# Data Partitioning on Heterogeneous Multicore and Multi-GPU Systems Using Functional Performance Models of Data-Parallel Applications

Ziming Zhong, Vladimir Rychkov, Alexey Lastovetsky
*Heterogeneous Computing Laboratory*
*School of Computer Science and Informatics, University College Dublin*
*Dublin, Ireland*
*Email: ziming.zhong@ucdconnect.ie, {vladimir.rychkov, alexey.lastovetsky}@ucd.ie*

***Abstract*—Transition to hybrid CPU/GPU platforms in high performance computing is challenging in the aspect of efficient utilisation of the heterogeneous hardware and existing optimised software. During recent years, scientific software has been ported to multicore and GPU architectures and now should be reused on hybrid platforms. In this paper, we model the performance of such scientific applications in order to execute them efficiently on hybrid platforms. We consider a hybrid platform as a heterogeneous distributed-memory system and apply the approach of functional performance models, which was originally designed for uniprocessor machines. The functional performance model (FPM) represents the processor speed by a function of problem size and integrates many important features characterising the performance of the architecture and the application. We demonstrate that FPMs facilitate performance evaluation of scientific applications on hybrid platforms. FPM-based data partitioning algorithms have been proved to be accurate for load balancing on heterogeneous networks of uniprocessor computers. We apply FPM-based data partitioning to balance the load between cores and GPUs in the hybrid architecture. In our experiments with parallel matrix multiplication, we couple the existing software optimised for multicores and GPUs and achieve high performance of the whole hybrid system.**

***Keywords*-performance model; data-parallel application; multicore; GPU; data partitioning.**

## I. Introduction

Heterogeneous multiprocessor systems, where multicore CPUs are coupled with GPU accelerators, have been widely used in high performance computing due to better power efficiency and performance/price ratio. Transition to hybrid CPU/GPU architectures is challenging in the aspects of the efficient utilisation of the hardware and the reuse of the software stack. Our target architecture is a dedicated hybrid HPC platform, characterised by a stable performance in time and a significant overhead associated with data migration between computing devices.

During recent years, scientific software has been ported to multicore and GPU architectures. To develop applications for multicores, some legacy programming techniques were adopted, such as OpenMP and MPI. Nevertheless, introduction of multicores in HPC resulted in significant refactoring of existing parallel applications. To facilitate development of general-purpose applications on GPUs, new programming models, such as Brook+, CUDA, and OpenCL, were created. A large number of algorithms and specific applications have been successfully ported to GPUs claiming substantial speedup over their optimised CPU counterparts. Since multicore and GPU applications are highly optimised for their platforms, they should be reused on hybrid systems. In order to achieve the maximum performance of hybrid multicore and multi-GPU systems, it is essential to balance workloads between CPU cores and GPUs, taking into account heterogeneity of processors. However, load balancing on this platform is complicated by several factors, such as resource contention, limited local memory on GPUs, and ever-increasing gap between GPU performance and communication speed of PCI Express.

In this paper, we target data-parallel scientific applications, such as linear algebra routines, digital signal processing, computational fluid dynamics They are characterised by divisible computational workload, which is directly proportional to the size of data and dependent on data locality. Computational kernels optimised 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. We consider a hybrid platform as a heterogeneous distributed-memory system, and therefore apply data partitioning, a static 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 [1], constants representing the sustained performance of the application on CPU/GPU were used to partition data. The constants were found a priori. In [2], 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 the partitioning of the problem results in some tasks fitting into different levels of memory hierarchy (i) or processors/devices switch between different codes to solve the same computational problem (ii).

In [3], the execution time of CPU/GPU was approximated by linear functions of problem size, and an empirical approach to estimate linear performance models was proposed. This model was successfully used to balance the load in the neighbourhood of some problem size. A more elaborate analytical predictive model was proposed in [4]. 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 [4], it was admitted that the linear models might not fit the actual performance in the case of resource contention (iii), and therefore, data partitioning algorithms might fail to balance the load.

In this paper, we apply data partitioning based on functional performance models, which was originally designed and proved to be accurate for uniprocessor machines [5]. The functional performance model (FPM) represents the processor speed by a function of problem size. It is built empirically and integrates many important features characterising 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)-(ii). Since a multicore and a GPU have separate memory and different programming models, we can build their FPMs independently. We show that this approach is accurate despite that resources are shared in hybrid architectures (iii).

