Data Partitioning on Multicore and Multi-GPU Platforms Using Functional Performance Models

Ziming Zhong, Vladimir Rychkov, and Alexey Lastovetsky, *Member, IEEE*

Abstract—Heterogeneous multiprocessor systems, which are composed of a mix of processing elements, such as commodity multicore processors, graphics processing units (GPUs), and others, have been widely used in scientific computing community. Software applications incorporate the code designed and optimized for different types of processing elements in order to exploit the computing power of such heterogeneous computing systems. In this paper, we consider the problem of optimal distribution of the workload of data-parallel scientific applications between processing elements of such heterogeneous computing systems. We present a solution that uses functional performance models (FPMs) of processing elements and FPM-based data partitioning algorithms. Efficiency of this approach is demonstrated by experiments with parallel matrix multiplication and numerical simulation of lid-driven cavity flow on hybrid servers and clusters.

Index Terms—HPC, heterogeneous computing, GPU-accelerated multicore system, performance modeling, data partitioning

1 Introduction

Heterogeneous multiprocessor systems, where multicore processors are coupled with graphics processing units (GPUs), have been widely used in high performance computing as one approach to continuing performance improvement while managing the new challenge of energy efficiency. Considerable efforts have been made over recent years to port critical scientific software to both multicore and GPU architectures. This often requires re-engineering of the existing parallel applications and development of new programming models, tools and algorithms. Therefore, the existing highly optimized code should be reused for development of scientific software for hybrid platforms. In order to achieve the maximum performance of scientific applications on heterogeneous multicore and multi-GPU platforms, it is essential to balance the workload between heterogeneous processing elements. However, load balancing on such platforms is complicated by several factors, including resource contention, non-uniform memory access (NUMA), limited GPU memory, and low bandwidth of PCIe that connects the host and the GPU, etc.

In this paper, we target data-parallel scientific applications which are characterized by divisible computational workload, such as linear algebra, digital image processing, computational fluid dynamics. The computational workload in these applications is proportional to the size of data. In order to distribute workload between heterogeneous processing elements optimally, we use the data partitioning method, which is a static load balancing method widely used on distributed-memory platforms.

Data partitioning algorithms, including those already proposed for heterogeneous systems, rely on accurate performance models of processors. In [1], constants found *a priori* and representing the sustained performance of the application on CPUs and GPUs were used to partition data. In [2], a similar constant performance model (CPM) was proposed, but it was built adaptively, using the history of performance measurements. The fundamental assumptions of the data partitioning algorithms based on constant performance models are that (i) the absolute speed of processing elements does not depend on the size of a computational task, and therefore, can be represented by a constant; (ii) the processing elements are independent of each other, and thus, their speed can be measured separately. However, they become invalid in the following situations:

1) the partitioning of the problem results in tasks fitting into different levels of memory hierarchy;
2) processing elements switch between different codes to solve the same computational problem;
3) processing elements contend for shared system resources with each other.

Data partitioning algorithms based on functional performance models (FPMs) were originally designed and proved to be accurate for heterogeneous network of uniprocessors [3]. The functional performance model represents the processor speed by 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 performance model is only applicable in situations (1) and (2).

In this work, we extend FPM-based data partitioning to heterogeneous multicore and multi-GPU platforms, where processing elements are coupled and share system

• *Z. Zhong, V. Rychkov and A. Lastovetsky are with the School of Computer Science and Informatics, University College Dublin, Belfield, Dublin 4, Ireland.*
E-mail: ziming.zhong@ucdconnect.ie, vladimir.rychkov@ucd.ie, alexey.lastovetsky@ucd.ie

resources. In such platforms, the speed of one processing element may depend on the load of others due to resource contention, therefore, they cannot be considered as independent processors and their speed cannot be measured separately. In addition, a parallel application on such a platform allows for many different configurations, depending on the computational kernels used in the application and their mapping to the processing elements of the platform. In this work, we focus on the problem of optimal data distribution between kernels of the data-parallel application assuming that the configuration of the application is fixed. Comparison of different configurations and the problem of finding the optimal configuration of the application are out of the scope of this paper. In our experiments, however, we only use configurations that we believe are optimal. For example, we would never run a NUMA-unaware multi-threaded computational kernel across multiple NUMA nodes. Instead, we will use multiple instances of this kernel, one per NUMA node.

We propose to model a multicore and multi-GPU system by a set of heterogeneous abstract processors determined by the configuration of the parallel application. Namely, a group of processing elements executing one computational kernel of the application makes a combined processing unit and will be represented in the model by one abstract processor. For example, if a single-threaded computational kernel is used, then each CPU core executing this kernel will be represented in the model by an abstract processor. If a multi-threaded computational kernel is used, then each group of CPU cores executing the kernel will make a combined processing unit represented in the model by one abstract processor. A GPU is usually controlled by a host process running on a dedicated CPU core. This process instructs the GPU to perform computations and handles data transfers between the host and device memory. In the case of a single-GPU computational kernel, the GPU and its dedicated CPU core will make a combined processing unit represented by an abstract processor. If a multi-GPU computational kernel is used in the application, the GPUs and their dedicated CPU core will make a combined processing unit represented by an abstract processor. We build functional performance models of the abstract processors and perform model-based data partitioning in order to balance the workload between the combined processing units represented by these abstract processors.

To build the performance models of the abstract processors, we have to measure the performance of the processing units representing these processors. To measure the performance of the processing units accurately, we propose to group them by shared system resources, so that the resources be shared within each group but not shared between groups. The performance of processing units in a group is measured when all processing units in the group are executing some workload simultaneously, thereby taking into account the influence of resource contention. For example, processing units that share memory or PCIe link are grouped together during performance measurement.

To illustrate the importance of accurate performance measurement, let us consider a parallel application configured to be executed with one multi-threaded CPU kernel and one multi-GPU computational kernel on a multicore and multi-GPU computer. In this configuration, all GPUs and their dedicated CPU core will make one processing unit and be modeled by one abstract processor, and all other CPU cores will make another processing unit and be modeled by another abstract processor. As these two processing units include CPU cores and all the cores share memory, the processing units also share memory and cannot be considered independent. Therefore, the performance of these processing units should be measured simultaneously. If the measurements are not synchronized and conducted separately, the measured performance of these processing units will not reflect their actual performance during the execution of the application, and therefore load balancing decisions based on the corresponding performance models will be inaccurate.

Using the proposed method for measuring performance, the speed of processing units is measured for a wide range of problem sizes. The functional performance models built from this empirical data will be applicable in situation (3). The performance models built in this way can be used in FPM-based data partitioning algorithms to balance the workload in heterogeneous multicore and multi-GPU platforms.

The contributions of this work are as follows:

1) We propose a method for performance modeling on dedicated heterogeneous multicore and multi-GPU systems. A multicore and multi-GPU system is modeled by a number of abstract processors. Functional performance models of these abstract processors are built and used as the input of data partitioning algorithm.
2) We propose a method for accurate performance measurement on dedicated heterogeneous multicore and multi-GPU systems. Performance of processing units are measured simultaneously rather than separately, thereby taking into account resource contention. To ensure the reliability of measurements, we bind processes to CPU cores and repeat measurements multiple times.
3) From analysis of functional performance models built in different configurations, we reveal the impact of resource contention on the performance of CPU and GPU processing units, and the impact of process mapping on GPU-accelerated multicore systems of NUMA architecture on the performance of the GPU processing unit.
4) We demonstrate that data partitioning algorithms based on functional performance models of abstract processors, each representing a group of processing elements, are able to balance the work-

load on heterogeneous multicore and multi-GPU platforms with two typical data parallel applications, namely, parallel matrix multiplication and numerical simulation of lid-driven cavity flow.

The rest of this paper is organized as follows. Section 2 introduces the basic concepts of functional performance model (FPM) and FPM-based data partitioning algorithms. Section 3 describes the proposed method for performance modeling on multicore and multi-GPU systems, and analyzes the impact of resource contention and NUMA mapping. Section 4 presents experimental results of FPM-based data partitioning. Section 5 presents the related work. Section 6 concludes the paper.

