each core loads a different starting sequence from main memory, the semantic of the algorithm is preserved.

For our purposes we can subdivide the generation of V random numbers in two distinct phases:

- *generation phase*: the generation of V 32-bit random numbers in the range $[0, 1]$. The random numbers are embedded in $V/V_r = V/4$ vector data-words. Each vector data-word contains four scalar 32-bit random numbers
- *shuffling phase*: Each scalar 32-bit random number is compared with three values , and the resulting vectors are shuffled to produce the vector data-words `R0` and `R1` that will be used for comparison with energy differences (see 2.3.2 and note that `R0` and `R1` are used to respresent respectively the least and the most significant bits of the variable R)

The first phase corresponds to the equation 4.2 applied to an array of SIMD data-word of four 32-bit `unsigned int` scalar data-words. The random number $R(k)$ have to be converted from `unsigned int` to `float`, and then be multiplied by $(1/(2^{32} - 1))$, in order to be normalized into the range $[0, 1]$. In total four instruction are required, neglecting memory accesses and the related addressings. Trivially, the generation of a vector of four 32-bit random numbers can be realized with the following fragment of code:

```
ui_rnd = spu_xor ( IRA[ip++] = spu_add (IRA[ip1++], IRA[ip2++]),
                   IRA[ip3++] );
f_rnd = spu_mul ( spu_convtf (ui_rnd, 0), C_inv_max_rand );
```

Given a SIMD-granularity V, Table 4.2 shows how many instructions are ideally needed for the *generation phase*. Because a SIMD-granularity $V_r = 4$ is mandatory in our implementation of spin glasses on CBE processor, the number of instances of a type of instruction depends on SIMD-granularity V.

The optimal scheduling of this fragment of code is shown in Table 4.3, while the scheduling obtained with `gcc 4.1.1` is displayed in Table 4.4. In both tables the

Randon Number Generation		
Type	**Pipe**	**Number**
ADD	even	$1 \times (V/4)$
XOR	even	$1 \times (V/4)$
FMUL	even	$1 \times (V/4)$
Even total		$3 \times (V/4)$
Odd total		0
Total		$(3 \times (V/4))$

Table 4.2: The ideal number of instructions required to produce V scalar 32-bit random numbers. Note that instructions of odd pipeline are not required.

last instruction is the multiplication that normalizes the random number, which has a latency of six cycles. Some extra instructions also appears in the tables: they are needed to load/store data from/to the local store and to perform type conversions. Additional instructions, like integer sums and rotations, that are used to calculate memory addresses, are not shown.

Note that there is a trivial false dependence between the store instruction and the last load. The compiler cannot foresee the access pattern of the array, so it simply sequentialize load/store instructions. However this can be avoided if all the required values are loaded in registers before calculating the random number and the result is stored back to the array only at the end of the update procedure. This is a technique that we will use pervasively in our implementation, because it allows to expose all the possible concurrency between instructions to the compiler.

Another issue that does not allow to obtain an optimal scheduling is the computation of memory addresses. To help the compiler to use the minimum number of instructions, it is a good practice to use the lowest possible level of abstraction when dealing with pointers arithmetics. In our case we have used byte resolution, that allows to obtain the scheduling shown in Table 4.5.

The *shuffling phase* takes $V/4$ random number vector-data words as input, and the following comparisons are performed in sequence to obtain a vector data-word $\texttt{IDX}_i$ from each input vector data-word $\texttt{R}_i$:

1. `IDX = 0`
2. If $\texttt{R}_i > e^{-4\times\beta}$ then $\texttt{IDX}_i$ `= 1`
3. If $\texttt{R}_i > e^{-8\times\beta}$ then $\texttt{IDX}_i$ `= 2`
4. If $\texttt{R}_i > e^{-12\times\beta}$ then $\texttt{IDX}_i$ `= 3`

Cycle	Even Pipe	Odd Pipe
0		lqx $36, $43, $10
1		lqx $37, $42, $10
2		lqx $34, $26, $10
3		
4		
5		
6		
7	a $33, $36, $37	
8		
9	xor $31, $33, $34	stqx $33, $32, $10
10		
11	cuflt $27, $31, 0	
12		
13		
14		
15		
16		
17		
18	fm $24, $27, $17	

Table 4.3: The optimal scheduling of the instructions required to generate a vector data-word containing four 32-bit scalar random numbers. Note that, assuming that local store addresses are available, the three loads `lqx`s are started consecutively. The first arithmetic instruction, the integer add `a` at cycle 7, starts just after the completion of the second load, due to data dependency. The `XOR`, the conversion `cuflt` and the float multiplication `fm` cannot start consecutively due to data dependency. The store `stqx` has to wait the completion of the integer sum. Note that there are 12 stall cycles over 19 total cycles, although instructions used for data addressing are not taken into consideration.

Cycle	Even Pipe	Odd Pipe
0		lqx $37, $43, $9
1		lqx $38, $42, $9
2		
3		
4		
5		
6		
7	a $29, $37, $38	
8		
9		stqx $29, $35, $9
10		lqx $25, $34, $9
11		
12		
13		
14		
15		
16	xor, $27, $29, $25	
17		
18	cuflt $26, $27, 0	
19		
20		
21		
22		
23		
24		
25	fm $23, $26, $16	

Table 4.4: The scheduling of reference code used to generate a single vector data-word containing four 32-bit random numbers, obtained with `gcc` compiler. Note that the second `LOAD` is delayed due to address calculation (not shown) and that the third is placed after the STORE due to a false dependence. As a consequence the XOR is also delayed but it starts as soon as the LOAD is complete. The other instructions are in the expected locations.

Cycle	Even Pipe	Odd Pipe
0		lqx $42, $16, $8
1		
2		lqx $43, $16, $9
3		lqx $41, $16, $13
4		
5		
6		
7		
8	a $40, $42, $43	
9		
10	xor $38, $40, $41	stqx $40, $16, $14
11		
12	cuflt $34, $38, 0	
13		
14		
15		
16		
17		
18		
19	fm $28, $34, $20	

Table 4.5: The scheduling obtained with `gcc` compiler of the optimized random number generation code. The three `LOAD`s are not consecutive due to additional instructions (not shown) required for addressing, but all other instructions are started as soon as possible, allowing a scheduling very similar to the ideal one.

IDX_i	R0	R1
0	0x00...00	0x00...00
1	0xff...ff	0x00...00
2	0x00...00	0xff...ff
3	0xff...ff	0xff...ff

Table 4.6: Generation of vector data-words R0 and R1, used to concurrently compare the energy differences of all the spins embedded in a scalar variable with the random numbers contained in the corresponding scalar slot of IDX.

The SPE instruction set allows to execute comparisons and assignments independently for each scalar slot of vector data-words. Each scalar data-word embedded into vector data word IDX can assume values in the set $0, 1, 2, 3$ and it is used to produce two scalar data-words embedded in two the vector data-words R0 and R1. Each pair of corresponding bits from R0 and R1 represents the least and the most significant bits of a scalar slot of IDX, and correspond to the variable R[2] introduced in section 2.3.2.

Because random numbers are shared between samples, $w = W/u$ spins shares the same value of a scalar slot of IDX, so for each scalar data-word embedded in IDX w bits of R0 and R1 have to be set in the way shown in Table 4.6

If $V = 4, 8, 16$ (SIMD-granularities supported by the architecture), then the shuffle instruction can be used to compose the IDX vector. The shuffle instruction takes three vectors as arguments: two input data vectors and a mask. The results is a vector composed by bytes from the two input vectors mixed in the way specified by the mask argument. Figure 4.12 shows how in the case of $V_r = 16$ four random vector data-words are manipulated to produce a single IDX vector.

After shuffling, the vector IDX is splitted into R0 and R1 vectors using two AND and two CMPEQ instructions, that operate independently on each scalar slot. The two ANDs allows to isolate the first or the second bit of the value represented by IDX. The CMP instructions allows to replicate the same bit on all the w bits of a scalar variable (because they share the same random number). This is possible because the CMPEQ instruction sets to 1 all the bit of the scalar slot if the condition is met for that slot:

```
R0 = spu_and (IDX, one_v);
R1 = spu_and (IDX, two_v);

R0 = spu_cmpeq (R0, one_v);
R1 = spu_cmpeq (R1, two_v);
```

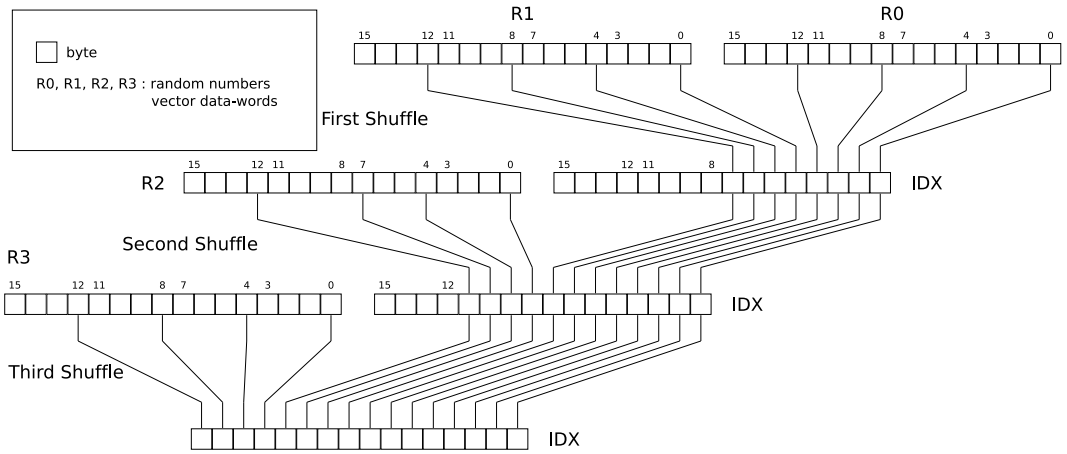

Figure 4.12: The shuffling of $V_r \times$ 32-bit random numbers, to compose the vector data-word `IDX` that is used to perform comparison with the energy differences vectors.

If V is not supported by the architecture, then it is possible to to produce $V/4$ vector data-words of 32-bit random numbers. The same procedure used when $V = 16$ is used to produce $V/16$ independent couples of `R0` and `R1` vectors. Finally, a single pair of `R0` and `R1` is obtained using the `SEL` instruction, that is similar to the `shuffle` but operates at bit level. However, in this way we are not taking full advantage from the SIMD nature of the architecture, so overheads have to be expected.

Table 4.7 shows the scheduling obtained with the real code to generate random numbers when $V = 4$. As we can see, although this is the simplest case, the generation of `R0` and `R1` take a non negligible amount of cycles.

In general, the overhead caused by addressing and stall cycles can be mitigated when SIMD-granularities higher than $V = 4$ are used. The code that generates one random vector data-word has to be repeated two or more times, and as a consequence stall cycles are reduced due to pipelining. The ideal number of instructions needed to generated V scalar 32-bit random numbers and to generate the vector data-words `R0` and `R1` is shown in Table 4.8. The instructions needed to read and write the local store are not counted, as they are implementation dependent. In the real code additional load/store, integer and logical instructions are used to exchange data between the register file and the local store. The performance of this fraction of code is expressed in terms of the number of nanoseconds required to generate a 32-bit random number (ns/rnd), that is directly comparable to the system spin update time ($ns/spin$).

Table 4.9 shows the the behavior of actual code when generating different quantities of random numbers, corresponding to increasing SIMD-granularities, and to compose the `R0` and `R1` vectors. Note that the pipelining and the partial overlapping of the instruction used to generate independent random number vector data-words

Cycle	Even Pipe	Odd Pipe
0		lqx $42, $16, $8
1		
2		lqx $43, $16, $9
3		lqx $41, $16, $13
4		
5		
6		
7		
8	a $40, $42, $43	
9		
10	xor $38, $40, $41	stqx $40, $16, $14
11		
12	cuflt $34, $38, 0	
13		
14		
15		
16		
17		
18		
19	fm $28, $34, $20	
20		
21		
22		
24		
25	fcgt $29, $28, $30	
26		
27	selb $26, $11, $18, $29	
28	fcgt $27, $28, $29	
29	fcgt $24, $28, $26	
30	selb $25, $10, $26, $27	
31		
32	selb $4, $17, $25, $24	
33		
34	and $22, $4, $11	
35	and $21, $4, $10	
36	ceq $2, $22, $11	
37	ceq $3, $21, $10	

Table 4.7: The scheduling of the actual random number generation code. Instructions used for addressing are not shown. Note that the loads are not consecutive, although only a stall cycle is present, which means that the addressing is not too heavy. The other instructions are started as soon as possible, as we expected.

Instruction	Pipe	V=4	V=8	V=16	V=32	V=64	V=128
ADD	even	1	2	4	8	16	32
XOR	even	1	2	4	8	16	32
FMUL	even	1	2	4	8	16	32
SEL	even	3	6	12	26	54	126
AND	even	2	2	2	4	8	16
CMP	odd	5	8	14	28	56	112
SHUFFLE	odd	0	1	3	6	12	24
Total Even		8	14	26	54	110	238
Total Odd		5	9	17	34	68	136
ns/random		0.63	0.55	0.51	0.53	0.54	0.58
speed-up			1.15	1.26	1.19	1.17	1.09

Table 4.8: The ideal number of instructions required to produce V scalar 32-bit random numbers and the consequently generate `R0` and `R1` vector data-words. Note that the code is always dominated by the instructions of the even pipeline, so that the estimate of the nanoseconds required to generate a single 32-bit random number is obtained by multiplication the number of even instruction by the length of a clock cycle, and then dividing the result by V.

allows to obtain a better usage of the odd pipeline when increasing V, which implies that the time required to produce a scalar 32-bit random number slightly decrease.

The odd pipe is not heavily used and the program is dominated by the even pipe instructions. In particular the even pipe is almost full starting from $V = 4$ and as a consequence the performance gain of higher SIMD granularities is quite limited. Note that the data manipulation is tipycally performed by instruction of the even pipe, so for higher values of V there is an increase in the number of odd pipe instructions and the pipe as a higher usage percentage. In conclusion, we can assume that higher SIMD-granularities are the best choice, although the gain is not dramatic. Table 4.10 shows the results of compiling the same code with the IBM `XLC V9.0` compiler. Although the behavior is similar to the previous case, in general there is much more overhead and the time required to generate a scalar 32-bit random number is slightly higher.

Finally, Table 4.11 shows the count of instruction of real code. The number of instructions increases as expected, although there are much more `ADD`s and `XOR` than expected, and they constitutes the higher fraction of the addressing overhead.

In conclusion, as random number generation is a fraction of the spin update procedure, we have obtained a lower bound for the system spin update time $\tau(V)$ for a set of different SIMD-granularities. When in the next section an estimate of the performance of the fraction of code that performs the spin update will be proposed, we will be able to compare it with the amount of time taken by random

V	I_E	$E_{\%}$	I_O	$O_{\%}$	$D_{\%}$	I	$cycles$	ns/rnd	$S.up$
4	35	87	8	20	16	43	40	3.1	
8	55	100	14	25	25	69	55	2.2	1.4
16	90	100	27	30	30	117	90	1.8	1.7
32	168	100	49	29	29	217	168	1.6	1.9
64	324	100	103	32	32	427	324	1.6	1.9
128	636	100	201	32	32	837	636	1.6	1.9

Table 4.9: The behavior of the code that produces V scalar 32-bit random numbers and generates the `R0` and `R1` vector data-words. For each pipe, the number of instructions (I_E and I_O) and the pipeline usage ($E_{\%}$ and $O_{\%}$) are reported. The first column (V) indicates he quantity of 32-bit random number that are generated. The sixth column ($D_{\%}$) allows to evaluate how well the concurrency between the two pipelines is exploited. Finally, the number of total cycles allows to determine how many nanoseconds are required to generate a scalar 32-bit random number. The final columns report the speedup that is achieved pipelining the generation of an increasing quantity of random numbers. The best performance is achieved when generating 32 or more random numbers.

V	I_E	$E_{\%}$	I_O	$O_{\%}$	$D_{\%}$	I	$cycles$	ns/rnd	$S.up$
4	29	53	28	51	32	57	55	4.3	
8	65	100	19	29	30	84	65	2.5	1.7
16	100	100	36	33	36	146	110	2.2	2.1
32	204	100	65	32	31	269	205	2.0	2.2
64	392	100	144	37	36	536	394	1.9	2.3
128	768	100	273	36	36	1041	768	1.9	2.3

Table 4.10: The details of the code that produces V scalar 32-bit random numbers and generates the `R0` and `R1` vector data-words, compiled with IBM `XLC` compiler. The results should be compared with those of Table 4.9. Note that the performance are worse, although the behavior in function of V is very similar.