Applying the functional performance model to hybrid platforms is complicated by resource contention. In our previous work [6], we proposed a method for accurate measurement of the speed of multicore nodes, which takes into account resource contention. This method was used to build the functional performance models of individual multicore nodes of a heterogeneous multicore cluster. The FPMs built this way were used for inter-node data partitioning. In this paper, we apply this approach to a hybrid platform that consists of several multicore sockets with their own separate local memory. First, we build the functional performance models of the multicore sockets, which are considered as heterogeneous nodes because some of them are coupled with GPUs. Then, we execute the FPM-based data partitioning algorithm to balance the load between sockets.

In hybrid CPU/GPU systems, GPUs are usually used as accelerators for executing tasks. 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 GPU. It can be extended to infinity for the out-of-core applications, which can handle the large amount of data stored in the low-speed memory.

In this paper, we demonstrate how to utilise efficiently a typical hybrid platform that consists of several multicore sockets and multiple GPUs. Without loss of generality we experiment with a heterogeneous parallel matrix multiplication application. We build the functional performance models of multicore sockets and GPUs, executing the optimised BLAS kernels, ACML and CUBLAS respectively. Using these models in the FPM-based data partitioning algorithm, we obtain the optimal distribution of matrices between multicore sockets and GPUs, and balance the load in the hybrid system. We use this application for demonstration purposes. Design of optimal matrix multiplication algorithms for hybrid platforms is out of scope of this paper.

The rest of this paper is organised as follows. In Section II, related work is discussed. In Section III, we describe the techniques for performance measurement on hybrid multicore and multi-GPU systems. In Section IV, we describe a heterogeneous parallel application to be executed on a hybrid platform. In Section V, we present the speed functions of multiple cores and GPUs built for this application on the hybrid platform and analyse how different factors affect their shape. In Section VI, we demonstrate that data partitioning based on functional performance models allows for optimal execution of heterogeneous applications on hybrid platforms.

## II. Related Work

In this section, we review performance models used in heterogeneous programming systems and applications for hybrid platforms. Then, we briefly describe the functional performance model and the FPM-based data partitioning, which will be applied to a hybrid platform in this work.

Performance modelling is a very common technique used for optimisation of parallel applications on HPC platforms. Building such models usually requires certain knowledge of both the application algorithm and the underlying architecture. It may involve significant programming efforts and incur an extra performance overhead. A number of studies defined the ratio of the number of memory operations to the number of floating-point operations to describe the performance of the program by showing how fast the data is supplied and processed in the application. The Roofline model [7] 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. However, rewriting the code may result to minor performance improvements but major development costs, especially on hierarchical heterogeneous platforms.

For hybrid platforms, a number of heterogeneous programming systems have been developed [3], [8], [9], [10]. They require reimplementing the application in a high-level language, which is translated into the instruction set architecture (ISA) or the application programming interface (API) of the target platform. These systems use performance models in order to execute the translated code efficiently. In the performance models, the speed of a device is given by a constant positive number and calculated from the history of measurements of representative kernels.

The load balancing algorithms used in heterogeneous programming systems can be classified as static or dynamic. ***Static*** algorithms (for example, those based on data partitioning), which are used in [1], [2], [3], [4] require 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. ***Dynamic*** algorithms (such as task scheduling and work stealing), which are used in [8], [11], [12] 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 due to data migration. Dynamic algorithms often use static partitioning for their initial step due to its provably near-optimal communication cost, bounded tiny load imbalance, and lesser scheduling overhead [13].

In this paper, we focus on data-parallel scientific applications, where computational workload is directly proportional to the size of data and dependent on data locality. Our target architecture is a dedicated hybrid HPC platform. Instead of reimplementing our target applications for this platform, we propose using data partitioning algorithms in order to optimally distribute computations between computing devices. Data partitioning algorithms map all data to processors and do not redistribute it during the calculation. However, they rely on the accuracy of performance models, and therefore, depend on how the models are defined and built. With simplistic constant performance models, data partitioning algorithms may return solutions that are far away from optimal or may even fail [14].