2 BACKGROUND

In this section, we briefly introduce the concepts of functional performance model and data partitioning based on such models. These concepts, proposed in [3] and [4] for dedicated heterogeneous network of uniprocessors, form the basis of the research presented in this paper. Comparison with the state-of-the-art will be given in Section 5, and the system architecture used in this research will be defined in detail in Sections 3 and 4.

2.1 Functional Performance Model

Under the functional performance model (FPM), 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 one time unit. The computation unit can be defined differently for different applications. An arithmetical operation and the matrix update $C = C + A \times B$, where A, B, and C are $r \times r$ matrices of the fixed size r, give us examples of computation units.

The problem size is understood as a set of one, two, or more parameters characterizing the amount and layout of data stored and processed during the execution of the computational task. The number and the semantics of the problem size parameters are problem- or even application-specific. It is assumed that the amount of stored data will increase with the increase of any of the problem size parameters.

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. The benchmarking can be done more efficiently by using a serial code, which performs much less computations but still representative for the application. We call such a code a ***kernel***. For example, computationally intensive applications often perform the same performance-critical computation multiple times in a loop. A benchmark made of one such core computation can be representative of the performance of the whole application and can be used as a kernel. The speed function of the application can be built more efficiently by timing this kernel.

2.2 FPM-based Data Partitioning Algorithm

The problem of data partitioning using functional performance models was formulated in [3] as follows. A total problem size n is given as the number of computation units to be distributed between p ($p \ll n$) processes $P_1, \ldots, P_p$. The speeds of processors are represented by positive continuous functions 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 complete 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:

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

The solution of these equations can be represented geometrically by intersection of the speed functions with a line passing through the origin of the coordinate system.

The geometrical algorithm solving this data partitioning problem was proposed in [3] 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, 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 $d_1, \ldots, d_p$: $d_1 + \ldots + d_p = n$. Correctness proof and complexity analysis of this algorithm are presented in [3].

2.3 Partial FPM and Dynamic Data Partitioning

Functional performance models 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 optimised 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. 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 [4], [5]. 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 $\bar{s}_i(x)$.
3) Using the current partial estimates of the speed functions, the FPM-based partitioning algorithm calculates a new distribution.

This algorithm allows for efficient load balancing and is suitable for use in self-adaptable applications, which run without *a priori* information of the platforms.

3 Performance Modeling of Multicore and Multi-GPU Platforms

In this section, we present a new method of performance modeling on multicore and multi-GPU platforms, and investigate the impact of memory contention, PCIe contention and process placement on the performance of CPU and GPU processing units.

For illustration, a GPU-accelerated multicore server of NUMA architecture, Pluto, is used. As specified in Table 1, it consists of eight NUMA nodes connected by AMD HyperTransport(HT) links, with 6 cores and 16 GB local memory each. It is equipped with a NVIDIA Tesla S2050 server, which consists of two pairs of GPUs.

TABLE 1
Specifications of the Pluto server *pluto.icl.utk.edu*

Hybird Server	Pluto (AMD CPU + NVIDIA GPU)	
Architecture	Opteron 6172	Tesla S2050
Core Clock	2.1 GHz	575 MHz
Number of Cores	8×6 cores	4×448 cores
Memory Size	8×16 GB	2667 MB
Mem. Bandwidth		4×148.4 GB/s
PCIe	$2 \times$ I/O hubs	$2 \times$ switches

Each pair is connected by a PCIe switch and linked to a separate NUMA node by a PCIe bus.

3.1 Performance Modeling of CPU Cores

In scientific applications, both single- and multi-threaded computational kernels are commonly used. A single-threaded computational kernel is executed by one process on a CPU core. A multi-threaded kernel is also executed by one process but on several cores using multiple threads. Therefore, all CPU cores are naturally partitioned into a number of CPU processing units, each made of one or several CPU cores and executing one computational kernel of the application.

The performance of the CPU processing units is measured for a wide range of problem sizes to build their functional performance models, which are then used as the input of a FPM-based data partitioning algorithm. Thus, at the stage of data partitioning the CPU cores of a multicore platform are modeled by a set of heterogeneous abstract processors, each characterized by its speed function and representing a processing unit made of one or several CPU cores.

Performance modeling on multicore systems is complicated by resource contention, because the performance of a CPU core may depend on the load of other CPU cores. Therefore, we propose to group processing units by shared resources. The performance of processing units in a group are measured when all processing units in the group are executing some workload simultaneously, thereby taking resource contention into account.

In this work, when looking for the optimal distribution of the workload, we only consider solutions that evenly distribute the workload between identical CPU processing units. This simplification significantly reduces the complexity of the data partitioning problem. It is based on our extensive experiments that have shown no evidence that uneven distribution between identical processing units could speed up applications. We also found no such evidence in literature. Therefore, identical processing units that share system resources are always given the same amount of workload during performance measurements.

Performance measurements on processing units that share system resources are synchronized. With the same amount of workload, measurements will be completed

with roughly the same amount of elapsed time, which realistically simulates resource contention.

Figure 1 shows a GPU-accelerated multicore server of NUMA architecture executing a parallel application in two different configurations. The configuration shown in Figure 1(a) is based on the single-threaded and single-GPU computational kernels. It consists of ten processes running the CPU kernels on ten cores of both NUMA nodes, and two processes running the GPU kernels on accelerators and their dedicated cores on the second NUMA node. The configuration in Figure 1(b) is based on the multi-threaded and multi-GPU computational kernels. It consists of one process running the 6-thread CPU kernel on one NUMA node, one process running the 5-thread CPU kernel on another NUMA node, and one process running the GPU kernel on the GPUs and their single dedicated core. All processing elements in these diagrams are enumerated. Each number indicates the combined processing unit to which the processing element belongs. For example, in the first configuration, the cores in NUMA node 0 make six processing units, and each GPU with its dedicated CPU core in NUMA node 1 make a combined processing unit.

In the first configuration, the cores in NUMA node 0 execute six identical processes and are modeled by six abstract processors. These cores are tightly coupled and share memory, therefore, they cannot be considered independent. On the other hand, this group of processing elements is relatively independent of other processing elements of the server. Therefore, their performance should be measured simultaneously in a group but can be measured separately from the others. In the second configuration, these six cores execute one process and modeled as one combined processing unit. Its performance can be measured separately from other processing elements of the server.

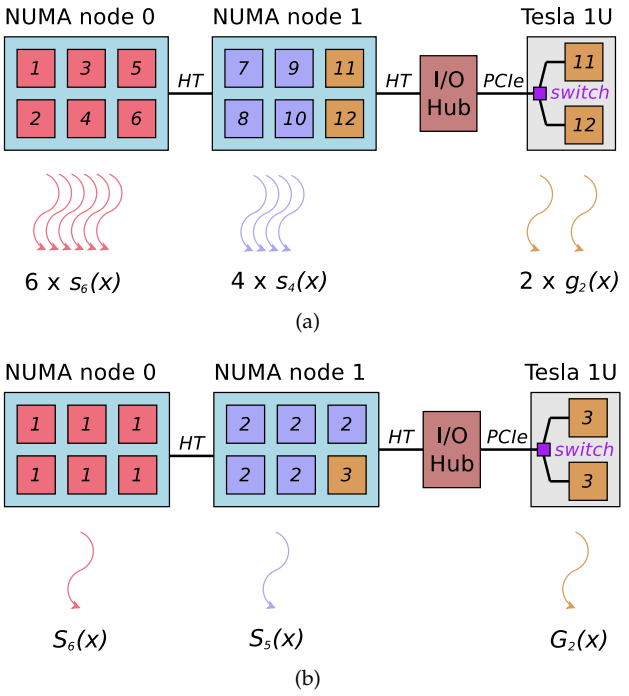


Fig. 1. Performance modeling on a GPU-accelerated multicore server of NUMA architecture: (a) single-threaded and single-GPU computational kernels executed; each GPU handled by a dedicated CPU core (b) multi-threaded and multi-GPU computational kernels executed; two GPUs handled by a single dedicated CPU core

In order to build the functional performance model of an abstract processor, we perform measurements for a wide range of problem sizes. To prevent the operating system from migrating processes excessively, processes are bound to CPU cores. 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. To ensure the reliability of the results, measurements are repeated multiple times, and average execution times are used. We find the confidence interval and stop the measurements if the sample mean lies in the interval with the confidence level 95%. In this work, for simplicity, we use Student's t-test, assuming that the individual observations are independent and their population follows the normal distribution.