Instruction	**Pipe**	**V=4**	**V=8**	**V=16**	**V=32**	**V=64**	**V=128**
ADD	even	7	13	23	43	83	163
SUB	even	2	2	2	2	2	2
AND	even	8	12	20	38	74	146
XOR	even	3	4	6	10	18	34
FMUL	even	1	2	4	8	16	32
FCMP	even	3	6	12	24	48	96
CMP	even	2	2	2	4	8	16
BSEL	even	3	6	12	26	54	110
ROTATION	even	1	1	1	1	1	1
SHIFT	even	2	2	2	2	2	4
CUFLT	even	1	2	4	8	16	32
NOP	even	2	3	2	2	2	2
SHUFFLE	odd	0	1	3	6	12	24
LOAD	odd	3	6	12	24	48	96
STORE	odd	3	4	6	10	18	34
BRANCH HINT	odd	2	2	2	3	3	3
LNOP	even	0	1	4	6	22	42
STALLS		3	0	0	0	0	0
Total Even		35	55	90	168	324	638
Total Odd		8	14	27	49	103	199

Table 4.11: The number of each type of instruction in the code that generate V scalar 32-bit random numbers and to compose `R0` and `R1` vector data words. Note that the number of `XOR`s scales as expected, if we trascurate two extra `XOR`s that are present in all the cases. There are more integer sums than those strictly required for the random number generator, but this is expected, because they are also used for addressing. Note that their number almost doubles at the doubling of V. There is a large number of `AND`s, used for addressing. The other instructions show the predicted behavior, although is some cases there are a few more instructions than expected. Note that there are two extra STOREs to save `R0` and `R1` vector-data words (the output of this fragment of code), that in the spin update procedure are only two temporary values.

number generation, in order to discover the real amount of performance gain that can expected when increasing the SIMD granularity.

4.1.4 Local Memory: Computational Core

Now the spin update procedure will be analyzed in order to estimate the number of instructions that it requires and the consequent system spin update time for different couples of SIMD-granularities and lattice sizes (V, L). The Figure 4.13 show a theoretic DAG [2] of the spin update procedure. As we can see, to calculate the value of ΔE at least 13 instructions are needed, and additional instructions are required to decide if the spin has to be flipped and (possibly) to calculate the random numbers. Probably some operations can be mapped into `MULADD`s, thus decreasing the number of total instructions .

Ignoring latencies and random number generation phase, at least 13 `LOAD`s are required to get the 13 input vectors, and extra instructions are needed to determine memory addresses. As a consequence, given that the instructions needed to perform real computations are ≤ 13, it is likely that the spin update fraction of code will be dominated by local store accesses.

Multispin coding translates integer operations into a larger set of bitwise logical instructions. From the point of view of the input data access time, there are no differences when packing one spin or w spins inside a scalar data-word, while the increase of the number of instructions performing computations can negatively impact performance. However, multispin coding has two advantages. Firstly, in CBE processor integer multiplications takes 6 cycles to complete, while all the logical instructions have a very small latency of only 2 cycles, so pipeline stalls are less probable. Moreover, integer arithmetic is supported only up to a SIMD-granularity $V = 16$, so multispin coding, that does not have such limitation, allows a better exploitation of synchronous parallelism.

The ideal number of instructions required to implement the multispin coding update procedure for a single vector data-word, given that random numbers have already been calculated and packed, is shown in Table 4.12. The procedure does not depend on V, so the number of instruction is constant.

At least 43 instructions are required to update a vector data-word, and if we assume that they are perfectly pipelined and if we ignore their latencies, so that each instruction requires only one cycle, it is possible to estimate another lower bound for the system spin update time (that have to be added to the lower bound determined for random number generation, in order to obtain a global estimate), as shown in Table 4.13.

Although this estimate is ideal and does not take into account the overheads associated to data access, it allows to evaluate the impact of random number gen-

[2] DAG is a Direct Acyclic Graphs that represents the dependency between instructions

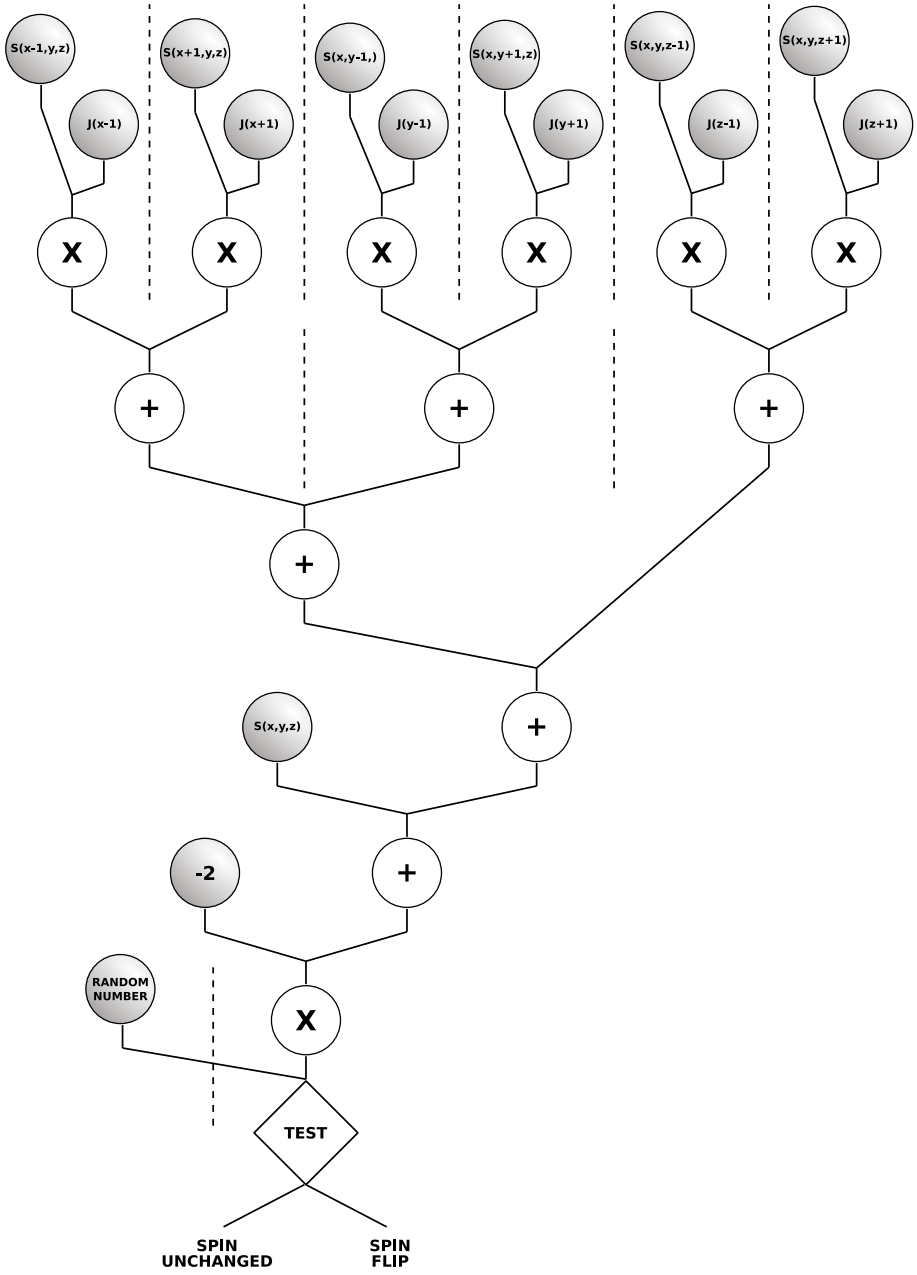

Figure 4.13: The DAG of the spin update procedure. Note that the dotted lines indicate operations that can in principle be executed in parallel, so that instruction level parallelism can be exploited.

spin update		
Type	**Pipe**	**Number**
XOR	even	28
AND	even	12
OR	even	3
Even total		43
Odd total		0
Total		43

Table 4.12: The number of instruction required to update a vector-data word, given that a vector containing enough random numbers is available. The instruction count does not depend on SIMD-granularity V. Data access is ignored, as long as it is implementation-dependent. As a consequence there are not odd pipe instructions.

Ideal spin update						
V	**4**	**8**	**16**	**32**	**64**	**128**
ns/spin	3.36	1.68	0.84	0.42	0.21	0.10

Table 4.13: An estimate of the performance of the spin update fraction of the code, in terms of nanoseconds needed to update a spin of a single system. The basic idea is that the code in invariant with respect to V, so the performance should scale linearly to it. However, these estimates do not take into account addressing, data access and (by definition) random number generation, so they are only an upper bound for the performance achievable with the real program.

Ideal spin update time $\tau(V)$												
V	R_E	R_O	R_T	$R_{\%}$	U_E	U_O	U_T	$U_{\%}$	Tot_E	Tot_O	$\tau(V)$	$S.up$
4	8	5	0.63	0.16	43	0	3.36	0.84	50	1	3.98	1.00
8	14	9	0.55	0.25	43	0	1.68	0.75	53	3	2.23	1.79
16	26	17	0.51	0.38	43	0	0.84	0.62	59	5	1.35	2.96
32	54	34	0.53	0.56	43	0	0.42	0.44	75	10	0.95	4.21
64	110	68	0.54	0.72	43	0	0.21	0.28	107	20	0.75	5.33
128	238	136	0.61	0.85	43	0	0.10	0.15	171	40	0.71	5.61

Table 4.14: A comparison between the ideal number of instructions required for random number generation and spin update fractions of code. R_E and R_O indicate respectively the number of instructions of the even and of the odd pipe used for the random number generation. R_T indicates the time required to generate a 32-bit random number and $R_{\%}$ is the fraction of global time occupied by random number generation. U_E, U_O, U_T and $U_{\%}$ give the same informations for the spin update fraction of code. Finally, Tot_E, Tot_O are the total number of even and odd instruction, while $\tau(V)$ is an ideal estimate of the system spin update time (in $(ns/spin)$).

eration on system spin update time. Table 4.14 summarizes the number of even and odd pipe instructions required to execute the whole update procedure with increasing SIMD-granularities, taking into account the ideal number of instructions for both random number generation and spin update fractions of code.

Up to $V = 64$ a higher SIMD granularity implies a better system spin update time, although the transition from $V = 16$ to $V = 32$ reverses the balance between the two phases and random generation takes the predominant fraction of time.

Data Access

The actual code for the generation of random numbers is several times slower than its ideal estimate, although its efficiency increase with SIMD-granularity. The code that performs the spin update also has a certain amount of overhead associated data access, but as long as the number of instructions is invariant into respect of SIMD-granularity, it should scale linearly with it. We can expect that the difference between the theoretic system spin update time $\tau(V)$ and the one of the actual code is determined by:

- load/store instruction;
- integer and bitwise logic instructions used for addressing

The previous estimates of the ideal number of instructions assumed that all the input data-words were already loaded into the register file, but in real implementation the access to local store data structures is not negligible. To update a single vector data-word, at least eight different half-planes are accessed, as shown earlier,

so we should assume that at least eight data pointers are used. Because thirteen vector data-words containing spins or coupling terms constitute the input data set (neglecting the special cases), it is safer to assume that the program has to keep track of at least thirteen pointers while updating the vector data-words that compose a half-plane. Moreover, to generate a vector of four 32-bit random numbers four offsets are used to access the `IRA` array used by the random number generator. As the latter is a circular array with 256 vector entries, modulus is required, that can simply be implemented with logical `AND`s.

The data set can be accessed with a quite regular pattern, because the data layout that we have proposed implies that the relative positions of neighbor spins and coupling terms is the same for all the internal vector data-words of a half-line. In our code we use thirteen pointers, to independently point to input data, and each pointer is simply incremented of one position after the update of a data-word, thus avoiding the using of multiplications in addressing. The use of C-style array indexes would have required several sums and multiplications.

Another interesting property is that input data-words are relatively near, in terms of memory address, because they are neighbors in the lattice and because the local store is relatively small. In particular, when we are updating a white spin vector data word, its neighbors spins are all located in three adjacent half-planes, so there is the chance to use a single pointer, with different offsets computed at compile-time, to load all the required data-words.

The SPU ISA supports the *d-form* to compose the address used by a load or by a store instruction [31]:

```
lqd RT,symbol(RA)}
```

where the address is obtained by adding the 10-bit signed value of `symbol`, with 4 zero bits appended, to the value in the preferred slot of register `RA` and forcing the least significant 4 bits of the sum to zero. The 16 bytes at the computed address are placed into register RT.

The *d-form* is interesting because the register can be the address of the black spin half-planes. The offsets required to access black spins data-words are known at compile-time. An offset is expressed in terms of quadwords, and a signed value of 10-bit allows to load data-words that are within a distance of 512 quadwords. For example, if $L = 64$ and $V = 16$ then each half-plane is composed by only 128 vector data-words, so it is effectively possible to use the *d-form*.

The simplest way to analyze the overhead associated with data access is the determination of the difference between the ideal number of instructions required by spin update procedure and the number of instructions and stalls of the actual code. The instructions used to generate memory address typically are:

- integer instructions (sums, multiplications, ...)
- shifts (for fast multiplication and division by powers of 2)
- logical instruction (ANDs, ...)

We should expect that the actual mode has much more instructions of this type then the optimal one, while the other types of instructions should scale as estimated. To analyze the real code, for several combinations of V and L we have isolated the innermost section of code (intended as a set of instructions with only an entry point and one exit point) that performs the update of one or more vector data-words. As seen in section 4.1.2 the location of the neighbors varies with the position and the oddness of the vector data-words and with the length of the half-line. As a consequence, there are different access patterns to take in consideration:

- if a half-line is composed by a vector data-word only, the section contains only the update of that vector and shuffling is needed. As we are not implementing unrolling along Y or Z directions, it is not possible to exploit pipelining with the update of other data-words
- if a half-line is composed by two vector data-words, the instructions that update the two vectors are inside the same section and can be pipelined. A certain amount of manipulation of input data is needed, as not all the required data-words are located in the correct scalar slots
- if a half-line is composed by three vector data-words, they can be updated within a single section and the corresponding instruction can be pipelined. Permutations of input data are required
- if a line is composed by four or more vector data-words, a basic section is represented by the code that performs the update of a internal vector data words. For this vector permutation of input data is not needed, but the instruction cannot be pipelined with those that update other vectors. If the internal loop is unrolled, then there is the chance of a better instruction level parallelism, but the case of even and odd number of vector data-word per line have to be separated

Odd and even half-lines require have different memory access patterns, so the compiler may produce separate sections of code for the two cases. In CBE processor the cost of branches is quite high, because they cause the pipelines to be emptied, so it is very important to keep the code that loads data as linear as possible [74].

To benefit from the potential better scheduling allowed by the presence of independent update procedures inside the same section, it is necessary to expose to

the compiler the independence of instructions, so it is important to avoid variables reuse at to keep access to arrays outside the core computations. All the required black and white spins and coupling terms can be loaded at the start of the section and stored back to local store only at the end of computations. The generation of random numbers have instrinsically a sequential nature due to the access to `IRA` array, so there an unavoidable degree of sequentially in the generation of independent random numbers, but since it do does not occupies a predominant fraction of time, it should not prevent pipelining.

Now we will analyze the behavior of the code when using the three SIMD-granularities supported by the architecture. In principle the number of instructions of the update code does not depend on V, but the performance should scale linearly with V. We have already seen that random number generation is better for high SIMD-granularities but does not scale linearly, so we do not expect the system spin update time to be simply inversely proportional to V.

The results are shown in Table 4.15. This version of the program is very simple and does not use a different set of variables for each vector data-word of a half-line. An improved implementation was developed, using a dedicated set of variables for each vector data-word and explicitly unrolling the update of a line up to four vectors, and adding the support SIMD-granularities higher than those of the architectures. The results are shown in Table 4.16.

The comparison between the two tables makes it clear that the pipeline is so full that there are very little improvements when unrolling the inner computational cycle. The benefits, although measurable, generally decrease the system spin update time of less than 10%. As a consequence, it is presumable that more aggressive unrolling does not grant enough benefits. We will do not explicitly unroll the loops along Y and Z directions.

The performance extrapolated from real code can be used to evaluate the behavior of the fraction of instructions dedicate to spin update. Table 4.17 extrapolates the subtracting from the global time the random number generation time shown in Table 4.9. We can observe that the update time scales more than linearly, probably because the update instructions have been overlapped to the instruction used to generate random numbers.

To evaluate the overhead associated with data access, Table 4.18 shows the number of instructions required to update a vector data-word with increasing SIMD-granularities. Note that most instructions scales in the correct way, but there are much more integer `ADD`s and logical `AND`s than expected . There is a `STORE` for each random number vector produced plus a final `STORE` to save the update spin vector, but there are too many `LOAD`s, which means that the procedure requires more variables than those that can be kept into the register file, so some of them have to be loaded just before their usage without being modified, as the are there no additional `STORE`s.