The functional performance model (FPM) [5] more accurately approximates the processor speed because it represents the speed by a function of problem size. The processor speed is defined as the number of computation units performed by the processor per second, including all contributions from clock cycles, memory operations and hierarchy, and operating system overhead. The processor speed is application specific since computation units can be defined differently for different applications. The problem size is understood as a set of parameters characterizing the amount and layout of data stored and processed during the execution of the computational task. The number and semantics of the problem size parameters are application specific. It is assumed that the amount of stored data and the amount of computation will increase with the increase of any of the problem size parameters. High performance of parallel scientific applications on hybrid platforms can be achieved when all processors complete their work within the same time. This requirement is satisfied by partitioning the computational workload and, hence, data unevenly across all processors. FPM-based data partitioning algorithms has been proved to be accurate for load balancing on heterogeneous uniprocessor [5] and multicore [6] clusters.

The processor speed is found experimentally by measuring the execution time. This time can be found by benchmarking the full application. This benchmarking can be done more efficiently by using a serial code, the speed of execution of which is the same as that of the application but the execution time of which is significantly less. We call such a code, performing much less computations but still representative for the application, a ***kernel***. For example, computationally intensive applications often perform the same core 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.

On multicore platforms, parallel processes interfere with each other through shared memory so that the speed of individual cores cannot be measured independently, and independent performance models cannot be defined for cores. In our previous work [6], we proposed to evaluate the performance of cores in a group, when all cores are executing the benchmarks in parallel. We proposed to measure a combined workload including both arithmetic and memory operations. Process binding and synchronization were employed to ensure consistent performance. This approach can be applied to the intra-node interactions between threads, involving shared memory programming and multithreaded runtime systems.

Interactions between CPUs and GPUs include data transfers between the host and GPU memory over PCI Express, launching of GPU kernels, and some other operations. Existing approaches and tools for performance measurement on heterogeneous GPU-accelerated systems are analysed in [15]. In the synchronous approach, a host CPU core observes the beginning and the end of an operation. This approach does not require any special measurement mechanisms and accurately reflects the performance of kernels implemented in synchronous libraries, such as CUBLAS. In the event queue and callback approaches, a GPU initiates the time measurement. In the first case, the code recording the state of the GPU is injected before and after the operation to measure. In the second case, the callback functions are registered at the GPU to be triggered before and after the operation. These approaches address multiple operations launched in a GPU stream, but entirely rely on the device manufacture to provide support for the event queue and callback concepts.

In this work, we apply the measurement technique for multicores [6] and the synchronous measurement approach for GPU [15] to a hybrid platform. Obtaining the timings for different parts of the hybrid platform, we build the functional performance models of the optimised kernels executed on CPUs and GPUs. Then, we apply the FPM-based data partitioning algorithm to balance the load between the processors.

## III. Performance Measurement on Hybrid Platforms

In this section, we describe how we measure the speed of cores and GPUs on hybrid platforms, present our experimental platform and analyse the results of measurements.

In a typical hybrid multicore and multi-GPU node, the host has multiple identical cores and a hierarchical memory and is connected with heterogeneous GPUs via the PCI Express connections. The host and devices have disjoint memory locations and explicit memory transfers are required for communication between them. The node executes a heterogeneous parallel application that invokes the libraries optimised for multicore and GPU respectively. We need to measure the speed of the processing elements in the system.

Our approach to performance measurement can be summarised 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 synchronised to minimise the idle computational cycles, aiming at the highest floating point rate for the application. Synchronisation 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.

GPU depends on a host process, which handles data transfer between the host and device and launches kernels on the device. A CPU core is usually dedicated to deal with the GPU, and can undertake partial computations simultaneously with the GPU. Therefore, we measure the combined performance of the dedicated core and GPU, including the overhead incurred by data transfer between them. Due to limited GPU memory, the execution time of GPU kernels can be measured only within some range of problem sizes, unless out-of-core implementations, which address this limitation, are available.

Our experimental platform is a hybrid multicore and multi-GPU node of NUMA architecture specified in Table I, which consists of 4 sockets with a six-core AMD processor and 16 GB memory each. It is accelerated by 2 NVIDIA GPUs. For speed measurements, we used the GEMM kernel from the ACML 4.4 (AMD Core Math Library) for CPU and from CUBLAS 4.1 (NVIDIA CUDA BLAS) for GPUs.

