CHAPTER 1

DESIGN AND OPTIMIZATION OF SCIENTIFIC APPLICATIONS FOR HIGHLY HETEROGENEOUS AND HIERARCHICAL HPC PLATFORMS USING FUNCTIONAL COMPUTATION PERFORMANCE MODELS

D. Clarke [1], A. Ilic [2], A. Lastovetsky [1], V. Rychkov [1], L. Sousa [2] Z. Zhong [1],

[1] School of Computer Science and Informatics, University College Dublin, Belfield, Dublin 4, Ireland

[2] INESC-ID, IST/Technical University of Lisbon, Rua Alves Redol, 9, 1000-029 Lisbon, Portugal

HPC platforms are getting increasingly heterogeneous and hierarchical. The main source of heterogeneity in many individual computing nodes is due to the utilization of specialized accelerators such as GPUs alongside general purpose CPUs. Heterogeneous many-core processors will be another source of intra-node heterogeneity in the near future. As modern HPC clusters become more heterogeneous, due to increasing number of different processing devices, hierarchical approach needs to be taken with respect to memory and communication interconnects to reduce complexity. During recent years, many scientific codes have been ported to multicore and GPU architectures. To achieve optimum performance of these applications on CPU/GPU hybrid platforms software heterogeneity needs to be accounted for. Therefore, design and implementation of data parallel scientific applications for such highly heterogeneous and hierarchical platforms represent a significant scientific and engineering challenge. This chapter will present the state of the art in the solution of this problem based on the functional performance models of computing devices and nodes.

Complex HPC.

1.1 Introduction

Highly heterogeneous and hierarchical HPC platforms, namely hardware-accelerated multicore clusters, are widely used in high performance computing due to better power efficiency and performance/price ratio. Introduction of multicores in HPC resulted in significant refactoring of existing parallel applications. For general-purpose computing on GPUs, new programming models, such as CUDA and OpenCL were proposed. A large number of algorithms and specific applications have been successfully ported to GPUs delivering substantial speedup over their optimized CPU counterparts. Transition to hybrid CPU/GPU architectures is challenging in the aspects of efficient utilization of the heterogeneous hardware and reuse of the software stack. In existing programming and execution environments for hybrid platforms, such as OpenCL, StarPU [1] and CHPS [2], the problem of efficient cross-device data partitioning and load balancing still remains.

We target data-parallel scientific applications, such as linear algebra routines, digital signal processing, computational fluid dynamics etc. They are characterized by divisible computational workload, which is directly proportional to the size of data and dependent on data locality. Computation kernels optimized for multicore and GPU are available for these applications. To execute such applications efficiently on hybrid multicore and multi-GPU platforms, workload has to be distributed unevenly between highly heterogeneous computing devices. Our target architecture is a dedicated heterogeneous CPU/GPU cluster, characterized by a stable performance in time and a complex hierarchy of heterogeneous computing devices. We consider such platform as a distributed-memory system, and therefore apply data partitioning, a load balancing method widely used on distributed-memory supercomputers.

Data partitioning algorithms, including those already proposed for hybrid platforms rely on performance models of processors. In [3], *a priori* found constants, representing the sustained performance of the application on CPU/GPU, were used to partition data. In [4], a similar constant performance model (CPM) was proposed, but it was built adaptively, using the history of performance measurements. The fundamental assumption of the data partitioning algorithms based on constant performance models is that the absolute speed of processors/devices does not depend on the size of a computational task. However, it becomes less accurate when (i) the partitioning of the problem results in some tasks fitting into different levels of memory hierarchy or (ii) processors/devices switch between different codes to solve the same computational problem.

An analytical predictive model was proposed in [5]. In contrast to others, this model is application-specific: the number of parameters and the predictive formulas for the execution time of processors/devices are defined for each application. This approach requires a detailed knowledge of the computational algorithm, in order to provide an accurate prediction. In [5], it was also acknowledged that the linear models might not fit the actual performance in the case of resource contention, and therefore, data partitioning algorithms might fail to balance the load.

In the work presented in this chapter, we apply data partitioning based on functional performance models, which was originally designed and proved to be accu-

rate for heterogeneous clusters of uniprocessor machines [6]. The functional performance model (FPM) represents the processor speed as a function of problem size. It is built empirically and integrates many important features characterizing the performance of both the architecture and the application. This model can be used with any data-parallel application and applicable in the situations (i) and (ii). In this work, we target CPU/GPU clusters, which consist of heterogeneous devices with separate memory and different programming models. Here we extend the functional performance model, originally designed for uniprocessors, to these platforms, and demonstrate how to design and optimize parallel scientific applications using FPM-based data partitioning.

Although hardware accelerators are significantly faster than traditional multicores, the computing power of the mulitcores should not be ignored. To obtain maximum performance from a parallel scientific application on an accelerated heterogeneous platform, the workload needs to be distributed over the hierarchy of devices. Evaluation of device performance is complicated by resource contention and device-specific limitations (for example, limited GPU memory). Furthermore, multiple computation kernels optimized for different devices and based on different programming models need to be used simultaneously.

In previous work [7], we proposed a method for building functional performance models of multicore nodes, which takes into account resource contention. The FPMs built this way were used for inter-node data partitioning. In this chapter we present how to apply this approach to a hierarchical system that consists of several multicore sockets coupled with GPUs. A GPU is controlled by a host process that handles data transfer between the host and device, and instructs the GPU to execute kernels. In this work, we measure the speed of the host process and build the performance model for the GPU coupled with its dedicated core, which includes the contributions from the kernel running on GPU and from the memory transfers. In general, this model can be defined only for the range of problem sizes that fit the local memory of the GPU. It can be extended to infinity for out-of-core applications, which can handle a large amount of data stored in low-speed memory [8].

Functional performance models are hardware and application specific and are built by empirically benchmarking the kernel. Building accurate models for the full range of problem sizes is expensive. This approach is not suitable for applications that will be run a small number of times on a given platform, for example, in grid environments, where different processors are assigned for different runs of the application. Such applications should be able to optimally distribute computations between the processors of the executing platform assuming that this platform is different and *a priori* unknown for each run of the application. In [9], we proposed an algorithm which efficiently builds only the necessary parts of the speed functions (partial FPMs) to the required level of accuracy in order to achieve load balancing. In this work, we present an adaptation of this algorithm to hierarchical platforms [10].

In order to demonstrate how to design and optimize scientific applications for highly heterogeneous and hierarchical HPC platforms using functional computation performance models, we modify parallel matrix multiplication to be used with FPM-based data partitioning. We show how to extract and benchmark the computation

kernel of the application on different devices. The kernel will call the hardware-specific optimized code for each device (BLAS GEMM). We design a hierarchical data partitioning scheme, which allows for nested parallelism, and apply FPM-based data partitioning algorithms. This method was developed in collaboration between University College Dublin (Ireland) and Technical University of Lisbon (Portugal), funded by the ComplexHPC COST action IC0805.

This chapter is structured as follows. In Sect. 1.2, we review related work and conclude that data partitioning algorithms are more suited for balancing data-parallel scientific applications on heterogeneous platforms, but they may fail on highly heterogeneous hierarchical platforms if simplistic performance models are used. However, data partitioning based on the functional performance model, described in Sect. 1.3, can be applied successfully on such platforms. The main contribution of this chapter is the adaptation of the FPM-based data partitioning to hybrid CPU/GPU nodes and clusters. We demonstrate how to design a scientific application to make use of FPM-based data partitioning on a heterogeneous hierarchical platform. More specifically, we use the well-known parallel matrix multiplication application, which is presented in Sect. 1.4. Building performance models for such an application on heterogeneous devices is challenging. In Sect. 1.5–1.7, we present a solution for a hybrid multi-CPU/GPU node. Building full functional performance models can be expensive and hierarchical platforms add extra complexity to this, so much so as to prohibit the use of full models. Fortunately, we have developed an efficient method that builds the models to sufficient level of accuracy in the relevant range of problem sizes (partial FPM) as summarized in Sect. 1.8. Partial FPMs were originally designed for heterogeneous uniprocessors; another contribution of this chapter is the application of partial FPMs to hierarchical platforms. In Sect. 1.9, we design a hierarchical version of the matrix multiplication application for hybrid clusters. In this application, we use hierarchical data partitioning scheme and partial FPMs of devices and nodes.

