LAPORAN PENELITIAN

"High Performance and Distributed Computing"

Didin Agustian P. Ph.D

INSTITUT TEKNOLOGI NASIONAL
BANDUNG - 2020

PART 3

Computer Science

11

High Performance and Distributed Computing

11.1. High performance computing

The recent developments in numerical modeling for geo-sciences are increasingly offering new approaches to study the complex and nonlinear processing involving Mother Earth, difficult or even impossible to scale down for studying in laboratory experiments. However, the developments of computer capabilities in terms of both computing resources and storage have opened the doors for large-scale parallel computations in Earth Sciences. This is leading to the use of increasing complex models where a greater number of tunable parameters can be taken into account.

High performance computing (HPC) most generally refers to the practice of aggregating computing power in a way that delivers much higher performance than we could get out of a typical desktop computer or workstation in order to solve large problems in science, engineering or business. In this section, we give an overview of the most common processor options used today for HPC.

The architecture of processors has changed dramatically during the last years, and this has a relevant impact not only on the hardware side and specially on the design of the processor itself, but also on the software side and in particular on how applications have to be programmed.

The evolution has been driven initially by the famous *Moore's Law*, an observation made in 1965 by Gordon Moore, co-founder of Intel, stating that the number of transistors double every 1.5 years (green line). This, combined with a

Chapter written by Sebastiano Fabio SCHIFANO, Eleonora LUPPI, Didin Agustian PERMADI, Thi Kim Oanh NGUYEN, Nhat Ha Chi NGUYEN and Luca TOMASSETTI.

constant shrinking of transistor size, translates in more hardware resources to integrate in a chip. Moreover, according to Dennard's law, as transistors get smaller, their power density stay constant, so that the power use stays in proportion with area, and both voltage and current scale downwards with the length of transistors. This roughly translates to the fact that decreasing the transistor size by a factor, the number of transistors increase by a factor of 2, the clock-speed increases by a factor and the energy used does not change. In summary, the increase in the performance comes from more hardware resources integrated into the chips and a constant increase in the clock frequency. However, Dennard's law is true as long as the leakage current is negligible. But, as the size of transistor becomes smaller and smaller, the leakage current becomes a relevant factor, and the chip can no longer dissipate the heat that accumulates when components operate shoulder-to-shoulder in a constricted area.

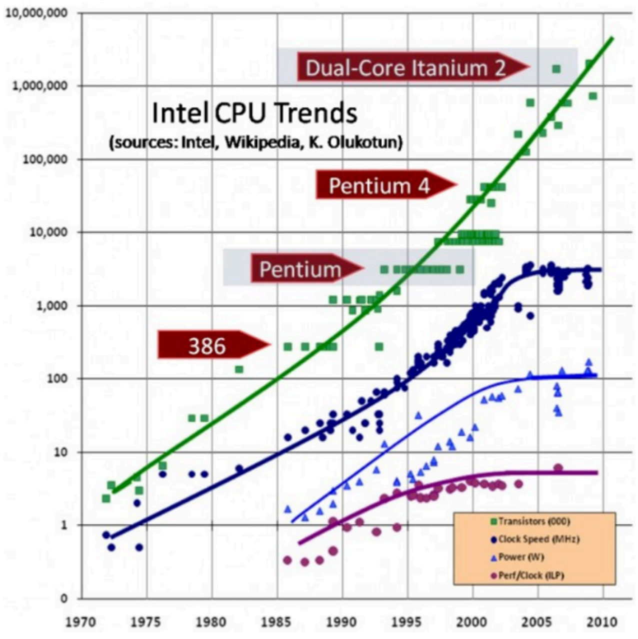

Figure 11.1. *Evolution of processors developed over the last decades. For a color version of this figure, see www.iste.co.uk/laffly/torus1.zip*

This happened around the year 2005, and has created the so-called power-wall limiting the operating clock frequency of chips to approximately 4 GHz. Since the computing performance of a processor is the result of the number of operations per clock cycle, times the frequency of the processor, this limitation seriously harmed the increase in computing performances. At this time, to overcome this limitation, the architecture of processors has been strongly changed introducing two important design changes: multi-core and vectorization. The first changes the architecture of processors from uni-core to multi-core. This means that into a single chip several full CPUs called core are integrated together, increasing the aggregate computing performance delivered. The latter change conversely introduces vector-units, giving to CPUs the capability to operate on vector operands increasing the number of operations performed in a clock cycle.