First, we measured the execution time of the ACML 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 CUBLAS 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 [16], 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 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

Table I
Specifications of the hybrid platform *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 | $4 \times 6$ cores | 1536 cores | 128 cores |
| Memory Size | $4 \times 16$ GB | 2048 MB | 1536 MB |
| Mem. Bandwidth | | 192.3 GB/s | 76.8 GB/s |

cores for resources. This observation allows us to measure the speed of multiple cores and GPUs independently with some accuracy.

## IV. Heterogeneous Parallel Application: Column-Based Matrix Multiplication

We demonstrate how FPMs can be used to evaluate the performance of parallel scientific applications on hybrid platforms by the example of the heterogeneous parallel column-based matrix multiplication with optimized communication volume proposed in [17]. This application was originally designed for heterogeneous network of processors. It takes the FPMs of heterogeneous processors as input, partitions the matrices using the FPM-based data partitioning algorithm, and performs the blocked matrix multiplication, using the GEMM kernel. We will execute this application on the hybrid platform presented in Section 3, using the kernels optimised for multicores and GPUs.

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. For simplicity, we work with square matrices. Figure 1(a) shows one iteration of the blocked matrix multiplication, with the blocking factor $b$. At each iteration of the main loop, pivot column of matrix $A$ and pivot row of matrix $B$ are broadcast horizontally and vertically, and then matrix $C$ is updated in parallel. The partitioning algorithm used in this application arranges the submatrices to be as square as possible, minimising the total volume of communications and balancing the computations on the heterogeneous processors. The blocking factor $b$ is a parameter of the application adjusting the granularity of communications and computations [18]. The optimal values of this parameter are found experimentally for each platform and GEMM implementation.

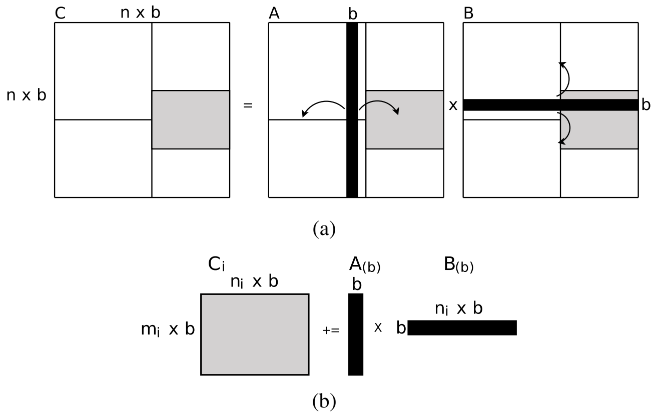


Figure 1. Heterogeneous parallel column-based matrix multiplication (a) and its computational kernel (b)

The absolute speed of the processor is defined as the total number of computations executed by this processor during the application divided by the total execution time. In order to measure the speed more efficiently, we make an assumption that the total execution time of the application can be approximated by multiplying the number of iterations of the application with the execution time of a single run of the computational kernel. As shown in Figure 1(b), the computational kernel for the processor $i$ performs one update of the submatrix $C_i$ with the parts of pivot column $A_{(b)}$ and pivot row $B_{(b)}$: $C_i+=A_{(b)} \times B_{(b)}$. Therefore, this speed is estimated more efficiently by measuring just one run of the kernel. This kernel is implemented by the GEMM routine of the Basic Linear Algebra Subprograms (BLAS). Having the same memory access pattern as the whole application, it reflects the whole computational workload. In this paper, we focus on data partitioning with respect to computational performance of processing elements, and to this end, we do not model the communication between processing elements; instead we arrange elements so that the communication volume is minimised [17].

## V. Functional Performance Models of Multiple Cores and GPUs

In this section, we build the functional performance models of the matrix multiplication application for multiple cores and GPUs, using the representative computational kernel of the application. We analyse different factors that affect the shape of the speed functions, which include configuration of the application, resource contention, and optimisation of the kernel.

We divide the FPMs for the computing devices of the hybrid platform specified in Table I into two groups:

1) **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.
2) **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) $x$.

Since the speed of the kernel for a given matrix area $x$ does not vary with the nearly square shapes of submatrices [17], we build the speed functions by timing the kernel with the submatrices of size $\sqrt{x} \times \sqrt{x}$.