On-Chip Memory: static estimate $\tau_S(V)$ (not optimized)										
V	L	I	I_E	$E_{\%}$	I_O	$O_{\%}$	$D_{\%}$	*cycles*	$\tau_S(V)$	*S.up*
4	8	121	91	93	30	31	25	98	7.7	
4	16	228	166	92	62	35	29	178	7.0	1.1
4	24	316	231	93	85	34	28	249	6.5	1.2
4	32	220	166	94	54	31	26	176	6.9	1.1
8	16	139	105	100	34	32	32	105	4.1	
8	32	266	197	100	69	35	35	197	3.9	1.0
8	48	382	281	100	101	36	36	281	3.7	1.1
16	32	193	142	99	51	36	36	143	2.8	
16	64	372	269	99	103	38	38	273	2.7	1.0

Table 4.15: The analysis of the performance of the actual code that uses on local memories. The column labeled as I indicates the total number or instructions, while I_E and I_O are respectively referred to the number of instructions of the even and odd pipelines. Columns $E_{\%}$ and $O_{\%}$ indicates the usage of the two pipelines, while $D_{\%}$ indicates the percentage of dual issues. $\tau(V)$ is an esistimation of the system spin update time (in $ns/spin$) based on the analysis of assembler code. Note that the code si dominated by the instruction of the even pipeline, that has an high usage percentage, indicating that there are not many stall cycles. Note also that the lattice linear size L has a very little impact on performance.

Finally, Table 4.19 shows a comparison between the best system spin update times obtained with the static analysis of assembler code with with the best times obtained effectively running the program on the hardware (in this case a QS22 blade server) and measuring only the time required to perform the isolated computational kernel. The time measurements are taken using the *decrementer register*, that is readable and writeable through the channel interface of each SPE. The decrement register decrements is value `timebase` times in a second. The `timebase` differs in each version of CBE processor. The CBEs of QS22 blades have a `timebase` value of 26664863, which implies that it the decrementer registers decrements each R ns:

$$R = \frac{1 \times 10^9 \text{ ns}}{26664863} = 37.5 \text{ ns} \tag{4.2}$$

As this is a relatively large amount of time (it is equivalent to 120 clock cycles), it is better measure the update of large lattices. The computation time is dominated by the update of internal half-planes. Moreover, the update of an half-plane requires to repeat the procedure that updates a half-line L times, so at least $L-1$ jumps are required, which can cause a high overhead, because it interrupts the pipelines. Although we can choose an L that keeps constant the number of vector data-words inside a half-line independently by SIMD-granularity V, the size of the lattice increases proportionally to V, so a high granularity implies a higher number of internal

On-Chip Memory: static $\tau_S(V)$ estimate										
V	L	I	I_E	$E_{\%}$	I_O	$O_{\%}$	$D_{\%}$	$cycles$	$\tau_S(V)$	$S.up$
4	8	121	91	93	30	31	25	98	7.6	
4	16	216	162	99	54	33	33	164	6.4	1.2
4	24	302	231	99	71	30	30	233	6.0	1.3
4	32	474	306	100	168	55	55	306	6.0	1.3
8	16	141	107	100	34	32	32	107	4.2	
8	32	260	196	99	64	32	32	198	3.9	1.1
8	48	372	281	100	91	32	32	281	3.7	1.1
16	32	193	142	100	51	36	36	142	2.8	
16	64	362	267	100	95	35	35	268	2.6	1.1
32	64	297	221	100	76	35	35	221	2.2	
64	128	509	377	100	132	35	35	377	1.8	
128	256	924	688	99	236	34	34	692	1.7	

Table 4.16: An analysis of the performance of actual code that uses only on-chip memory. The inner computational cycle, that updates an X line, has been explicitly unrolled, and the support to SIMD-granularities higher than $V = 16$ has been added. See the caption of Table 4.15 for the meaning of column headings. $\tau_S(V)$ is a static estimate of the system spin update time (in $ns/spin$) based on the analysis of the assembler code. Note that the code is dominated by the instruction of the even pipeline, that has an high usage percentage. The lattice linear length L has a little impact on performance (as expected), and the higher SIMD-granularity allows to achieve the best performances.

On-chip memory: static $\tau_S(V)$ estimate								
V	$\tau_S(V)$	R_T	$R_{\%}$	U_T	$U_{\%}$	$R_{S.up}$	$U_{S.up}$	$\tau_S(V)_{S.up}$
4	7.6	3.1	41	4.5	59			
8	4.2	2.2	53	2.0	47	1.4	2.2	1.8
16	2.8	1.8	64	1.0	36	1.7	4.5	2.7
32	2.2	1.6	73	0.6	37	1.9	7.5	3.4
64	1.8	1.6	89	0.2	11	1.9	22.5	4.2
128	1.7	1.6	94	0.1	6	1.9	45.0	4.7

Table 4.17: A static analysis of the scaling of spin update time $\tau(V)$, dividing the fractions of time spent generating the random numbers and updating the spins. R_T and U_T indicate respectively the time required to generate a 32-bit random number and to update a single spin of a system, while $R_{\%}$ and $U_{\%}$ show their percentage into respect of system spin update time $\tau(V)$. $R_{S.up}$, $U_{S.up}$ and $\tau(V)_{S.up}$ shows the speedup of random number generation, spin update and $\tau(V)$ when increasing the SIMD-granularity V. Note that starting from $V = 32$ the random number generation time is predominant and its performance do not increase for higher granularities. Spin update, instead, improves significatively when using high values of V, thus allowing a better spin update time.

Instruction	**pipe**	**V=4**	**V=8**	**V=16**	**V=32**	**V=64**	**V=128**
ADD	even	15	20	30	50	90	170
SUB	even	3	3	3	2	2	2
AND	even	18	22	30	48	84	156
OR	even	3	4	3	3	4	3
XOR	even	29	30	32	36	44	60
FMUL	even	1	2	4	8	16	32
FCMP	even	3	6	12	24	48	96
CMP	even	4	6	4	6	10	18
BSEL	even	10	13	19	33	61	117
ROTATION	even	3	3	3	3	3	3
SHIFT	even	0	0	1	4	5	5
CUFLT	even	1	2	4	8	16	32
NOP	even	4	1	5	2	1	52
SHUFFLE	odd	0	1	3	6	12	24
LOAD	odd	19	20	30	42	66	115
STORE	odd	2	3	5	9	17	34
BRANCH HINT	odd	2	3	3	3	3	2
LNOP	even	2	2	0	7	24	0
STALLS		1	0	0	0	0	3

Table 4.18: The number of instructions of the actual code that updates a vector data-word with SIMD-granularity V.

On-chip memory: $T_S(V)$ Vs. $T_R(V)$				
V	L	$\tau_S(V)$	$\tau_R(V)$	$Overhead(\%)$
4	32	6.0	6.3	5
8	64	3.7	3.8	3
16	64	2.6	2.8	8
32	64	2.2	2.3	5
64	128	1.8	1.9	6

Table 4.19: A comparison between the the static $\tau_S(V)$ and the run-time $\tau_R(V)$ estimates of system spin update time The static one is obtained analyzing the scheduling of the code, while the second one is obtained measuring (with the *decrementer register*) the execution time of the computational kernel. Note that the difference between the two estimates is alway under 10% and that we used the largest possible line lattice sizes.

half-planes, thus increasing the precision of measurements. Table 4.19 compares the static and run-time analysis of system spin update time. Note that in the case of static analysis it was to possibile to assume linear sizes L and SIMD-granularities V that in pratice are not applicable due to memory constraints or because are beyond our targets (like $L \geq 256$ and $V = 128$).

For small SIMD-granularities there is a large difference between the two estimates, but it can probably be attributable to the precision of the measurements. These estimates of the system spin update time $\tau(V)$ will be later applied to the balance equations to determine which lattice sizes allows to completely hide data transfers time inside computation time.

4.1.5 Main Memory: Performance and Data Access

The previous considerations about the ideal number of instructions required to update a vector data-word are still valid for the implementation of the program that keeps the data set in main memory, but as long as the data structures are different, we can expect a impact of addressing on performance. It does not make sense to use the main memory strategy for small lattices that can be kept in local memories, because they can obviously be updated faster using only local resources. The target of this version of the program is to update bigger lattices, keeping in mind that the upper bound is always $L = 128$. Not only this strategy allows to update larger lattices, but also allows to update a small lattice with less cores than when using local memory, that can be useful in those cases that can be managed by local memory version only using dual-processors configurations. Incidentally, it also allows to update small lattices with a higher number of samples, but although the increase of statistics is useful, as already remarked it is not our main target.

When the lattice is stored in main memory the local store data structures are

Off-chip memory: static $\tau_S(V)$ estimate									
V	I	I_E	$E_{\%}$	O_E	$O_{\%}$	$D_{\%}$	$cycles$	$\tau_S(V)$	$S.up$
4	105	81	96	24	26	27	84	6.6	
8	127	98	100	29	30	30	98	3.8	1.7
16	181	136	100	45	33	33	136	2.7	2.4
32	289	217	100	72	33	33	217	2.1	3.1
64	499	373	100	126	34	34	373	1.8	3.7
128	913	685	100	228	33	33	685	1.7	3.9

Table 4.20: $\tau_S(V)$ is a static estimate of spin update time $\tau(V)$. In this case the lattice data is stored in main memory. The table refers to the update of a single vector data-word. As in the case of local memory, the instruction of the even pipe dominates the code, and that pipeline is almost full. The system spin update time, as expected, is only a little higher then those estimated when using local memories (Table 4.16), due to the different data structures.

essentially the buffers containing the spins and the coupling terms, that are rotated at the end of the update of each half-plane, so we can expect a relatively complicated access pattern and more instructions associated to addressing than in previous case. We can also expect that the synchronization between cores and that the management of DMA transfer decrease the system spin update time, but this issues will be analyzed later.

Table 4.20 shows the behavior of the computational core of the program that stores the data set in main memory. In particular, the number of instructions, the pipelines usage and the total cycles of the actual code that updates a vector-data word are illustrated. Although higher, the system spin update time is comparable to those of local memory program.

Table 4.21 compare the estimate of system spin update time $\tau_S(V)$ based on static assembler analysis with those measured run-time. Only the best times have been taken into consideration, because they are obtained with the longest sections of code, that minimize the overheads and allow to make run-time measurements with more precision. In particular, the larger lattices allow to measure longer periods of time, thus reducing the inaccuracy related to the resolution of the decrementer register. Moreover, as each line is composed by more vector data-words, the impact of the branch instruction at the end of the update of each line is less relevant. As a consequence, the difference between the two type of estimate of $\tau(V)$ is very small.

4.1.6 Memory Usage

As previously stated, a very important programming technique consists in loading all the data from arrays (that are allocated in local store) to local variables at the

Off-chip memory: $T_S(V)$ Vs. $T_R(V)$				
V	L	$\tau_S(V)$	$\tau_R(V)$	$Overhead(\%)$
4	72	6.6	8.8	33
8	96	3.8	4.0	5
16	128	2.7	2.8	4
32	128	2.1	2.2	5
64	128	1.9	1.9	0
128	256	1.7	1.7	0

Table 4.21: A comparison between the static and the run-time estimate of $\tau(V)$ for the program that use main memory to store lattice data. Note that the run-time overhead is very small. Except the case of $V = 4$, the overhead is generally smaller that those of the on-chip memory program (see Table 4.19) because in this case it is possible to manage bigger lattices, that allow to reduces the impact of branches and to measure longer periods of time.

begin of a procedure, performing the computations and storing the results back to data array only at the end . Keeping input and temporary data into local variables avoids ambiguity regarding the memory accesses (that might require with very complicate addressing and pointers arithmetic), so that a very compact scheduling can be achieved. The drawback is that if too much variables have to be kept into the register file, then there can be a performance drop due to register spilling [3]. For these reasons, it is important to make an estimate of the number of local variables that are required to update a single vector data-word. The spin update procedure require at least:

- 7 spins (the current one and its six first neighbors)
- 6 coupling terms between the spin and its neighbors
- 1 register containing the combination of V random numbers
- at least 5 temporary variables for random number generation (that increase when $V > 4$).
- at least 8 temporary variables for the computation of ΔE

We can estimate that at least 27 variables are needed to update a single vector data-word. Moreover, there are at least 8 pointers to data structures, and various physics parameters that are likely to be kept in register file. Additional registers are needed when the SIMD-granularity $V \neq 4$, due to the random number vectors that

[3]Register spilling occurs where there are more live variables than machine registers, and as a consequence variables have to be temporary moved into memory

have to be independently generated and combined together. We can realistically estimate that at least 40 variables are required to update a single vector data word. Most of the variables are mapped into registers as long as they are required to perform calculation, and as a consequence at a given instant it is likely that less than 40 register are required. We expect that register file is large enough to support short loop unrolling so that, if the pipelines are enough free, 2 or 4 vector data-word are independently updated in the same section. The CBE ABI [75] states that registers in the range $[R3, R79]$ can be used as local variables of the current function, while registers in the range $[R80, R127]$ must be preserved across function call. In our case, the computational kernel can be embedded in a single function, using macros or inline functions, we can expect that the compiler can almost use the whole register file to perform computations.

Table 4.22 shows the local store usage of our two programs. As we can see, to manage lattices in our target range ($L \in [16, 128]$) it is not necessary to take into consideration the *slice* version of the main memory program. Because the computational cores would be the same, and only larger data transfers would be required, it will no longer be taken into consideration.

With this table it is possible to determine, given the tuple (V, L, C), if it is possible to use only local memories and thus get the best performan, or if it is necessary to use main memory.

4.2 Interaction Between Cores

In this section we will apply the estimate of system spin update time $\tau(V)$ to the balance equations proposed in chapter 3. We will take in consideration only ideal bandwidths. The behavior of CBE for data transfers between main and local memory is well documented in [76, 49, 56, 57, 33]. It is not trivial to exploit the peak bandwidth. First of all, the addresses involved in DMA transfer should be 128-byte aligned, and the transfer size should be a multiple of 128-byte, to match the underlying hardware. As a general rule, a high number of transactions should be issued. In particular, when accessing the main memory, the peak 25.6 GB/s bandwidth cannot be achieved using a single SPEs. The benchmarks shows that all the SPEs have to be used and the data transfer have to be at least of 2 KBytes.

In our programs the basic data transfer unit is always the half-plane, that occupies $w \times L^2/2$ bits of memory. Note that in our data structures the vector data-words that compose a half-plane are always in consecutive memory locations. Table 4.23 shows the size of half-planes for different SIMD-granularities V and lattice linear sizes L. As we can see, only for value of $L \geq 32$ a half plane is greater than 2K, so for smaller lattices we have to expect a sub-optimal usage of bandwidth. For the larger lattices, insted, half-planes are bigger than the maximum length of a single

Local Store usage (in KBytes) for each SPE											
		V=4		V=8		V=16		V=32		V=64	
L	C	LM	MM	LM	MM	LM	MM	LM	MM	LM	MM
8	8	**2**	**2**								
16	8	**1**	**9**	**5**	**4**						
24	8	**34**	**20**								
32	8	**76**	**36**	**38**	**18**	**19**	**9**				
40	8	**144**	**56**								
48	8	243	**81**	**121**	**40**						
48	16	**135**	**810**								
56	8		**110**								
56	16		**110**								
64	8		**144**	280	**72**	**140**	**36**	**70**	**18**		
64	16		**144**	**152**	**72**			**38**	**18**		
72	8		**182**								
72	16		**182**								
80	8			537	**112**						
80	16			287	**112**						
96	8			918	**162**	459	**81**				
96	16			486	**162**	243	**81**				
112	8										
112	16										
128	8					1072	**144**	536	**72**	268	**36**
128	16					560	**144**	280	**72**	**140**	**36**

Table 4.22: The local store space required (in KBytes) bye each SPE for several tuples (V, L, C) when using only local memories (LM) or also main memory (MM). Only the cases reported with boldface fonts are valid (data is small enough to stay in local store).

Half-planes sizes (bytes)					
L	**V=4**	**V=8**	**V=16**	**V=32**	**V=64**
8	128				
16	512	256			
32	2048	1024	512		
48	4608	2304			
64	8192	4096	2048	4096	
72	10368	5184			
80	12800	6400			
96	18432	9216	4608		
112	25088	12544			
128	32768	16384	8192	16384	8192

Table 4.23: The size in bytes of a half-plane for several combinations of linear lattice size L and SIMD-granularity V. A half-plane is a basic data transfer unit: if it is smaller that 16384 bytes it is transferred with a single DMA operation, otherwise it is splitted in smaller transfers. Because an high efficiency in DMA operantions is achieved only when transferring at least 2048 bytes, we cannot expected to be able to exploit the full bandwith for smaller lattice ($L \leq 32$).

DMA transfers (16384 bytes), so they cannot be transferred with a single operation.

4.2.1 Local Memory Version

When the data set is stored in local memories, the transfer of data is overlapped to the update of the internal half-planes, so the estimates of $\tau(V)$ allow to compute the minimum value of L that keeps equation (3.6) balanced given a V and C:

$$L = \frac{16 \times C}{\tau(V) \times V \times B} + 2 \times C \tag{4.3}$$

Table 4.24 shows the minimum value of L for various SIMD-granularities V and number of cores C and reports the minimum L that keeps the equation balanced and shows the range of available L, taking into account the minimum L that guarantee balance for the low boundary and the local store occupation for the high one. Moreover, other constraints are taken into account: L must be a multiple of $2 \times V$ and $L/C \geq 2$. The value of $\tau(V)$ used for the estimate is the one measured run-time, because although it does not take into account all the overheads of the program, it is the more realistic estimate available.

Figure 4.14 describes an example of the sequence of steps needed to perform a full Monte Carlo step.