Three types of functional performance models for CPU cores are defined as follows:

1) $s(x)$ approximates the speed of a uniprocessor executing a single-threaded computational kernel. The speed $s(x) = x/t$, where x is the number of computation units, and t is the execution time.
2) $s_c(x)$ approximates the speed of one of c CPU cores all executing the same single-threaded computational kernel simultaneously. The speed $s_c(x) = x/t$, where x is the number of computation units executed by each CPU core, and t is the execution time.
3) $S_c(x)$ approximates the collective speed of c CPU cores executing a multi-threaded computational kernel. The speed $S_c(x) = x/t$, where x is the total number of computation units executed by all c CPU cores, and t is the execution time. $S_c(cx)/c$ is used to approxcimate the average speed of a CPU core.

Figure 2 shows speed functions of a CPU core built in different configurations on Pluto. Speed functions $s_1(x)$, $s_6(x)$, and $s_{12}(x)$ are built by executing a single-threaded *gemm* kernel per CPU core on only one CPU core, on six CPU cores of a NUMA node, and on twelve CPU cores of two NUMA nodes respectively. Speed functions $S_6(6x)/6$ and $S_{12}(12x)/12$ approximate the average speed of a CPU core, when one multi-threaded *gemm* kernel is executed on one and two NUMA nodes respectively. The single- and multi-threaded *gemm* kernels used are from the ACML 4.4 library.

We can see that $s_6(x)$ is clearly lower than $s_1(x)$, which indicates extensive memory contention between CPU cores of the same NUMA node. By contrast, there is

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2014.2375202, IEEE Transactions on Computers

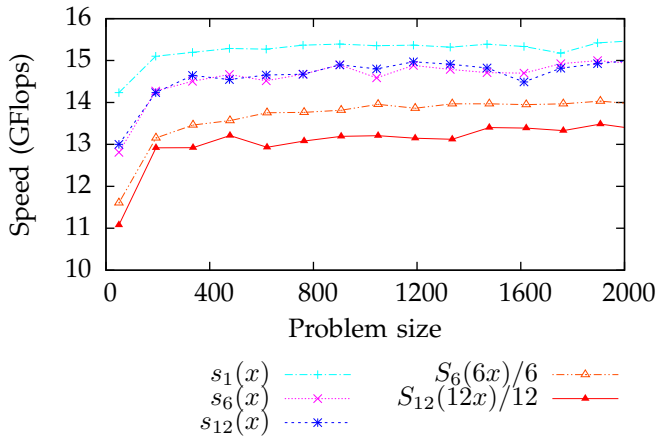


Fig. 2. Speed functions of a CPU core built in different configurations

no difference between $s_6(x)$ and $s_{12}(x)$, which indicates no contention between CPU cores of different NUMA nodes. Therefore, the performance model of the CPU cores of a NUMA node can be built separately from other CPUs. $S_{12}(12x)/12$ is lower than $S_6(6x)/6$ because the multi-threaded *gemm* kernel is NUMA-unaware.

Thus we can conclude that depending on the configuration of the application the speed of individual cores can vary significantly. Therefore, to achieve optimal distribution of computations it is very important to build and use speed functions which accurately reflect the performance of the CPU cores.

3.2 Performance Modeling of GPUs

In GPU-accelerated multicore systems, a GPU is usually controlled by a host process that is executed on a dedicated CPU core. Since GPU has a separate memory, input data is transfered to the device for computation, and the result is copied back to the host memory afterwards. The transfers are performed over a PCIe link and may dominate the execution time. In this work, we model a GPU and its dedicated CPU core by an abstract processor, consequently the data transfer time is included in the execution time during performance measurement. We measure the speed of the combined GPU processing unit (i.e. a GPU and its dedicated core) for a number of problem sizes to build its functional performance model. In general, the model can be defined only for the range of problem sizes that fit in the device memory. However, it can be extended for out-of-core applications, which can handle a larger amount of data stored in host memory through multiple host-device data transfers.

As more and more GPUs added to GPU-accelerated multicore systems, additional PCIe lanes are required to maintain the available bandwidth to each GPU. There are two common strategies for increasing the number of PCIe connections [6]. One approach is to introduce additional I/O hubs so that each GPU is connected to the host processor via a separate PCIe link. In this case, GPU processing units do not share PCIe and can be considered independent; the performance models of GPU abstract processors can be built separately. Another approach is to utilize PCIe switch. All data traffic traverses a single PCIe connection to the PCIe switch, and then is routed to GPUs connected to the PCIe switch. In this case, GPU processing units cannot be considered independent due to PCIe contention; the performance models of GPU abstract processors cannot be built separately. Therefore, we propose to group GPU processing units by shared PCIe link for performance measurements. GPU processing units that share a PCIe link are grouped together and their performance is measured when all GPU processing units in the group are executing the same amount of workload simultaneously. For example, as shown in Figure 1(a), GPU processing units 11 and 12 on NUMA node 1 share a single PCIe link, therefore, they are grouped together and their performance should be measured simultaneously.

Wide use of multi-GPU systems encourages development of optimized computational kernels that could efficiently distribute workload between multiple GPUs that share a PCIe link, minimize PCIe contention and overlap the host-device data transfers and device computations. If such a kernel is used, the GPUs and their dedicated CPU core will make a combined processing unit and will be modeled by an abstract processor. For example, as shown in Figure 1(b), GPU processing unit 3 is composed of two GPUs and a CPU core of NUMA node 1.

Three types of functional performance models for GPUs are defined as follows:

1) $g(x)$ approximates the speed of a combined processing unit made of a GPU and its dedicated CPU core that execute a single-GPU computational kernel, exclusively using a PCIe link. The speed $g(x) = x/t$, where x is the number of computation units, and t is the execution time.
2) $g_d(x)$ approximates the speed of one of d combined processing units, each made of a GPU and its dedicated CPU core. All processing units execute identical single-GPU computational kernels simultaneously. The speed $g_d(x) = x/t$, where x is the number of computation units executed by each GPU processing unit, and t is the execution time.
3) $G_d(x)$ approximates the speed of a combined processing unit made of d GPUs and their dedicated CPU core that collectively execute a multi-GPU computational kernel. The speed $G_d(x) = x/t$, where x is the total number of computation units processed by all d GPUs, and t is the execution time. $G_d(dx)/d$ is used to approximate the average speed of a GPU.

Figure 3 shows the speed functions of a combined GPU processing unit built in different configurations on Pluto. Speed functions $g_1(x)$, $g_2(x)$, and $g_4(x)$ are built by executing a single-GPU *gemm* computational kernel per GPU processing unit on only one GPU processing

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2014.2375202, IEEE Transactions on Computers

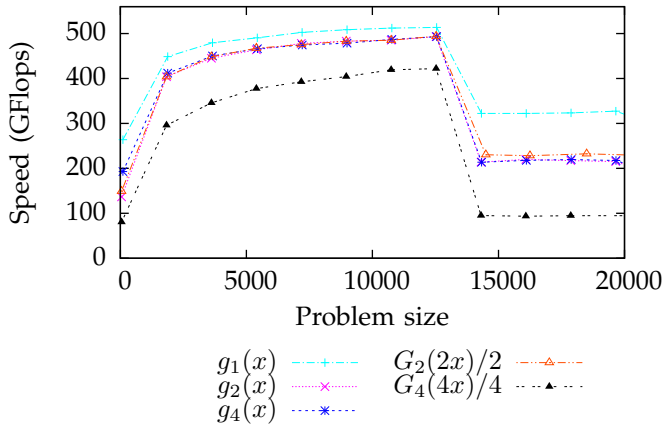


Fig. 3. Speed functions of a GPU processing unit built in different configurations

unit, on two GPU processing units that share a PCIe link, and on two pairs of GPU processing units, each sharing a PCIe link. The dedicated CPU cores are located on NUMA nodes directly connected to the GPUs, therefore, the GPU processing units in a pair share not only PCIe but also memory. $G_2(2x)/2$ and $G_4(4x)/4$ approximate the average speed of a GPU, when one multi-GPU *gemm* kernel is executed on two GPUs that share a PCIe link, and on two pairs of GPUs, each sharing a PCIe link.

