CHAPTER 1

MULTI- AND MANY-CORES, ARCHITECTURAL OVERVIEW FOR PROGRAMMERS

Lasse Natvig, Alexandru Iordan, Mujahed Eleyat, Magnus Jahre and Jorn Amundsen

1.1 INTRODUCTION

1.1.1 Fundamental Techniques

Parallelism has been used since the early days of computing to enhance performance. From the first computers to the most modern sequential processors (also called *uniprocessors*), the main concepts introduced by von Neumann [20] are still in use. However, the ever-increasing demand for computing performance has pushed computer architects toward implementing different techniques of parallelism. The von Neumann architecture was initially a sequential machine operating on scalar data with bit-serial operations [20]. *Word-parallel* operations were made possible by using more complex logic that could perform binary operations in parallel on all the bits in a computer word, and it was just the start of an adventure of innovations in parallel computer architectures.

Programming Multicore and Many-core Computing Systems,
First Edition. Edited by Sabri Pllana and Fatos Xhafa.

Prefetching is a 'look-ahead technique' that was introduced quite early and is a way of parallelism that is used at several levels and in different components of a computer today. Both data and instructions are very often accessed sequentially. Therefore, when accessing an element (instruction or data) at address k, an automatic access to address $k+1$ will bring the element to where it is needed *before* it is accessed and thus eliminates or reduces waiting time. Many clever techniques for *hardware prefetching* have been researched [5, 17] and can be exploited in the context of the new multicore processors. However, the opportunities and challenges given by the new technology in multicores require both a review of old techniques and a development of new ones [9, 21]. *Software prefetching* exploits sequential access patterns in a similar way but either it is controlled by the compiler inserting prefetch operations or it can be explicitly controlled by the programmer [10].

Block access is also a fundamental technique that in some sense is a parallel operation. Instead of bringing one word closer to the processor, for example, from memory or cache, a *cache line* (block of words) is transferred. Block access also gives a prefetching effect since the access to the first element in the block will bring in the succeeding elements. The evolution of processor and memory technology during the last 20 years has caused a large and still increasing gap between processor and memory speed-making techniques such as prefetching and block access even more important than before. This *processor–memory gap*, also called the *memory wall*, is further discussed in Section 1.2.

Functional parallelism is a very general technique that has been used for a long time and is exploited at different levels and in different components of almost all computers today. The principle is to have different functional units in the processor that can operate concurrently. Consequently, more than one instruction can be executed at the same time, for example, one unit can execute an arithmetic integer operation while another unit executes a floating-point operation. This is to exploit what has later been called *instruction level parallelism* (ILP).

Pipelining is one main variant of functional parallelism and has been used extensively at different levels and in different components of computers to improve performance. It is perhaps most widely known from the *instruction pipeline* used in almost all contemporary processors. Instructions are processed as a sequence of steps or stages, such as instruction fetch, instruction decoding, execution and write back of results. Modern microprocessors can use more than 20 pipeline stages so that more than 20 instructions are being processed concurrently. Pipelining gives potentially a large performance gain but also added complexity since interdependencies between instructions must be handled to ensure correct execution of the program.

The term *scalar processor* denotes computers that operate on one computer word at a time. When functional parallelism is used as described in the preceding text to exploit ILP, we have a *superscalar processor*. A *k-way superscalar* processor can issue up to k instructions at the same time (during one clock cycle). Also instruction fetching, decoding and other nonarithmetic operations are parallelized by adding more functional units.

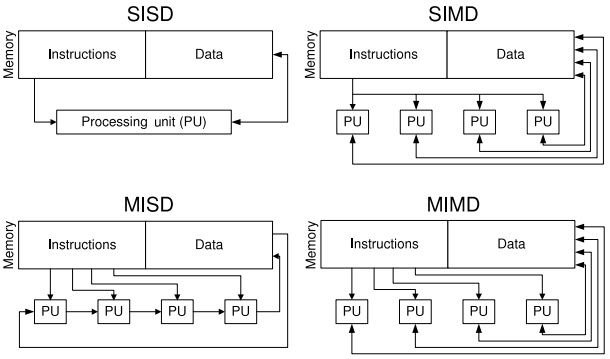

Figure 1.1 Flynn's taxonomy.

1.1.2 Multiprogramming, Multiprocessors and Clusters

Multiprogramming is a technique invented in the 1960s to interleave the execution of the programs and I/O operations among different users by time multiplexing. In this way many users can share a single computer and get acceptable response time, and the concept of a *time-sharing operating system* controlling such a computer was a milestone in the history of computers.

Multiprocessors are computers with two or more distinct physical processors, and they are capable of executing real parallel programs. Here, at the cost of additional hardware, a performance gain can be achieved by executing the parallel processes in different processors.

Many multiprocessors were developed during the 1960s and early 1970s, and in the start most of the commercial multiprocessors had only two processors. Different research prototypes were also developed, and the first computer with a large number of processors was the Illiac IV developed at the University of Illinois [6]. The project development stretched roughly 10 years, and the computer was designed to have 256 processors but was never built with more than 64 processors.

1.1.2.1 Flynn's Taxonomy

Flynn divided multiprocessors into four categories based on the multiplicity of instruction streams and data streams – and this has become known as the famous *Flynn's taxonomy* [14, 15] illustrated in Figure 1.1.

A conventional computer (uniprocessor or von Neumann machine) is termed a *Single Instruction Single Data (SISD)* machine. It has one execution or processing unit (PU) that is controlled by a single sequence of instructions, and it operates on a single sequence of data in memory. In the early days of computing, the control logic needed to decode the instructions into control signals that manage the execution and data traffic in a processor was a costly component. When introducing parallel processing, it was therefore natural to let multiple execution units operate on different data (multiple data streams) while they were controlled by the same single *control unit*, that is, a single instruction stream. A fundamental limitation of these *SIMD archi-*

tectures is that different PUs cannot execute different instructions and, at the same time, they are all bound to one single instruction stream.

SIMD machines evolved in many variants. A main distinction is between SIMD with shared memory as shown in Figure 1.1 and SIMD computers with distributed memory. In the latter variant, the main memory is distributed to the different PUs. The advantage of this architecture is that it is much easier to implement compared to multiple data streams to one shared memory. A disadvantage is that it gives the need for some mechanism such as special instructions for communicating between the different PUs.

The *Multiple Instruction Single Data (MISD)* category of machines has been given a mixed treatment in the literature. Some textbooks simply say that no machines of this category have been built, while others present examples. In our view MISD is an important category representing different parallel architectures. One of the example architectures presented in the classical paper by Flynn [14] is very similar to the variant shown in Figure 1.1. Here a source data stream is sent from the memory to the first PU, then a *derived* data stream is sent to the next PU, where it is processed by another program (instruction stream) and so on until it is streamed back to memory. This kind of computation has by some authors been called a *software pipeline* [26]. It can be efficient for applications such as real-time processing of a stream of images (video) data, where data is streamed through different PUs executing different image processing functions (e.g. filtering or feature extraction).

Another type of parallel architectures that can be classified as MISD is *systolic arrays*. These are specialized hardware structures, often implemented as an application specific integrated circuit (ASIC), and use highly pipelined and parallel execution of specific algorithms such as pattern matching or sorting [36, 22].

The *Multiple Instruction Multiple Data (MIMD)* category comprises most contemporary parallel computer architectures, and its inability to categorize these has been a source for the proposal of different alternative taxonomies [43]. In a MIMD computer, every PU has its own control unit that reads a separate stream of instructions dictating the execution in its PU. Just as for SIMD machines, a main subdivision of MIMD machines is into those having shared memory or distributed memory. In the latter variant each PU can have a local memory storing both instructions and data. This leads us to another main categorization of multiprocessors, –shared memory multiprocessors and message passing multiprocessors.

1.1.2.2 Shared Memory versus Message Passing

When discussing communication and memory in multiprocessors, it is important to distinguish the *programmers view* (logical view or *programming model*) from the actual implementation (physical view or *architecture*). We will use Figure 1.2 as a base for our discussion.