On-chip memory: min L				
V	C	$\tau_R(V)$	min L	L range
4	2	5.5	4.0	8 - 24
4	4	5.5	8.1	8 - 32
4	8	5.5	16.5	32 - 40
4	16	5.5	33.8	40 - 48
8	2	3.8	4.0	16 - 32
8	4	3.8	8.1	16 - 32
8	8	3.8	16.3	32 - 48
8	16	3.8	33.3	40 - 64
16	2	2.8	4.0	32
16	4	2.8	8.1	32
16	8	2.8	16.2	32 - 64
16	16	2.8	32.9	64 - 80
32	4	2.3	8.0	64
32	8	2.3	16.1	64
32	16	2.3	36.7	64
64	16	1.9	32.3	128

Table 4.24: The minimum lattice size L that keeps the system balanced with given values of SIMD-granularities V and of number of cores C.

Off-chip memory: max C			
V	$\tau_R(V)$	max C	L range
4	8.0	5.7	16 - 72
8	4.9	7.0	32 - 96
16	3.4	9.7	64 - 128

Table 4.25: For several SIMD-granularities V, the maximum amount of cores C that, according to the balance equation, can access the main memory while completely overlapping the data transfer time into computation time. To compute the value of C the run-time estimate $\tau_R(V)$ of the system spin update time has been used, while the main memory peak bandwith has been assumed.

4.2.2 Main Memory Version

When the lattice is stored into main memory, all the cores concurrently access this shared resource to load data in their local memories and to store the updated half-planes. For this reason, the key of balance is the number of cores that, given a system spin update time $\tau(V)$, main memory bandwidth is able to satisfy. Using the equation (3.18) it is possible to determine the highest number of cores that keeps the system balanced:

$$\tau(V) = \frac{\tau(V) \times B \times V}{144} \tag{4.4}$$

The results are shown in Table 4.25. Only with a SIMD-granularity of at least $V = 16$ it is possible to use the eight cores of a CBE processor. However, as the ideal bandwith will probably not be achieved, we can expect that in the real programs the main memory bandwith is not sufficient to feed all the available cores. This topic will be discussed in more detail in the next chapter.

Finally, Figures 4.15, 4.16, 4.17 and 4.18 shown an example of the steps that a core performs to execute a Monte Carlo step on its white half-planes. Note that at the start and at the end of the procedure data transfers are not overlapped, so at least the transfer of some half-planes it is not overlapped to computations, thus increasing the total time.

For each Monte Carlo iteration, when a core has update all its planes of a given color, it has to synchronize with its two neighbors before to proceed to the other color or to the next iteration. In practice each core send to each neighbor a mail that notifies the availability of the just update data. Then, the core waits for analog mails from its neighbors. However, before the send of the mail, and `mfcsyn` instructions has to be issue, to ensure that all data transfers have complete (i.e. alla data is available in main memory).

Gaussian Model			
On-chip Memory			
L	$\tau_S(V)$	$\tau_R(V)$	Overhead (%)
16	5.2	7.2	38
24	4.7	6.3	34
32	4.7	9.2	34
40	5.9	9.2	56
48	5.9	9.2	56

Table 4.26: The comparison between the static and run-time estimates of the system spin update time $\tau(V)$, for Gaussian model when the lattice data is stored in local memory. In this case the inner loop is not unrolled, so the code is more efficient for very small lattice sizes, when an half-line can be updated without branches. However, for $\tau_S(V)$ the difference between the best and the worst cases is under the 15%.

Gaussian Model			
On-chip Memory			
V	C	min L	L range
4	8	18.1	16-40
4	16	42.2	16-48

Table 4.27: Balancing of processing and communication for the Gaussian model for the case of the whole lattice stored into local memory.

4.3 Gaussian Model

All the previous considerations are valid for Gaussian model fixing $V = 4$ and using all the $128/V = 32$-bit of each scalar data-word to represent spins and couplings as 32-bit floating point numbers. An estimate of spin update time when using local memory is given in Table 4.26, while Table 4.27 lists the ranges of L that balance data transfer and computation times.

Finally, Table 4.28 shows an estimate of spin update time when keeping lattice data structures in main memory.

The estimate of the system update time applied to the balance equation shows that in all cases at most $C = 3$ can be used. The code that implements the Gaussian model is faster than the Binomial code for $V = 4$, so the bottleneck represented by the concurrent access to main memory is more critical. However, the Binomial model takes advantage from higher SIMD-granularities, that allow to obtain both a faster computational code and to significatively reduce the amount of data transfers.

Gaussian Model			
Off-chip Memory			
L	$\tau_S(V)$	$\tau_R(V)$	Overhead (%)
16	4.5	5.4	20
24	4.2	4.9	17
32	4.1	4.6	12
40	4.2	4.7	12
48	4.2	4.6	10
56	4.3	4.7	9
64	4.0	5.2	30
72	3.9	4.9	26
80	3.9	4.9	26

Table 4.28: Static and run-time estimates of the spin Update Time $\tau(V)$ for the Gaussian Model when lattice data is stored into main memory.

4.4 Conclusions

In this chapter we have described the development and analyzed the performance of the spin update procedure for a wide range of SIMD-granularities V and for different lattice linear sizes L, for both local and main memory strategies. The run-time estimates of system spin update times $\tau_R(V)$ have been applied to the balance equations introduced in chapter 3 to evaluate in which cases data transfers do not dominate the execution time of the program, in order to deteremine which parameters make it possible to efficiently use all the availabel cores.

In the next chapter the run-time performances of the programs, including data exchange between cores and main memory, will be illustrated, discussed an compared to the estimates presented in this chapter.

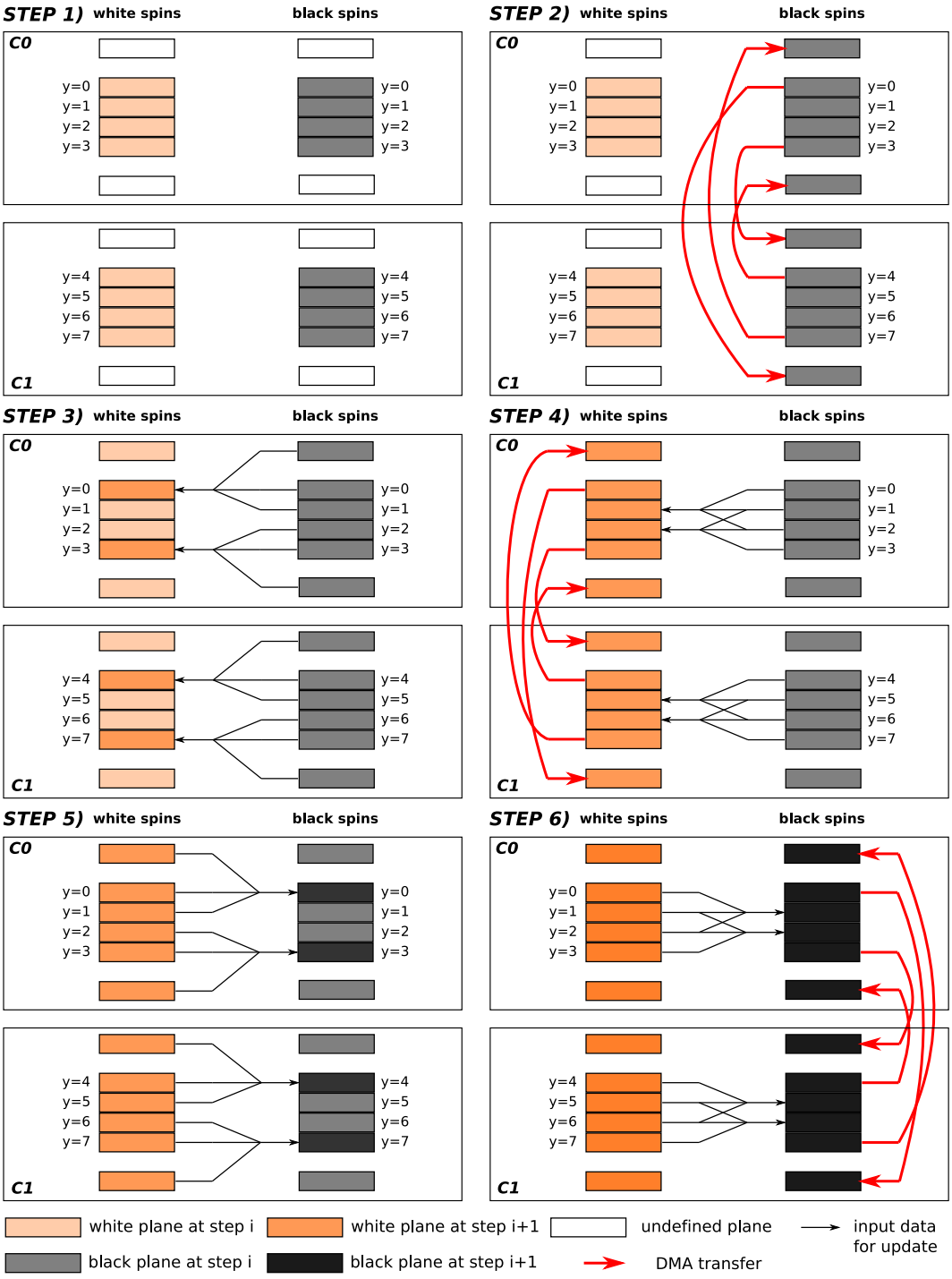

Figure 4.14: The basic step needed to perform a complete Monte Carlo update on a whole lattice stored in local memories when $L = 8$ and $C = 2$, and the range of lattice sizes that can be managed with a given SIMD-granularity V.

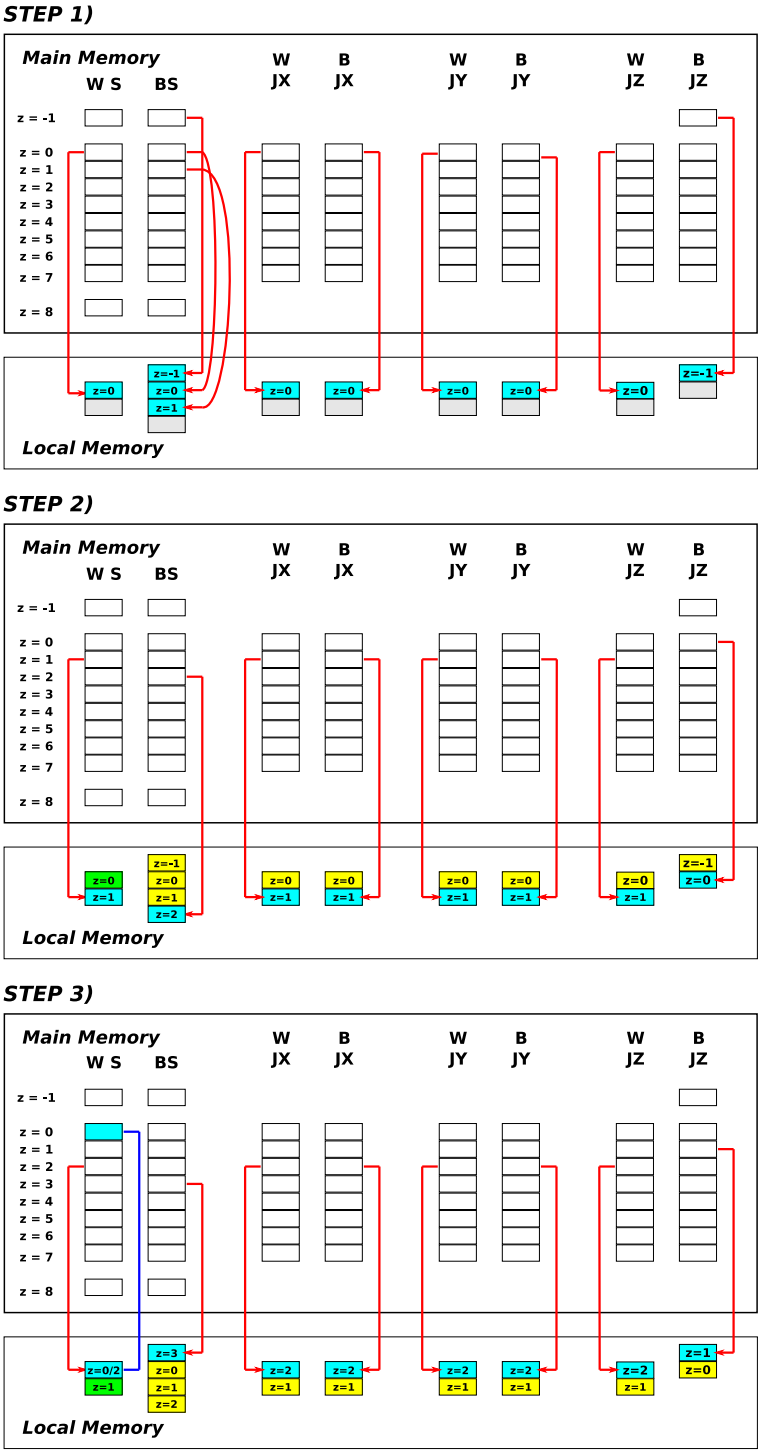

Figure 4.15: Steps 1, 2 and 3 of the update of the half-planes of a core. In the first step they have to be loaded from main to local memory, and no computations are performed.

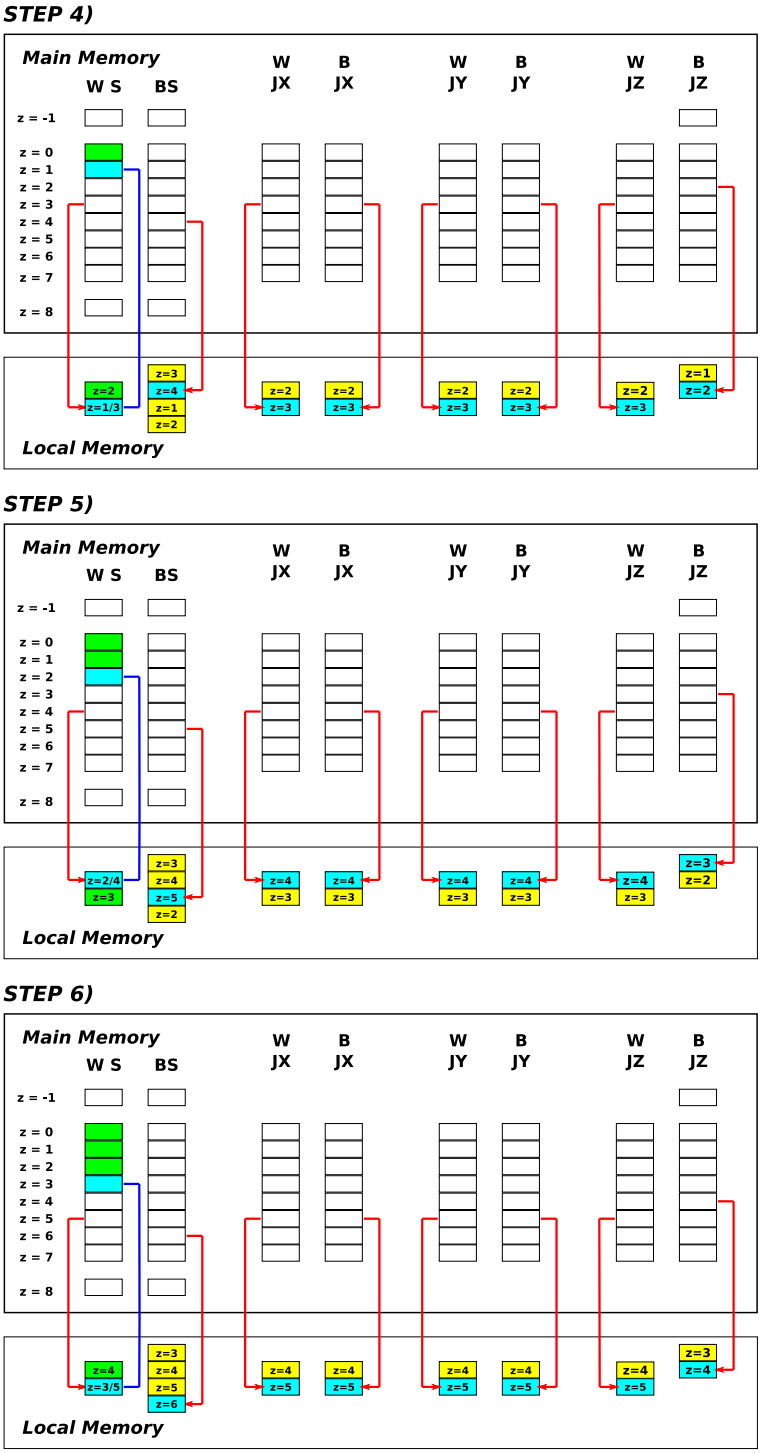

Figure 4.16: Steps 4,5 and 6 of the update of the half-planes of a core. This is the most common case: while a half-plane is updated, 9 half-planes are concurrently exchanged with main memory.