Figure 2 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. In addition, the performance depends on the blocking factor $b$, a parameter of the application. To exploit optimisations implemented in the ACML kernel, which take

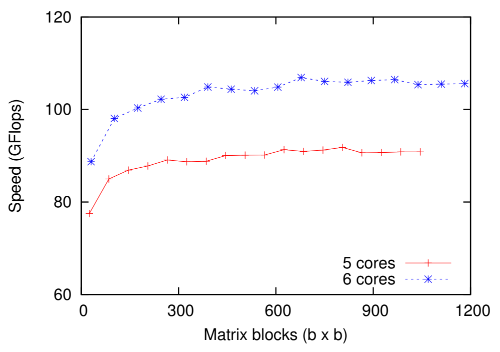


Figure 2. Speed functions of a socket, $s_5(x)$ and $s_6(x)$, in single precision with blocking factor $b = 640$

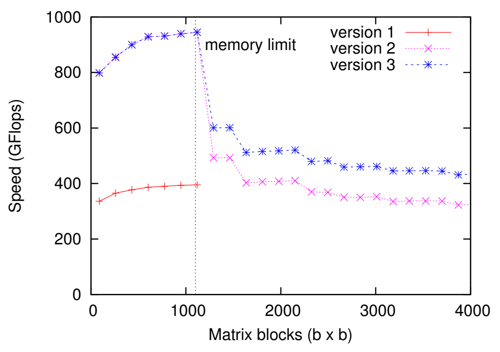


Figure 3. Speed functions of GeForce GTX680 (single precision, $b = 640$) built for kernels accumulating the intermediate results in the host memory (version 1); in device memory with out-of-core extension (versions 2); with overlapping of communications and computations (version 3)

into account memory hierarchy of a multicore architecture, we experimented with the blocking factor $b = 640$.

In Figure 3, 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.

The kernel shown in Figure 1(b) 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. Using the CUBLAS GEMM, the functional performance model of a GPU can be built only for the range of problem sizes that fit the device memory. In order to extend the model, we implemented the out-of-core version of the kernel.

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 Figure 3, the performance doubles when 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 Figure 4(a)). This implementation requires multiple transfers of the rectangles of the submatrix $C_i$ to and from the device memory, which explains the performance drop in the range of large problem sizes. In order to make the kernel more realistic, we keep the last two rectangles accumulating intermediate results in the device in each iteration and reverse the updating order every other iteration, which can save two transfers in each direction between the host and device memory every iteration. Also both two dimensions of these rectangles are ensured to be multiples of 32, taking into account the impact of memory alignment issues of CUDA on the Level 3 BLAS implementation of CUBLAS (especially the GEMM kernel)[19].

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 Figure 4(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

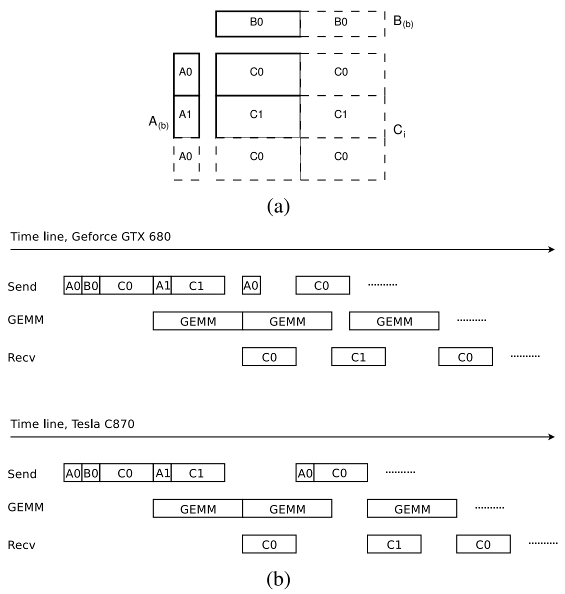


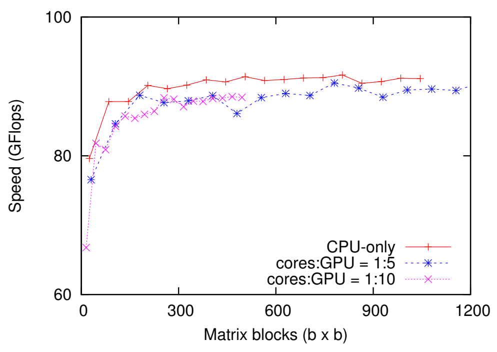
Figure 4. Out-of-core implementation of the kernel on GPU (a). Concurrent data transfers and kernel executions on GPUs (b)

$C_i$. Overlapping communications and computations in the out-of-core version of the kernel is illustrated in Figure 4(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 Figure 4(b)).