Recent development trends for multi-core processors are based on an increasing number of independent cores together with wider vector units within each core. Both changes allow us to increase the computing performance delivered by processors and to still scale according to the Moore’s law.

High performance computing (HPC) has seen in recent years an increasingly large role played by general purpose graphics processor unit (GP-GPU or GPU). GPU are processors originally designed and specialized to accelerate the creation and manipulation of images to output to a display device. GPUs have been popularized by their inventor, the NVIDIA corporation, that has evolved the architecture of these devices to also include capabilities to support general purpose programming. The hardware architecture of GPUs follows that of CPUs, sharing a common multicore architecture together with wide vector units.

In the next two sections, we analyze the architecture of most common CPUs, and GPUs, and of programming frameworks required to develop codes for these systems.

11.2. Systems based on multi-core CPUs

The typical HPC system for scientific computing today is a large cluster of nodes based on commodity multi-core CPUs and interconnected by a high-speed switched network, such as InfiniBand.

In recent years, multi-core architecture has become the standard approach to develop high performance processors. Several cores are integrated on a single chip device; this has allowed peak performance at node level to scale according to Moore’s law.

In many cases, the cores are x86-CPUs, usually including two levels of caches, and a floating-point unit able to execute vector instructions; the vector size increases steadily as newer processor versions are released.

Today, 256-bit and 512-bit vectors have been adopted in the Intel micro-architecture; cores within a device share a large L3-cache and external memory.

Many system vendors provide so-called multi-socket platforms, based on several multi-core CPUs. Typically, two, four or eight sockets can be installed on a motherboard; each socket has its own memory-bank and is connected to the other sockets by dedicated busses. Appropriate protocols allow us to share the global memory space and keep the content of caches coherent. From a programming point of view, these systems operate as symmetric multiprocessors (SMP), which means that they can be programmed as a single processor with Nc cores (Nc equal to the sum of cores of all sockets) sharing the full memory space and controlled by a single instance of the Linux operating system.

Effective performances heavily rely on the ability of programmers, compilers and run-time support to carefully exploit parallelism on all available hardware features, and are a combination of several factors:

$$P = f \times \#cores \times nInstrPerCycle \times nFlopPerInstr$$

where P is the performance, f the clock frequency, #cores the numbers of available cores, nInstrPerCycle the number of instructions executed per clock cycle and nFlopPerInstr the number of operations encoded on a single instruction.

Relevant optimization strategies to exploit performances of these systems are:

– core parallelism: the code should allow all cores to work in parallel exploiting MIMD or SPMD multi-task parallelism; in the first case, the application is decomposed in several sub-tasks and each one is executed by a different core; in the latter case, that can be also combined with the previous one, the data-set is typically partitioned among the cores, each one executing the same task;

– vector programming: each core has to process the data-set of the application using vector instructions and exploiting streaming-parallelism (SIMD). The number of data-items that can be processed by vector instructions depends on the architecture of the CPU; on the latest Sandybridge architecture the vector size is 256-bit, that is up to 8 single-precision or 4 double-precision floating point numbers;

– efficient use of caches: memory hierarchies of commodity systems are based on the concept of cache to minimize over-heads associated with accessing main memories. The application code has to exploit cache-reuse in order to save time in memory access; this may have a serious impact on the organization of the data-layout for the application;

– NUMA control: multi-socket platforms are non-uniform memory access (NUMA) systems; this means that the time to access data in memory is related to the allocation of the threads that perform the memory access; the time is shorter to access data stored onto memory attached to the socket where the thread is running.

11.2.1. *Systems based on GPUs*

GPUs are used as accelerators to upload and boost the execution time of some parts of applications. GPU devices are hosted into standard CPU cabinets and are physically attached to CPU processor through a PCI-e bus or more recently through the NVLINK bus developed by NVIDIA, one of major developers of GPUs.