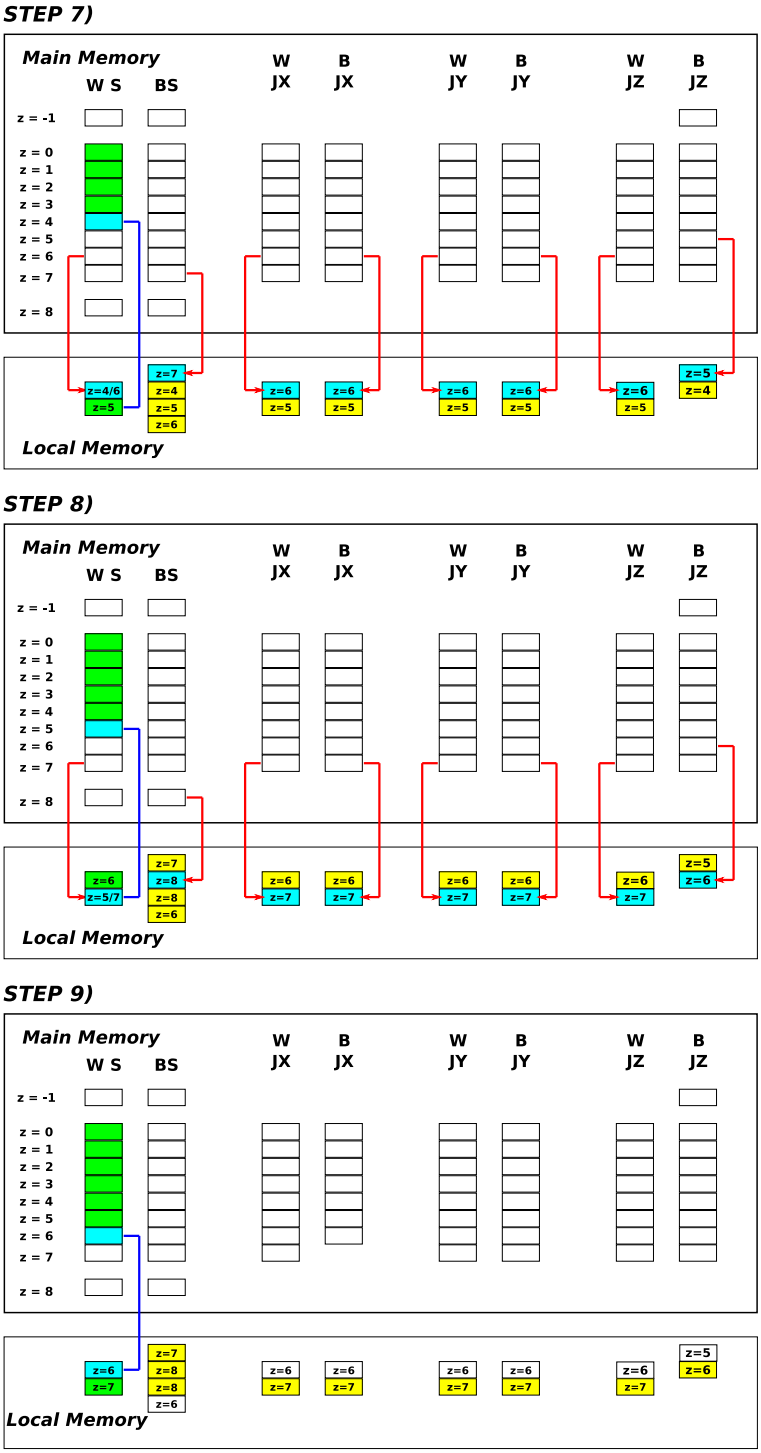

Figure 4.17: Steps 4,5 and 6 of the update of the half-planes of a core. Data transfers and computations are still concurrent.

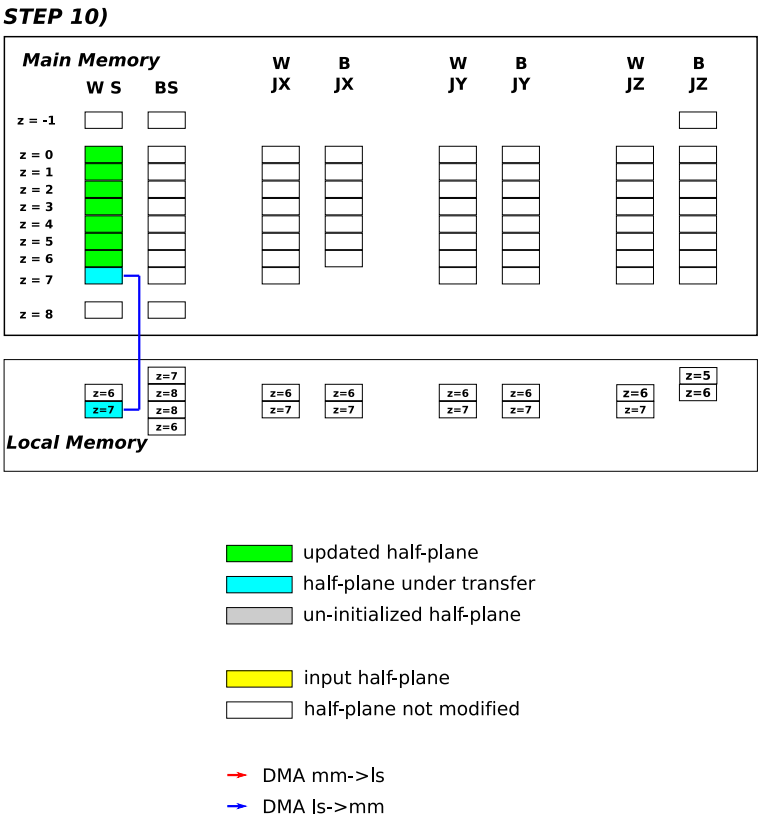

Figure 4.18: The final step of the update of the half-planes of a core. All the half planes have been updated and the last one is stored back to main memory.

Chapter 5

Spin Glasses Performance on CBE

In this chapter the run-time performance of the spin glass programs will be measured and compared with the behavior predicted in the previous chapters. For tests we a single IBM BladeCenter QS22, which has two directly connected IBM PowerXCell 8i processors. In this configuration, it is possible to use up to 16 cores, although each SPE can exchange data at full speed only with the SPEs and the main memory of the same processor.

We have made extensive performance tests of most allowed combinations of linear lattice size L, SIMD-granularity V and number of cores C. We expect large variation in overall efficiency as we move in this parameter space, due to the strong dependency of the balance with the parameters V and C, as seen in balance equations, that reflects the constraints of the architecture.

When not specified, the programs were compiled using the `GCC` toolchain provided with IBM SDK for Multicore Acceleration Version 3.1.

5.1 Binary Model

In this section we will measure the run-time performance of both local and main memory version of the program that implements the Binary model. In particular we will determine the impact of SIMD-granularity on systems spin update time $\tau(V)$.

Note that all the tests were performed with al lest two cores ($C \geq 2$), because using only one core it is not considered an interesting case, as it does not expose teh concurrency between cores.

5.1.1 Local Memory

Let us analyze the performance when lattice data is stored only in local memories. The values reported in Table 5.1 are measured on PPE side, and are comprensive of synchronization between cores and of data transfers that do not overlap with

computations (i.e. when the balance is not guaranteed). For each linear lattice length L, the performance obtained with all the possible combinations of SIMD-granularity V and number of cores C are shown. some tuples (L, V, C) may be missing due to the two constraints of the program: at least two planes have to be assigned to each core, and L has to be a multiple of C (in order to assign homogeneously distribute the plane among cores). Note that the table also takes into consideration tuples (V, C) for which we previously predicted that data transfer should dominate computation time.

On-Chip Memory					
System Spin Update Time (ns/spin)					
C	V=4	V=8	V=16	V=32	V=64
L = 16					
2	**3.753**	**2.561**	×	×	×
4	**2.051**	**1.404**	×	×	×
8	**1.135**	**0.834**	×	×	×
16	**1.480**	**1.169**	×	×	×
L = 32					
2	m	**2.072**	**1.550**	×	×
4	**1.590**	**1.058**	**0.787**	×	×
8	**0.807**	**0.542**	**0.406**	×	×
16	**0.499**	**0.357**	**0.263**	×	×
L = 64					
4	m	m	m	**0.576**	×
8	m	m	**0.348**	**0.289**	×
16	m	**0.244**	**0.179**	**0.149**	×
L = 128					
16	m	m	m	m	**0.120**

Table 5.1: The system spin update time $\tau(V)$ of several tuples (L, V, C) when storing lattice data in local memories. The tuples that required too much local store space are labeled with m, while the invalid combination of L and V are indicated with the symbol $\times$. The invalid combinations of L and C are not reported.

To analyze more in more detail the behavior of performance, Table 5.2 compares the system spin update times obtained with eight cores (usually the best option when using a single processor) to those that would be obtained using only one core, while Table 5.3 depicts the slow-down in respect to the system spin update time predicted in the previous chapter.

In an ideal case we would assume that system spin update time is inversely proportional to C, if data transfers and synchronization are not a bottleneck. Although

C	L	V=4	V=8	V=16	V=32
8	16	9.080	6.672		
8	24	6.752			
8	32	**6.456**	**4.336**	**3.248**	
8	40	**6.272**			
8	48		**3.856**		
8	64			**2.784**	**2.312**

Table 5.2: The system spin update time for a single core, computed as $\tau(V) \times C$, when using all the SPEs of a single CBE processor. The values in bold are relative to the tuples (L, V, C) in which we expect that the global time in dominated by computation, as calculated from balance equations.

L	C	V=4	V=8	V=16	V=32
16	8	1.65	1.76		
24	8	1.23			
32	8	**1.18**	**1.4**	**1.16**	
40	8	**1.14**			
48	8		**1.0**		
64	8			**1.0**	**1.2**

Table 5.3: An estimation of the overhead of the run-time measured system spin update time $\tau(V)$ with respect of the static one determined in Chapter 4. The values in bold are relative to the tuples (L, V, C) that, according to balance equations, should be dominated by computation time and, as a consequence, a very little overhead in terms of data transfers and synchronization, when using all the available SPEs. The measured system spin update time confirms our prediction, as in most cases the overhead is less than 20%.

for $L = 16$ the performance is different from that expected, in the other cases the overhead is quite limited, usually under 20%. The behavior is coherent with our predictions: Table 4.24 of Chapter 4 indicates that for $V = 4, 8, 16$ the balance is assured if $L \geq 32$, confirmed by the low overheads of the run-time tests.

The behavior of the system spin update time $\tau(V)$ as function of the number of cores C is shown Figure 5.1. As we can see, except for the smallest lattice $L = 16$, adding cores always gives better performance, and it is also possible to go faster using a dual-CBE configuration.

In Figure 5.2 is emphasized the difference between the speed-up obtained adding more cores and the ideal speed-up. As we can see, in most cases the difference is under the 50%.

The difference into respect of the ideal scaling when the number of cores C is fixed and the SIMD-granularity V varies is illustrated in Figure 5.4. The results

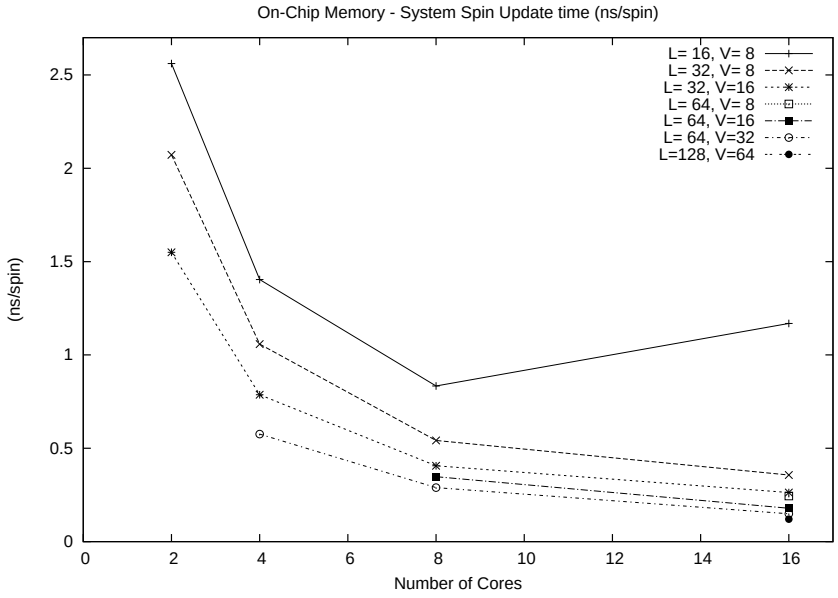

Figure 5.1: System Spin update time $\tau(V)$ in function of the number of cores C for several linear lattice sizes L and SIMD-granularities V.

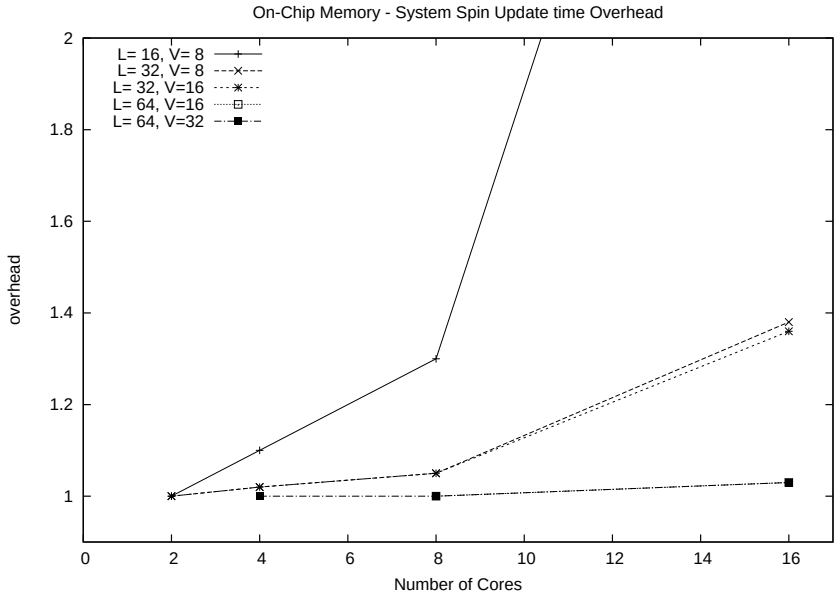

Figure 5.2: The overhead of the system Spin update time $\tau(V)$ in to respect to the ideal one (assuming perfect scaling) in function of the number of cores C for several linear lattice sizes L and SIMD-granularities V.

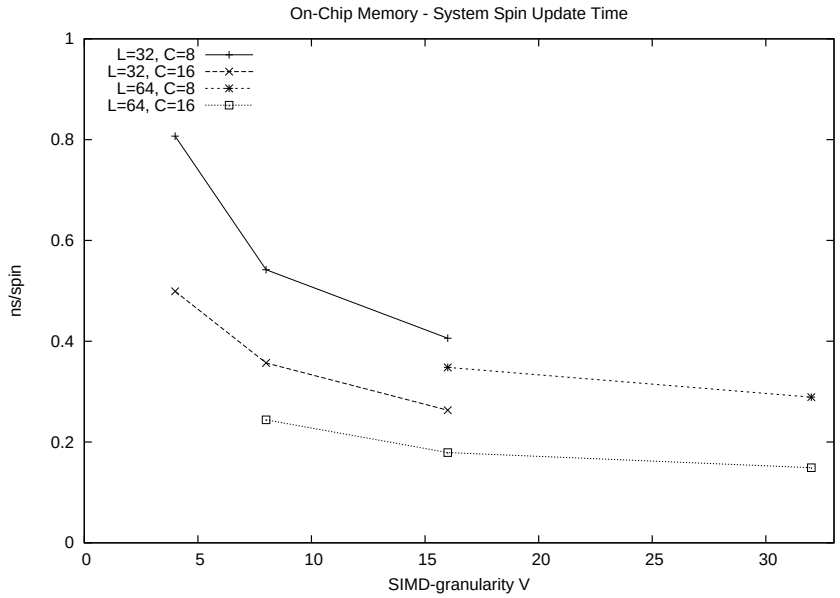

Figure 5.3: The system Spin update time $\tau(V)$ as a function of SIMD-granularity V for several linear lattice sizes L and number of cores C.

are not too different from what expected, because for high granularities the random number generation time is the dominant fraction, so there are little improvements when increasing granularites, but it is nonetheless better to use the highest possibile SIMD-granularity.

To individuate the origin of the overheads, the time dedicated by a SPE to pure computation has been measured at run-time with the decrementer register. The results are shown in Table 5.4. Note that the fraction of time dedicated to computation is better for larger lattices, because the basic unit of data transmission, the half-plane, is bigger and allows a better usage of the EIB (see Table 4.23 from Chapter 4).

5.1.2 Main Memory Version

When lattice data is stored in main memory, for a given SIMD-granularity V we expect that the main memory bandwidth to be large enough to feed a certain number of cores C, as seen in Table 4.25. Given that the theoric bandwith is achievable only with lar ge lattices, we expect worse system spin update time for smaller lattices ($L \leq 32$), according to Table 4.23. However, in those cases it is preferable to use only local memories, the allow better performances.

The system spin update times for many of the possible tuples (L, C, V) are shown in Table 5.5.