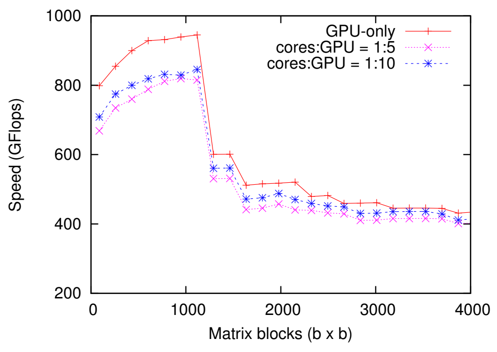
We can see from Figure 3 that the performance of Geforce GTX680 improves by around 30% when using overlapping. Based on our experiments, the speed function shapes of Tesla C870 are similar to Geforce GTX680's. However, there is less performance improvement from overlapping because Tesla C870 does not support concurrent data transfers.

In the application with the out-of-core kernel, the total volume of communications between the host and device memory is determined by the blocking factor $b$. In each iteration of parallel matrix multiplication, the submatrix $C_i$ to be updated by GPU will be transferred between the host and device once in each direction, by transferring its sub-rectangles serially. The total number of transfers of $C_i$ doubles the number of iterations of the application. The total volume of communications can be decreased by increasing the blocking factor so as to decrease the number of iterations, which can improve the performance since data transfer time occupies a large part of the whole GPU execution time. Meanwhile, with a larger $b$, all processing elements perform better, benefiting from the optimised GEMM kernels, and the communication operations (such as broadcast) between processing elements decrease. However, too large blocking factor result in coarse-grained partitioning of matrices, and therefore, may reduce the level of parallelism and leave less opportunity to balance the load. The blocking factor $b$ should be tuned to achieve a better performance depending on different platforms and parallel routines, which is out of the scope of this paper.

In conclusion, we analyse the impact of resource contention between CPUs and GPUs on the shape of their speed functions. It can be captured when the CPU and GPU kernels are executed simultaneously within a socket, with more workload being allocated to GPU. Figure 5(a) shows the speed of 5 cores that execute the ACML kernel with 1/11 and 1/6 of the total workload, shared with GeForce GTX680. Figure 5(b) shows the combined performance of GeForce GTX680 and its dedicated core, when they execute the CUBLAS kernel and receive 10/11 and 5/6 of the total workload. The amounts of work are proportional to the speeds measured exclusively for the cores and the GPU. The first distribution (1:10) corresponds to the problem sizes that fit in the GPU memory, while the second distribution (1:5) corresponds to the large problem sizes. The CPU cores demonstrate almost the same performance as when the GPU is idle. Therefore, the speed function built exclusively for 5 cores provides a good approximation of their performance even in the case of resource contention with GPU. At the same time, the GPU is affected by resource contention. Therefore, the function built exclusively for the GPU, when CPU cores are idle, can approximate the speed of the GPU in the case of resource contention with 85% accuracy.



(a)



(b)

Figure 5. Impact of resource contention on the speed functions (single precision, $b = 640$) of multiple cores (a) and GPU (b) on the same socket

## VI. FPM-based Data Partitioning on Hybrid Platforms

In this section, we present the experimental results demonstrating that the heterogeneous parallel matrix multiplication application can be balanced on hybrid platforms by data partitioning algorithm [5] based on functional performance models of multiple cores and GPUs.

Table II shows the execution time of the heterogeneous matrix multiplication application described in Section IV measured on different configurations of the hybrid platform described in Section III. The experiments were performed for square matrices with blocking factor $b = 640$. The

Table II
EXECUTION TIME OF PARALLEL MATRIX MULTIPLICATION

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

Table III
HETEROGENEOUS DATA PARTITIONING ON THE HYBRID NODE

| Matrix | CPM-based (blocks) | | | | FPM-based (blocks) | | | |
|---|---|---|---|---|---|---|---|---|
| $n \times n$ | $G1$ | $G2$ | $S5$ | $S6$ | $G1$ | $G2$ | $S5$ | $S6$ |
| $40 \times 40$ | 928 | 226 | 105 | 120 | 1000 | 210 | 95 | 102 |
| $50 \times 50$ | 1460 | 352 | 160 | 186 | 1250 | 429 | 190 | 222 |
| $60 \times 60$ | 2085 | 501 | 235 | 270 | 1627 | 657 | 295 | 342 |
| $70 \times 70$ | 2848 | 677 | 320 | 366 | 2250 | 806 | 425 | 504 |