1.2 Related Work

In this section, we review a number of algorithms for load balancing of parallel scientific and engineering problems on heterogeneous platforms. The older algorithms referenced were designed for either heterogeneous networks of workstations or shared-memory supercomputers. The newer algorithms target hybrid CPU/GPU platforms.

Static algorithms, for example, those based on data partitioning [3, 6, 11, 5], use a priori information about the parallel application and platform. This information can be gathered either at compile-time or run-time. Static algorithms are also known as predicting-the-future because they rely on accurate performance models as input to predict the future execution of the application. Static algorithms are particularly useful for applications where data locality is important because they do not require data redistribution. However, these algorithms are unable to balance on non-dedicated platforms, where load changes with time, and for applications with

non-deterministic workload. ***Dynamic*** algorithms, such as task queue scheduling and work stealing [12, 13, 14, 15, 16, 17] balance the load by moving fine-grained tasks between processors during the calculation. Dynamic algorithms do not require a priori information about execution but may incur significant communication overhead on distributed-memory platforms due to data migration. Dynamic algorithms often use static data partitioning for their initial step to minimize the amount of data redistributions needed. For example, in the state-of-the-art load balancing techniques for multi-node, multicore, and multi-GPU platforms, the performance gain is mainly due to better initial data partitioning. It was shown that even the static distribution based on simplistic performance models (single values specifying the maximum performance of a dominant computational kernel on CPUs and GPUs) improves the performance of traditional dynamic scheduling techniques by up to 250% [18].

In this work we focus on parallel scientific applications, where computational workload is directly proportional to the size of data and dependent on data locality. The general scheme of such applications can be summarized as follows: (i) all data is partitioned over the processors, (ii) some independent calculations are carried out in parallel, and (iii) some data synchronization takes place. Our target architecture is a dedicated heterogeneous distributed-memory HPC platform, such as heterogeneous clusters, interconnected clusters, multicores with GPU and FPGA accelerators. These HPC platforms have the following features: (i) the performance of the application is stable in time and is not affected by varying system load; (ii) there is a significant overhead associated with data migration between computing devices; (iii) optimized architecture-specific libraries implementing the same kernels may be available for different computing devices. On these platforms, for most scientific applications, static load balancing algorithms outperform dynamic ones because they do not involve data migration. Therefore, for this type of applications, we find that centralized static algorithms, such as data partitioning, are the most appropriate.

Most of the state of the art data partitioning algorithms [3, 4, 11, 19, 20, 21, 22] make the assumption that the speed of a process does not change with problem size, and hence are based on constant performance models (CPM). In [4] they make redistribution decisions based on the average of recorded times of the previous iterations. In [3] they acknowledge performance changes with problem size, however for their partitioning calculations they use single-value sustained performance of computational kernel on devices. In [11] they use a linear model of time, which is equivalent to constant speed.

The fundamental assumption of the conventional CPM-based algorithms is that the absolute speed of the processors does not depend on the size of the computational task. This assumption is typically satisfied when medium-sized scientific problems are solved on a heterogeneous network of workstations. However, it becomes much less accurate in the following situations: (i) The partitioning of the problem results in some tasks either: not fitting into the available memory of the assigned device and hence causing out-of-core computations; or fully fitting into faster levels of its memory hierarchy. (ii) Some processing devices involved in computations are not traditional general-purpose processors (say, accelerators such as GPUs or specialized cores). In this case, the relative speed of a traditional processor and a non-traditional

one may differ for two different sizes of the same computational task even if both sizes fully fit into the available memory. (iii) Different devices use different codes to solve the same problem locally.

The above situations become more and more common in modern and especially perspective high-performance heterogeneous platforms. As a result, applicability of the traditional CPM-based distribution algorithms becomes more restricted. Indeed, if we consider two really heterogeneous processing devices P_i and P_j, then the more different they are, the smaller will be the range R_{ij} of sizes of the computational task where their relative speeds can be accurately approximated by constants. In the case of several different heterogeneous processing devices, the range of sizes where CPM-based algorithms can be applied will be given by the intersection of these pair-wise ranges, $\bigcap_{i,j=1}^{p} R_{ij}$. Therefore, if a high-performance computing platform includes even a few significantly heterogeneous processing devices, the area of applicability of CPM-based algorithms may become quite small or even empty. For such platforms, new algorithms are needed that would be able to optimally distribute computations between processing devices for the full range of problem sizes.

The functional performance model (FPM) has proven to be more realistic than the constant performance model, because it integrates many important features, such as the hardware and software heterogeneity, the heterogeneity of memory structure, the effects of paging and so on [6]. FPMs can be used for data partitioning between devices within a CPU/GPU node [8], we present how this can be done in Sect. 1.7. Moreover, we also proposed several FPM-based load balancing approaches for multicore CPU and multi-GPU environments, which are capable of exploiting the systems capabilities at multiple levels of parallel execution [23, 24]. For a cluster of hybrid nodes, hierarchical partitioning can be advantageous because it works well in scheduling divisible workload. For example, hierarchical scheduling was shown to be efficient for load balancing on homogeneous [25] and heterogeneous [12] hybrid clusters. In this work, we will demonstrate how FPMs can be used for hierarchical data partitioning.

1.3 Data Partitioning Based on Functional Performance Model

The functional performance model [6] is application and hardware specific. It is associated with a process executing the application on the particular piece of hardware. Under the functional performance model, the speed of each process is represented by a continuous function of the problem size. The speed is defined as the number of computation units performed by the process per second. The ***computation unit*** is the smallest amount of work that can be given to a process. All units require the exact same number of arithmetic calculations and have the same input and output data storage requirements. The computation unit can be defined differently for different applications. The compute time for a fixed amount of computation units on a given process must remain constant.

Performance models consist of a series of speed measurements taken over a range of problem sizes. The speed is found experimentally by measuring the execution

time. This can be done by benchmarking the full application for each problem size. However, since computationally intensive applications often perform the same core computation multiple times in a loop, a benchmark made of one such core computation will be representative of the performance of the whole application. We call this core computation, which performs much less computations but is still representative of the application, a ***kernel***. If the nature of the application allows, FPMs can be built more efficiently by only benchmarking the kernel.

The problem of data partitioning using functional performance models was formulated in [6] as follows. In this formulation, a total problem size n is given as the number of computation units to be distributed between p ($p < n$) processes $P_1, \ldots, P_p$. The speed of each process is represented by a positive continuous function of problem size $s_1(x), \ldots, s_p(x) : s_i(x) = x/t_i(x)$, where $t_i(x)$ is the execution time of processing x units on the processor i. Speed functions are defined at $[0, n]$. The output of the algorithm is a distribution of computation units, $d_1, \ldots, d_p$, so that $d_1 + d_2 + \ldots + d_p = n$. Load balancing is achieved when all processors execute their work at the same time: $t_1(d_1) \approx t_2(d_2) \approx \ldots \approx t_p(d_p)$. This can be expressed as:

$$\begin{cases} \dfrac{d_1}{s_1(d_1)} \approx \dfrac{d_2}{s_2(d_2)} \approx \ldots \approx \dfrac{d_p}{s_p(d_p)} \\ \\ d_1 + d_2 + \ldots + d_p = n \end{cases} \tag{1.1}$$

The solution of these equations, $d_1, \ldots, d_p$, can be represented geometrically by intersection of the speed functions with a line passing through the origin of the coordinate system. This is illustrated in Fig. 1.1 for $p = 4$.