The programmers, view of a *shared memory multiprocessor* is that all processes or threads share the same single main memory. The simplest and cheapest way of building such a machine is to attach a set of processors to one single memory through a bus. A fundamental limitation of a bus is that it allows only one transaction (communication operation or memory access) to be handled at a time. Consequently, its performance does not scale with the number of processors. When multiproces-

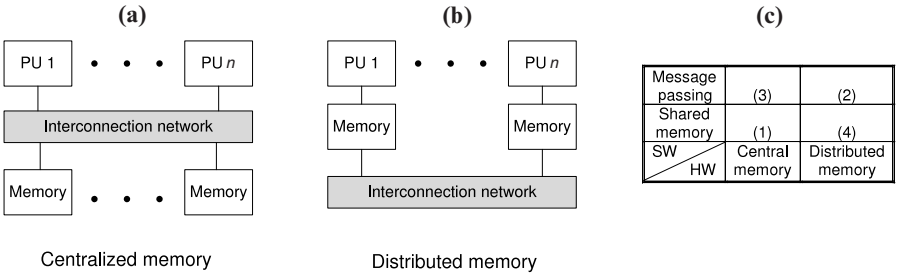

Figure 1.2 Multiprocessor memory architectures and programming models.

sors with higher number of processors were built – the bus was often replaced by an interconnection network that could handle several transactions simultaneously. Examples are a crossbar switch (all-to-all communication), multistage networks, hypercubes and meshes (see [23] Appendix E for more details). The development of these parallel interconnection networks is another example of increased use of parallelism in computers, and they are highly relevant also in multi- and many-core architectures.

When attaching many processors to a single memory module through a parallel interconnection network, the memory could easily become a bottleneck. Consequently, it is common to use several physical memory modules as shown in Figure 1.2(a). Although it has multiple memory modules, this architecture can be called a *centralized memory* system since the modules (memory banks) are assembled as one subsystem that is equally accessible from all the processors. Due to this uniformity of access, these systems are often called *symmetric multiprocessors (SMP)* or *uniform memory access (UMA)* architectures. This programming model (SW) using shared memory implemented on top of centralized memory (HW) is marked as alternative (1) in Figure 1.2(c).

The parallel interconnection network and the multiplicity of memory modules can be used to let the processors work independently and in parallel with different parts of the memory, or a single processor can distribute its memory accesses across the memory banks. This latter technique was one of the early methods to exploit parallelism in memory systems and is called *memory interleaving*. It was motivated by memory modules being much slower than the processors and was together with *memory pipelining* used to speed up memory access in early multiprocessors [26]. As seen in the next section, such techniques are even more important today.

The main alternative to centralized memory is called *distributed memory* and is shown in Figure 1.2(b). Here, the memory modules are located together with the processors. This architecture became popular during the late 1980s and 1990s, when the combination of the RISC processor and VLSI technology made it possible to implement a complete processor with local memory and network interconnect (NIC) on a single board. The machines typically ran multiprocessor variants of the UNIX operating system, and parallel programming was facilitated by *message passing libraries*, standardized with the message passing interface (MPI) [47]. Typical for these machines is that access to a processors local memory module is much

faster than access to the memory module of another processor, thus giving the name *NonUniform Memory Access (NUMA)* machines. This multiprocessor variant with message passing SW and a physically distributed memory is marked as (2) in the right part of Figure 1.2(c).

The distributed architectures are generally easier to build, especially for computers designed to be scalable to a large number of processors. When the number of processors grows in these machines either the cost of the interconnection network will increase rapidly (as with crossbar) or it will become both more costly and slower (as with multistage network). A slower network will make every memory access slower if we use centralized memory.

However, with distributed memory, a slower network can to some extent be hidden if a large fraction of the accesses can be directed to the local memory module. When this design choice is made, we can use cheaper networks and even a hierarchy of interconnection networks, and the programmer is likely to develop software that exploits the given NUMA architecture. A disadvantage is that the distribution and use of data might become a crucial factor to achieve good performance – and in that way making programming more difficult. Also, the ability of porting code to other architectures without loosing performance is reduced.

Shared memory is generally considered to make parallel programming easier compared to message passing, since cooperation and synchronization between the processors can be done through shared data structures, explicit message passing code can be avoided, and memory access latency is relatively uniform. In such *distributed shared memory (DSM)* machines, the programmers view is one single address space, and the machine implements this using specialized hardware and/or system software such as message passing. The last alternative (3) – to offer message passing on top of centralized memory-is much less common but can have the advantage of offering increased portability of message passing code. As an example, MPI has been implemented on multicores with shared memory [42].

The term *multicomputer* has been used to denote parallel computers built of autonomous processors, often called nodes [26]. Here, each node is an independent computer with its own processor and address space, but message passing can be used to provide the view of *one* distributed memory to the multicomputer programmer. The nodes normally also have I/O units, and today the mostly used term for these parallel machines is *cluster*. Many clusters are built of commercial-off-the-shelf (COTS) components, such as standard PCs or workstations and a fast local area network or switch. This is probably the most cost-efficient way of building a large supercomputer if the goal is maximum compute power on applications that are easy to parallelize. However, although the network technology has improved steadily, these machines have in general a much lower internode communication speed and capacity compared to the computational capacity (processor speed) of the nodes. As a consequence, more tightly coupled multiprocessors have often been chosen for the most communication intensive applications.

1.1.2.3 Multithreading

Multithreading is quite similar to multiprogramming, that is, multiple processes or threads share the functional units of one processor by using

overlapped execution. The purpose can be to execute several programs on one processor as in multiprogramming or can be to execute a single application organized as a multithreaded program (real parallel program). The threads in multithreading are sometimes called HW threads, while the threads of an application can be called SW threads or processes. The HW threads are under execution in the processor, while SW threads can be waiting in a queue outside the processor or even swapped to disk.

When implementing multithreading in a processor, it is common to add internal storage making it possible to save the current architectural state of a thread in a very fast way, making rapid switches between threads possible.

A switch between processes, normally denoted *context switch* in operating systems terminology, can typically use hundreds or even thousands of clock cycles, while there is multithreaded processors that can switch to another thread within one clock cycle. Processes can belong to different users (applications) while threads belong to the same user (application). The use of multithreading is now commonly called *thread-level parallelism (TLP),* and it can be said to be a higher level of parallelism than ILP since the execution of each single thread can exploit ILP.

Fine-grained multithreading denotes cases where the processor switches between threads at every instruction, while in *coarse grained multithreading* the processor executes several instructions from the same thread between switches, normally when the thread has to wait for a lengthy memory access. Both ILP and TLP can be combined as in *simultaneous multithreading (SMT)* processors where the k issue slots of a k-way superscalar processor can be filled with instructions from different threads. In this way, it offers 'real parallelism' in the same way as a multiprocessor. In a SMT processor, the threads will compete for the different subcomponents of the processor, and this might at first sight seem to be a poor solution compared to a multiprocessor where a process or thread can run at top speed without competition from other threads. The advantage of SMT is the good resource utilization of such architectures – very often the processor will stall on lengthy memory operations, and more than one thread is needed to fill in the execution gap. *Hyper-threading* is Intel's terminology (officially called hyper-threading technology) and corresponds to SMT [48].

1.2 WHY MULTICORES?

In recent years, general-purpose processor manufacturers have started to provide chips with multiple processor cores. This type of processor is commonly referred to as a *multicore architecture* or a *chip multiprocessor (CMP)* [38]. Multicores have become a necessity due to four technological and economical constraints, and the purpose of this section is to give a high-level introduction to these.

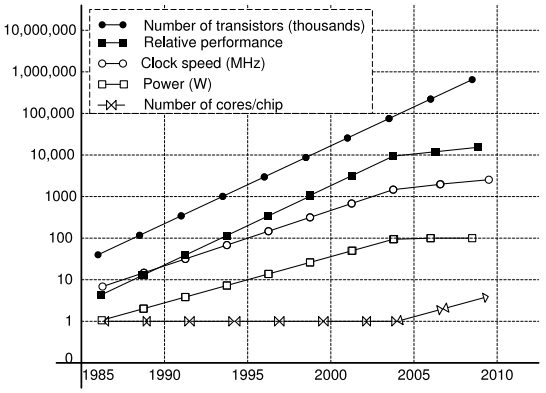

Figure 1.3 Technological trends for microprocessors. Simplified version of Figure 1 in [18].