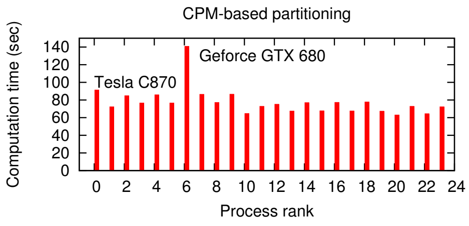
first column shows the matrix size $n \times n$ in square blocks of $640 \times 640$. Column 2 shows the application execution time for the homogeneous matrix distribution between 24 CPU cores. Column 3 shows the execution time on GeForce GTX680 and a dedicated core. The last column shows the execution 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.

To demonstrate the accuracy of the FPM-based data partitioning on hybrid platforms, we compare it with the traditional data partitioning. Traditional data partitioning algorithms distribute a workload between processors in proportion to the constants that define the performance of these processors. The constants (constant performance models) are obtained in advance, from the speed measurements when some workload is distributed evenly between the processors. In Table III, we present the results of the CPM- and FPM-based partitioning algorithms on the hybrid node for different problem sizes. Column 1 shows the matrix size in square blocks; columns $G1$ and $G2$ present the numbers of matrix blocks assigned to GeForce GTX680 and Tesla C870 respectively; columns $S5$ present the numbers of blocks assigned to the sockets with one core dedicated to a GPU; and columns $S6$ present the numbers of blocks distributed between the rest of sockets.

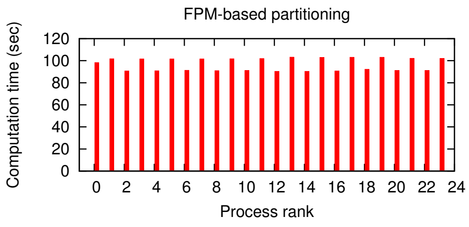
According to the speed functions (see Figure 2 and 3), GeForce GTX680 ($G1$) is around 9 times faster than a socket ($S6$) when the problem fits in its local memory ($40 \times 40$), and around $6 \sim 4$ times faster when the problem exceeds its memory (from $50 \times 50$ to $70 \times 70$). Table III shows that the CPM-based fata partitioning results in overloading GeForce GTX680, starting from matrix size $50 \times 50$. Namely, the ratio of the number of blocks partitioned to $G1$ and $S6$ is nearly 8 when problem size is $70 \times 70$. This is because the CPM-based algorithm is based on inaccurate performance models, using the speed of the GPU kernel when it fitted in the GPU memory. Since the functional performance model accurately captures the performance change under a wide range of problem sizes, the FPM-based partitioning algorithm always balances the load.

Figure 6 illustrates the computation time (communication time between processes excluded) of each process when matrix size is $60 \times 60$ and the workload is distributed by using CPM- and FPM-based partitioning algorithms. In both experiments, process 0 and 6 were bound to cores dedicated to Tesla C870 and GeForce GTX680 respectively. As shown in Figure 6(a), GeForce GTX680 took a longer time than other processes to finish its job because it was overloaded. The CPM-based data partitioning failed to balance the load. The FPM-based data partitioning achieved load balancing and reduced the total computation time by 40%.

Figure 7 shows the execution time (including communication time between processes) of the parallel matrix multiplication application when the workload is distributed by using different data partitioning algorithms. The execution of the application based on homogeneous partitioning



(a)



(b)

Figure 6. The computation time of each process when matrix size is $60 \times 60$ and the workload is distributed by (a) CPM-based data partitioning algorithm (b) FPM-based data partitioning algorithm.