The single- and multi-GPU *gemm* computational kernels used are implemented in CUDA 4.1. The *gemm* operation in the kernels is implemented by invoking the *gemm* routine from the CUBLAS 4.1 library. The multi-GPU *gemm* kernel is designed for multiple GPUs that share a single data link and are handled by a single dedicated CPU core. The kernel schedules data transfers to eliminate PCIe contention between GPUs.

We can see that $g_2(x)$ is lower than $g_1(x)$, especially for large problem sizes. This indicates significant resource contention between two GPU processing units, dominated by PCIe but also including memory. There is no difference between $g_2(x)$ and $g_4(x)$, indicating no resource contention between the two pairs of GPU processing units. In the 4-unit configuration, each pair of GPUs is connected to its own NUMA node, therefore, the performance of the two pairs of GPU processing units can be measured independently.

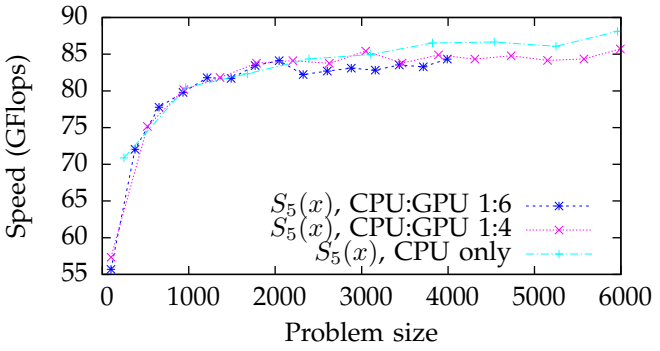
$G_4(4x)/4$ is lower than $G_2(2x)/2$ because of inappropriate use of the multi-GPU computational kernel. In the 4-GPU configuration, due to the contention-free scheduling of data transfers, the two data links are used alternately, remaining under-utilized during the execution of the kernel. In addition, the 4-GPU configuration uses the PCIe slots of two NUMA nodes but the memory of only one of them. One of the data links includes an extra HT link between the two NUMA nodes, which incurs extra communication overhead.

In the next section, we investigate the impact of resource contention between heterogeneous processing units, which is the case of NUMA node 1 in Figure 1, where the CPU and GPU processing units compete for the main memory.

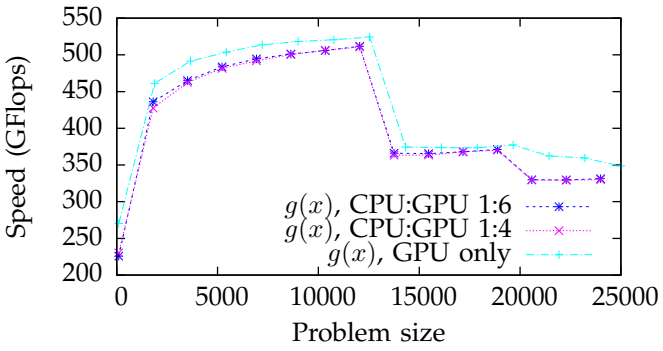
3.3 Impact of Resource Contention between CPU and GPU Processing Units

To achieve the maximum performance on a multicore and multi-GPU system, it is necessary to employ both CPUs and GPUs for computation. During the execution, while a CPU computational kernel performs computations using all levels of memory hierarchy, a GPU computational kernel mainly offloads work to GPUs. Therefore, CPU and GPU processing units are heterogeneous in terms of computing power and memory access pattern. As the CPU cores included in these two types of processing units share memory, they cannot be considered independent.

Figure 4(a) shows the speed functions of a 5-core CPU processing unit executing the multi-threaded CPU *gemm* kernel, $S_5(x)$, which are built while the GPU and another core of the same NUMA node are idle or executing the GPU *gemm* kernel. The corresponding speed functions of the GPU processing unit, $g(x)$, are shown in Figure 4(b), complemented by the function built when the CPU cores are idle. The workload assigned to the CPU cores and the GPU is proportional to their speed measured exclusively. The first distribution (1:6) corresponds to the the speeds



(a)



(b)

Fig. 4. Impact of resource contention on the performance of the CPU (a) and GPU (b) processing units

observed for the problems that fit in the device memory; the second distribution (1:4) – for the large problem sizes, which exceed the device memory. For both distributions, we measured the speed of the CPU and GPU processing units simultaneously.

As Figure 4 suggests, the resource contention affects the performance of both processing units, however, performance degradation depends on the types of the processing elements and the computational kernels. Namely, the performance of the CPU processing unit is nearly the same in all cases. Therefore, the speed function built exclusively for 5 cores provides a good approximation of their performance even in the case of resource contention with the GPU unit. In contrast, to build the accurate performance model of a GPU unit, it is necessary to measure its performance simultaneously with CPU cores.

3.4 Impact of NUMA Mapping

On multicore systems of NUMA architecture, data is transferred between NUMA nodes over fast links, such as AMD HyperTransport(HT), whose bandwidth is usually lower than memory bandwidth. Integration of multiple GPUs into multicore systems of NUMA architecture introduces complex performance phenomena. If the host process that handles the GPU is bound to a CPU core that resides in the NUMA node connected to the GPU directly through an I/O hub, the data processed by the GPU will only traverse links between the NUMA node, the I/O hub, and the GPU. Otherwise, it will traverse extra links between NUMA nodes, incurring extra communication overhead. We will refer to these two types of configurations as *local* and *remote*.

Figure 5 demonstrates the impact of NUMA mapping on the performance of a GPU processing unit, comprised of a CPU core and a GPU of Tesla S2050 deployed in Pluto. $g_1(x)$ is built by executing one single-GPU *gemm* kernel, which uses exclusively the data link and the memory of a local or remote NUMA node. $g_2(x)$ is built

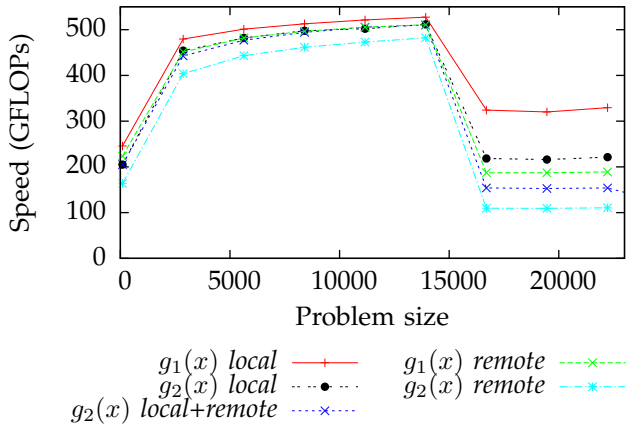


Fig. 5. Speed functions of a GPU processing unit built in different configurations

by executing two single-GPU kernels simultaneously on two GPU units that share the PCIe link and the memory of the same NUMA node, local or remote. In the remote configuration, the GPU units also share an extra HT link to the remote NUMA node. Speed function $g_2(x)$ is also built in the configuration when two dedicated CPU cores are located on different NUMA nodes, which is denoted as *local + remote*. In this case, the processing units share PCIe but do not share memory.

The difference between speed functions $g_1(x)$ and $g_2(x)$ reflects the performance degradation due to the contention for PCIe, HT and memory. Significant difference is observed for large problem sizes when many data transfers are required. Communication overhead between NUMA nodes can be estimated by the difference between $g_1(x)$ in *local* and *remote* configurations. The combined effect of both phenomena is reflected by the $g_2(x)$ functions in different configurations.

4 Experimental Results

In this section, the proposed methods are evaluated with two typical data parallel applications, namely parallel matrix multiplication and numerical simulation of lid-driven cavity flow. The building of functional performance models of the computational kernels of the applications, and the FPM-based data partitioning on experimental platforms are realized with the help of the FuPerMod software [7]. In this work, the dynamic voltage and frequency scaling (DVFS) is not considered.

4.1 Parallel Matrix Multiplication

Heterogeneous parallel matrix multiplication [8] takes the functional performance models of heterogeneous processors as input, partitions the matrices using the FPM-based data partitioning algorithm, and then performs the blocked matrix multiplication.