The system spin update time $\tau(V)$ as a function of the number of cores C is

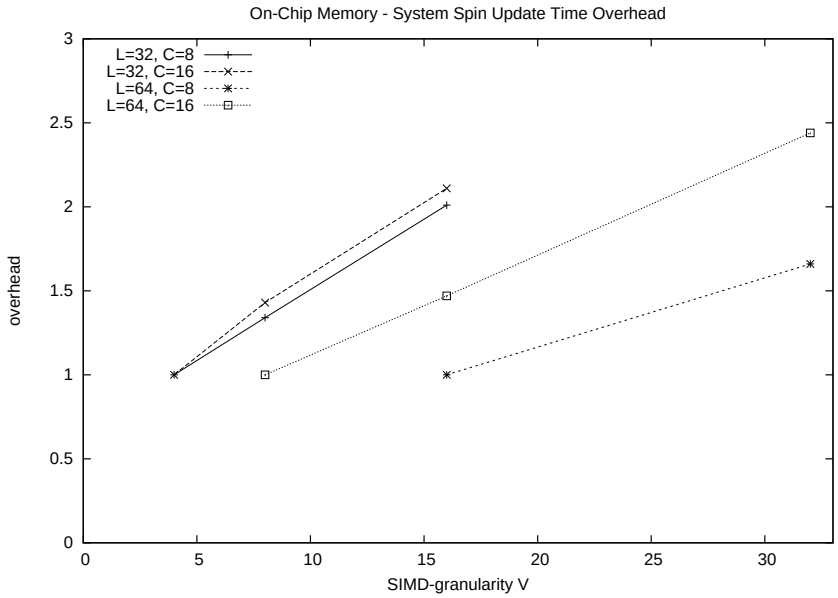

Figure 5.4: The overhead of the system Spin update time $\tau(V)$ in to respect of the ideal one (assuming perfect scaling) as a function of SIMD-granularity V for several linear lattice sizes L and number of cores C.

shown in Figure 5.5 for three representative lattices of linear size $L = 64, 96, 128$. As we can see, the best performance is achieved when using all the eight cores of a CBE, while the two processors configuration not only does not grant any benefit, but due to memory bandwidth saturation leads to worse performances. The overheads with respect of ideal scaling is shown in Figures 5.6, where SIMD-granularity is fixed to $V = 16$ and three different lattices are taken into consideration. In Figure 5.7. only $L = 128$ is taken into account, but SIMD-granularity is $V = 64$. As we can see, a high granularity highly reduce the overhead, because it implies a much smaller set of data to be exchanged between local and main memories.

Note that for small lattices the performances are not comparable to those of the local memory implementation, while for $L \geq 64$ the overhead is much smaller. Tables 5.6 and 5.6 show the difference the fraction of the global time dedicated to computation, measured on a SPE with the decrementer register. As expected, the fraction is smaller when using a large number of cores, due to the conflicting memory accesses and to synchronization. However, for $L \geq 4$, the time dedicated to computation is always above the 50% when using eight cores. Moreover, for large lattice the computation fraction is proportional to SIMD-granularity V, because less data has to be transferred between main and local memories. Note also that the use of two processors grant none or little improvements. This is expected for many reasons: first of all, the SPEs of the second CBE have to use a non-local memory, with the bottleneck of the connection between the two processors. Also,

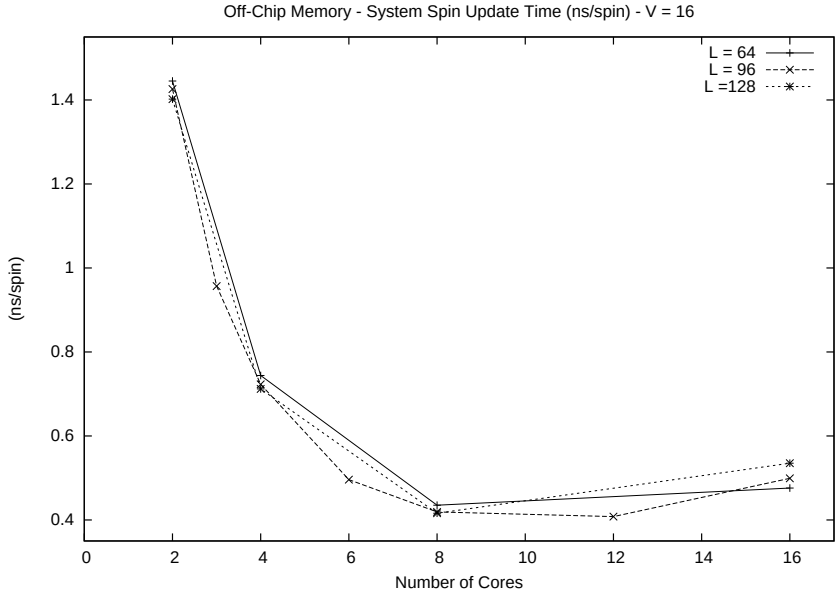

Figure 5.5: The system spin update time $\tau(V)$ as a function of the number of cores C with fixed SIMD-granularity $V = 16$ and for several lattices sizes L.

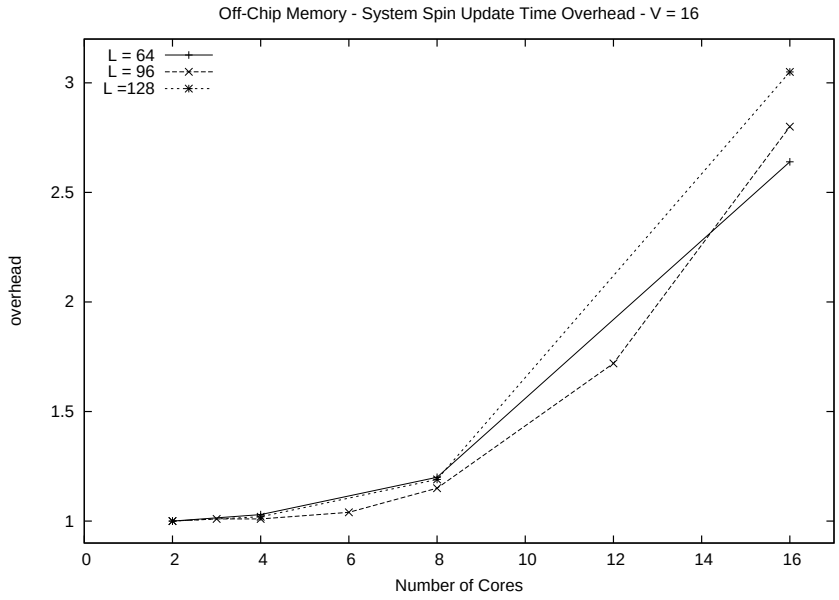

Figure 5.6: The system spin update time overhead as a function of the number of cores C with fixed SIMD-granularity $V = 16$ and for several lattices sizes L.

On-Chip Memory					
Computation %					
C	V=4	V=8	V=16	V=32	V=64
L = 16					
2	**88.36**	**83.60**			
4	**78.83**	**71.80**			
8	**65.74**	**56.36**			
16	**50.08**	**44.00**			
L = 32					
2	**97.28**	**96.50**			
4	**96.49**	**94.53**	**92.90**		
8	**92.67**	**89.68**	**86.63**		
16	**74.32**	**69.20**	**63.51**		
L = 64					
4				**98.78**	
8			**97.94**	**97.53**	
16		**95.77**	**94.29**	**93.15**	
L = 128					
16					**97.02**

Table 5.4: The percentage of the global time of a SPE dedicated to the execution of the computational core. The code is the same used in Chapter 4 to estimate the ideal system spin update time $\tau(V)$, so a high fraction implies that the performance of the whole program are near to the ideal one.

using more SPEs increases the pressure on main memory, and the balance equations have already shown that at most 9 SPEs can be efficiently used (Table 4.25).

Table 5.7 shows the fraction of global time used to wait the end of data transfers (also called *stall time*). This time is determined by the data transfers that cannot complete in time or that are only partially overlapped to computations. The first case is relateted to the possible unbalance of compututations and data transfers. If the bandwidth is not efficiently exploited or there is too much data too transfers due to the algorithmic parameters, after the end of computations it is necessary to wait the end of all DMA transfers, thus introducing an overhead. The second case, instead, refers to the start-up sequence of each Monte Carlo iteration, when the first three half-planes are loaded into local memories: until all the required data is available, computations cannot begin. Moreover, after the update of the last half-plane, it is necessary to wait the end of the transfer before to proceed to the other color or to the next Monte Carlo iteration.

The Table shows that for high SIMD-granularities and large lattices the stall time

Off-Chip Memory					
System Spin Update Time (ns/spin)					
C	V=4	V=8	V=16	V=32	V=64
L = 32					
2	**3.630**	**2.365**	**1.825**	×	×
4	**1.953**	**1.283**	**1.042**	×	×
8	**1.759**	**1.052**	**0.673**	×	×
16	**2.047**	**0.958**	**0.585**	×	×
L = 64					
2	**3.480**	**2.096**	**1.445**	**1.180**	×
4	**1.968**	**1.147**	**0.744**	**0.601**	×
8	**1.683**	**0.874**	**0.435**	**0.323**	×
16	**1.866**	**1.044**	**0.476**	**0.230**	×
L = 96					
2	m	**2.079**	**1.426**	×	×
3	m	**1.407**	**0.957**	×	×
4	m	**1.063**	**0.723**	×	×
6	m	**0.821**	**0.496**	×	×
8	m	**0.799**	**0.419**	×	×
12	m	**0.798**	**0.408**	×	×
16	m	**1.028**	**0.499**	×	×
L = 128					
2	m	m	**1.402**	**1.101**	**0.957**
4	m	m	**0.712**	**0.557**	**0.482**
8	m	m	**0.416**	**0.288**	**0.246**
16	m	m	**0.535**	**0.253**	**0.135**

Table 5.5: The system spin update time $\tau(V)$ when the lattice data set is stored in main memories, for several tuples (L, V, C).

is less than 10%, when using at most eight cores. Finally, in Table 5.8 the fractions of the global time, measured on the SPE side, dedicated to synchronization between cores, are illustrated. They are always less than 10% and for $L \geq 80$ they are always less than 1%, so they do not represent a performance bottleneck.

5.1.3 Performance Comparison

For spin glasses we have two important terms of comparisons: a program based on the same multispin coding principles compiled and executed on high-end commodity

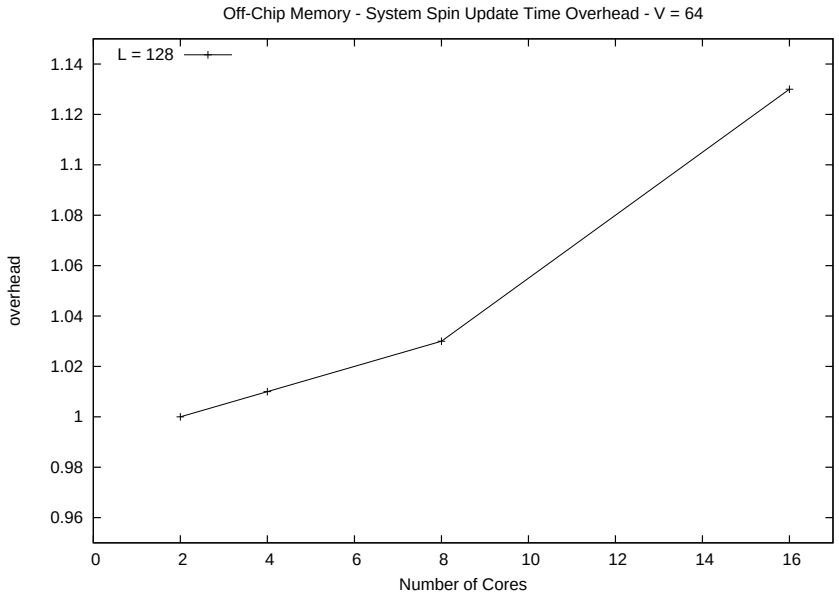

Figure 5.7: The system spin update time overhead as a function of the number of cores C with fixed SIMD-granularity $V = 64$ and lattice linear size $L = 128$.

processors, and Janus, a FPGA-based massively parallel machine developed specifically for spin glass simulations. The first one allows to make a comparison with widely available architectures of a comparable price, to evaluate the value of CBE in this context, and to a special pourpose machine, that today is able to achieve state-of-the-art performance in this specific application.

Let us start with the binary model. On an Intel Xeon 3.0 GHz processor a Metropolis Monte Carlo routine implementing synchronous multi-spin coding and processing 64 spins of a single system of linear size 64 parallel performs at 4.8 ns/spin (system spin update time). Note that lattice size does not impact performance as long as a large cache memory is available and that the program runs on a single core and does not explicitly make use of SIMD instructions.

Note that when $C = 16$ and $V = 64$ the number of spins of the same system concurrently update is 1024, that is comparable to those of Janus. However, due to random number generation and to data access (that requires many instructions for both addressing and for load/stores), the high clock speed of CBE it is not sufficient to achieve the performance of a dedicated architecture.

Comparing the Xeon performance with the one given in the previous chapter for the CBE, we conclude that one CBE is approximately 5-19 times faster than one Xeon processor when considering the system update times, depending on lattice size. The dual-CBE configuration are directly usable without changes in the programs, and as a result it is possible to go up to 40 times faster than a single Xeon core. However, the benefits of using two processors are significative only for lattices with

Off-Chip Memory					
Computation %					
C	V=4	V=8	V=16	V=32	V=64
L = 32					
2	**86.80**	**82.14**	**77.28**		
4	**80.09**	**75.40**	**67.86**		
8	**44.95**	**48.98**	**52.42**		
16	**19.48**	**26.60**	**30.43**		
L = 64					
2	**93.93**	**92.79**	**93.36**	**92.43**	
4	**82.74**	**84.16**	**90.55**	**91.04**	
8	**47.66**	**57.24**	**78.27**	**83.88**	
16	**18.16**	**23.61**	**35.90**	**60.76**	
L = 96					
2		**66.89**	**94.99**		
3		**94.63**	**94.30**		
4		**93.40**	**93.74**		
6		**81.35**	**91.19**		
8		**62.48**	**81.82**		
12		**41.82**	**55.51**		
16		**24.30**	**34.37**		
L = 128					
2			**94.10**	**91.50**	**87.66**
4			**96.03**	**96.37**	**96.76**
8			**82.14**	**93.13**	**94.55**
16			**31.93**	**53.35**	**86.40**

Table 5.6: The percentage of the global time of a SPE dedicated to the execution of the computational core for several tuples (L, V, C).

linear sizes such that high SIMD-granularities ($V = 32$ and $V = 64$) can be used, because they minimize the volume of data transfers.

On the other hand, the dedicated Janus machines outperforms the CBE implementation approximately by a factor 15 - 53, depending on the lattice size, when using a single processor, and by a factor, 8 - 53 for dual-processor configurations.

Off-Chip Memory					
Stall %					
C	V=4	V=8	V=16	V=32	V=64
L = 32					
2	**5.01**	**6.35**	**8.71**		
4	**10.60**	**11.27**	**17.04**		
8	**44.23**	**41.07**	**32.18**		
16	**44.40**	**40.86**	**29.24**		
L = 64					
2	**3.99**	**3.63**	**1.99**	**1.963**	
4	**14.82**	**12.04**	**3.94**	**3.384**	
8	**49.16**	**39.04**	**12.74**	**8.448**	
16	**53.40**	**49.40**	**30.36**	**14.986**	
L = 96					
2		**16.59**	**1.86**		
3		**3.22**	**2.56**		
4		**4.44**	**3.09**		
6		**16.12**	**5.23**		
8		**33.61**	**11.98**		
12		**31.85**	**15.30**		
16		**45.90**	**35.55**		
L = 128					
2			**3.02**	**3.27**	**3.54**
4			**2.62**	**1.87**	**1.29**
8			**14.19**	**4.76**	**3.25**
16			**38.67**	**18.58**	**4.85**

Table 5.7: The percentage of the *stall time* in to respect of the global time of a SPE. It measures the time in which a SPE is not performing useful computations but it is only waiting the and of DMA transfers.

5.2 Gaussian Model

Finally, let's consider the Gaussian model. In this case the SIMD-granularity is fixed to $V = 4$, so we can expect overheads related to data transfers. The results for the implementation that uses only local memory are illustrated in Table 5.10. The spin update time achieved by a single core is extrapolated from the global time, and compared with the theoric value estimated in the previous chapter. As we can see, the difference between the real and the ideal cases are quite small. Although the fixed SIMD-granularity does not allow to reduce the amount of transferred data

Off-Chip Memory					
Synchronization %					
C	V=4	V=8	V=16	V=32	V=64
L = 32					
2	**0.61**	**1.84**	**1.21**		
4	**1.94**	**4.03**	**2.95**		
8	**3.59**	**2.15**	**4.35**		
16	**3.54**	**3.97**	**4.92**		
L = 64					
2	**0.15**	**0.32**	**0.29**	**0.53**	
4	**0.67**	**0.23**	**0.36**	**0.47**	
8	**0.23**	**0.39**	**1.05**	**2.12**	
16	**0.37**	**0.75**	**1.47**	**2.25**	
L = 96					
2		**0.59**	**0.59**		
3		**0.17**	**0.17**		
4		**0.22**	**0.22**		
6		**0.28**	**0.28**		
8		**0.26**	**0.26**		
12		**0.14**	**0.14**		
16		**0.13**	**0.13**		
L = 128					
2			**0.10**	**0.19**	**0.33**
4			**0.14**	**0.18**	**0.07**
8			**0.221**	**0.12**	**0.15**
16			**0.173**	**0.21**	**0.38**

Table 5.8: The percentage of the global time of a SPE dedicated to the synchronization with the other cores. No Computations or data transfers are performed during this fraction of time.

using a higher value of V, the time required to perform computations is long enough to effectively mask data transfers. Figure 5.8 shows the behaviour of the spin update time for several lattice manageable with the local memory version of the program. On a Xeon processor, using a single core, the best achieved spin update time is 65 ns, so when using local memories the CBE implementation is 64-100 times faster.