1.2.1 The Power Wall

High-performance single-core processors consume a great deal of power, and high power consumption necessitates expensive packaging and powerful cooling solutions. During the 1990s and into the 21st century, the strategy of scaling down the gate size of integrated circuits, reducing the supply voltage and increasing the clock rate, was successful and resulted in faster single-core processors. However, around year 2004, it became infeasible to continue reducing the supply voltage, and this made it difficult to continue increasing the clock speed without increasing power dissipation. As a result, the power dissipation started to grow beyond practical limits [18], and the single-core processors were said to hit the *power wall*. In a CMP, multiple cores can cooperate to achieve high performance at a lower clock frequency.

Figure 1.3 illustrates the evolution of processors and the recent shift toward multicores. First, the figure illustrates that Moore's law still holds since the number of transistors is increasing exponentially. However, the relative performance, clock speed and power curves have a distinct knee in 2004 and has been flat or slowly increasing since then. As these curves flatten, the number of cores per chip curve has started to rise. The *aggregate* chip performance is the product of the relative performance per core and the number of cores on a chip, and this scales roughly with Moore's law. Consequently, Figure 1.3 illustrates that multicores are able to increase aggregate performance without increasing power consumption. This exponential performance potential can only be realized for a single application through scalable parallel programming.

1.2.2 The Memory Wall

Processor performance has been improving at a faster rate than the main memory access time for more than 20 years [23]. Consequently, the gap between processor performance and main memory latency is large and growing. This trend is referred to as the *processor–memory gap* or *memory wall*. Figure 1.4 contains the classical plot by Hennessy and Patterson that illustrates the memory wall. The effects of the

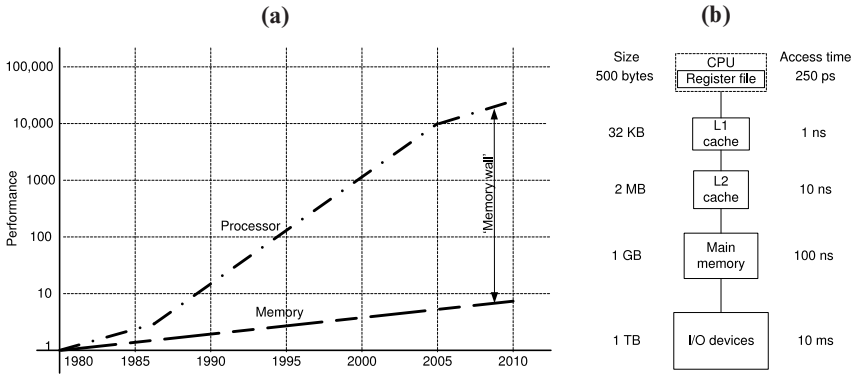

Figure 1.4 The processor–memory gap (a) and a typical memory hierarchy (b).

memory wall have traditionally been handled with latency hiding techniques such as pipelining, out-of-order execution and multilevel caches. The most evident effect of the processor–memory gap is the increasing complexity of the memory hierarchy, shown in Figure 1.4(b). As the gap increased, more levels of cache were added. In recent years, it has been common with a third level of cache, L3 cache. The figure gives some typical numbers for storage capacity and access latency at the different levels [23].

The memory wall also affects multicores, and they invest a significant amount of resources to hide memory latencies. Fortunately, since multicores use lower clock frequencies, the processor–memory gap is growing at a slower rate for multicores than for traditional single cores. However, *aggregate* processor performance is growing at roughly the same rate as Moore's Law. Therefore, multicores to some extent transform a latency hiding problem into an increased bandwidth demand. This is helpful because off-chip bandwidth is expected to scale significantly better than memory latencies [29, 40]. The multicore memory system must provide enough bandwidth to support the needs of an increasing number of concurrent threads. Therefore, there is a need to use the available bandwidth in an efficient manner [30].

1.2.3 The ILP Wall and the Complexity Wall

It has become increasingly difficult to improve performance with techniques that exploit ILP beyond what is common today. Although there is a considerable ILP available in the instruction stream [55], extracting it has proven difficult with current process technologies [2]. This trend has been referred to as the *ILP wall*. Multicores alleviate this problem by shifting the focus from transparently extracting ILP from a serial instruction stream to letting the programmer provide the parallelism through TLP.

Designing and verifying a complex out-of-order processor is a significant task. This challenge has been referred to as the *complexity wall*. In a multicore, a processor core is designed once and reused as many times as there are cores on the chip.

These cores can also be simpler than their single-core counterparts. Consequently, multicores facilitate design reuse and reduce processor core complexity.

1.3 HOMOGENEOUS MULTICORES

Contemporary multicores can be divided into two main classes. This section introduces *homogeneous multicores* that are processors where all the cores are similar, that is, they execute the same instruction set, they run on the same clock frequency and they have the same amount of cache resources. Conceptually, these multicores are quite similar to SMPs. The section starts by introducing a possible categorization of such multicores, before we describe a selected set of modern multicores at a high level. All of these are rather complex products, and both *the scope of this chapter and the space available make it impossible to give a complete and thorough description. Our goal is to introduce the reader to the richness and variety of the market – motivating for further studies.* The other mainclass, heterogeneous multicores, is discussed in the next section. A tabular summary of a larger number of commercial multicores can be found in a recent paper by Sodan et-al. [48].

1.3.1 Early Generations

In the paper *Chip Multithreading: Opportunities and Challenges* by Spracklen and Abraham [50], the authors introduced a categorization of what they called chip multi threaded processors (CMT processors) that also can be used to categorize multicore architectures. As shown in Figure 1.5, the first generation multicores typically had processor cores that did not share any on-chip resources except the off-chip datapaths. It was normally two cores per chip and they were derived from earlier uniprocessor designs. Also the PUs used in the second generation multicores were from earlier uniprocessor designs, but they were more tightly integrated through use of a *shared L2 cache*. It could be more than two processors, and the shared L2 made intracore communication very fast. The cores sometimes run the same program (SPMD), so the demand for cache capacity for storing instructions can be reduced. Both these advantages of the shared L2 cache can reduce the demand of off-chip bandwidth. However, more than one core using the L2 cache introduce new challenges such as cache partitioning, fairness and quality of service (Qos) [12, 11, 30].

The third generation multicores can be said to be those using cores that are designed from the ground up and optimized to sit in a multicore processor. These may typically be simpler cores running at a lower frequency and hence with a much lower power consumption. Further, they are typically using SMT. Olukotun and Hammond [37] call these three generations for simple CMP, shared-cache CMP and multithreaded shared-cache CMP, respectively.

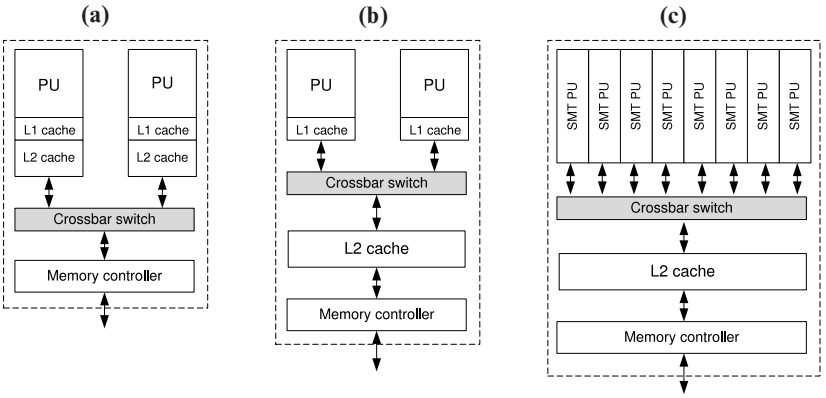

Figure 1.5 Multicore processor generations: first (a), second (b), third (c).