The geometrical algorithm solving this data partitioning problem was proposed in [6] and can be summarized as follows. Any line passing through the origin and intersecting the speed functions represents an optimum distribution for a particular problem size. Therefore, the space of solutions of the data partitioning problem consists of all such lines. The two outer bounds of the solution space are selected as the starting point of algorithm. The upper line, U, represents the optimal data distribution $d_1^u, \ldots, d_p^u$ for some problem size $n_u < n$, $n_u = d_1^u + \ldots + d_p^u$, while the lower line, L, gives the solution $d_1^l, \ldots, d_p^l$ for $n_l > n$, $n_l = d_1^l + \ldots + d_p^l$. The region between two lines is iteratively bisected by new lines B_k. At the iteration k,

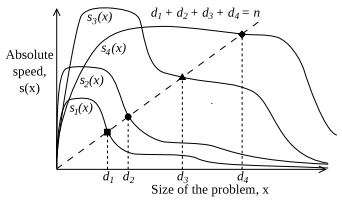

Figure 1.1 Optimal data distribution: the number of computation units is geometrically proportional to the speeds of the processors.

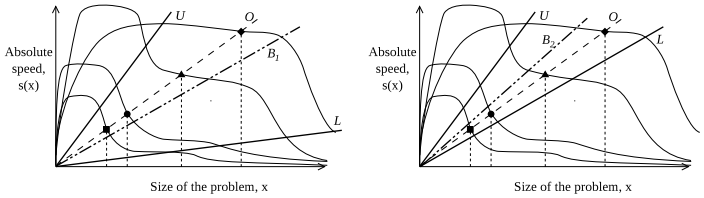

Figure 1.2 Two steps of the iterative geometrical data partitioning algorithm. The dashed line O represents the optimal solution. (a) Upper U and lower L represent the outer bounds of the solution space. Line (B_1) represents the first bisection. (b) Line B_1 becomes L. Solution space is bisected by line B_2, which, at the next step, will become U. Through this method the partitioner converges on the solution.

the problem size corresponding to the new line intersecting the speed functions at the points $d_1^k, \ldots, d_p^k$ is calculated as $n_k = d_1^k + \ldots + d_p^k$. Depending on whether n_k is less than or greater than n, this line becomes a new upper or lower bound. Making n_k close to n, this algorithm finds the optimal partition of the given problem $d_1, \ldots, d_p$: $d_1 + \ldots + d_p = n$. Fig. 1.2 illustrates the work of the bisection algorithm. Correctness proof and complexity analysis of this algorithm are presented in [6].

In the following section, we present a typical computationally intensive parallel application, and define its computation unit and kernel.

1.4 Example Application: Heterogeneous Parallel Matrix Multiplication

In this section, we describe a column-based heterogeneous modification [26] of the two-dimensional blocked matrix multiplication [27]. It will be used in subsequent sections to demonstrate how to design parallel scientific applications for heterogeneous hierarchical platforms, using the proposed data partitioning algorithms.

Parallelism in this application is achieved by slicing the matrices, with a one-to-one mapping between slices and processes. For efficiency and scalability, the application uses two-dimensional slicing of the matrices. The general solution for finding the optimum matrix partitioning for a set of heterogeneous processors has been shown to be NP-complete [20]. By applying a column-based matrix partitioning restriction, an algorithm with polynomial complexity can be used to find optimum partitioning [28]. In this algorithm each process is responsible for calculations associated with a rectangular submatrix. These rectangles are arranged into columns and the area of the rectangles is proportional to the speed of the device upon which the process is running (Fig. 1.3(a)). A communication minimizing algorithm [20] uses this column-based partitioning and finds the shape and ordering of these rectangles such that the total volume of communication for parallel matrix multiplication is minimized.

The application performs the matrix multiplication $C = A \times B$. Without loss of generality we will work with square $N \times N$ matrices and we assume that N

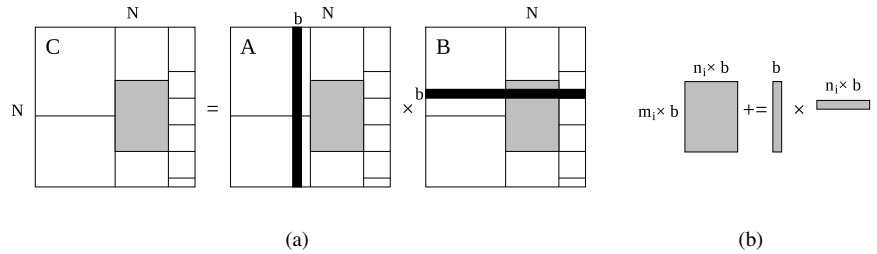

Figure 1.3 Heterogeneous parallel column-based matrix multiplication. (a) One step of the algorithm. (b) Computational kernel.

is a multiple of the blocking factor b. Dense matrices A, B and C are partitioned into p submatrices A_i, B_i, C_i, each of height bm_i and width bn_i, where p is the number of processes, m_i and n_i are the height and width of submatrices in blocks (Fig. 1.3(b)). The application consists of N/b iterations, with a column of blocks (the pivot column) of matrix A being communicated horizontally, and a row of blocks (the pivot row) of matrix B being communicated vertically. The pivot column and row move horizontally and vertically respectively with each iteration. If process i, with submatrix A_i, forms part of the pivot column, then it will send its part of the pivot column horizontally. If process i with submatrix B_i forms part of the pivot row, then it will send its part of the pivot row vertically. At each iteration all processes will receive into a buffer $A_{(b)}$ of size $bm_i \times b$ and $B_{(b)}$ of size $b \times bn_i$. Then the following GEMM operation is then performed by all processes in parallel: $C_i = C_i + A_{(b)} \times B_{(b)}$. This forms the computation kernel of the application.

For this kernel, we define the computation unit as an update of a $b \times b$ block of C_i. The amount of computations to update the whole *i*-th rectangle is equal to its area measured in these units $d_i = m_i \times n_i$. The communication minimizing algorithm [20] arranges the processes into columns and sets the rectangles' dimensions (m_i, n_i) using the optimal areas of the rectangles $d_1, \ldots, d_p$ provided as input. The computational kernel is representative of the execution of one iteration of the application on a given processor while being independent of the performance of the processors neighbors.

While porting this application to hierarchical multicore multi-GPU platforms, we face the following challenges:

- On a hybrid node, parallel processes interfere with each other due to sharing resources. The speed of individual devices cannot be measured independently. However, the devices can be divided into groups so that there is no significant interference between the groups. The functional performance model of a group of devices can be defined. For example, on a multi-socket node, a model of a socket can be built instead of the models of individual CPU cores.

- Interactions between CPUs and GPUs include: data transfers between the host and GPU over PCI Express; launching of GPU kernels; and other operations.

The effect on performance by these interactions is included in the functional performance model by designing an appropriate kernel for benchmarking.

- In order to achieve load balancing between hybrid nodes, the partitioning algorithm has to account for the hierarchy of processing devices, and hence, functional performance models need to be defined for each level of hierarchy.

We address these challenges in the following sections and demonstrate how to use FPM-based data partitioning on hierarchical multicore and multi-GPU platforms, using the example application.

1.5 Performance Measurement on CPUs/GPUs System

Here we apply the approach proposed in previous work [7] to measure the performance of multiple cores in a system by taking into account the resource contention between parallel processes. Each GPU is assumed to be controlled by a dedicated CPU core, which instructs the kernel execution on the GPU and handles the data transfers between them; therefore, we measure the combined performance of the GPU with its dedicated core, which includes the contributions from the kernel running on GPU and the memory transfers.

Our experimental system is a hybrid multicore and multi-GPU node of NUMA architecture, consisting of 4 sockets of six-core AMD processors with 16 GB memory each and accelerated by 2 different GPUs (Table 1.1). We used the GEMM kernel from ACML and CUBLAS for CPU and GPU respectively.

