International Journal of Networking and Computing – www.ijnc.org
ISSN 2185-2839 (print) ISSN 2185-2847 (online)
Volume 1, Number 1, pages 96–113, January 2011

CHPS: An Environment for Collaborative Execution on Heterogeneous Desktop Systems

ALEKSANDAR ILIĆ
INESC-ID, IST/TU Lisbon, Rua Alves Redol 9
Lisbon, 1000-029, Portugal

LEONEL SOUSA
INESC-ID, IST/TU Lisbon, Rua Alves Redol 9
Lisbon, 1000-029, Portugal

Received: July 11, 2010
Revised: October 30, 2010
Accepted: December 13, 2010
Communicated by Akihiro Fujiwara

Abstract

Modern commodity desktop computers equipped with multi-core Central Processing Units (CPUs) and specialized but programmable co-processors are capable of providing a remarkable computational performance. However, approaching this performance is not a trivial task as it requires the coordination of architecturally different devices for cooperative execution. Coordinating the use of the full set of processing units demands careful coalescing of diverse programing models and addressing the challenges imposed by the overall system complexity.

In order to exploit the computational power of a heterogeneous desktop system, such as a platform consisting of a multi-core CPU and a Graphics Processing Unit (GPU), we propose herein a collaborative execution environment that allows to cooperatively execute a single application by exploiting both task and data parallelism. In particular, the proposed environment is able to use the different native programming models according to the device type, e.g., the application processing interfaces such as OpenMP for the CPU and Compute Unified Device Architecture (CUDA) for the GPU devices. The data and task level parallelism is exploited for both types of processors by relying on the task description scheme defined by the proposed environment.

The relevance of the proposed approach is demonstrated in a heterogeneous system with a quad-core CPU and a GPU for linear algebra and digital signal processing applications. We obtain significant performance gains in comparison to both single core and multi-core executions when computing matrix multiplication and Fast Fourier Transform (FFT).

Keywords: Heterogeneous desktop systems, Unified execution environment, Graphics Processing Units

1 Introduction

Parallel computing systems are becoming increasingly heterogeneous and hierarchical. At the level of a single device, a widely accepted trend for an increase of device performance is to assemble more processing cores onto a single chip and/or to improve the architecture, e.g. multi-core CPUs. On the

other hand, relatively low-cost and efficient accelerators, such as GPUs [15], Cell/BE processors [12] and Field Programmable Gate Arrays (FPGAs) have already shown the ability to provide remarkable results in the high performance computing domain. The common ground for all those devices lies in their widespread availability and programmability, thus making them perfectly suitable to be a part of a modern desktop computer. As a result, current multi-core designs equipped with specialized but programmable processing accelerators cause to perceive practically each desktop computer as an individual heterogeneous system.

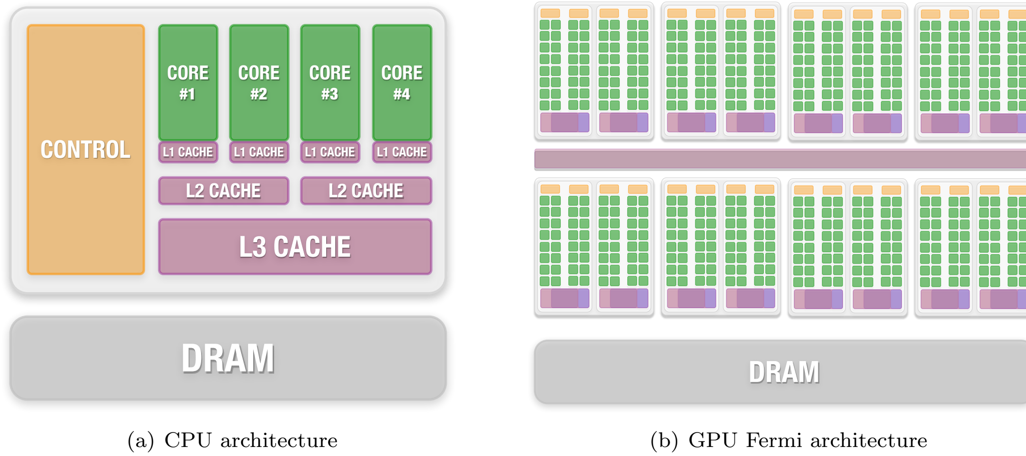
However, approaching the potential collaborative peak performance of these systems is not a trivial task due to the complexity of the overall system's architecture. In particular, the overall system is composed of devices with different architectures, characteristics and processing potentials, tightly coupled to build a single, but heterogeneous execution environment. This means that not only hard-to-solve programming challenges common for every parallel system have to be dealt with, but also several vendor-specific programming models and/or high performance libraries must usually be employed at the same time. Moreover, the application optimization requires load balancing, and usually relies on per-device performance modeling to provide adequate task distribution. The above-mentioned problems are probably the major reasons why the scientific community is still predominantly aimed on using a single device at the time and exploring their capabilities for domain-specific computations, thus leaving the collaborative potential practically unexplored.