1.3.2 Many Thin Cores or Few Fat Cores?

The choice between a few powerful and many less powerful processors or cores has been discussed widely both during the multiprocessor era and the multicore era. In his classical paper Amdahl [3] gave a simple formula explaining how the serial fraction of an application severely constraints the maximum speedup that can be achieved by a multiprocessor. The serial fraction is a code that cannot be parallelized, and Amdahl's law might motivate for having at least one core that is faster than the others, that is, go for a heterogeneous multicore. For executing the so-called *embarrassingly parallel applications*, that is, applications that are very easy to parallelize since they have no or a very tiny serial part – a multicore with a large number of small cores might be most efficient, especially if power efficiency is in focus. However, if there is significant serial fraction, a smaller number of more powerful cores might be best. A recent paper by Hill and Marty [24] titled *Amdahl's Law in the Multicore Era* demonstrates the influence of Amdahl's law on this trade-off in an elegant way.

1.3.3 Example Multicore Architectures

1.3.3.1 IBM(R) Power(R) Performance Optimization With Enhanced RISC (POWER) is an IBM processor architecture for technical computing workloads implementing superscalar RISC. The POWER architecture was the starting point in 1991 of the Apple®, IBM and Motorola® (now Freescale Semiconductor®) joint effort to develop a new RISC processor architecture, the PowerPC® architecture [49]. The design goals of PowerPC were to create a single chip providing multiprocessing extensions and 64-bit support (addressing and operations). It was later expanded with vector instructions, originally trademarked AltiVec™. In 2006, POWER and PowerPC was unified into a new brand, the *Power Architecture*, owned by Power.org.

The POWER*n* series of processors are IBM's main product line implementing the Power architecture. The first product in this series was the multichip, super-

scalar and out-of-order POWER1 processor, introduced in 1990. The POWER7®, introduced in 2010, is the latest development in this series and is also the processor to power the first DARPA High Productivity Computing System (HPCS) petaflops computer. A stripped-down POWER7-core is expected to be used in the Blue Gene®/Q system, replacing the BlueGene/P massively parallel supercomputer in 2012.

The POWER7 processor provides 4, 6 or 8 cores per chip, each with 4-way hardware multithreading (SMT) [1]. A core might under software control be set to

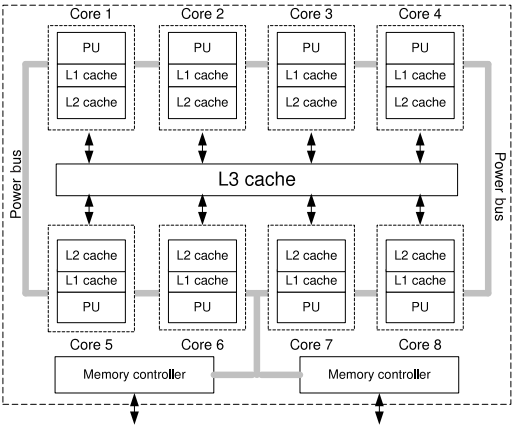

Figure 1.6 Power 7 multicore, simplified block diagram.

operate at different degrees of multithreading from single-threaded mode (ST) to 4-ways SMT.

The chip is implemented in 45 nm technology, with cores running at a nominal frequency of 3.0 – 4.14 GHz, depending on the configuration. The cache hierarchy consists of 32K 4-way L1 data and instruction caches, a 256K 8-way L2 cache and 32 MB shared L3 cache, partitioned into 8×4 MB 8-way partitions. The L3 cache is implemented with embedded DRAM technology (eDRAM). The chip is organized as 8 cores (called *chiplets*), each containing the PU, L1 and L2 caches and one of the 8 L3-cache partitions (Fig. 1.6). A consequence of this design is that the L3 has a nonuniform latency. A pair of DDR3 DRAM controllers, each with four 6.4 GHz channels provides a sustained main memory bandwidth of over 100 GB/s.

In addition to POWER6® VMX (AltiVec) and decimal floating point (DFU), the POWER7 core provides the new VSX vector facility. VSX is mainly an extension for 64-bit vector floating-point arithmetic; it does not provide 64-bit integer arithmetic like Intel® and AMD processors.

Energy efficiency is implemented at the core or chiplet level where each core frequency might be individually changed. The modes *sleep*, *nap* and *turbo* allows dynamic voltage and frequency adjustment, from off, to –50% and +10% for maximum performance.

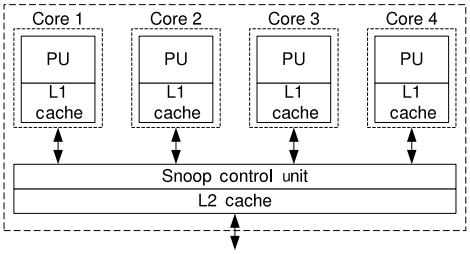

Figure 1.7 ARM Cortex A15, simplified block diagram.

1.3.3.2 ARM(R) Cortex™-A15 MPCoreTM Processor

ARM became one of the first companies to implement multicore technology with the launch of the ARM11™ MPCore™ processor in 2004. The latest version of the ARM MPCore technology is the ARM Cortex™-A15 MPCore processor, targeting markets ranging from mobile computing, high-end digital home, servers and wireless infrastructure.

The processor can be implemented to include up to four cores (see Figure 1.7). The multicore architecture enables the processor to exceed the performance of single -core high-performance embedded devices while consuming significantly less power. Every Cortex-A series processor has power management features including dynamic voltage and frequency scaling and the ability for each core to go independently into standby, dormant or power off energy management states. Like its predecessors Cortex-A15 is based on the ARMv7A processor architecture giving full application compatibility with all ARM Cortex-A processors. This compatibility enables access to an established developer and software ecosystem.

Each processor core has an out-of-order superscalar pipeline and low-latency access through a bus to a shared L2 cache that can be up to 4 MB. The cores provide floating-point support and special SIMD instructions for media performance [4].

1.3.3.3 Sun UltraSPARC(R) T2

Sun's UltraSPARC T2 is a homogeneous multithreaded multicore specially designed to exploit the TLP present in almost every server type application. Sun introduced its first multicore, multithreaded microprocessor the UltraSPARC T1 (codenamed Niagara) in November 2005 [33]. The UltraSPARC T1 uses the SPARC V9® instruction set and was available with 4, 6 and 8 processing cores, each able to execute four threads simultaneously [48]. The UltraSPARC T2 includes a network interface unit and a PCI express interface unit, and this is why the T2 is sometimes referred to as a system on chip [45]. It was available in October 2007 and produced in 65 nm technology.

The UltraSPARC T2 is comprised of 8 64-bit cores, and each core can execute 8 independent threads. Thus, T2 is able to execute 64 threads simultaneously. The cores are connected by a crossbar to an 8-banked shared L2 cache, 4 DRAM controllers and 2 interface units (Fig. 1.8).

In order to minimize power requirements and to meet temperature constraints, the UltraSPARC T2 uses a core frequency of only 1.4 GHz. A complete implementation of the UltraSparc T2 processor in Verilog™ (a HW description language) along

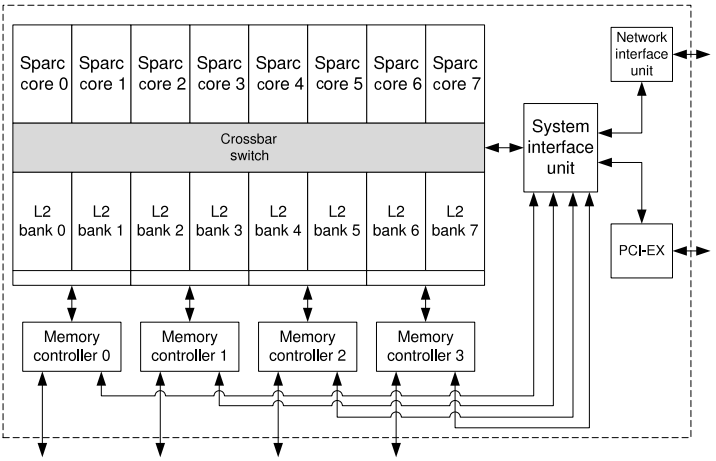

Figure 1.8 Sun UltraSPARC T2 architecture, simplified block diagram.

with tools is freely available from the OpenSPARC® project [54]. This gives the interested researcher a rare opportunity to study the inner details of a modern multicore processor.