In this application, matrices A, B and C are partitioned over a 2D arrangement of heterogeneous processors so that the area of each rectangle is proportional to the speed of the processor that handles the rectangle. This speed is given by the speed function of the processor for the assigned problem size, i.e. the number of matrix blocks of size $b \times b$. Figure 6(a) shows one iteration of the application. At each iteration of the main loop, pivot block column of matrix A and pivot block row of matrix B are broadcast horizontally and vertically, then all processors update their own parts of matrix C in parallel. The blocking factor b is a parameter used to adjust the granularity of communications and computations [9], whose optimal value can be found experimentally.

As shown in Figure 6(b), the computational kernel performs one update of the submatrix C_i with the portions of pivot block column A_i and pivot block row B_i: $C_i += A_i \times B_i$. In this work, the vendor-optimized single- and multi-threaded *gemm* kernels from the ACML 4.4 library are used straightforwardly for the implementation of CPU computational kernels. The GPU computational

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2014.2375202, IEEE Transactions on Computers

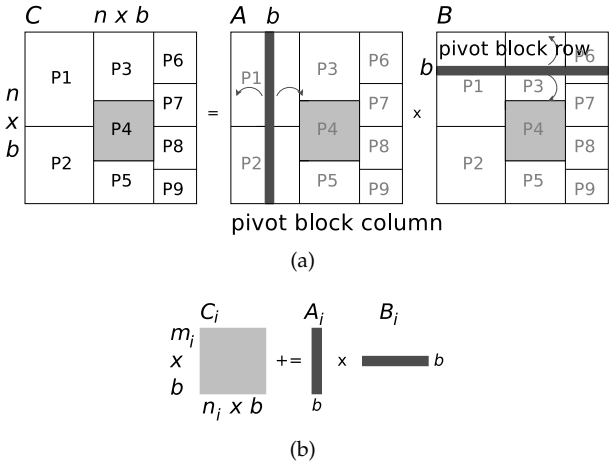


Fig. 6. (a) Heterogeneous parallel matrix multiplication (b) The computational kernel

kernels are implemented in CUDA 4.1. The memory capacity of the GPU is usually small compared to that of the host. To overcome this limit, we developed out-of-core computational kernels that are able to handle large problems. In this work, we only optimize the GPU computational kernels at a high level. We do not develop low-level GPU BLAS kernels but rely on the vendor-optimized CUBLAS library. The *gemm* operation in the GPU computational kernels is implemented by invoking the *gemm* kernel from CUBLAS 4.1. To maximize parallel execution between the host, the device and the PCIe bus, asynchronous function calls are used and executed concurrently through CUDA *streams*. Apart from these, other optimization techniques have not been considered.

In the out-of-core computational kernel, when the problem size fits in the device memory, the *gemm* execution is preceded by transferring submatrices A_i, B_i, and C_i from the host to the device memory and followed by transferring the updated C_i back. When the problem size exceeds the device memory, C_i is split in two dimensions into a number of small rectangular matrix blocks, A_i and B_i are split in one dimension into horizontal and vertical slices respectively. Then, C_i is updated block by block in the device, leading to a large number of data transfers between the host and device memory. In order to reduce the communication cost, we transfer the blocks over the PCIe link asynchronously in both directions, using multiple CUDA *streams*, and overlap data transfers with the *gemm* executions. More detail can be found in [10]. We also developed an out-of-core computational kernel for multiple GPUs that share a PCIe link. The submatrices are also split into a number of rectangular blocks. Then, C_i is updated block by block using all GPUs. We schedule data transfers and *gemm* executions so that the PCIe link is used exclusively by one device at a time, thereby avoiding PCIe contention.

Figure 3 shows the speed functions of a GPU processing unit executing the out-of-core computational kernels. As we can see, when the problem size fits in device memory, the performance is relatively high. However, when the problem size exceeds the device memory, the performance decreases dramatically due to the extra communication overhead incurred by transferring the rectangular blocks between the host and device memory.

4.1.1 FPM-based Data Partitioning on a Hybrid Server

In this section, we present experimental results of FPM-based data partitioning on Ig, a multicore and multi-GPU server specified in Table 2. Ig consists of four NUMA nodes with 6 cores and 16 GB memory each, and is equipped with two heterogeneous GPUs. The functional performance models were built using proposed performance modeling methods.

In this experiment, the single-threaded CPU and single-GPU computational kernels were used. Therefore, each GPU and its dedicated CPU core made a combined processing unit, which was modeled by an abstract processor. Each of other 22 CPU cores was modeled by an abstract processor. The matrices were partitioned based on the performance models of these abstract processors in order to balance the workload. The two CPU cores dedicated to GPUs were from two different NUMA nodes. For each of the two NUMA nodes, a speed function, $g(x)$, was built for the GPU processing unit by executing the GPU computational kernel separately. Moreover, a speed function, $s_5(x)$, was built for other 5 CPU cores by executing 5 CPU computational kernels simultaneously. For each of other two NUMA nodes, a speed function, $s_6(x)$, was built for all CPU cores by executing 6 CPU computational kernels simultaneously. Note that $g(x)$ and $s_5(x)$ were built separately because the impact of resource contention between CPU and GPU processing units is insignificant in this case.

To demonstrate the efficiency of the FPM-based data partitioning, we compared the execution time of this application when the matrices were partitioned using different partitioning algorithms. The results are presented in Figure 7. The execution of the application based on homogeneous partitioning was unbalanced, being dominated by the slowest processing units (i.e. CPU cores). Both the CPM-based and FPM-based data partitioning balanced the workload when problem sizes were relatively small. However, starting from problem size 50×50, the CPM-based algorithm failed to balance the workload. The FPM-based data partitioning algorithm

TABLE 2
Specifications of the hybrid server Ig

Hybird Server	Ig (AMD CPU + NVIDIA GPU)		
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
Number of PCIe		1	1

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2014.2375202, IEEE Transactions on Computers

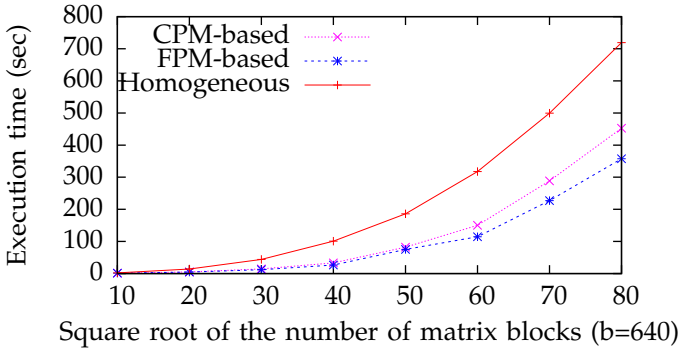


Fig. 7. Execution time of the parallel matrix multiplication with different data partitioning algorithms

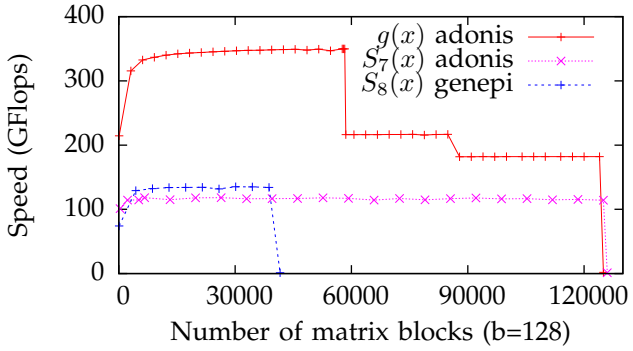


Fig. 8. Pre-built functional performance models of CPU and GPU processing units of Grenoble nodes

reduced the execution time of the application over the CPM-based and homogeneous partitioning algorithms by up to 21% and 50% respectively.

4.1.2 Partial FPM-based Data Partitioning on Heterogeneous GPU-accelerated Multicore Clusters

In this section, we present experimental results of partial FPM-based data partitioning on heterogeneous GPU-accelerated multicore clusters. A total number of 40 dedicated nodes from two clusters Adonis and Genepi were used, specified in Table 3. Each Adonis node is equipped with a GPU. All nodes are connected with Infiniband 40G/20G network. The platform is heterogeneous in terms of computing devices and memory capacity. The partial functional performance models were built dynamically at application run-time to a sufficient accuracy level using proposed performance modeling methods.