Our approach to performance measurement on heterogeneous multicore and multi-GPU system can be summarized as follows. (i) Since automatic rearranging of the processes provided by operating system may result in performance degradation, processes are bound to cores. (ii) Processes are synchronized to minimize the idle computational cycles, aiming at the highest floating point rate for the application. Synchronization also ensures that the resources will be shared between the maximum number of processes, generating the highest memory traffic. (iii) To ensure the reliability of the measurement, experiments are repeated multiple times until the results are statistically reliable.

Table 1.1 Specifications of the hybrid system *ig.icl.utk.edu*

	CPU (AMD)	GPUs (NVIDIA)	
Architecture	Opteron 8439SE	GF GTX680	Tesla C870
Core Clock	2.8 GHz	1006 MHz	600 MHz
Number of Cores	4×6 cores	1536 cores	128 cores
Memory Size	4×16 GB	2048 MB	1536 MB
Mem. Bandwidth		192.3 GB/s	76.8 GB/s

First, we measured the execution time of the kernel on a single and multiple CPU cores. We observed that the speed of a core depended on the number of cores executing the kernel on the same socket, because they compete for shared resources. However, the performance of the core was not affected by the execution of the kernel on other sockets, due to the NUMA architecture and a large capacity of memory. Therefore, we can accurately measure the time and, hence, the speed of a socket executing the same kernel simultaneously on its cores. This approach realistically reflects the performance of parallel applications designed for multicores.

Next, we experimented with the kernel on a GPU, with one core being dedicated to the GPU, and other cores on the same socket being idle. Since the kernel does not provide data transfer between the host and device, we implemented sending/receiving of matrices and measured the combined execution time on the dedicated core. Communication operations with GPU take a large proportion of the whole execution time for most applications [29], therefore, the time measured this way realistically reflects the performance of the kernel. This approach allows us to measure the speed of a single GPU.

Finally, we simultaneously executed the GEMM kernels on both a GPU and the cores located on the same socket. The cores, except for one dedicated to the GPU, executed the ACML kernel. The dedicated core and the GPU executed the CUBLAS kernel. The amounts of work given to the CPUs and the GPU were proportional to their speeds obtained from the previous experiments for a single core and for a single GPU. This may be not very accurate but realistic distribution of workload, which reflects the hybrid parallel applications. We measured the execution time on all cores and observed that the performance of the GPU dropped by 7–15% because of resource contention, while the CPU cores were not so much affected by the GPU process. In this experiment, we exploited the distributed-memory feature of the hybrid architecture. Namely, having received the data from the host memory, the GPU performed the computations in its local memory, and the dedicated core did not compete with other cores for resources. This observation allows us to measure the speed of multiple cores and GPUs independently with some accuracy.

1.6 Functional Performance Models of Multiple Cores and GPUs

We build the functional performance models of the parallel matrix multiplication application described in Sect. 1.4 for multiple cores and GPUs respectively, using the representative computational kernel of the application. Since the speed of the kernel for a given matrix area x does not vary with the nearly square shapes of submatrices [26], we build the speed functions by timing the kernel with the submatrices of size $\sqrt{x} \times \sqrt{x}$.

Speed functions of multiple cores: $s_c(x)$. These functions approximate the speed of a socket executing the ACML kernels simultaneously on c cores, with the problem size (matrix area) x/c on each core.

Speed functions of GPUs: $g_1(x)$, $g_2(x)$. Each function approximates the combined performance of a GPU and its dedicated core, while the GPU executing the CUBLAS kernel, with the problem size (matrix area) x.

Figure 1.4(a) shows the FPMs of a socket, $s_5(x)$ and $s_6(x)$, executing the ACML kernel on 5 and 6 cores simultaneously. The maximum performance of the socket is observed when all cores are involved in computations. It does not increase linearly with the number of active cores, because of resource contention. Additionally the performance depends on the blocking factor b, an application parameter. To exploit optimizations implemented in the ACML kernel, which take into account memory hierarchy of a multicore architecture, we experimented with $b = 640$.

In Fig. 1.4(b), the speed functions built for different modifications of the kernel on GeForce GTX680 are presented. The speed was measured on a dedicated core, while other cores stayed idle. In ***version 1***, the pivot column $A_{(b)}$ and row $B_{(b)}$ and the submatrix C_i are stored in the host memory. Before the execution of GEMM on the device, the pivot column and row are transferred to the device. After the execution, the updated submatrix is transferred back from the device. Therefore, the speed of the first version includes all transfers between the host and device memory.

In the application, the kernel is executed multiple times with different pivot columns and rows, updating the same submatrix C_i. Therefore, the submatrix can be stored in the device memory, accumulating the intermediate results. The transfer of C_i can be excluded from the kernel and from the speed measurements. In ***version 2***, submatrix C_i is stored and intermediate results are accumulated in the device until the device memory is exceeded. As shown in Fig. 1.4(b), the performance doubles when problem sizes fit in the GPU memory. After that, it splits the pivot column $A_{(b)}$ and row $B_{(b)}$ and the submatrix C_i into rectangles that fit the device memory and performs the CUBLAS GEMM multiple times to update these rectangles serially (see Fig. 1.5(a)). This implementation requires multiple transfers of the rectangles of the

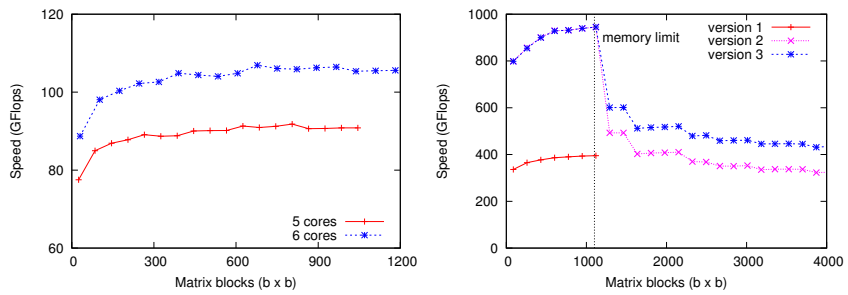

Figure 1.4 (a) Speed functions of a socket, $s_5(x)$ and $s_6(x)$, with blocking factor $b = 640$. (b) Speed functions of GeForce GTX680 ($b = 640$) built for kernels accumulating the intermediate results in the host memory (version 1); in device memory with out-of-core extension (versions 2); with overlapping of communications and computations (version 3).

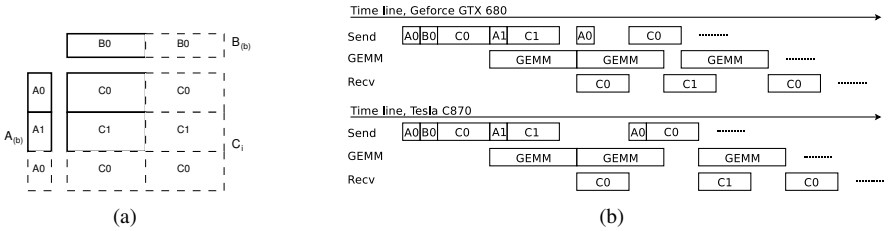

Figure 1.5 (a) Out-of-core implementation of the kernel on GPU. (b) Concurrent data transfers and kernel executions on GPUs.

submatrix C_i to and from the device memory, which explains the performance drop in the range of large problem sizes.