In autumn 2010, Oracle launched the SPARC T3, previously known as UltraSPARC T3. It has 16 cores each capable of 8-way SMT giving a total of 128-way multithreading [39].

1.3.3.4 AMD Istanbul The Istanbul processor is the first 6-core AMD Opteron™ processor and is available for 2-, 4- and 8-socket systems, with clock speeds ranging from 2.0 to 2.8 GHz. It was introduced in June 2009 and is manufactured in a 45 nm process and based on the AMD 64-bit K10 architecture. The K10 architecture supports the full AMD64 instruction set and SIMD instructions for both integer and floating-point operations [25].

Figure 1.9 shows a simplified block diagram. The processor has six cores, three levels of cache, a crossbar connecting the cores, the system request interface, the memory controller and the three HyperTransport™ 3.0 links. The memory controller supports DDR2 memory with a bandwidth of up to 12.8 GB/s. In addition, the HyperTransport 3.0 links provide an aggregate bandwidth of 57.6 GB/s and are used to allow communication between different Istanbul processors.

The 6 MB of L3 cache is shared among the 6 cores: there are a 512 KB L2 cache per core and 64 KB L1 data cache and a 64 KB L1 instruction cache for each core.

1.3.3.5 Intel(R) Nehalem In November 2008, with the release of Core™ i7, Intel introduced the new microprocessor architecture Nehalem [28]. The Nehalem architecture (Fig. 1.10) has been used in a large number of processor variants in the mobile, desktop and servers markets and is mainly produced in 45 nm technology. The core count is typically 2 for mobile products, 2 – 4 cores for desktop and 4,

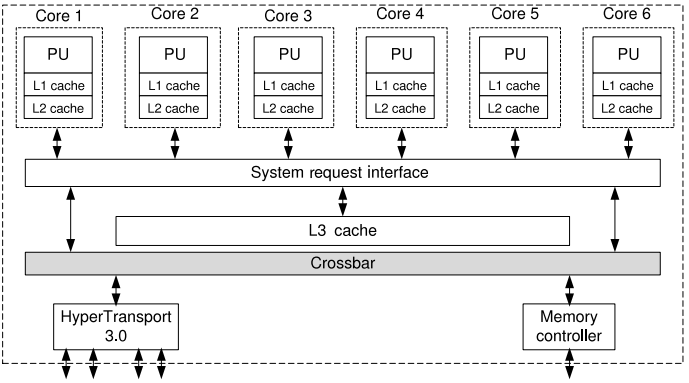

Figure 1.9 AMD Opteron Istanbul processor, simplified block diagram.

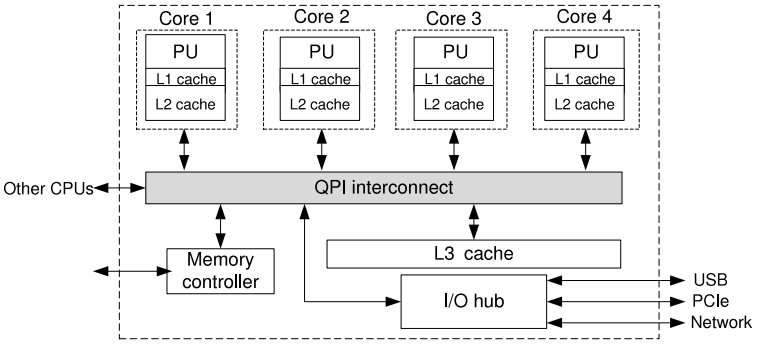

Figure 1.10 Intel Nehalem architecture – 4 cores, simplified block diagram.

6 or 8 for servers. At the high end, the Nehalem architecture shrinked to 32 nm technology (also called Westmere) has been announced to provide a 10-core chip.

Intel introduced with Nehalem the *turbo boost technology* (TBT) to allow adjustments of core frequency at runtime [27]. Considering the number of active cores, estimated current usage, estimated power requirements and CPU temperature, TBT determines the maximum frequency that the processor can run at. Core frequency can be increased in steps of 133 MHz and to a higher level if few cores are active. This allows for a boost in performance while still maintaining the power envelope. To save energy, it is possible to power down cores when they are idle, but when needed again they are turned on, and the frequency of the processor is reduced accordingly [52].

The *QuickPath interconnect* (QPI) was introduced in Nehalem to provide high speed, point-to-point connections between all cores, the I/O hub, the memory controller and the large shared L3 cache (Fig. 1.10). The L3 cache is inclusive. Nehalem-based processors have up to 3.5 times more memory bandwidth than previous generation processors.

The Nehalem architecture reintroduced hyper-threading, a technique that allows each core to run two threads simultaneously, improving on resource utilization and reducing latency. Although it was introduced in Intel processors as early as in 2002, it was not used in the *Intel core* architecture that preceded Nehalem.

For faster computation of media applications, the Nehalem architecture supports the SSE4 instruction set introduced in the previous generation processors. SSE is an abbreviation for *streaming SIMD extensions* and is an SIMD instruction set extension to the ×86 architecture that is used by compilers and assembly coders for vectorization.

1.3.3.6 Tilera(R) TILE64 TM Tilera [53] has developed and is currently shipping the TILEPro36™ and TILEPro64™ series of embedded many-core processors. The Tilera devices may contain up to 64 individual 32-bit processors on a single silicon device and are targeted at embedded markets which require programmability, high performance and demanding power constraints. All Tilera devices contain numerous integrated IO interfaces, allowing system designers to save board real estate and complexity by integrating the IO and processing into a single device. Current target markets for the TILEPro™ family of devices include video and network processing. The TILEPro family of devices is fabricated in TSMC's 90 nm technology and comes in 700 and 866 MHz frequency grades.

Each Tilera device contains multiple individual processor cores. Each core supports the TILE instruction set architecture (ISA), a Tilera proprietary ISA sharing many similarities with modern RISC ISAs. The Tilera ISA is a 3-wide VLIW format, where each 64-bit VLIW instruction encodes three operations. Correspondingly, there are three execution pipelines per processor core, two arithmetic pipelines and one load/store pipeline. When running at 866 MHz, a TILEPro64 is capable of 166 billion 32-bit operations per second. Additionally, the Tilera ISA contains SIMD operations, enabling 32b, 16b and 8b arithmetic. The physical address of the TILEPro devices is 36 bits, giving a TILEPro device access to up to 64 GB of memory. The TILEPro processor is an in-order machine, issuing 64-bit VLIW instructions in program order. However, the TILEPro cache subsystem is out of order, allowing the processor to continue to fetch, issue and execute instructions in the presence of multiple cache misses. The TILE cores do not have HW FPU support.

The TILEPro device is a complete system on a chip, containing multiple integrated IO interfaces. TILEPro64 contains four integrated DDR2 memory controllers, capable of supporting 800 MHz operation. Memory space may be configured to be automatically interleaved across the four controllers or programmatically assigned on a page-by-page mapping from page to controller.

A TILEPro processor core contains a 16 KB L1 instruction cache, an 8 KB L1 data cache and a 64 KB unified L2 cache (used for both instructions and data). All processor cores on a TILEPro device are cache coherent, enabling running of standard, shared-memory programs such as POSIX threads across the entire device. The cores may be configured into multiple coherence domains, allowing a single SMP Linux image to run across all cores within the system, or only a subset. Tilera hypervisor technology enables the ability to run multiple Linux images in parallel. Coherency

is maintained between the processor cores via a unique directory-based coherency protocol, called dynamic distributed cache (DDC). The DDC protocol tracks address sharers within the system via a distributed directory and maintains coherence by properly invalidating/updating shared data upon modification. Additionally, the Tilera cache subsystem provides the ability for one core's L2 cache to serve as a backing L3 cache for another core within the system. In this context, the L2 storage structures may contain both L2 and L3 cache blocks.