In this paper we tackle the problems of collaborative execution across a set of heterogenous devices in a modern desktop system. In detail, we propose herein the Collaborative Execution Environment for Heterogeneous Parallel Systems (CHPS) that allows to cooperatively execute a single application by employing architecturally different processing devices, namely multi-core CPUs and GPUs. At the device level, the proposed environment is capable of integrating the native programming models on a per device type basis to attain the most efficient execution, such as OpenMP for multi-core CPUs and CUDA for GPU devices. At the collaborative execution level, the CHPS relies on a task description scheme that accommodates different execution principles for different task types, and allows to exploit both data and task parallelism. Furthermore, the proposed environment integrates the specific scheduling module which assigns different workload distributions to the devices thus allowing to achieve the optimal load-balancing via exhaustive search. The proposed approach is tested in a heterogeneous system comprising a quad core CPU and a GPU for executing two of the most common applications for scientific and digital signal processing, namely dense matrix multiplication and complex FFT. We show that our approach is capable of achieving significant performance gains compared not only to a single core, but also to multi-core execution.

2 Heterogeneous Desktop Systems

Contemporary multi-core CPUs, as the integral parts of every desktop system, incorporate a collection of identical processors sharing the same primary memory. Currently, AMD and Intel CPU designs are built around the small number of individual superscalar cores that may share a single coherent on-device cache or may have completely separate caches, as depicted in Fig. 1(a). The common practice in programming those devices anticipates the use of PThreads [6] and OpenMP [20], which can be considered as de-facto standard for symmetric multi-core CPU architectures. The wide acceptance of OpenMP lies in its simplicity to define and manipulate parallel tasks using a minimal set of directives. Tasks can be specified explicitly or created automatically on reaching the worksharing constructs, where the work inside the construct is divided among the threads and executed cooperatively. However, OpenMP programmers still need to check for data dependencies, data conflicts, race conditions, or deadlocks.

The large majority of scientific research efforts in the area of heterogeneous computing is targeting the distributed memory systems, such as clusters of computers, consisting of several interconnected architectures equipped with only CPUs. The heterogeneous aspect of this type of distributed systems is only evidenced by the presence of multi-cores with diverse computational capabilities, vendor-specific design strategies or even architectural principles, such as different cache hierarchies. Regardless of the level of diversity, all employed CPUs are sharing the same programming paradigms,



(a) CPU architecture

(b) GPU Fermi architecture

Figure 1: Architectural diversity of devices in an heterogeneous desktop system.

thus the collaborative execution is practically inherently supported in CPU-only heterogenous systems.

On the other hand, modern desktop computers even support a higher level of heterogeneity, because a typical desktop system usually embraces architecturally different devices as its integral parts, such as GPUs next to the multi-core CPUs. In contrast to the overall CPU architecture, GPUs are massively parallel multi-threaded devices, which use hundreds of simple cores to provide huge processing power (see Fig. 1(b). For example, NVIDIA GPUs with Fermi architecture [16] has 512 streaming processing cores integrated on a chip. Performing general-purpose computations on a GPU require the use of specific programming models such as NVIDIA CUDA [17] or ATI Stream Technology [4]. CUDA offers an easy-to-program C-based framework, where efficient implementations demand from a programmer to have an insight on the GPU-specific features, such as the streaming processor array organization and the memory hierarchy composed of local, shared, and global memories. Each kernel run in the GPU embraces the concurrent execution of thousands of extremely lightweight threads in order to achieve efficiency. This very fine-grained data-level parallelism favors throughput over latency, requiring an application with high degree of data parallelism.