The parallel matrix multiplication application was executed with one multi-threaded CPU and one single-GPU computational kernels on each Adonis node, and one multi-threaded computational kernel on each Genepi node. Therefore, on each Adonis node, the GPU and its dedicated CPU core made a combined processing unit and were modeled by an abstract processor. The other CPU cores made another combined processing unit and were modeled by another abstract processor. On each Genepi node, all CPU cores made a combined processing unit and were modeled by an abstract processor. The matrices were partitioned based on the performance models of these abstract processors in order to balance the workload. For each Adonis node, a speed function, $g(x)$, was built for the GPU processing unit by executing the single-GPU computational kernel. Moreover, a speed function, $S_7(x)$, was built for the CPU processing unit by executing the multi-threaded computational kernel. For each Genepi node, a speed function, $S_8(x)$, was built by executing the multi-threaded CPU computational kernel. The pre-built speed functions are presented in Figure 8 for reference. Note that $g(x)$ and $S_7(x)$ were built separately because the impact of resource contention between CPU and GPU processing units was insignificant in this case.

To demonstrate the efficiency of the partial FPM-based data partitioning, we compared the execution time of this application when the matrices were partitioned using different partitioning algorithms. The results are presented in Figure 9. The execution of the application based on homogeneous partitioning was unbalanced, being dominated by the slowest processing units, i.e. the CPU processing unit of each Adonis node. Both CPM-based and partial FPM-based data partitioning balanced the workload when problem sizes were relatively small, i.e. problem sizes up to 1100×1100. However, starting from problem size 1200×1200, the CPM-based algorithm failed to balance the workload. The partial FPM-based

TABLE 3
Specifications of the GPU-accelerated multicore cluster

Cluster	Adonis (CPU + GPU)		Genepi (CPU)
Processor	Intel Xeon E5520	Nvidia Tesla C1060	Intel XeonE5420 QC
Cores	2 × 4 cores	240 cores	2 × 4 cores
Clock Rate	2.27 GHz	602MHz	2.5 GHz
Memory Size	24 GB	4 GB	8 GB
Nodes	9	1 GPU/node	31
Network	Infiniband 40G		Infiniband 20G

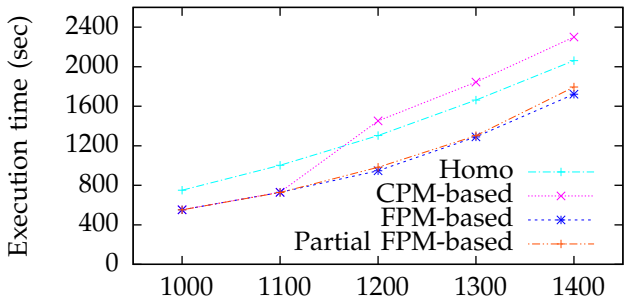


Fig. 9. Execution time of the parallel matrix multiplication application with different data partitioning algorithms

TABLE 4
Estimated overhead for building then partial FPMs

Problem size	1000^2	1100^2	1200^2	1300^2	1400^2
Iterations	2	2	11	11	9
Overhead	0.2%	0.18%	0.92%	0.85%	0.64%

data partitioning reduced the execution time over homogeneous and CPM-based partitioning by up to 13% and 22% respectively. Data partitioning based on partial and pre-built functional performance models resulted in similar execution time.

Table 4 shows the estimated overhead for building the partial functional performance models. The first row shows the problem size, whose square root is the number of iterations of the application for the given problem size. The second row shows the number of iterations that were actually run for building the partial FPMs. The third row shows the ratio of the number of iterations for building the partial FPMs to the number of iterations of the application. This ratio could be roughly seen as the ratio of the time for building partial FPMs to the computation time of the application, because the same computational kernel was executed when building partial FPMs and running the application. We can see that the time for building partial FPMs is neglectable compared to the computation time of the application.

4.2 Numerical Simulation of Lid-driven Cavity Flow

Computational Fluid Dynamics (CFD) is the analysis of systems involving fluid flow, heat transfer, and associated phenomena by means of computer-based numerical simulation. Over the past few decades, computational fluid dynamics has become a practical cornerstone of most fluid and mechanical engineering applications.

The lid-driven cavity flow is a classical test problem for incompressible, lamina flow of Newtonian fluids, which has been investigated by many researchers and accurate solutions are available in the literature (see [11]). The standard case is fluid contained in a square domain with Dirichlet boundary conditions on all sides, with three stationary sides and a lid moving with a tangential unit velocity. The fluid motion in the driven cavity is governed by the Navier-Stokes equations:

$$\begin{cases} \partial_t \boldsymbol{U} + (\boldsymbol{U} \cdot \nabla)\boldsymbol{U} - \dfrac{1}{Re}\Delta \boldsymbol{U} + \nabla p = 0 \\ \nabla \cdot \boldsymbol{U} = 0 \end{cases}$$

to be solved in $\Omega = [0,1] \times [0,1]$. In these equations, $\boldsymbol{U}$ is the velocity vector, t is time, Re is the Reynolds number, and p is the pressure.

The Navier-Stokes equations are difficult to solve analytically, so numerical methods are often used. In numerical methods, the geometric domain and the conservation equations are discretized, producing a system of algebraic equations whose solution is used to approximate the solution of the conservation equations. For this test case, a system of pressure-velocity coupled equations is produced from discretization. In this work, we use PISO [12] to solve this linear equations system. In PISO, the velocity and pressure are calculated by solving linear systems of velocity equations and pressure-correction equations. We choose conjugate gradient algorithm (CG) [13] to solve the symmetric linear pressure-correction equations, and bi-conjugate gradient stabilized algorithm (BiCGSTAB) [14] to solve the non-symmetric linear velocity equations in this work.

Parallel computing in CFD is usually based on domain decomposition, which is essentially data parallelism. The geometry domain is divided into a number of subdomains, each assigned to a process. The problem is solved on the entire domain from problem solutions on subdomains. Since processes needs data that resides in other subdomains, exchange of data between processes is required. Solving the sparse linear equation systems is the compute-intensive and time-consuming part in CFD applications. Therefore, we build functional performance models of processing units executing the linear equation solvers of this application, i.e. CG and BICGSTAB. Then, we perform heterogeneous domain decomposition based on functional performance models in order to balance the workload on heterogeneous platforms. The speed of processing units is calculated by dividing the number of floating-point operations by the computation time of one iteration of the linear equation solver (communication time eliminated). In this work, one-dimensional heterogeneous domain decomposition is used, which is able to handle rectangle geometry domains, but the communication overhead is not optimized.

In PISO, both CG and BiCGSTAB solvers will be invoked at each time step. Because the BiCGSTAB solver requires almost exactly twice as many linear algebra operations of each type per iteration as the CG solver, both the complexity and computation time per iteration of the BiCGSTAB solver will twice as much as the CG solver. As a result, their performance are similar, which is proved by our experimental results. Since the two linear equation solvers have almost the same performance, it is reasonable to partition the workload based on performance models of either solver. In this work, we choose to use the FPMs of the CG solver.

The experimental platform is the Adonis cluster, specified in Table 3. Most linear algebra operations involved in the linear equation solvers are implemented using CUSP 0.3, a C++ template library for sparse matrix computations for both CPU and GPU computing systems. Experiments were conducted on both a single Adonis node and a cluster of Adonis nodes.

In the experiment on a single Adonis node, the CFD test case was executed with one GPU CG solver on the GPU with its dedicated CPU core, and one CPU CG solver on each of other seven CPU cores. Therefore, the GPU and its dedicated CPU core made a combined processing and were modeled by an abstract processor. Each of other seven cores was modeled by an abstract

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2014.2375202, IEEE Transactions on Computers

processor. The domain was decomposed based on the performance models of these abstract processors in order to balance the workload. An Anodis node consists of two NUMA nodes. For the NUMA node to which the GPU was connected, a speed function, $g(x)$, for the GPU processing unit, and a speed function, $s_3(x)$, for the other 3 CPU cores were built by executing the GPU CG solver on the GPU processing unit and three CPU CG solvers on other 3 CPU cores simultaneously. For CPU cores of the other NUMA node, a speed function, $s_4(x)$, was built by executing 4 CPU CG solvers simultaneously. These speed functions are presented in Figure 10.