The TILEPro processor cores communicate with each other and the IO interfaces via multiple on-chip, packet-switched networks. These networks, called the iMesh™, are proprietary interconnects used to carry communication within the system such as memory read requests, memory read responses, tile-to-tile read responses, etc. The networks are configured in a mesh topology, providing performance scalability as the number of cores is increased. The TILEPro devices contain three separate mesh networks for memory and cache communication, as well as two networks for user-level messaging. These networks are synchronous with the processor cores and run at the same frequency, and the latency for a message through the mesh networks is one processor cycle per node.

1.4 HETEROGENEOUS MULTICORES

This section introduces *heterogeneous multicores* – processors where one or some of the cores are significantly different than the others. The difference can be as fundamental as the instruction set used, or it can be the processor speed or cache/memory capacity of the different cores. We start by introducing some of the main types of heterogeneity, before we present three different contemporary products in this category of processors.

1.4.1 Types of Heterogeneity in Multicores

Single-ISA heterogeneous multicores are processors where all the cores have the same ISA, that is, they can execute the same instructions, but they can have different clock frequencies and/or cache sizes. Also, the cores might have different architectures implementing the same ISA. Typically there is one or a few high-performance cores (fat cores) that are superscalar out-of-order processors and a larger number of smaller and simpler cores that can be in-order processors with a shorter pipeline [34]. As discussed in Section 1.3.2, this can be beneficial for speeding up applications where there is a significant part of the computation that is serial or if some of the threads put more demand on the memory system. This kind of multicores is called by some authors *asymmetric multicore processors (AMP)*. They have gained increased interest lately since they potentially can be more energy efficient than conventional homogeneous multicores [13].

Multiple-ISA multicores such as the Cell/BE™ microprocessor presented in Section 1.4.2.1 have two or more different instruction sets. They require a toolchain for each core type and are in general harder to program. In addition, many of these

processors, including Cell™, have explicitly managed memory hierarchies where the programmer is responsible for placement and transfer of data. This will in general increase programmer effort and code complexity compared to a cache-based system that are automatic and hidden from the programmer. Recent research has shown that comparable performance can be achieved through programming environments where compiler and runtime support implicitly manage locality [44].

In the embedded systems market, there is a long tradition of using highly heterogeneous multicores with different kinds of simple or complex cores and HW units integrated on a single chip. These *multiprocessor system-on-chip* (MPSoC) systems often achieve a very high level of power efficiency through specialization [35], but again the price to pay is often more difficult programming. MPSoC systems have been available as commercial products for longer than multicores, and some few MPSoCs are homogeneous. We refer the reader to a recent survey of MPSoCs by Wolf, Jerraya and Martin for this rich branch of multicore processors [57].

Graphics processing units (GPU) and accelerators are also considered examples of heterogeneous multicores, even though in most cases they in general need a host processor to be able to run a complete application. The principle of *hardware acceleration* – adding a special purpose HW unit to off load the processor or to speed up computation by doing specific functions in HW instead of software - has a long history. About 30 years ago, a common practice for speeding up floating-point operations in a PC was to add a floating-point coprocessor unit (FPU). Today, the inclusion of different accelerator subunits in a CMP is becoming increasingly popular, and IBM has recently announced a processor architecture where processing cores and hardware accelerators are closely coupled [16].

Similarly, the GPU was added to accelerate the processing of graphics. GPUs have during the last two decades been through a substantial development from specialized units for graphics processing only to more programmable units being popular for general-purpose GPU (GPGPU). Their programming has become substantially improved through languages such as CUDA™ and OpenCL™ [32, 8].

1.4.2 Examples of Multicore Architectures

1.4.2.1 The Cell™ Processor Architecture

The Cell Broadband Engine™ (Cell/BE) is a heterogeneous processor that was jointly developed by Sony®, Toshiba® and IBM®. As shown in Figure 1.11, it is mainly composed of one main core (power processing element (PPE)), 8 specialized cores (called synergistic processing elements (SPEs)), an on-chip memory controller and a controller for a configurable I/O interface, all linked together by an element interconnection bus (EIB) [46]. The main core is a 64-bit Power processor with vector processing extensions and two levels of hardware-managed caches, a 32 KB L1 data cache and a 512 KB L2 cache. In addition, it is a dual-issue, dual-threaded processor that has a single-precision peak of 25.6 Gflops/s and a double-precision peak of 6.4 Gflops/s.

The 8 SPEs are SIMD cores (SPU) which each possess a 256 KB local store (LS) for storing both data and instructions, a 128 × 128-bit register file and a memory flow controller (MFC). MFC has the capability to move code and data between main

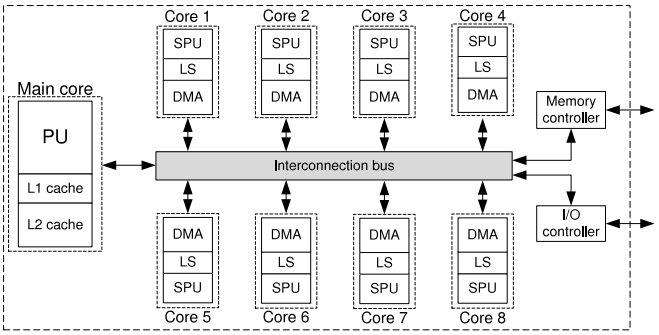

Figure 1.11 Simplified block diagram of Cell/BE.

memory and LS using a direct memory access (DMA) controller. Moreover, each SPE has a single-precision peak of 25.6 Gflops/s and a double-precision peak of only 1.83 Gflops/s. The EIB is composed of 4 unidirectional rings that are used as a communication bus between elements that are connected to it, and it can deliver 25.6 GB/s to each of them.

The memory controller is used to connect to a dual-channel Rambus extreme data rate (XDR) memory which can deliver a bandwidth of 25.6 GB/s. In addition, the Cell has an I/O controller which can be dedicated to connect up to two separate logical interfaces [31]. These interfaces provide chip-to-chip connections and can be used to design an efficient dual-processor system.

The main core (PPE) is usually responsible for running the operating system and controlling the other cores (SPEs); it can start, stop, interrupt and schedule processes running on them. In fact, SPEs achieve their work only by following PPE commands. The PPE can read and write the main memory and the local memories of SPEs through the standard load/store instructions. However, data movement to and from an SPE (LS) is achieved explicitly using DMA commands. The explicit transfer of data and limited size of SPE LS poses a major challenge to software development on the Cell/BE processor.

The *PowerXCell*^{TM} *8i* is a revised variant of the Cell/BE processor that was announced by IBM in 2008 and made available in IBM QS22 blade servers. The SPEs in the new variant have a much better double-precision floating-point peak performance (102.4 GFLOPS) compared to the previous one (14.64 GFLOPS). In addition, it has support for up to 32 GB of slotted DDR2 memory. The PowerXCell 8i processor has been used in several supercomputers. For example, the Roadrunner supercomputer, the world's fastest in 2008–2009, has 12,240 PowerXCell 8i processors in addition to 6562 AMD Opteron processors. PoweXCell 8i supercomputers have also occupied many of the top positions on the Green500 list of the most energy-efficient supercomputers in the world [51].

1.4.2.2 NVIDIA(R) Fermi The GPU is a highly specialized PU dedicated to execute or accelerate video applications. Since these applications have an increased

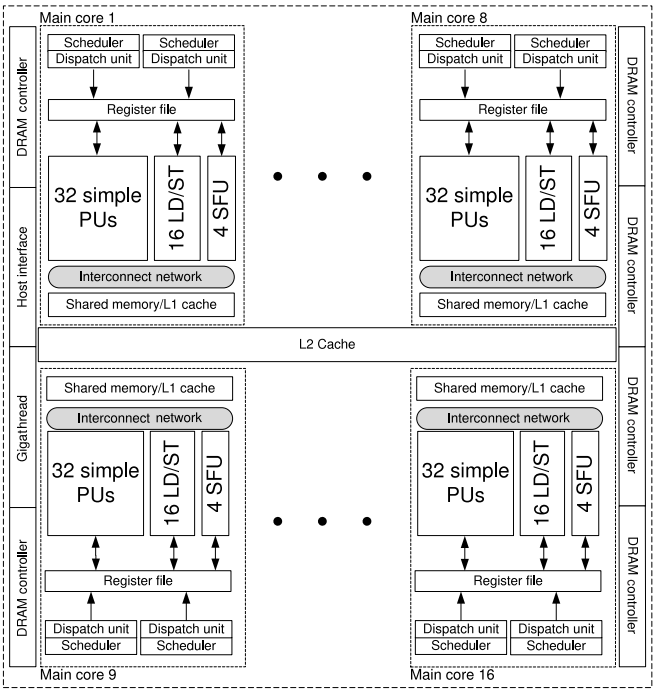

Figure 1.12 NVIDIA Fermi, simplified block diagram.

degree of parallelism, GPUs use tens to hundreds of cores to perform the advanced floating-point operations specific to video rendering. With the introduction of NVIDIA's *unified shader architecture* in 2006 and CUDA in 2007, the potential performance of GPU's massively parallel architecture was also made available to other fields, like high-performance computing (HPC) [19].