NVIDIA GPUs are multi-core processors with each core capable of processing several operations in parallel. Core units are called streaming multiprocessors (SM) or extended streaming multiprocessors (SMX) on more recent systems, as they have enhanced capabilities. Each multiprocessor has several compute units called CUDA-cores in NVIDIA jargon. At each clock-cycle, multiprocessors execute multiple warps, with each warp being a group of operations called CUDA-threads processed in single instructions multiple threads (SIMT) fashion. SIMT execution is similar to SIMD but more flexible, and for example, different CUDA-threads of a SIMT group are allowed to take different branches at a performance penalty. At variance with CPU threads, context switches among active CUDA-threads are very fast, as these architectures maintain many thread states. Typically, one CUDA-thread processes one element of the data-set of the application. This helps exploit available parallelism of the algorithm, and hides latencies by switching between threads waiting for data coming from memory and threads ready to run. This structure has remained stable across all generations.

Table 11.1 summarizes only a few relevant parameters for several generations of NVIDIA GPUs.

	C2050 / C2070	K20X	K40	K80		P100	V100
GPU	GF100	GK110	GK110B	GK210	$\times$ 2	P100	V100
Number of SMs	16	14	15	13	$\times$ 2	56	80
Number of CUDA-cores	448	2688	2880	2496	$\times$ 2	3584	5120
Base clock frequency (MHz)	1.15	735	745	562		1328	1370
Base DP performance (Gflops)	515	1310	1430	935	$\times$ 2	4755	7000
Boosted clock frequency (MHz)	–	–	875	875		1480	1455
Boosted DP performance(Gflops)	–	–	1660	1455	$\times$ 2	5300	7500
Total available memory (GB)	3 / 6	6	12	12	$\times$ 2	16	16
Memory bus width (bit)	384	384	384	384	$\times$ 2	4096	4096
Peak mem. BW (ECC-off) (GB/s)	144	250	288	240	$\times$ 2	732	900
Max Power (Watt)	215	235	235	300		300	300

Table 11.1. *Some relevant parameters for several generations of NVIDIA GPUs*

The C2050 and C2070 boards based on the Fermi architecture differ in the amount of available global memory. The K20, K40 and K80 are boards based on Kepler processors. Several enhancements are available in the more recent Kepler and

Pascal architecture that have 256 32-bit registers addressable by each CUDA-thread (a 4X increase over Fermi) and each SMX has 65536 registers (a 2X increase). Kepler and Pascal GPUs are also able to increase their clock frequency beyond the nominal value (this is usually referred to as GPUBoost), if power and thermal constraints allow us to do so. The K40 processor has more global memory than the K20 and slightly improves memory bandwidth and floating-point throughput, meanwhile, the K80 has two enhanced Kepler GPUs with more registers and shared memory than K20/K40 and extended GPUBoost features. Up to now the only board based on the Pascal processor is the P100. The Tesla C2050 system has a peak performance of $\approx$ 1 Tflops in single-precision (SP), and $\approx$ 500 Gflops in double-precision (DP); on the Kepler K20 and K40, the peak SP (DP) performance is ≈ 5 Tflops (≈ 1.5 Tflops), while on the K80 the aggregate SP (DP) performance of the two GPUs is ≈ 5.6 Tflops (≈ 1.9 Tflops). The P100 delivers up to ≈ 10.5 Tflops (SP) and ≈ 5.3 (DP). Fast access to memory is strongly correlated with performance: peak bandwidth is 144 GB/s for the C2050 and C2070 processors, and 250 and 288 GB/s respectively for the K20X and the K40; on the K80, the aggregate peak is 480 GB/s, increased to 732 GB/s on the P100. The memory system has an error detection and correction system (ECC) to increase reliability when running large codes. This feature is usually always on, even though it slightly reduces available memory and bandwidth (e.g. on the Tesla C2050 available memory is reduced by ≈ 12 % and we measure a typical bandwidth cost $\approx 20 - 25\%$). The next generation NVIDIA GPU architecture Volta further increases the computing throughput to 7.5 Tflops DP, and the memory bandwidth to 900 GB/s, a factor 1.4X and 1.2X w.r.t. the Pascal architecture.