Figure 11 shows the actual speedup and the estimated upper bound on the speedup in execution time of the CFD test case when using FPM-based heterogeneous decomposition over homogeneous decomposition. It is worth noting that the speedup that could be achieved depends on the level of the processor heterogeneity of the experiment platform.

In the experiment on a single Adonis node, the speedup is up to 1.23. For a certain problem size, if the domain is decomposed by FPM-based decomposition method, let g and s_3 denote the speed of the GPU processing unit and the CPU cores of the same NUMA node, s_4 denote the speed of the CPU cores of the other NUMA node, with a heterogeneous workload distribution. For the same problem size, if the domain is decomposed by homogeneous decomposition method, let $\bar{g}$, $\bar{s}_3$ and $\bar{s}_4$ denote the speed of the corresponding processing units. We can calculate that the speedup in computation time $S_1 = (g+3*s_3+4*s_4)/(8*\bar{s}_4)$ is up to 1.4. We can estimate that the speedup in executing time $S_2 = (S_1+r_f)/(1+r_f)$, where r_f is the ratio of the communication time to the computation time measured in experiments when the solution domain is decomposed using FPM-based decomposition method. Based on experimental results, r_f is around 0.6, therefore, S_2 is up to 1.25. As we can see, the actual speedup and the estimated upper bound are quite close, which demonstrates that the FPM-based heterogeneous domain decomposition is able to balance the load on a GPU-accelerated multicore node.

The tool for mesh generation we rely on in the experiments can only run on a single node, so the range of problem size with which we can experiment is limited by the capacity of the main memory of a single node. If the largest number of control volumes that can be generated on a single node are distributed evenly to all processing elements of the Adonis cluster, each receives a relatively small number of control volumes. In this case, the computing capacity of GPU is barely fulfilled and the level of performance heterogeneity of CPU and GPU is low, which is of no interest in this study.

In order to study the FPM-based domain decomposition on a hybrid cluster with a relatively wide range of problem size, one GPU (with a dedicated CPU core) and seven CPU cores were used, each from a Adonis node. For heterogeneous domain decomposition, a speed function, $s_1(x)$, was built by executing the CPU CG solver on a single CPU core, and a speed function, $g(x)$, was built by executing the GPU CG solver independently. These speed functions are presented in Figure 10. Compared to the performance measured independently, the performance of the GPU CG solver measured under contention from neighboring CPU cores reduces by around 30% because of resource contention from other CPU cores.

On the Adonis cluster, the speedup in execution time is around 1.1. Using the same method as above, we can calculate that the speedup in computation time is up to 1.25. Based on experimental results, the ratio of the communication time to the computation time r_f is around 0.67. Therefore, we can estimate that the speedup in execution time S_2 is up to 1.15. As we can see, the actual speedup and the estimated upper bound are reasonably close, which demonstrates the effectiveness of FPM-based heterogeneous domain decomposition.

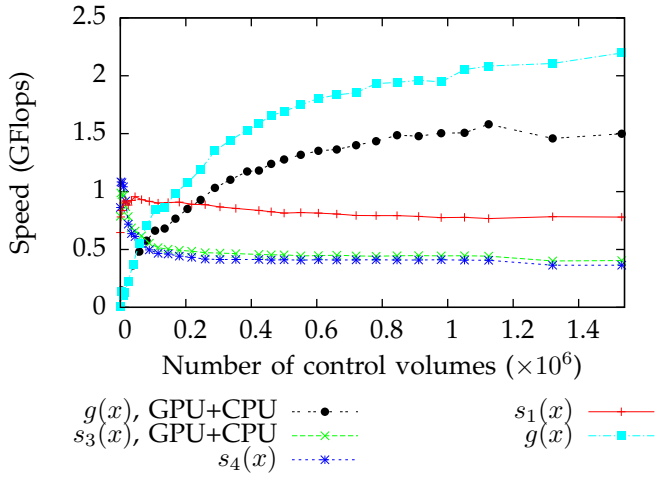


Fig. 10. Speed functions of the CG solver built in different configurations on an Adonis node

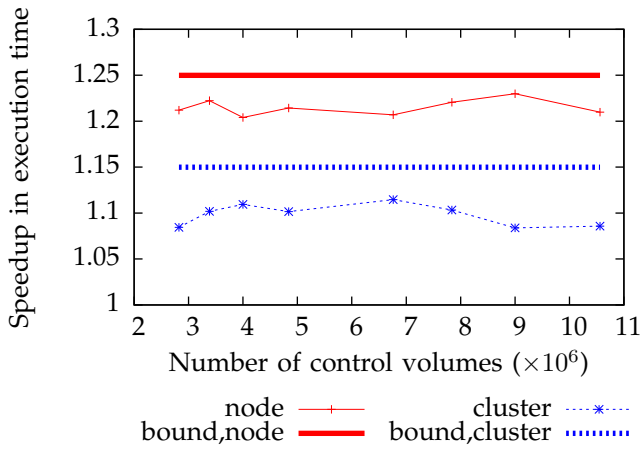


Fig. 11. The actual speedup and the estimated upper bound on the speedup in execution time on a single Adonis node and on the Adonis cluster

5 Related Work

Performance modeling is a very common technique used for optimization of parallel applications on HPC platforms. A large number of performance models have been proposed for multicore and GPU architectures. For example, for multicore systems, the Roofline model [15] ties floating-point and memory performance together in a two dimensional graph with the bounds representing a set of recommendations how to reengineer the application; [16] proposed an integrated power and performance prediction model for GPU architecture.

The load balancing algorithms can be classified into two categories, namely *static* and *dynamic* algorithms. Static algorithms, such as data partitioning [1], [2], [17], require a priori information about the parallel application and platform. Static algorithms rely on accurate performance models as input to predict the future execution of the application. Static algorithms are particularly useful for applications for which data locality is critical because data redistribution incurs significant overhead. However, these algorithms are unable to balance on non-dedicated platforms, where load changes with time. Dynamic algorithms, such as task scheduling and work stealing [18], [19], [20], balance the load by moving fine-grained tasks between processors during the execution. Dynamic algorithms do not require a priori information about execution but may incur large communication overhead due to data migration. Dynamic algorithms can use static partitioning for the initial step due to its provably near-optimal communication cost and bounded tiny load imbalance.

Most of the state-of-the-art data partitioning algorithms for GPU-accelerated HPC platforms, such as [1], [2], [21], [22], make the assumption that the speed of a process does not change with problem size, therefore, the relative speed of processing elements is constant. They are referred to as data partitioning based on constant performance models (CPM). Graph partitioning software, e.g. Metis [23], can be used to partition graphs and meshes for heterogeneous platforms. Similarly, the speed of heterogeneous processors is represented by different weights (constants), based on which graph partitioning algorithms balance workload and optimize total communication volume. However, as demonstrated in [5], CPM-based data partitioning algorithms may return solutions that are far away from optimal or may fail to converge.

Several high-level programming systems for heterogeneous hardware-accelerated multicore platforms have been developed. Qilin [24] automatically generates code and uses adaptive mapping for performance tuning. During the training run, Qilin executes the program at different input sizes on CPUs and GPUs separately, and build performance models to determine workload partitioning between CPUs and GPUs. Peppher [25] employs component implementation variants of performance-critical parts of applications tailored to different architectures, and relies on the compiler and runtime system to select and schedule component tasks on available computing resources. PetaBricks [26], an implicitly parallel language and compiler, uses an empirical autotuning approach to search the space of possible implementations at installation time to construct poly-algorithms that combine many different algorithmic techniques to obtain better performance. Other heterogeneous programming systems with similar functionality include Merge [18], StarPU [27], and Elastic Computing [28].

The integration of multiple GPUs into multicore systems introduces complex performance phenomena, including non-uniform memory (NUMA) access and contention for shared system resources. In [29], the differences in GPU bandwidth due to NUMA effect are briefly documented. In [6], the effects of NUMA mapping on data transfers for a multi-GPU system of dual-IOH architecture are quantified.