Fermi (or GeForce® 400) is the latest architecture from NVIDIA [19]. It has up to 512 PUs distributed among 16 main cores, which NVIDIA calls *streaming multiprocessors*. It has a two-level cache hierarchy and very fast double-precision floating-point math operations [56]. Fermi is a good fit for some HPC applications, and NVIDIA offers the Tesla line of products as dedicated GPGPUs to be used as accelerators in HPC supercomputers. In October 2010, after being upgraded with more than 7000 Fermi-based Tesla GPGPUs, Tianhe-1A became the fastest supercomputer in the world, as ranked by the TOP500 list.

Fermi has a transistor count of more than 3 billions that are used to create 16 main cores, a shared L2 cache, 6 memory (DRAM) controllers, a hardware thread scheduler (called GigaThread) and the host interface (Fig. 1.12). However, not all of these resources are always activated.

Each main core consists of 32 very simple PUs, capable of performing integer or floating-point operations. All PUs share a single-register file, 16 load/store units, 4

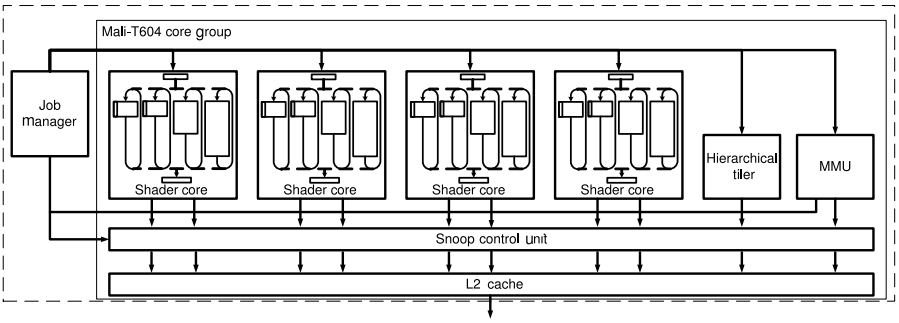

Figure 1.13 ARM Mali T604, simplified block diagram.

special function units (for advanced math like square root or sine), 2 thread schedulers and dispatch units and an L1 cache. The main core can been seen as a 32-issue superscalar processor. This main core design, coupled with the GigaThread scheduler, allows a Fermi-class GPU to switch very fast between threads and to handle more than 24.000 parallel threads in an efficient way.

In order to improve the HPC performance, Fermi uses a more standard memory hierarchy which includes a shared L2 cache. Since the memory penalty is greater with GPUs than with CPUs, NVIDIA added new keywords to its CUDA implementation that allow a programmer to specify where data will be stored. Also, in order to make it more programmer friendly, NVIDIA improved Fermi's ISA (improved atomic integer instructions [41]) and added support for C++ object-oriented programming.

1.4.2.3 ARM(R)Mali TM-T604 GPU The ARM Mali-T604 GPU is a licensable, low-power multicore GPU targeting system-on-chip (SoC) providers and a broad range of applications including mobile, digital TV, gaming and navigation. It is the first GPU from ARM with an architecture designed to enhance GPU computing through, for example, the Khronos^{TM} OpenCL^{TM} API. This is in addition to graphics standards such as Khronos OpenGL ES^{TM} and OpenVG^{TM}.

The ARM multicore design philosophy previously used with CPUs has been applied to ARM Mali GPUs, and the result is the Mali-T604. This multicore GPU has a customer-configurable number of cores that share a coherent Level 2 memory subsystem. A single job manager handles the host CPU interface and load balancing on the GPU, while a hierarchical tiler (HT) accelerates the tile-based graphics processing and a memory management unit (MMU) handles virtual address translation and processes separation (see Fig. 1.13).

The Level 2 memory subsystem can maintain full coherency between cores through a Snoop Control Unit (SCU). This approach is inspired by ARM multiprocessor CPUs and is different from traditional GPUs where local memory is noncoherent. Each shader core is a multipipeline, multithreaded unit with the ability to execute hundreds of threads simultaneously. This is particularly beneficial for throughput-oriented computing with an abundance of data-level parallelism. The

processor supports a wide range of data types, including integer and IEEE-754 floationg point up to 64-bit allowing for algorithms requiring single and double precision. Support for 64-bit integer arithmetic is provided.

Computing on the Mali-T604 GPU is highly efficient compared to high-end desktop or server GPUs. The moving of data between CPU and GPU memory is avoided by the use of a unified memory system coherent between the GPU and CPU, such as the ARM Cortex™-A15 CPU. Fast atomic operations with Mali-T604 mean that algorithms requiring interthread communication will be much more efficient than on a traditional GPU.

1.5 CONCLUDING REMARKS

Current trends in multi- and many-core architectures are increased parallelism, increased heterogeneity, use of accelerators and energy efficiency as a first-order design constraint. The '*5 P's of parallel processing: performance, predictability, power efficiency, programmability and portability*' are all important goals that we strive to meet when we build or program multicore systems. To meet the challenge of partly conflicting goals, we need more research in parallel programming models that adopts a holistic view – covering aspects from hardware and power consumption through system software and up to the programmers wish for programmability and portability.

A challenge is that optimizing for one of these goals very often will reduce the possibility of achieving some of the others. The present state of the art is very diverse and dynamic and in many ways less stable than 10 years ago. As an example, heterogeneous processors with explicitly managed memories like the Cell processor have achieved outstanding results for power efficiency [51] at the cost of reduced programmability. However, researchers are continuously looking for ways to achieve many of these goals at the same time. Power efficiency innovations in runtime systems is one of the many promising directions. Borkar and Chien [7] outline a hypothetical heterogeneous processor consisting of a few large cores and many small cores, where the supply voltage and frequency of each core are controlled individually. This fine-grained power management improves energy efficiency without burdening the application programmer, since it is controlled by the runtime system. However, significant breakthroughs are needed on the software level to make such systems practical.

To end the chapter, we would like to quote the recent and highly motivating paper *Computing Performance: Game Over or Next Level?* [18]: the era of sequential computing must give way to an era in which parallelism holds the forefront.