In ***version 3***, another out-of-core implementation of the kernel, we use the concurrency feature of NVIDIA GPUs on top of version 2. This feature enables to perform multiple CUDA operations simultaneously and, hence, to overlap communications with computations on host and device. In addition, modern generations of NVIDIA GPUs, such as GeForce GTX680, support concurrent data transfers to and from the device memory. As shown in Fig. 1.5(a), five buffers are allocated in the device memory, using its maximum capacity: A0 and A1 for rectangles of the pivot column $A_{(b)}$, B0 for the pivot row $B_{(b)}$, C0 and C1 for the submatrix C_i. Overlapping communications and computations in the out-of-core version of the kernel is illustrated in Fig. 1.5(b). In the beginning of each column, the first rectangles of the pivot column and row and the submatrix are sent to the buffers A0, B0 and C0. While GEMM is executed with these buffers, the next rectangles of the pivot column and the submatrix are sent to A1 and C1. Next, three operations are overlapped. (i) The rectangle of the submatrix updated during the previous execution of GEMM is transferred from C0 to the host memory. (ii) GEMM is executed with the new rectangles of the pivot column and the submatrix, using the buffers A1, B0, C1. (iii) The next rectangles of the pivot column and the submatrix are sent to A0 and C0. On the Tesla C870, which supports only one DMA engine, the latter operation is performed after (i) is complete (see Fig. 1.5(b)). We can see from Fig. 1.4(b) that the performance of GeForce GTX680 improves by around 30% when using overlapping. Based on our experiments, the speed function shapes of Tesla C870 are similar to GeForce GTX680's. However, there is less performance improvement from overlapping because Tesla C870 does not support concurrent data transfers.

1.7 FPM-Based Data Partitioning on CPUs/GPUs System

Table 1.2 shows the execution time of the heterogeneous matrix multiplication application [26] measured on different configurations of the hybrid system. The experiments were performed for square matrices with blocking factor $b = 640$. The first column shows the matrix size $n \times n$ in square blocks of 640×640. Column 2 shows

Table 1.2 Execution time of parallel matrix multiplication

Matrix (blks)	CPUs (sec)	GTX680 (sec)	Hybrid-FPM (sec)
40×40	99.5	74.2	26.6
50×50	195.4	162.7	77.8
60×60	300.1	316.8	114.4
70×70	491.6	554.8	226.1

the application execution time for the homogeneous matrix distribution between 24 CPU cores. Column 3 shows the execution time on GeForce GTX680 and a dedicated core. The last column shows the execution time for the heterogeneous matrix distribution between 22 CPU cores and 2 GPUs, with the rest 2 CPU cores being dedicated to GPUs. The distribution was obtained from the FPM-based data partitioning algorithm with the speed functions of 2 GPUs, $g_1(x)$, $g_2(x)$, 2 sockets with 5 active cores, $2 \times s_5(x)$, and 2 sockets with 6 active cores, $2 \times s_6(x)$. GeForce GTX680 outperforms 24 CPU cores when the problem fits in the device memory. When the problem exceeds the device memory, CPUs perform better. Functional performance models capture these variations, and therefore, the FPM-based data partitioning algorithm successfully distributes computations for all problem sizes, and the application delivers high performance.

In this section, we presented how to model performance of devices and how to use these models to find the optimal data partitioning within a node. On hardware-accelerated multicore clusters, there is a two-level hierarchy consisting of nodes and devices. To enable FPM-based data partitioning in this case, we need to introduce the model of a node. This model has to represent the optimal performance of the node, which is achieved by balancing the load between its internal devices. For each point in the node model, it is necessary to build the models of devices and perform data partitioning. Hence, the cost of building the node model may be prohibitively high.

One method of building the functional performance models efficiently is to estimate them at run-time, only in some region, with a sufficient degree of accuracy [9]. We refer to these estimates as the partial functional performance models. In the following section, we give a brief description of this method, which was originally designed for heterogeneous uniprocessor clusters of workstations. Then, in Sect. 1.9, we use the partial models to reduce the cost of building the two-level hierarchical models. We redesign the example application to use hierarchical data partitioning based on partial models of devices and nodes.

1.8 Efficient Building of Functional Performance Models

Functional performance models are hardware and application specific and are built empirically by benchmarking the kernel for a range of problem sizes. The accuracy

of the model depends on the number of experimental points used to build it. Despite the kernel being lightweight, building the full model can be very expensive. The applicability of FPMs built for the full range of problem sizes is limited to parallel applications executed many times on stable in time heterogeneous platforms. In this case, the time of construction of the full FPMs can become very small compared to the accumulated performance gains during the multiple executions of the optimized application. However, this approach is not suitable for applications that will be run a small number of times on a given platform, for example, in grid environments, where different processors are assigned for different runs of the application, or in hierarchical platforms, where the performance of a node depends not only on the workload assigned to the node but also on the distribution of this workload between the processing devices on the node. Such applications should be able to optimally distribute computations between the processors of the executing platform assuming that this platform is different and *a priori* unknown for each run of the application.

Partial estimates of the full speed functions can be built dynamically at application run-time to a sufficient level of accuracy to achieve load balancing [9, 30]. We refer to these approximations as partial functional performance models. The partial FPMs are based on a few points connected by linear segments and estimate the real functions in detail only in the relevant regions: $\bar{s}_i(x) \approx s_i(x)$, $1 \leq i \leq p$, $\forall x \in [a, b]$. Both the partial models and the regions are determined at runtime.

The algorithm to build the partial FPMs is iterative and alternates between (i) benchmarking the kernel on each process for a given distribution of workload and (ii) repartitioning the data. At each iteration, the current distribution $d_1, \ldots, d_p$ is updated, converging to the optimum, while the partial models $\bar{s}_1(x), \ldots, \bar{s}_p(x)$ become more detailed. Initially the workload is distributed evenly between all processes. Then the algorithm iterates as follows:

1. The time to execute the kernel for the current distribution is measured on each process. If the difference between timings is less than some ϵ, the current distribution solves the load balancing problem and the algorithm stops.

2. The speeds are calculated from the execution times and the points (d_i, s_i) are added to the corresponding partial models $\bar{s}_i(x)$ (Fig. 1.6 (b, d, f)).

3. Using on the current partial estimates of the speed functions, the FPM-based partitioning algorithm calculates a new distribution (Fig. 1.6 (a, c, e)).

This algorithm allows for efficient load balancing and suitable for use in self-adaptable applications, which run without a priori information about the heterogeneous platform. In the next section, we use the partial functional performance models for data partitioning on hierarchical platform.

1.9 FPM-Based Data Partitioning on Hierarchical Platforms

In this section our target platform is a two level hierarchical heterogeneous cluster of CPU/GPU compute nodes. This distributed platform can be described as having q

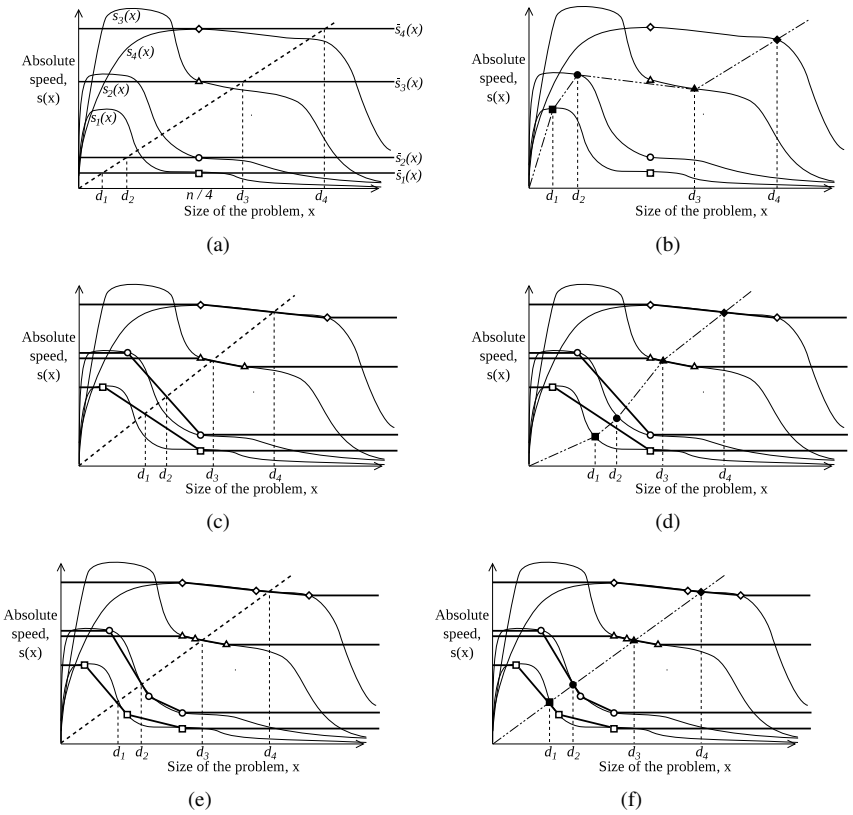

Figure 1.6 Steps of the partial FPM-based data partitioning algorithm illustrated using four heterogeneous processors.

nodes, $Q_1, \ldots, Q_q$, where a node Q_i has p_i devices, $P_{i1}, \ldots, P_{ip_i}$. The problem to be solved by this algorithm is to partition a matrix between these nodes and devices with respect to the performance of each of these processing elements. The hierarchical partitioning algorithm is iterative and converges towards an optimum distribution which balances the workload. It consists of two iterative algorithms, *inter-node partitioning algorithm (INPA)* and *inter-device partitioning algorithm (IDPA)*. The IDPA algorithm is nested inside the INPA algorithm [10].