To manage larger lattices it is necessary to use the main memory. The performance achieved in this case are illustrated in Table 5.2. The best spin update time is 1.7 ns/spin, which make this program 38 times faster than the Xeon counterpart. Note that with $C = 4$ the main memory bandwith is saturated, because there is no

System Spin Update Time								
L	C	V	Memory Type	CBE $\tau(V)$ (ns/spin)	Xeon $\tau(V)$ (ns/spin)	CBE vs. Xeon speed-up	Janus $\tau(V)$ (ns/spin)	CBE vs. Janus speed-up
16	8	8	on-chip	0.834	4.8	**5.76**	0.016	**0.019**
16	16	8	on-chip	1.169	4.8	**4.10**	0.016	**0.014**
24	8	4	on-chip	0.844	4.8	**5.69**	0.016	**0.019**
24	12	4	on-chip	0.738	4.8	**6.50**	0.016	**0.022**
32	8	16	on-chip	0.406	4.8	**11.82**	0.016	**0.039**
32	16	16	on-chip	0.263	4.8	**18.25**	0.016	**0.061**
40	8	4	on-chip	0.784	4.8	**6.12**	0.016	**0.020**
40	10	4	on-chip	0.641	4.8	**7.49**	0.016	**0.025**
48	8	8	on-chip	0.482	4.8	**9.96**	0.016	**0.033**
48	16	8	on-chip	0.252	4.8	**19.05**	0.016	**0.064**
64	8	32	on-chip	0.289	4.8	**16.60**	0.016	**0.055**
64	16	32	on-chip	0.149	4.8	**32.22**	0.016	**0.107**
80	8	8	off-chip	0.823	4.8	**5.83**	0.016	**0.019**
80	10	8	off-chip	0.797	4.8	**6.02**	0.016	**0.020**
96	8	16	off-chip	0.419	4.8	**11.46**	0.016	**0.038**
92	12	16	off-chip	0.408	4.8	**11.77**	0.016	**0.039**
112	8	8	off-chip	0.800	4.8	**6.00**	0.016	**0.020**
112	14	8	off-chip	0.907	4.8	**5.30**	0.016	**0.018**
128	8	64	off-chip	0.246	4.8	**19.12**	0.016	**0.065**
128	16	64	on-chip	0.120	4.8	**40.00**	0.016	**0.130**

Table 5.9: The comparison between the system spin update time $\tau(V)$ achieved with CBE, Xeon and Janus. For each linear lattice size L are reported the results obtained using one and two CBE processors. For the cases in which both local and main memory versions where available, we reported only the fastest one.

Gaussian Model			
On-Chip Memory			
System Spin Update Time			
C	(ns/spin)	overhead	speed-up
L = 16			
2	**3.131**	1.0	
4	**1.689**	1.1	1.85
8	**1.003**	1.3	3.12
16	**1.341**	3.4	2.34
L = 24			
3	**2.295**		
4	**1.722**		1.33
6	**1.180**		1.95
8	**0.890**		2.58
12	**0.769**		2.98
L = 32			
4	**1.269**	1.02	
8	**0.648**	1.04	1.96
16	**0.422**	1.36	3.02
L = 40			
5	**1.480**	0.99	
8	**0.931**	1.00	1.59
10	**0.768**	1.03	1.93
L = 48			
12	**0.531**	1.03	
16	**0.402**	1.04	1.32

Table 5.10: The spin update time $\tau(V)$ for the Gaussian model, when data set is distributed among the local stores of the SPEs. The overhead in to respect of the ideal performance estimated in Chapter 4 is also illustrated. In the last column we report the speed-up achieved incrementing the number of cores, referred to the performance obtained using the minimum possible number of cores. Note that the cases in which the size of the local memories was too small to store the whole lattice are not reported.

(or a very little) improvement when using more cores.

The scaling of the spin update time $\tau(V)$ as a function of the number of cores C is shown in Figure 5.9. The best performance are obtained with all the eight core of a CBE, while the two-processors configurations are not very useful.

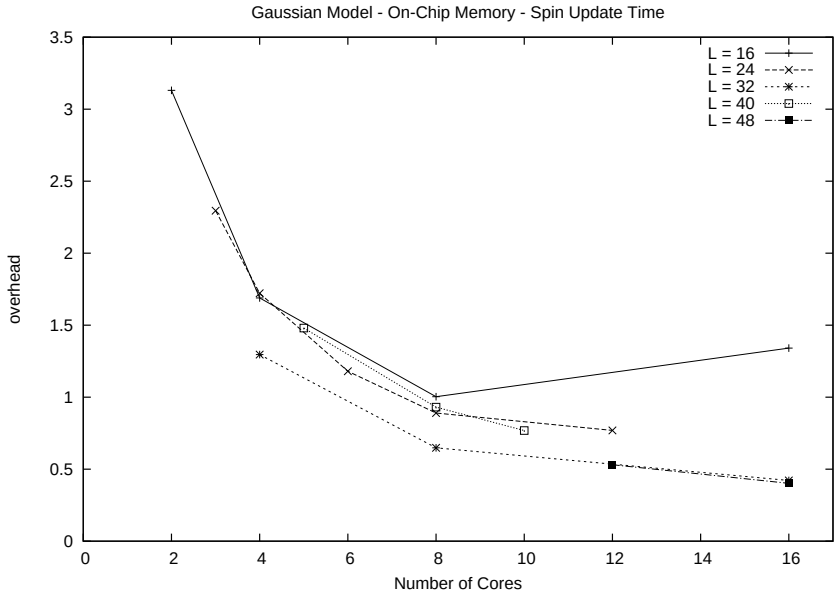

Figure 5.8: System Spin update time $\tau(V)$ as a function of the number of cores C for several linear lattice sizes L.

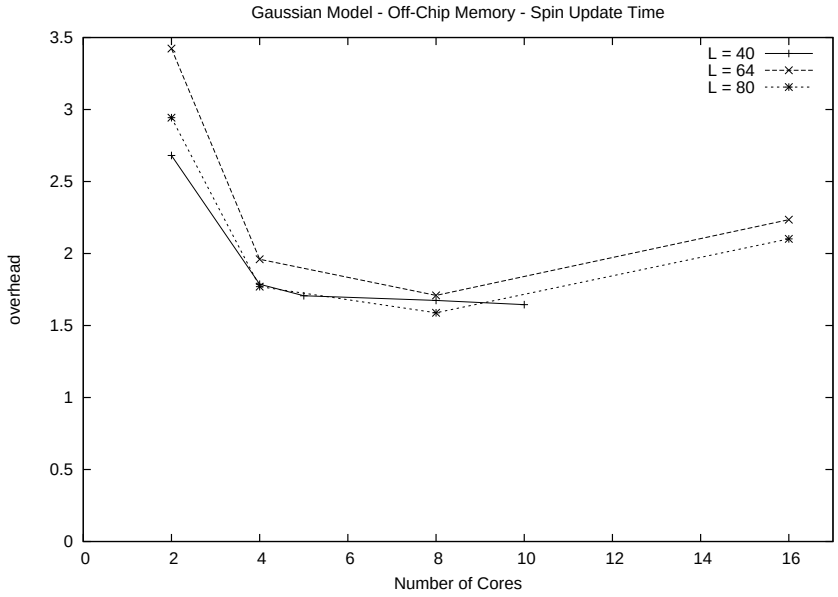

Figure 5.9: System Spin update time $\tau(V)$ as a function of the number of cores C for several linear lattice sizes L.

Gaussian Model			
Off-Chip Memory			
System Spin Update Time			
C	(ns/spin)	overhead	speed-up
L = 40			
2	**2.681**	1.13	
4	**1.787**	1.51	1.500
5	**1.707**	1.80	1.571
8	**1.674**	2.83	1.602
10	**1.645**	3.48	1.630
L = 48			
2	**2.604**	1.13	
3	**1.937**	1.27	1.344
4	**1.746**	1.52	1.491
6	**1.660**	2.17	1.569
8	**1.659**	2.89	1.570
12	**1.654**	4.32	1.574
16	**2.103**	7.33	1.238
L = 64			
2	**3.423**	1.33	
4	**1.960**	1.52	1.746
8	**1.709**	2.65	2.003
16	**2.235**	6.93	1.532
L = 80			
2	**2.943**	1.20	
4	**1.770**	1.44	1.663
8	**1.588**	2.59	1.853
16	**2.102**	6.85	1.400

The spin update time $\tau(V)$ for the Gauss model, when data set is stored in main memory. The overhead into respect of the ideal performance and the speed-up obtained using more cores are also reported.

5.3 Conclusion

In this chapter we have shown the measurements of the performance achieved executing the programs on a dual-processor CBE system, for several combinations of the linear lattice size L, the number of cores C and the SIMD-granularity V. The results confirms what we predicted with the proposed balance equations. If the data set fits the local memories, we can expect to efficiently use all the available core,

while if data have to be exchanged with main memory the SIMD-granularity V become a critical parameters, as higher granularities allows to reduce the amount of data that have to be transferred, so that in many cases it is still possible to balance computations ad data transfers. Incidentally, an high SIMD-granularity also allow to obtain better performance in the update of the spins of a single spins, that is one of our main targets.

Finally, the Gaussian model is constrained to a SIMD-granularity that does not fit very well the properties of the hardware, so in the case that main memory has to be used there is a notable performance degradation.

Conclusions

Multi-core architectures delivers into a single chip tens or hundreds Gflops of peak computing performance, with high power dissipation efficiency, and they make available computational power previously available only on high-end multi-processor systems. As a consequence, multi-core processors are very interesting in the perspective of scientific programming. However, the sustained performance and the programming efforts needed to exploit their power have to be carefully evaluated.

As a test bench for the study of multi-core processors, in my Ph.D. thesis I have considered Monte Carlo simulations of spin glasses, that is a very computing demanding scientific application, and I have optimized it for Cell Broadband Engine, that is one of the first available and most interesting multi-core processors. In particular, the target was to determine the sustained performance that can be achieved with this class of processors, to analyze the issues and the constraints that have to be faced in order to exploit their capabilities, and to individuate a set of techniques and strategies that should be adopted when programming multi-core processors.

In the context of spin glass simulations, the target was to achieve the highest possible speed for the update of a single system with linear lattice size in the range $L = 16 \cdots 128$. We have defined a performance metric for the problem, the system spin update time $\tau(V)$, that is useful to evaluate the achieved performance into respect of two available terms of comparison. The first one is a program based on the same multispin coding principles described in Chapter 2, but compiled and executed on a high-end commodity processor, in order to evaluate the benefits of CBE into respect of traditional architectures. The second term of comparison is Janus, a FPGA-base massively parallel machine designed specifically for spin glasses simulations and that today is able to achieve state-of-the-art performance in spin glass simulations.

When considering Binary model, one CBE processor is approximately $5 \cdots 19$ times (depending on lattice size) faster than a single core of a Xeon processor running at 3.0 GHz, while dual-CBE configurations allow to go up to 40 times faster than a Xeon core. On the other hand, the dedicated Janus machine outperforms the CBE implementation approximately by a factor $15 \cdots 53$, depending on the lattice size, when using a single CBE processor, and by a factor $8 \cdots 53$ for dual-processor configurations.

For the Gaussian model, the CBE processor is $64 \cdots 100$ times faster than a Xeon core if the lattice can be stored in local memories, while it is 38 times faster if main memory has to be used. Gaussian model is very interesting because it is floating-point arithmetic intensive, so it is the type of applications the CBE was designed for. Although in this case data transfers have an heavier impact on global time than in the case of Binary model, the performance improvement into respect of traditional architectures is notable. As a consequence, the CBE processor should be well suited for general- purpose massively-parallel machines, as high-performance floating-point computations are an essential key of many scientific applications.

Given the obtained results, the performance/price ratio of CBE is very interesting for spin glasses, in particular when considering the PlayStation 3, that allow to buy a complete system (although with only six cores) for less than 400 $. We expect that in the near future very large simulations in spin glass will still be dominated by dedicated machines, but smaller simulation campaigns will benefit heavily from the use of CBE based machines, as they allow to build small clusters of $16 \cdots 64$ machines with a very convenient performance/price ratio.

The first issue in optimizing spin glass simulations for CBE was to define a data layout and a data transfer scheme that allows the distribution of the data set among local an main memories. Local memories have a key role into achieving high performance, but they are able to manage only a limited range of lattice sizes, so we identified two main cases: (i) data set can be stored using only local memories or (ii) main memory is required. In both cases we assume that at a given instant each cores has in its local memory enough data to update a partition of the lattice. Each core updates its partition as fast as possible, while MFCs independently prefetch the data required to perform the next step of the algorithm (that is the update of either the same or another sublattice). Our target was to overlap data transfer and computation, in order to exploit the computational power of all the available cores. When only local memories are used, a potential bottleneck is inter-core data transfers, while if main memory is used the concurrent accesses performed by cores can saturate the bandwidth.

To evaluate the balance between computation and data transfer we defined simple models that take into account both algorithmic parameters (linear lattice size L, SIMD-granularity V and the system spin update time $\tau(V)$) and architectural parameters (the number of cores C and the bandwidths of inter-core communication and of main memory). The models allow to determine the combination of algorithmic parameters for which the global time is not dominated by data transfer, so that the computational power of all the cores is exploited. The models neglect details such as synchronizations and latencies and it assumes ideal bandwiths, so the results extrapolated from them are an upper bound of what can be expected running real programs.

The programming efforts were headed into two directions: to obtain the high-

est possible speed with a single core, and to try to minimize the impact of data transfers. Today it is not realistic to expect for a compiler to be able to exploit all the parallelism available in a multi-core processor, so programs have to be written explicitly exposing the parallelism. Moreover, in the case of CBE all the data transfers have to be mananged by the software, so a remarkable effort is required to the programmer. However, the need to deal with low-level details allows to design programs that matches the properties of the hardware.

The exploitation of the capability of a core has to deal with architectural constraints and with the problem of mapping the intrinsic concurrency of the algorithm into the two the functional units of a core. As in CBE applications the data layout has to respect 16 bytes alignment to match the properties of the SIMD functional units, the order of the spins inside the lattice has been rearranged to minimize the permutations required by data access. The operations needed to update the spins have been mapped into logical instructions according to multispin coding, with the support of multiple SIMD-granularities, and *intrinsics* [77] have been explicitly used. Random number generation procedure has been written in a way that takes advantage of SIMD-instructions and with the support of several SIMD-granularties. It was also possible to distribute the instructions almost equivalently between the two pipelines of a SPE. Several techniques were useful to improve the performance: the low-level addressing, the partial unrolling of the inner loop, the intensive use of local variables to reduce the access to the local store and to expose to the compiler the independence of the instructions. Several macros have been defined, to perform compile-time optimizations based onto parameters such as linear lattice size, SIMD-granularity and the number of cores. The quality of the code and of instruction scheduling produced by the compiler has been statically analyzed using the tools of the SDK and also some scripts that we were specifically developed. As a results, the code reacts to the varying of the parameters accordingly to our prediction and the achieved perforamcne are reasonably close to the ideal estimates. The estimates of the performance parameter $\tau(V)$ on a single core were applied to our models to predict the balance of computations and data transfers.

The distribution of the data set among memories and the definition of data structures determines the impact of data transfers on global time. It is important to organize the algorithm in a way that allows to minimize the data transfers that cannot be executed concurrently to computations. According to this target, we used double-buffering techniques and inside each local memory we reserved enough space to support both the storing of previously computed results and the prefetch of the data required by the following update step. When only local memories are used, to allow the concurrency of computations and data transfers it is necessary to rearrange the order in which the spins are updated, while when using main memory the algorithm has been organized in a way that allows to fetch each half-plane only once to update 3 different half-planes.

The development of spin glass simulations for CBE allowed to learn which are the constraints that have to be faced when programming CBE and to define several guidelines that should be followed when developing and application for this processor. The need to explicitly manage data transfers requires an additional effort to the programmer, but gives the opportunity to efficiently use the available local memory, as it is possible to prefetch the required data, minimizing the impact of data access, even when the data transfer time is higher than computational time. However, in some cases the bandwith of main memory can be too small to feed all the eight cores.