On GPUs, the native programming model is strongly based on data-parallel models, with one thread typically processing one element of the application data domain. The native language for NVIDIA GPUs is CUDA-C, together with OpenCL. Both languages have a very similar programming model, but use a slightly different terminology. For instance, on OpenCL the CUDA-thread is called work-item, the CUDA-block work-group, and the CUDA-kernel is a device program. A CUDA-C or OpenCL program consists of one or more functions that run either on the host, a standard CPU, or on a GPU. Functions that exhibits no (or limited) parallelism run on the host, while those exhibiting a large degree of data parallelism can go onto the GPU. The program is a modified C (or C++, Fortran) program including keyword extensions defining data parallel functions, called kernels or device programs. Kernel functions typically translate into a large number of threads, i.e. a large number of independent operations processing independent data items. Threads are grouped into blocks which in turn form the execution grid. The grid can be configured as a one-, two- or three-dimensional array of blocks. Each block is itself a one-, two- or three-dimensional array of threads, running on the same SM, and sharing data on a fast shared memory. When all threads complete their execution, the corresponding grid terminates. Since threads run in parallel with host CPU threads, it is possible to overlap in time processing on the host and on the accelerator.

```
#pragma acc data copyin(x), copy(y) {
  #pragma acc kernels present(x) present(y) async(1)
  #pragma acc loop vector(256)
  for (int i = 0; i < N; ++i)
    y[i] = a*x[i] + y[i];

  #pragma wait(1);
}
```

Box 11.1. *For a color version of this figure, see www.iste.co.uk/laffly/torus1.zip*

New programming approaches are now emerging, mainly based on directives, moving the coding abstraction layer at an higher level. These approaches should make code development easier on heterogeneous computing systems (OpenACC, 2016), simplifying the porting of existing codes on different architectures. OpenACC is one of such programming models, increasingly used by several scientific communities. OpenACC is based on pragma directives that help the compiler to identify those parts of the code that can be implemented as parallel functions and offloaded on the accelerator or divided among CPU cores. The actual construction of the parallel code is left to the compiler making, at least in principle, the same code portable without modifications across different architectures and possibly offering more opportunities for performance portability.

The listing below is a sample of OpenACC code computing a saxpy function on vectors x and y. The pragma clauses control data transfers between host and accelerator, and identify the code regions to be run on the accelerator.

The listing above shows an example of the saxpy operation of the Basic Linear Algebra Subprogram (BLAS) set coded in OpenACC. The pragma acc kernels clause identifies the code fragment running on the accelerator, while pragma acc loop specifies that the iterations of the for-loop should execute in parallel. The openACC standard defines many such directives, allowing a fine tuning of applications. As an example, the number of threads launched by each device function and their grouping can be tuned by the vector, worker and gang directives, in a similar fashion as setting the number of work-items and work-groups in CUDA. Data transfers between host and device memories are automatically generated, and occur on entering and exiting the annotated code regions. Specific data directives help to optimize data transfers, for example overlapping transfers and computation. For example, in listing 1, the clause copyin(ptr) copies the array pointed by ptr from host onto accelerator memory before entering the following code region; copy (ptr) copies it back to the host memory after leaving the code region. An asynchronous directive async instructs the compiler to generate asynchronous data transfers or device function executions; a corresponding clause (i.e. #pragma wait (queue)) awaits for completion. OpenACC is similar to the OpenMP (Open Multi-Processing) framework widely used to manage parallel codes on multi-core CPUs (Wienke *et al*. 2014); both frameworks are directive based, but OpenACC targets accelerators in general, while at this stage OpenMP targets mainly multi-core CPUs; the OpenMP4 standard has introduced directives to manage also accelerators, but currently, compiler support is still limited. Regular C/C++ or Fortran code, already developed and tested on traditional CPU architectures, can be annotated with OpenACC pragma directives (e.g. parallel or kernels clauses) to instruct the compiler to transform loop iterations into distinct threads, belonging to one or more functions to run on an accelerator. With the above-mentioned features, OpenACC is particularly well-suited for developing scientific HPC codes for several reasons: it is highly hardware agnostic, allowing us to target several architectures, GPUs and CPUs, allowing us to develop and maintain one single code version; the programming overhead to offload code regions to accelerators is limited to few pragma lines, in contrast to CUDA and, in particular OpenCL, verbosity; and the code annotated with OpenACC pragmas can still be compiled and run as plain C code, ignoring pragma directives.