As a demonstration how to optimize scientific applications on the target platform we take the heterogeneous parallel matrix multiplication application described in Sect. 1.4 and make the following modifications. Since the platform has two levels of hierarchy, we create a two-level partitioning scheme. The matrix is partitioned between the nodes (Fig. 1.7a) and then sub-partitioned between the devices within each node (Fig. 1.7b). This gives the application nested parallelism.

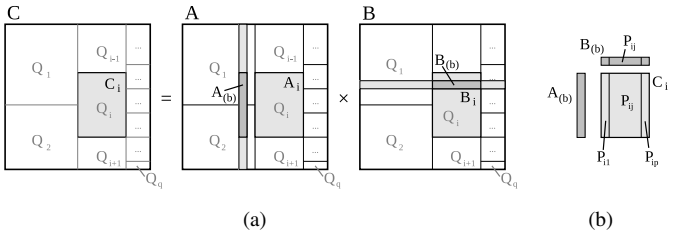

Figure 1.7 Parallel matrix multiplication algorithm: (a) two-dimensional blocked matrix multiplication between the nodes; (b) one-dimensional matrix multiplication within a node.

There is a total of W computation units to be distributed, where $W = (N/b) \times (N/b)$. The INPA partitions the total matrix into q submatrices to be processed on each heterogeneous computing node. The submatrix owned by node Q_i has an area equal to $w_i \times b \times b$, where $w_1 + \ldots + w_q = W$. The FPM partitioning algorithm (FPM-PA), from Sect. 1.3 uses, experimentally built speed functions to calculate a load balanced distribution $w_1, \ldots, w_q$. The shape and ordering of these submatrices is calculated by the *communication minimizing algorithm* (CMA) [20]. The CMA uses column-based 2D arrangement of nodes and outputs the heights bm_i and widths bn_i for each of the q nodes, such that $m_i \times n_i = w_i$, $bm = b \times m$ and $bn = b \times n$ (Fig. 1.8a). This two-dimensional partitioning algorithm uses a column-based arrangement of processors. The values of m_i and n_i are chosen so that the column widths sum up to N and heights of submatrices in a column sum to N.

The IDPA iteratively measures, on each device, the time of execution of the application specific core computational kernel with a given size while converging to a load balanced inter-device partitioning. It returns the kernel execution time of the last iteration to the INPA. IDPA calls the FPM-PA to partition the submatrix owned by Q_i into vertical slices of width d_{ij}, such that $d_{i1} + \ldots + d_{ip} = bn_i$ (Fig. 1.8b) to be processed on each device within the node. Device P_{ij} will be responsible for doing matrix operations on $bm_i \times d_{ij}$ matrix elements.

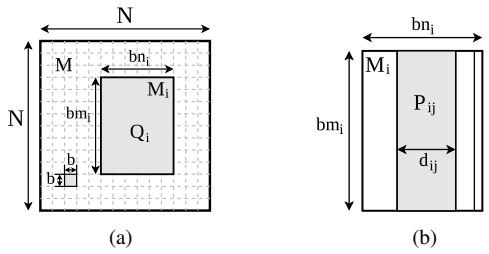

Figure 1.8 Two level matrix partitioning scheme: (a) two-dimensional partitioning between the nodes; (b) one dimensional partitioning between devices in a node.

We now present an outline of a parallel application using the proposed hierarchical partitioning algorithm. The partitioning is executed immediately before execution of the parallel algorithm. The outline is followed by a detailed description of the individual algorithms.

INPA$\big($IN: $N, b, q, p_1, \ldots, p_q$ OUT: $\{m_i, n_i, d_{i1}, \ldots, d_{ip}\}_{i=1}^{q}\big)$ {
WHILE inter-node imbalance
CMA$\Big($IN: $w_1, \ldots, w_q$ OUT: $(m_1, n_1), \ldots, (m_q, n_q)\Big)$;
On each node i (IDPA):
WHILE inter-device imbalance
On each device j: **kernel**$\Big($IN: bm_i, bn_i, d_{ij} OUT: $t_{ij}\Big)$;
FPM-PA$\Big($IN: p_i, bn_i, p_iFPMs OUT: $d_{i1}, \ldots, d_{iq})\Big)$;
END WHILE
FPM-PA$\Big($IN: q, W, qFPMs OUT: $w_1, \ldots, w_q\Big)$;
END WHILE
}
Parallel application$\Big($IN: $\{m_i, n_i, d_{i1}, \ldots, d_{ip}\}_{i=1}^{q}, \ldots\Big)$

Inter-Node Partitioning Algorithm (INPA)
Run in parallel on all nodes with distributed memory. Inputs: square matrix size N, number of nodes q, number of devices in each node $p_1, \ldots, p_q$ and block size b.

1. To add initial small point to the model, each node, in parallel, invokes the IDPA with an input (p_i, $bm_i = 1$, $bn_i = 1$). This algorithm returns a time which is sent to the head node.
2. The head node calculates speeds from these times as $s_i(1) = 1/t_i(1)$ and adds the first point, $(1, s(1))$, to the model of each node.
3. The head node then computes the initial homogeneous distribution by dividing the total number of blocks, W, between processors $w_i = W/q$.
4. The CMA is passed $w_1, \ldots, w_q$ and returns the inter-node distributions $(m_1, n_1), \ldots, (m_q, n_q)$ which are scattered to all nodes.
5. On each node, the IDPA is invoked with the input (p_i, bm_i, bn_i) and the returned time t_i is sent to the head node.
6. IF $\max\limits_{1 \le i,j \le q} \left| \frac{t_i(w_i) - t_j(w_i)}{t_i(w_i)} \right| \le \varepsilon_1$ THEN the current inter-node distribution solves the problem. All inter-device and inter-node distributions are saved and the algorithm stops;
ELSE the head node calculates the speeds of the nodes as $s_i(w_i) = w_i/t_i(w_i)$ and adds the point $(w_i, s_i(w_i))$ to each node-FPM.

7. On the head node, the FPM-PA is given the node-FPMs as input and returns a new distribution $w_1, \ldots, w_q$. GOTO 4

Inter-Device Partitioning Algorithm (IDPA)
This algorithm is run on a node with p devices. The input parameters are p and the submatrix sizes bm, bn. It computes the device distribution $d_1, \cdots, d_p$ and returns the time of last benchmark.

1. To add an initial small point to each device model, the *kernel* with parameters $(bm, bn, 1)$ is run in parallel on each device and its execution time is measured. The speed is computed as $s_j(1) = 1/t_j(1)$ and the point $(1, s_j(1))$ is added to each device model.