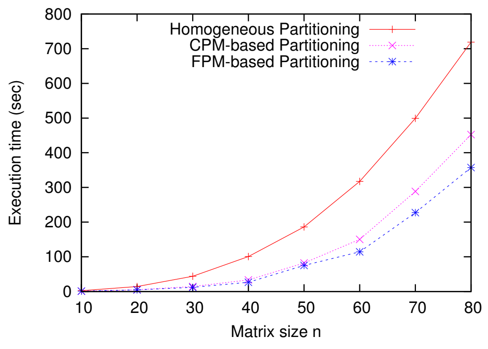


Figure 7. Execution time of the parallel matrix multiplication application with different data partitioning algorithms

(data distributed evenly) was unbalanced, being dominated by the slowest processing elements (CPU cores). Both the CPM-based and FPM-based data partitioning balanced the load when problem sizes were relatively small. However, starting from matrix size $50 \times 50$, the CPM-based algorithm failed to balance the load and the application took longer time to finish than the application based on the FPM-based algorithm. The FPM-based data partitioning algorithm reduced the execution time of the application over the CPM-based and homogeneous partitioning algorithms by 30% and 45% respectively, in the range of large problem sizes.

## VII. Conclusion

In this paper, we presented the performance measurement techniques on hybrid platforms. We defined and built functional performance models of heterogeneous processing elements for a fundamental data-parallel scientific application on a typical multicore and multi-GPU node, considering a hybrid node as a distributed-memory system. We demonstrated that FPMs can facilitate performance evaluation of scientific applications on hybrid platforms, and data partitioning algorithms based on accurate FPMs can deliver better performance than traditional partitioning approaches.

**Acknowledgements.** This research was partialy conducted with the financial support of Science Foundation Ireland 08/IN.1/I2054 and UCD-CSC scholarship.

## References

[1] M. Fatica, "Accelerating Linpack with CUDA on heterogenous clusters," in *GPGPU-2*. ACM, 2009, pp. 46–51.

[2] C. Yang, F. Wang, Y. Du *et al.*, "Adaptive optimization for petascale heterogeneous CPU/GPU computing," in *Cluster'10*, 2010, pp. 19–28.

[3] C.-K. Luk, S. Hong, and H. Kim, "Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping," in *MICRO-42*, 2009, pp. 45–55.

[4] Y. Ogata, T. Endo, N. Maruyama, and S. Matsuoka, "An efficient, model-based CPU-GPU heterogeneous FFT library," in *IPDPS 2008*, 2008, pp. 1 –10.

[5] 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.

[6] Z. Zhong, V. Rychkov, and A. Lastovetsky, "Data partitioning on heterogeneous multicore platforms," in *Cluster 2011*, 2011, pp. 580–584.

[7] 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.

[8] 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.

[9] C. Augonnet, S. Thibault, R. Namyst *et al.*, "StarPU: A unified platform for task scheduling on heterogeneous multicore architectures," in *EuroPar'09*, 2009, pp. 863–874.

[10] F. D. Igual, E. Chan, E. S. Quintana-Orti *et al.*, "The FLAME approach: From dense linear algebra algorithms to high-performance multi-accelerator implementations," *J Parallel Distr Com*, 2011.

[11] C. Augonnet, S. Thibault, and R. Namyst, "Automatic calibration of performance models on heterogeneous multicore architectures," in *EuroPar'09*, 2009, pp. 56–65.

[12] 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.

[13] F. Song, S. Tomov, and J. Dongarra, "Enabling and scaling matrix computations on heterogeneous multi-core and multi-GPU systems," in *ICS 2012*, 2012.

[14] D. Clarke, A. Lastovetsky, and V. Rychkov, "Dynamic load balancing of parallel computational iterative routines on highly heterogeneous HPC platforms," *Parallel Processing Letters*, vol. 21, pp. 195–217, 2011.

[15] A. D. Malony, S. Biersdorff, S. Shende *et al.*, "Parallel performance measurement of heterogeneous parallel systems with GPUs," in *ICPP '11*, 2011, pp. 176–185.

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

[17] D. Clarke, A. Lastovetsky, and V. Rychkov, "Column-based matrix partitioning for parallel matrix multiplication on heterogeneous processors based on functional performance models," in *HeteroPar'2011*, 2012, pp. 450–459.

[18] J. Choi, "A new parallel matrix multiplication algorithm on distributed-memory concurrent computers," in *HPC Asia '97*, 1997, pp. 224 –229.

[19] S. Barrachina, M. Castillo, F. Igual *et al.*, "Evaluation and tuning of the Level 3 CUBLAS for graphics processors," in *IPDPS 2008*, 2008, pp. 1–8.