The key factor in CBE programming is how much data can be kept into local memories. It is fundamental to define which fraction of the data set can fit the local memories, and to specify the algorithm that transfers data between main and local memories, because they constraint the amount of the computational power of the cores that can be reasonably exploited. In this optic, the definition of an analytic model is fundamental to determine the impact of data access into respect of the execution time, as the use of main memory instead of local memory can imply a notable drop of performance. Moreover, if the analytic model shows that the application is memory-bounded, a high optimization of the computational kernels is not useful.

The presence of the MFCs, that allows to move data independently by computations, makes very convenient techniques such as double and multi-buffering to mitigate the impact of data transfers. Access to main memory should be always avoided when inter-core data transfers are instead possible, because (i) main memory is a shared resource and (ii) several inter-core transfers can occur in parallel. Because the global ordering of data transfers is not guaranteed in CBE, it can be necessary to explicitly require a local or global ordering to ensure the correctness of the program. The synchronization should be distributed: instead of using a global, centralized synchronization involving PPE, it is better to directly synchronize small sets of cores, because in this way the time spent by cores waiting each other is reduced.

The optimization of the computational kernels that run on a single core requires to explicitly expose parallelism, but there are some techniques that proved to be successfull. First of all, the data layout has to be defined in a way that minimize the data access. If the scalar variables have to be rearranged into vector registers, there can be a performance drop. The large register file of the SPE can be exploited to efficiently pipeline instructions, so the access to arrays or to data structures with complicated address should be minimized inside the computational kernel, that should operate on local variables, that avoid false dependences and can directly be mapped into registers, avoiding unnecessary load and store instructions. SPEs do not support branch speculation, so the code should be kept as linear as possible. In particular, the computational kernel should not include function calls and also

conditional assignments should always be made using specific SIMD-instructions that avoid branches. For the same reason, loop unrolling can improve performance, although the size of the executable code should be kept small, in order to store as much data as possible into local memories.

We conclude that the parallelism of an algorithm can be effectively exploited by CBE architecture, if the balancing criteria between computing power and bandwidth of inter-core and core-to-memory communications are satisfied and if the computational kernels are opportunely optimized. Applications for CBE cannot be designed aiming to high flexibility and generality, because the CBE is very sensitive to the size of the problem, and there are considerable performance drops if access to main memory are too frequent. In a massively-parallel machine, however, it is likely that the data set can be distributed among the local memories of all the processors, that is the best case. The performance/Watt and performance/cost ratios of CBE are good enough to trigger the development of CBE-based massively parallel machines, and two have already been developed: RoadRunner [78], that is built around a standard interconnection network based on InfiniBand, and QPACE [79], that is equipped with a custom network designed for a specific application (the LQCD).

From this study of CBE processor we can conclude that with multi-core architectures it is effectively possible to achieve a very high computational power. However, to obtain such results it is necessary to explicitly expose the parallelism of the application and to optimize the computational kernels at low-level. Moreover, the applications should have a balance between computations and data transfers that fits the properties of the hardware. In particular, the bandwidth of main memory may be too small for several applications, and probably this will be one of the main concern of the next years, as the processors will be include more and more cores. However, the balance between data and computations it is not the only challenge of the future. The effort required to efficiently program the CBE is considerably higher than those needed for a standard processor. As the multi and many-core will dominate the near future, it is fundamental to develop adequate tools to develop application for these machines as efficiently and as fast as possible. As a consequence, we expect to see a burst of compiler technology in the next years.

Bibliography

[1] Michael Allen Barry Wilkinson. *Parallel Programming - Techniques and Appllications Using Networked Workstations ans Parallel Computers*. Prentice Hall, 1999.

[2] Michael J. Quinn. *Parallel Computing*. McGraw-Hill, Inc., 1994.

[3] *Grand Challanges in Computing*. The British Computer Society.

[4] Belletti et al. Computing for LQCD: apeNEXT. *Computing in Science and Engg.*, 8(1):18–29, 2006.

[5] P. A. Boyle. QCDOC: A 10 Teraflops Computer for Tightly-Coupled Calculations. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 40, Washington, DC, USA, 2004. IEEE Computer Society.

[6] Robert D. Mawhinney. The 1 Teraflops QCDSP computer. *Parallel Computing*, 25:1281, 1999.

[7] Junichiro Makino. The GRAPE Project. *Computing in Science and Engg.*, 8(1):30–40, 2006.

[8] A. Cruz et al. SUE: A Special Purpose Computer for Spin Glass Models. *Comp. Phys. Comm.*, (133):165–176, 2001.

[9] Francesco Belletti et al. IANUS: an FPGA-based System for High Performance Scientific Computing. *CoRR*, abs/0710.3535, 2007.

[10] Gianfranco Bilardi, Andrea Pietracaprina, Fabio Schifano, and Raffaele Tripiccione. The Potential of On-Chip Multiprocessing for QCD Machines, 2005.

[11] M. Mézard, G. Parisi, and M. Virasoro. *Spin Glass Theory and Beyond*. World Scientific, 1987.

[12] M. Schulz. *Statistical Physics and Economics: Concepts, Tools, and Applications*. Springer, 2003.

[13] D. L. Stein. *Spin Glasses and Biology*. World Scientific, 1992.

[14] F. Belletti, M. Cotallo, A. Cruz, L. A. Fernandez, A. Gordillo-Guerrero, M. Guidetti, A. Maiorano, F. Mantovani, E. Marinari, V. Martin-Mayor, A. Munoz Sudupe, D. Navarro, G. Parisi, S. Perez-Gaviro, J. J. Ruiz-Lorenzo, S. F. Schifano, D. Sciretti, A. Tarancon, R. Tripiccione, J. L. Velasco, and D. Yllanes. Nonequilibrium spin glass dynamics from picoseconds to 0.1 seconds. *Physical Review Letters*, 101:157201, 2008.

[15] F. Belletti et al. Simulating Spin Systems on Ianus, an FPGA-Based computer. *Comp. Phys. Comm.*, (178):208–216, 2008.

[16] Wm Wulf and Sally A. McKee. Hitting the Memory Wall: Implications of the Obvious. Technical report, Charlottesville, VA, USA, 1994.

[17] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann, 2007.

[18] McCalpin John, Moore Chuck, and Hester Phil. The Role of Multicore Processors in the Evolution of General-Purpose Computing. *CTWatch Quarterly*, Volume 3(Number 1), February 2007. `http://www.ctwatch.org/quarterly/articles/2007/02/`.

[19] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[20] Thomas Sterling, Peter Kogge, William J Dally, Steve Scott, William Gropp, David Keyes, and Pete Beckman. Multi-core for HPC: breakthrough or breakdown? In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 73, New York, NY, USA, 2006. ACM.

[21] William Gropp. Half full or half empty? SC06 (International Conference for High Performance Computing, Networking, Storage and Analysis).

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

[23] The Cisco QuantumFlow Processor: Cisco's Next Generation Network Processor.

[24] Dongarra J., Gannon D., G. Fox, and Kennedy K. The Impact of Multicore on Computational Science Software. *CTWatch Quarterly*, Volume 3(Number 1), February 2007. `http://www.ctwatch.org/quarterly/articles/2007/02/the-impact-of-multicore-on-computational-science-software/`.

[25] Manferdelli John L. The Many-Core Inflection Point for Mass Market Computer Systems. *CTWatch Quarterly*, Volume 3(Number 1), February 2007. `http://www.ctwatch.org/quarterly/articles/2007/02/`.

[26] Turek Dave. High Performance Computing and the Implications of Multicore Architectures. *CTWatch Quarterly*, Volume 3(Number 1), February 2007. `http://www.ctwatch.org/quarterly/articles/2007/02/`.

[27] John Shalf and David Patterson. Confronting Parallelism: The View from Berkeley, 2007. `http://www.hpcwire.com/features/17897779.html`.

[28] John Owens. Streaming architectures and technology trends. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*, page 9. ACM, 2005.

[29] Cray XD1 Datasheet. `www.cray.com`.

[30] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the cell multiprocessor. *IBM J. Res. Dev.*, 49(4/5):589–604, 2005.

[31] *Synergistic Processor Unit Instruction Set Architecture*. `http://www-128.ibm.com/developerworks/power/cell/documents.html`.

[32] T. Chen, R. Raghavan, J. N. Dale, and E. Iwata. Cell Broadband Engine Architecture and its first implementation: a performance view. *IBM J. Res. Dev.*, 51(5):559–572, 2007.

[33] Michael Kistler, Michael Perrone, and Fabrizio Petrini. Cell Multiprocessor Communication Network: Built for Speed. *IEEE Micro*, 26(3):10–23, 2006.

[34] C. R. Johns and D. A. Brokenshire. Introduction to the Cell Broadband Engine Architecture. *IBM J. Res. Dev.*, 51(5):503–519, 2007.

[35] *PowerPC Architecture book: Book III: PowerPC Operating Environment Architecture*. International Business Machines Corporation (IBM).

[36] *Cell Broadband Engine Programming Handbook Including the PowerXCell 8i Processor*. International Business Machines Corporation (IBM), 2008.

[37] *IBM Cell Broadband Engine Architecture*. `http://www-128.ibm.com/developerworks/power/cell/documents.html`.

[38] Richard Walsh, Earl C. Joseph, Steve Conway, and Jie Wu. With Its New PowerXCell 8i Product Line, IBM Intends to Take Accelerated Processing into the HPC Mainstream, 2008.

[39] A. K. Nanda, J. R. Moulic, R. E. Hanson, G. Goldrian, M. N. Day, B. D. D'Arnora, and S. Kesavarapu. Cell/B.E. blades: building blocks for scalable, real-time, interactive, and digital media servers. *IBM J. Res. Dev.*, 51(5):573–582, 2007.

[40] *Programming the Cell Broadband Engine Architecture: Examples and Best Practices*. IBM Redbooks.

[41] Kevin O'Brien, Kathryn O'Brien, Zehra Sura, Tong Chen, and Tao Zhang. Supporting OpenMP on Cell. *Int. J. Parallel Program.*, 36(3):289–311, 2008.

[42] M. Ohara, H. Inoue, Y. Sohda, H. Komatsu, and T. Nakatani. MPI Microtask for programming the Cell Broadband Engine Processor. *IBM Syst. J.*, 45(1):85–102, 2006.

[43] Pramod K. Bhatotia, Sanjeev K. Aggarwal, and Mainak Chaudhuri. Compiling Irregular Accesses for the Cell Broadband Engine.

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

[45] Alexandre E. Eichenberger, Kathryn O'Brien, Kevin O'Brien, Peng Wu, Tong Chen, Peter H. Oden, Daniel A. Prener, Janice C. Shepherd, Byoungro So, Zehra Sura, Amy Wang, Tao Zhang, Peng Zhao, and Michael Gschwind. Optimizing Compiler for the CELL Processor. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 161–172, Washington, DC, USA, 2005. IEEE Computer Society.

[46] Alastair Donaldson, Paul Keir, and Anton Lokhmotov. Compile-time and runtime issues in an auto-parallelisation system for the Cell BE processor. In *Proceedings of the 2nd Workshop on Highly Parallel Processing on a Chip (HPPC)*, Lecture Notes in Computer Science. Springer, 2008.

[47] Yuan Zhao and Ken Kennedy. Dependence-based Code Generation for a CELL processor. In *Proceedings of the 19th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2006)*. Springer-Verlag, Lecture Notes in Computer Science, 2006.

[48] Chen-Yong Cher and Michael Gschwind. Cell GC: using the Cell synergistic processor as a garbage collection coprocessor. In *VEE '08: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 141–150, New York, NY, USA, 2008. ACM.

[49] F. Belletti, G. Bilardi, M. Drochner, N. Eicker, Z. Fodor, D. Hierl, H. Kaldass, T. Lippert, T. Maurer, N. Meyer, A. Nobile, D. Pleiter, A. Schaefer, F. Schifano, H. Simma, S. Solbrig, T. Streuer, R. Tripiccione, and T. Wettig. QCD on the Cell Broadband Engine, 2007.

[50] Khaled Z. Ibrahim and Francois Bodin. Implementing Wilson-Dirac operator on the Cell Broadband Engine. In *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*, pages 4–14, New York, NY, USA, 2008. ACM.

[51] Buğra Gedik, Rajesh R. Bordawekar, and Philip S. Yu. CellSort: high performance sorting on the Cell processor. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 1286–1297. VLDB Endowment, 2007.

[52] David A. Bader, Virat Agarwal, and Kamesh Madduri. On the Design and Analysis of Irregular Algorithms on the Cell Processor: A Case Study of List Ranking.

[53] Neil Costigan and Michael Scott. Accelerating SSL using the Vector processors in IBM's Cell Broadband Engine for Sony's Playstation 3. Cryptology ePrint Archive, Report 2007/061, 2007. `http://eprint.iacr.org`.

[54] Samuel Williams, John Shalf, Leonid Oliker, Shoaib Kamil, Parry Husbands, and Katherine Yelick. Scientific computing Kernels on the Cell processor. *Int. J. Parallel Program.*, 35(3):263–298, 2007.

[55] Josiah Hosie Charles Lohr. Real Time Video Processing on the CBEA.

[56] Hubert Simma. Lattice QCD on the Cell Processor. Cell Cluster Meeting in Julich, 2007.

[57] Dirk Pleiter. Lattice QCD on the Cell BE. Power Architecture Developer Conference 07, 2007.

[58] Gregory Buehrer, Srinivasan Parthasarathy, and Matthew Goyder. Data mining on the Cell Broadband Engine. In *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*, pages 26–35. ACM, 2008.

[59] *NVIDIA CUDA Programming Guide.* `http://www.nvidia.com/object/cuda_develop.html`.

[60] *AMD Stream Computing User Guide.* `http://ati.amd.com/technology/streamcomputing/`.

[61] K. Binder and A. P. Young. Spin Glasses: Experimental Facts, Theoretical Concepts and Open Questions. *Rev. Mod. Phys.*, (58):801–976, 1986.

[62] Rodney J. Baxter. *Exactly Solved Models in Statistical Mechanics.* Dover Publications, December 2007.

[63] M. Mezard, Giorgio Parisi, and M. Virasoro. *Spin Glass Theory and Beyond (World Scientific Lecture Notes in Physics, Vol 9).* World Scientific Publishing Company.

[64] F. Y. Wu. The potts model. *Rev. Mod. Phys.*, (54):235–268, 1982.

[65] J. Barahona. On the computational complexity of Ising spin glass models. *J. Phys. A: Math. Gen.*, (15):3241–3253, 1982.

[66] R. Monasson, R. Zecchina, S. Kirkpatrick, B. Selman, and L. Troyanskyk. Determining computational complexity from characteristic 'phase transitions'. *Nature*, 400, 1999.

[67] C. P. Bachas. Computer-intractability of the frustration model of a spin glass. *J. Phys. A: Math. Gen.*, (17), 1984.

[68] Istrail Sorin. Statistical mechanics, three-dimensionality and NP-completeness: I. universality of intracatability for the partition function of the ising model across non-planar surfaces (extended abstract). In *STOC '00: Proceedings of the thirty-second annual ACM symposium on Theory of computing*, pages 87–96. ACM, 2000.

[69] Barry A. Cipra. The Ising model is NP-complete. *SIAM News*, 33(6), 2000.

[70] N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller, , and E. Teller. Equation of state calculations by fast computing machines. *J. Chem. Phys.*, pages 1087–1092, 1953.

[71] Barkema Newman. *Monte Carlo methods in Statistical Physics.* Oxford University Press, 1999.

[72] John E. Savage and Mohammad Zubair. A unified model for multicore architectures. In *IFMT '08: Proceedings of the 1st international forum on Next-generation multicore/manycore technologies*, pages 1–12, New York, NY, USA, 2008. ACM.

[73] Parisi G. and Rapuano F. Effects of the random number generator on computer simulations. *Phys. Lett. B*, (157):301–302, 1985.

[74] Mike Acton. Better Performance Through Branch Elimination. `http://www.cellperformance.com/articles/2006/07/tutorial\_branch\_elimination\_pa.html`.

[75] *SPU Application Binary Interface Specification*. `http://www-128.ibm.com/developerworks/power/cell/documents.html`.

[76] Nils Meyer. Local store peer-to-peer benchmarks and uni-directional DMA-transfer modelling for single source/target SPEs (performed on QS20 CellBlades), May/June 2007.

[77] *IBM C/C++ Language Extensions for Cell Broadband Engine Architecture*. `http://www-128.ibm.com/developerworks/power/cell/documents.html`.

[78] Entering the petaflop era: the architecture and performance of RoadRunner. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11. IEEE Press, 2008.

[79] H. Baier, H. Boettiger, M. Drochner, N. Eicker, U. Fischer, Z. Fodor, G. Goldrian, S. Heybrock, D. Hierl, T. Huth, B. Krill, J. Lauritsen, T. Lippert, T. Maurer, J. McFadden, N. Meyer, A. Nobile, I. Ouda, M. Pivanti, D. Pleiter, A. Schafer, H. Schick, F. Schifano, H. Simma, S. Solbrig, T. Streuer, K. H. Sulanke, R. Tripiccione, T. Wettig, and F. Winter. Status of the QPACE Project, 2008.