2. The initial homogeneous distribution $d_j = bn/p$, for all $1 \leq j \leq p$ is set.

3. In parallel on each device, the time $t_j(d_j)$ to execute the kernel with parameters (bm, bn, d_j) is measured.

4. IF $\max\limits_{1 \leq i,j \leq p} \left| \frac{t_i(d_i) - t_j(d_j)}{t_i(d_i)} \right| \leq \varepsilon_2$ THEN the current distribution of computations over devices solves the problem. This distribution $d_1, \cdots, d_p$ is saved and $\max\limits_{1 \leq j \leq p} t_j(d_j)$ is returned;
   ELSE the speeds $s_j(d_j) = d_j/t_j(d_j)$ are computed and the point $(d_j, s_j(d_j))$ is added to each device-FPM.

5. The FPM-PA takes bn and device-FPMs as input and returns a new distribution $d_1, \ldots, d_p$. GOTO 3

Functional Performance Model Partitioning Algorithm (FPM-PA)
This FPM partitioning algorithm is presented in detail in Sect. 1.3.

Communication Minimizing Algorithm (CMA)
This algorithm is specific to communication pattern of application and the topology of the communication network. It takes as input the number of computation units, w_i, to assign to each processing element and arranges them in such away, (m_i, n_i), as to minimize the communication cost. For example, for matrix multiplication, $\mathbf{A} \times \mathbf{B} = \mathbf{C}$, the total volume of data exchange is minimized by minimizing the sum of the half perimeters $H = \sum_{i=1}^{q} (m_i + n_i)$. A column-based restriction of this problem is solved in [20].

The Grid'5000 experimental testbed proved to be an ideal platform to test our application. We used 90 dedicated nodes from 3 clusters from the Grenoble site. 12 nodes from the Adonis cluster included NVIDIA Tesla GPUs, the rest were approximately homogeneous. In order to increase the impact of our experiments we chose to utilize only some of the CPU cores on some machines (Table 1.3). Such an approach is not unrealistic since it is possible to book individual CPU cores on this platform. For the local *dgemm* routine we used high performance vendor-provided

Table 1.3 Experimental hardware setup using 90 nodes from three clusters from Grenoble, Grid'5000. All nodes have 8 CPU cores, however, to increase heterogeneity only some of the CPU cores are utilized as tabulated below. One GPU was used with each node from the Adonis cluster, 10 have Tesla T10 and 2 have Tesla C2050 GPUs. A CPU core is devoted to control GPU. For example, we can read that 6 Edel nodes used just 1 CPU core. All nodes are connected with InfiniBand 20G & 40G

Cores:	0	1	2	3	4	5	6	7	8	Nodes	CPUs	GPUs	Hardware
Adonis	2	1	1	1	1	1	2	3	0	12	48	12	2.4GHz, 24GB
Edel	0	6	4	4	4	8	8	8	8	50	250	0	2.3GHz, 24GB
Genepi	0	3	3	3	3	4	4	4	4	28	134	0	2.5GHz, 8GB
Total										90	432	12	Intel Xeon

BLAS libraries, namely Intel MKL for CPU and CUBLAS for GPU devices. Open MPI was used for inter-node communication and OpenMP for inter-device parallelism. The GPU execution time includes the time to transfer data to the GPU. For these experiments, an out of core algorithm is not used when the GPU memory is exhausted. All nodes are interconnected by a high speed InfiniBand network which reduces the impact of communication on the total execution time, for $N = 1.5 \times 10^5$ all communications (including wait time due to any load imbalance) took 6% of total execution time. The full functional performance models of nodes, Fig. 1.9, illustrate the range of heterogeneity of our platform.

An appropriate block size of $b = 128$ proves to be a good balance between achieving near peak performance of optimized BLAS libraries while providing sufficient granularity for load balancing [10]. In order to demonstrate the effectiveness of the proposed FPM-based partitioning algorithm we compare it against 3 other par-

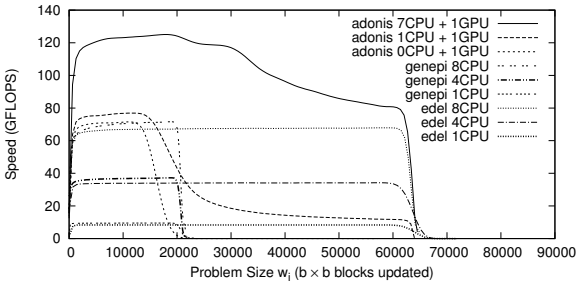

Figure 1.9 Full functional performance models for a number of nodes from Grid'5000 Grenoble site. Problem size is in number of $b \times b$ blocks of matrix C updated by a node. For each data point, it was necessary to build device models, find the optimum inter-device distribution and then measure the execution time of the kernel with this distribution.

titioning algorithms. All four algorithms invoke the *communication minimization algorithm* and are applied to an identical parallel matrix multiplication application. They differ on how load balancing decisions are made.

- **Multiple-CPM Partitioning** uses the same algorithm as proposed above, with step 7 of the INPA and step 5 of the IDPA replaced with $w_i = W \times \frac{s_i}{\sum_q s_i}$ and $d_j = bn \times \frac{s_j}{\sum_p s_j}$ respectively, where s_i and s_j are constants. This is equivalent to the approach used in [21, 22, 31].

- **Single-CPM Partitioning** does one iteration of the above multiple-CPM partitioning algorithm. This is equivalent to the approach used in [32, 20].

- **Homogeneous Partitioning** uses an even distribution between all nodes: $w_1 = w_2 = \cdots = w_q$ and between devices in a node: $d_{i1} = d_{i2} = \cdots = d_{ip_i}$.

Fig. 1.10 shows the speed achieved by the parallel matrix multiplication application when the four different algorithms are applied. It is worth emphasizing that the performance results related to the execution on GPU devices take into account the time to transfer the workload to/from the GPU. The speed of the application with the *homogeneous distribution* is governed by the speed of the slowest processor (a node from Edel cluster with 1CPU core). The *Single-CPM* and *multiple-CPM* partitioning algorithms are able to load balance for N up to 60000 and 75000 respectively, however this is only because the speed functions in these regions are horizontal. In general, for a full range of problem sizes, the simplistic algorithms are unable to converge to a balanced solution. By chance, for $N = 124032$, the multiple-CPM algorithm found a reasonably good partitioning after many iterations, but in general this is not the case. Meanwhile the *FPM-based partitioning* algorithm reliably found good partitioning for matrix multiplication involving in excess of 0.5TB of data.

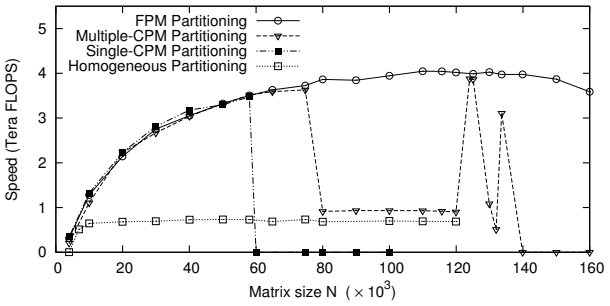

Figure 1.10 Absolute speed for a parallel matrix multiplication application based on four partitioning algorithms, measured on 90 heterogeneous nodes consisting of 432 CPU cores and 12 GPUs from 3 dedicated clusters.

1.10 Conclusion

In this chapter, we proposed several techniques to efficiently exploit the capabilities of modern heterogeneous systems equipped with multicore CPUs and several GPU devices for scientific computations. We investigated the process of efficient design and optimization of scientific applications not only at the level of a single CPU/GPU computing node, but also in highly heterogeneous and hierarchical HPC clusters. In contrast to other related works in this area, we based our approaches on functional performance modeling which integrates many important features characterizing the performance of the platform and the application, such as contention on shared resources, high performance disparity of architecturally different devices, limited memory of the accelerators or scenarios when different devices use different codes to solve the same computational problem.

For a single hybrid CPU/GPU node, we presented the performance measurement methods and analyzed the efficiency of different implementations of parallel matrix multiplication, chosen as a case study. We defined and built functional performance models of heterogeneous processing elements on a typical multicore and multi-GPU node. We showed that FPMs can facilitate performance evaluation of scientific applications on these hybrid platforms, and data partitioning algorithms based on accurate FPMs can deliver significant performance improvements when compared to the one obtained at the level of a single device.