In [10], we presented the preliminary work on performance modeling and FPM-based data partitioning on GPU-accelerated multicore systems. In that work, the data parallel applications were configured with single-threaded CPU and single-GPU computational kernels. The proposed methods were evaluated only with parallel matrix multiplication on a single multicore and multi-GPU server.

6 Conclusion

We aim at the maximum performance of data parallel scientific applications on heterogeneous multicore and multi-GPU platforms. Previously, we proposed FPM-based data partitioning to balance the workload of data parallel applications on uniprocessor heterogeneous platforms. In this work, we propose methods of performance modeling and performance measurement on multicore and multi-GPU systems, and extend the FPM-based data partitioning to heterogeneous multicore and multi-GPU platforms. We evaluate the proposed methods with two data parallel applications, namely, parallel matrix multiplication and numerical simulation of lid-driven cavity flow, on a multicore/multi-GPU server and on a heterogeneous GPU-accelerated multicore cluster. Experimental results demonstrate that FPM-based data partitioning is able to balance the workload on target platforms and deliver good performance. Energy considerations are out of the scope of the presented work.

The FPM-based data partitioning has been found useful and efficient for balancing the workload of many data parallel applications on modern heterogeneous platforms. However, there still needs more effort to be invested in improving this approach in several aspects. For example, so far only the computing performance of processing units is used for data partitioning. As the communication overhead between processes is not taken into account, we rely on other algorithms to optimize the total communication overhead. In our applications, the performance model of a processing unit is a one-dimensional function of the problem size, i.e. the total

number of the computation units. However, in some dense matrix applications on highly heterogeneous platforms, the performance of a processing unit may depend on the shape of the matrix block assigned to it. In that case, the multi-dimensional performance model with more than one parameter may be required to accurately describe the performance. For large complex applications where the computational kernel cannot be easily separated from the application or there exists more than one computational kernel, it may take more effort to balance the workload with this approach. All these problems will be studied in the future work.

Acknowledgments. This research was conducted with the financial support of SFI 08/IN.1/I2054, NSFC No.61403402, and UCD-CSC scholarship. Experiments were carried out on platforms provided by the Innovative Computing Lab (Univ. of Tennessee) and Grid5000.

REFERENCES

[1] M. Fatica, "Accelerating Linpack with CUDA on heterogenous clusters," in *GPGPU-2*. ACM, 2009, pp. 46–51.
[2] C. Yang *et al.*, "Adaptive optimization for petascale heterogeneous CPU/GPU computing," in *Cluster'10*, 2010, pp. 19–28.
[3] A. Lastovetsky and R. Reddy, "Data partitioning with a functional performance model of heterogeneous processors," *Int. J. High Perform. Comput.*, vol. 21, pp. 76–90, 2007.
[4] A. Lastovetsky and R. Reddy, "Distributed data partitioning for heterogeneous processors based on partial estimation of their functional performance models," in *Euro-Par'09*, 2009, pp. 91–101.
[5] D. Clarke *et al.*, "Dynamic load balancing of parallel computational iterative routines on highly heterogeneous HPC platforms," *Parallel Processing Letters*, vol. 21, pp. 195–217, 2011.
[6] K. Spafford *et al.*, "Quantifying NUMA and contention effects in multi-GPU systems," ser. GPGPU-4, 2011, pp. 11:1–11:7.
[7] D. Clarke, Z. Zhong, V. Rychkov, and A. Lastovetsky, "Fupermod: A framework for optimal data partitioning for parallel scientific applications on dedicated heterogeneous hpc platforms," in *PaCT-2013*, ser. LNCS. Springer, 2013, vol. 7979, pp. 182–196.
[8] D. Clarke *et al.*, "Column-based matrix partitioning for parallel matrix multiplication on heterogeneous processors based on functional performance models," in *HeteroPar'11*, 2011, pp. 450–459.
[9] J. Choi, "A new parallel matrix multiplication algorithm on distributed-memory concurrent computers," in *HPC Asia*, 1997, pp. 224 –229.
[10] Z. Zhong *et al.*, "Data partitioning on heterogeneous multicore and multi-gpu systems using functional performance models of data-parallel applications," in *Cluster*, 2012, pp. 191–199.
[11] U. Ghia, K. N. Ghia, and C. T. Shin, "High-Re solutions for incompressible flow using the Navier-Stokes equations and a multigrid method," *J. Comput. Phys.*, vol. 48, pp. 387–411, 1982.
[12] R. I. Issa, "Solution of the implicitly discretised fluid flow equations by operator-splitting," *J. Comput. Phys.*, vol. 62, no. 1, pp. 40–65, Jan. 1986.
[13] G. H. Golub and C. F. Van Loan, *Matrix Computations (3rd Ed.)*. Baltimore, MD, USA: Johns Hopkins University Press, 1996.
[14] H. A. van der Vorst, "Bi-cgstab: A fast and smoothly converging variant of bi-cg for the solution of nonsymmetric linear systems," *SIAM J. Sci. Stat. Comput.*, vol. 13, no. 2, pp. 631–644, Mar. 1992.
[15] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, pp. 65–76, Apr. 2009.
[16] S. Hong and H. Kim, "An integrated gpu power and performance model," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA '10. New York, NY, USA: ACM, 2010, pp. 280–289.
[17] Y. Ogata *et al.*, "An efficient, model-based CPU-GPU heterogeneous FFT library," in *IPDPS 2008*, 2008, pp. 1 –10.
[18] 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.
[19] C. Augonnet *et al.*, "Automatic calibration of performance models on heterogeneous multicore architectures," in *EuroPar*, 2009.
[20] G. Quintana-Ortí *et al.*, "Solving dense linear systems on platforms with multiple hardware accelerators," *SIGPLAN Not.*, vol. 44, pp. 121–130, 2009.
[21] I. Galindo *et al.*, "Dynamic Load Balancing on Dedicated Heterogeneous Systems," in *EuroPVM/MPI*. Springer, 2008, pp. 64–74.
[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] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM J. Sci. Comput.*, vol. 20, no. 1, pp. 359–392, Dec. 1998.
[24] C.-K. Luk, S. Hong, and H. Kim, "Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping," in *MICRO-42*, 2009, pp. 45–55.
[25] S. Benkner *et al.*, "High-level support for pipeline parallelism on many-core architectures," ser. Euro-Par'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 614–625.
[26] P. M. Phothilimthana *et al.*, "Portable performance on heterogeneous architectures," ser. ASPLOS '13. New York, NY, USA: ACM, 2013, pp. 431–444.
[27] C. Augonnet *et al.*, "StarPU: A unified platform for task scheduling on heterogeneous multicore architectures," in *EuroPar*, 2009, pp. 863–874.
[28] J. R. Wernsing and G. Stitt, "Elastic computing: A framework for transparent, portable, and adaptive multi-core heterogeneous computing," in *LCTES 2010*. ACM, 2010, pp. 115–124.
[29] V. Kindratenko *et al.*, "GPU clusters for high-performance computing," in *CLUSTER '09*, 2009, pp. 1–8.

Ziming Zhong received his bachelor degree in simulation engineering, and his master degree in control science and engineering from National University of Defense Technology, China, in 2007 and 2009 respectively. He is currently a PhD candidate in the School of Computer Science and Informatics, University College Dublin, Ireland. His research interests include performance modelling of processors in heterogeneous multicore and multi-GPU platforms.

Vladimir Rychkov received a PhD degree from the Russian Academy of Sciences in 2005. His main research interests include high performance computing (performance modeling and optimization of parallel applications), software engineering (advanced programming techniques and runtime environments), computer-aided engineering (finite element method, domain decomposition, unstructured mesh algorithms).

Alexey Lastovetsky received a PhD degree from the Moscow Aviation Institute in 1986, and a Doctor of Science degree from the Russian Academy of Sciences in 1997. His main research interests include algorithms, models, and programming tools for high performance heterogeneous computing. He has published over a hundred technical papers in refereed journals, edited books, and international conferences. He authored the monographs *Parallel computing on heterogeneous networks* (Wiley, 2003) and *High performance heterogeneous computing* (Wiley, 2009).