Trademark Notice

REFERENCES

1. J. Abeles et al. Performance Guide For HPC Applications On IBM Power 755 System. `http://power.org`, 2010.
2. V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger. Clock rate versus IPC: the end of the road for conventional microarchitectures. *ACIM SIGARCH Computer Architecture News*, 28(2):248–259, 2000.
3. G. M. Amdahl. Validity of the single processor approach to achieving large scale computer capabilities. In *AFIPS Joint Computer Conference Proceedings*, volume 30, pages 483–485, 1967.
4. ARM website. `http://www.arm.com`.
5. J. L. Baer and T. F. Chen. Effective hardware-based data prefetching for high–performance processors. *IEEE Transactions on Computers*, 44:609–623, May 1995.
6. G. H. Barnes, R. M. Brown , M. Kato , D. J. Kuck, D. L. Slotnick, and R. A. Stokes. The ILLIAC IV Computer. *IEEE Transactions on Computers*, C-17(8):746 – 757, August 1968.
7. S. Borkar and A. A. Chien. The future of microprocessors. *Communications of the ACM*, 54:67–77, May 2011.
8. A. R. Brodtkorb, C. Dyken, T. R. Hagen, J. M. Hjelmervik, and O. O. Storaasli. State-of -the-art in heterogeneous computing. *Scientific Programming*, 18:1–33, January 2010.
9. S. Byna, Y. Chen, and X. H. Sun. Taxonomy of data prefetching for multicore processors. *Journal of Computer Science and Technology*, 24:405–417, 2009.
10. D. Callahan, K. Kennedy, and A. Porterfield. Software prefetching. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS-IV, pages 40–52, New York, NY, USA, 1991. ACM.
11. H. Dybdahl. *Architectural Techniques to Improve Cache Utilization*. PhD thesis, Norwegian University of Science and Technology, Trondheim, Norway, 2010.
12. H. Dybdahl, P. Stenstrom, and L. Natvig. A cache-partitioning aware replacement policy for chip multiprocessors. In Y. Robert et al., editor, *High Performance Computing – HiPC 2006*, volume 4297 of *LNCS*, pages 22–34. Springer Berlin / Heidelberg, 2006.
13. A. Fedorova, J. C. Saez, D. Shelepov, and M. Prieto. Maximizing power efficiency with asymmetric multicore systems. *Communications of the ACM*, 52:48–57, December 2009.
14. M. J. Flynn. Very high-speed computing systems. In *Proceedings of the IEEE*, volume 54, pages 1901–1909, December 1966.
15. M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers,* C-21(9):948 –960, September 1972.
16. H. Franke, J. Xenidis, C. Basso, B. M. Bass, S. S. Woodward, J. D. Brown, and C. L. Johnson. Introduction to the wire-speed processor and architecture. *IBM Journal of Research and Development*, 54(1):3:1 –3:11, January–February 2010.
17. J. W. C. Fu, J. H. Patel, and B. L. Janssens. Stride directed prefetching in scalar processors. In *Proceedings of the 25th annual International Symposium on Microarchitecture,* MICRO 25, pages 102–110. IEEE Computer Society Press, Los Alamitos, CA, USA, 1992.

18. S. H. Fuller and L. I. Millett. Computing performance: game over or next level? *Computer*, 44(1):31–38, January 2011.

19. P. N. Glaskowsky. NVIDIA's Fermi: The First Complete GPU Computing Architecture. `http://www.nvidia.com/content/PDF/fermi_white_papers/P.Glaskowsky_NVIDIA's_Fermi-The_First_Complete_GPU_Architecture.pdf 2009.`

20. M. D. Godfrey and D. F. Hendry. The computer as von Neumann planned it. *IEEE Annals of the History of Computing,* 15(1):11–21, 1993.

21. M. Grannaes. *Reducing Memory Latency by Improving Resource Utilization*. PhD thesis, Norwegian University of Science and Technology, Trondheim, Norway, 2010.

22. A. Halaas, B. Svingen, M. Nedland, P. Saetrom, O. Snoeve Jr., and O.R. Birkeland. A recursive MISD architecture for pattern matching. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 12(7):727 – 734, July 2004.

23. J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, fourth edition, 2006.

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

25. P. G. Howard. Six-Core AMD Opteron Processor Istanbul. `http://www.microway.com/pdfs/microway_istanbul_whitepaper_2009-07.pdf`, 2009.

26. K. Hwang. *Advanced Computer Architecture: Parallelism, Scalability, Programmability*. McGraw-Hill Higher Education, first edition, 1992.

27. Intel corp. 2nd Generation Intel Core Processor Family Desktop. `http://download.intel.com/design/processor/datashts/324641.pdf`.

28. Intel corp. First the Tick, Now the Tock – Intel Microarchitecture (Nehalem). `http://www.intel.com/technology/architecture-silicon/next-gen/319724.pdf`.

29. ITRS. International Technology Roadmap for Semiconductors. `http://www.itrs.net/, 2006.`

30. M. Jahre. *Managing Shared Resources in Chip Multiprocessor Memory Systems*. PhD thesis, Norwegian University of Science and Technology, Trondheim, Norway, 2010.

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

32. D. B. Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., first edition, 2010.

33. P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: a 32-way multithreaded sparc processor. *IEEE Micro*, 25(2):21 – 29, 2005.

34. R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single-ISA heterogeneous multicore architectures: the potential for processor power reduction. *IEEE/ACM International Symposium on Microarchitecture*, pages 81–92, 2003.

35. R. Kumar, D. M. Tullsen, N. P. Jouppi, and P. Ranganathan. Heterogeneous chip multiprocessors. *Computer*, 38(11):32 – 38, November 2005.

36. H. T. Kung. Why systolic architectures? *Computer*, 15:37–46, 1982.

37. K. Olukotun and L. Hammond. The future of microprocessors. *Queue*, 3:26–29, September 2005.

38. K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The case for a single-chip multiprocessor. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS-VII, pages 2–11. ACM, New York, NY, USA, 1996.

39. Oracle Unveils SPARC T3 Systems and Storage Appliances. `http://www.oracle.com/us/corporate/features/sparc-t3-feature-173454.html`.

40. D. A. Patterson. Latency lags bandwith. *Communications of the ACM*, 47:71–75, October 2004.

41. D. A. Patterson. The Top 10 Innovations in the New NVIDIA Fermi Architecture, and the Top 3 Next Challenges. `http://www.nvidia.com/content/PDF/fermi_white_papers/D.Patterson_Top10InnovationsInNVIDIAFermi.pdf`, 2009.

42. J. Psota and A. Agarwal. rMPI: message passing on multicore processors with on-chip interconnect. In P. Stenstrom et al., editor, *High Performance Embedded Architectures and Compilers*, volume 4917 of *LNCS*, pages 22–37. Springer Berlin Heidelberg, 2008.

43. M. J. Quinn. *Designing Efficient Algorithms for Parallel Computers*. McGraw-Hill Book Company, New York, 1987.

44. S. Schneider, J. S. Yeom, and D. S. Nikolopoulos. Programming multiprocessors with explicitly managed memory hierarchies. *Computer*, 42:28–34, 2009.

45. M. Shah et al. UltraSPARC T2 – a highly-threaded, power-efficient, SPARC SOC. In *Solid-State Circuits Conference, ASSCC '07. IEEE Asian*, pages 22 –25, November 2007.

46. G. Shi, V. V. Kindratenko, I. S. Ufimtsev, T. J. Martinez, J. C. Phillips, and S. A. Gottlieb. Implementation of scientific computing applications on the cell broadband engine. *Scientific Programming*, 17:135–151, January 2009.

47. M. Snir et al. *MPI – The Complete Reference, Volume 1, The MPI Core*, volume 1. The MIT Press, second edition, 1999.

48. A.C.Sodan, J. Machina, A. Deshmeh, K. Macnaughton, and B. Esbaugh. Parallelism via multithreaded and multicore CPUs. *Computer*, 43(3):24 –32, march 2010.

49. F. Soltis. When Is PowerPC Not PowerPC? `http://systeminetwork.com`, 2002.

50. L. Spracklen and S.G. Abraham. Chip multithreading: opportunities and challenges. In *11th International Symposium on High-Performance Computer Architecture, HPCA-11*, pages 248 – 252, 2005.

51. The Green 500 List. `http://www.green500.org/`.

52. M. E. Thomadakis. The Architecture of the Nehalem Processor and Nehalem-EP SMP Platforms. `http://www.ece.tamu.edu/~tex/manual/node24.html`, 2011.

53. Tilera website. `http://www.tilera.com`.

54. UltraSPARC T2 architecture and performance modelings software tools. `http://www.opensparc.net/opensparc-t2/index.html`.

55. D. W. Wall. Limits of Instruction-Level Parallelism. Technical report, Digital Western Research Laboratory, 1993.

56. White paper. Looking Beyond Graphics. `http://www.nvidia.com/content/PDF/fermi_white_papers/T.Halfhill_Looking_Beyond_Graphics.pdf`.

57. W. Wolf, A. A. Jerraya, and G. Martin. Multiprocessor system-on-chip (MPSoC) technology. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(10):1701 –1713, october 2008.