To adapt parallel applications to hybrid heterogeneous clusters, we proposed a hierarchical data partitioning algorithm, which optimally distributed computation workload at two levels of the platform's hierarchy, namely, between nodes and between devices within each node. The presented approach is based on FPMs of processing elements which are efficiently built at runtime and realistically capture the high level of platform heterogeneity. The efficiency of the proposed algorithm was assessed on a real platform consisting of 90 highly heterogeneous nodes in 3 computing clusters and compared to the equivalent approaches based on traditional data partitioning algorithms. The results demonstrate that the presented algorithm minimized the overall communication volume and provided efficient load balancing decisions for very large problem sizes, while similar approaches were not able to find the adequate balancing solutions.

Acknowledgments. This work was supported by COST Action IC0805 "Open European Network for High Performance Computing on Complex Environments", Science Foundation Ireland (grant 08/IN.1/I2054), Portuguese National Foundation of Science and Technology (PIDDAC program and SFRH/BD/44568/2008 fellowship). Experiments were carried out on Grid'5000 developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see https://www.grid5000.fr).

REFERENCES

1. C. Augonnet, S. Thibault, R. Namyst, and P. Wacrenier, "StarPU: A unified platform for task scheduling on heterogeneous multicore architectures," *Euro-Par 2009*, pp. 863–874, 2009.
2. A. Ilic and L. Sousa, "Collaborative execution environment for heterogeneous parallel systems," in *IPDPS Workshops and Phd Forum (IPDPSW)*, pp. 1–8, 2010.
3. M. Fatica, "Accelerating Linpack with CUDA on heterogenous clusters," in *GPGPU-2*, pp. 46–51, ACM, 2009.
4. C. Yang, F. Wang, Y. Du, *et al.*, "Adaptive optimization for petascale heterogeneous CPU/GPU computing," in *Cluster'10*, pp. 19–28, 2010.
5. Y. Ogata, T. Endo, N. Maruyama, and S. Matsuoka, "An efficient, model-based CPU-GPU heterogeneous FFT library," in *IPDPS 2008*, pp. 1–10, 2008.
6. A. Lastovetsky and R. Reddy, "Data partitioning with a functional performance model of heterogeneous processors," *Int J High Perform C*, vol. 21, pp. 76–90, 2007.
7. Z. Zhong, V. Rychkov, and A. Lastovetsky, "Data partitioning on heterogeneous multicore platforms," in *Cluster 2011*, pp. 580–584, 2011.
8. Z. Zhong, V. Rychkov, and A. Lastovetsky, "Data partitioning on heterogeneous multicore and multi-gpu systems using functional performance models of data-parallel applications," in *Cluster 2012*, pp. 191–199, 2012.
9. A. Lastovetsky and R. Reddy, "Distributed Data Partitioning for Heterogeneous Processors Based on Partial Estimation of Their Functional Performance Models," in *EuroPar/HeteroPar 2009*, vol. 6043 of *LNCS*, pp. 91–101, Springer, 2010.

10. D. Clarke, A. Ilic, A. Lastovetsky, and L. Sousa, "Hierarchical partitioning algorithm for scientific computing on highly heterogeneous CPU + GPU clusters," in *Euro-Par 2012*, vol. 7484 of *LNCS*, pp. 489–501, Springer, 2012.
11. C.-K. Luk, S. Hong, and H. Kim, "Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping," in *MICRO-42*, pp. 45–55, 2009.
12. A. Ilic and L. Sousa, "On realistic divisible load scheduling in highly heterogeneous distributed systems," in *PDP 2012*, pp. 426–433, IEEE, 2012.
13. R. Blumofe and C. Leiserson, "Scheduling multithreaded computations by work stealing," *JACM*, vol. 46, no. 5, pp. 720–748, 1999.
14. J. Quintin and F. Wagner, "Hierarchical work-stealing," *Euro-Par 2010-Parallel Processing*, pp. 217–229, 2010.
15. M. D. Linderman, J. D. Collins, H. Wang, *et al.*, "Merge: a programming model for heterogeneous multi-core systems," *SIGPLAN Not.*, vol. 43, pp. 287–296, 2008.
16. C. Augonnet, S. Thibault, and R. Namyst, "Automatic calibration of performance models on heterogeneous multicore architectures," in *EuroPar'09*, pp. 56–65, 2009.
17. G. Quintana-Ortí, F. D. Igual, E. S. Quintana-Ortí, and R. A. van de Geijn, "Solving dense linear systems on platforms with multiple hardware accelerators," *SIGPLAN Not.*, vol. 44, pp. 121–130, 2009.
18. F. Song, S. Tomov, and J. Dongarra, "Enabling and scaling matrix computations on heterogeneous multi-core and multi-GPU systems," in *ICS '12*, pp. 365–376, ACM, 2012.
19. M. Cierniak, M. Zaki, and W. Li, "Compile-Time Scheduling Algorithms for Heterogeneous Network of Workstations," *Computer J.*, vol. 40, pp. 356–372, 1997.
20. O. Beaumont, V. Boudet, F. Rastello, and Y. Robert, "Matrix Multiplication on Heterogeneous Platforms," *IEEE Trans. Parallel Distrib. Syst.*, vol. 12, no. 10, pp. 1033–1051, 2001.
21. I. Galindo, F. Almeida, and J. Bada-Contelles, "Dynamic Load Balancing on Dedicated Heterogeneous Systems," in *EuroPVM/MPI 2008*, pp. 64–74, Springer, 2008.
22. J. Martínez, E. Garzón, A. Plaza, and I. García, "Automatic tuning of iterative computation on heterogeneous multiprocessors with ADITHE," *J. Supercomput.*, vol. 58, no. 2, pp. 151–159, 2011.
23. A. Ilic and L. Sousa, "Scheduling divisible loads on heterogeneous desktop systems with limited memory," in *Euro-Par/HeteroPar 2011*, pp. 491–501, Springer, 2011.
24. A. Ilic and L. Sousa, "Simultaneous multi-level divisible load balancing for heterogeneous desktop systems," in *ISPA 2012*, pp. 683–690, IEEE, 2012.
25. J. Dongarra, M. Faverge, T. Herault, J. Langou, and Y. Robert, "Hierarchical QR factorization algorithms for multi-core cluster systems," *Arxiv preprint arXiv:1110.1553*, 2011.
26. D. Clarke, A. Lastovetsky, and V. Rychkov, "Column-based matrix partitioning for parallel matrix multiplication on heterogeneous processors based on functional performance models," in *Euro-Par 2011: Parallel Processing Workshops*, vol. 7155 of *LNCS*, pp. 450–459, Springer, 2012.
27. J. Choi, "A new parallel matrix multiplication algorithm on distributed-memory concurrent computers," *Concurrency: Practice and Experience*, vol. 10, no. 8, pp. 655–670, 1998.

28. A. Kalinov and A. Lastovetsky, "Heterogeneous distribution of computations solving linear algebra problems on networks of heterogeneous computers," *Journal of Parallel and Distributed Computing*, vol. 61, no. 4, pp. 520–535, 2001.

29. C. Gregg and K. Hazelwood, "Where is the data? Why you cannot debate CPU vs. GPU performance without the answer," in *ISPASS '11*, pp. 134–144, 2011.

30. D. Clarke, A. Lastovetsky, and V. Rychkov, "Dynamic load balancing of parallel computational iterative routines on platforms with memory heterogeneity," in *Euro-Par/HeteroPar 2010*, pp. 41–50, Springer, 2011.

31. A. Legrand, H. Renard, Y. Robert, and F. Vivien, "Mapping and load-balancing iterative computations," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 15, no. 6, pp. 546–558, 2004.

32. S. Hummel, J. Schmidt, R. N. Uma, and J. Wein, "Load-sharing in heterogeneous systems via weighted factoring," in *SPAA96*, pp. 318–328, ACM, 1996.