At the global system level, the parallel execution paradigm on desktop computers relies on a heterogeneous serial-parallel model, where the CPU (host) sees the underlying devices as many-core co-processors. The host is responsible for initialization and program control, whereas the parallel portions of the code (kernels) are physically off-loaded to the devices. This requires different roles to be assigned to the available computational resources depending on their position and functionality, as presented in Fig. 2. In general, the CPU is the only resource allowed to transparently access the whole range of global memory, whereas the underlying devices perform the computations in their own local memories using different instruction sets. This naturally designates the CPU as the global execution controller as it has to retain the control over the complete collaborative execution on all employed functional units.

Communication between the host and devices is performed explicitly via bidirectional interconnection channels, i.e., all input and output data must be transferred from host to device memory and vice versa. In practice, the interconnection buses between host and device have limited bandwidth, thus introducing an additional overhead to the overall execution time, and even becoming a performance bottleneck for applications with low computation to communication ratio. Moreover, the communication path between devices in off-the-shelf heterogeneous systems is restricted by the overall architectural principle requiring the host processor as an unavoidable intermediate in the communication path. This implies that underlying computational units are usually not free to explicitly communicate between each other (even if they are of the same type). This host-bound communication model has several major implications to the overall system usability. First of all, device synchronization problem is almost completely supported by the host. Moreover, only a small sub-

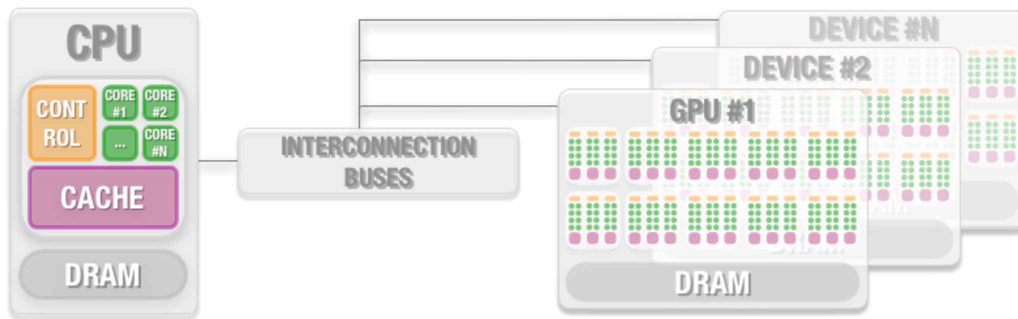


Figure 2: Architecture of the heterogeneous systems.

set of already developed parallelization methods for distributed environments can be implemented, although the address spaces are distributed.

Due to significant architectural differences, achieving peak performance on heterogeneous desktop systems usually involves careful coalescing of diverse, per-device programming models and/or vendor-provided high performance libraries. Combining those programming models to build a single application requires the knowledge of specific device and system architectural details and forces the entirely dissimilar programming and optimization techniques to be reconciled. Achieving an efficient execution across multiple devices also demands complex and careful computation partitioning prior to the actual execution. Partitioning must take into account several parameters for sustaining the overall system performance, such as different processing capabilities, scarce memory capacities of the computational resources, and limited bandwidth of interconnection channels. Nevertheless, certain devices are often specialized for certain set of applications, thus partitioning should be conducted in order to address only the devices on which the performance benefits are expected.

One can notice that significant differences at both architectural and programming model levels exist in modern heterogeneous desktop systems, even when considering only CPU and GPU devices. Moreover, those differences can even be wider taking into account the ability to connect other architecturally divergent devices, such as Cell/BEs [12], FPGA boards etc. As a result, the overall system's heterogeneity elevates the complexity of collaborative execution challenges to the much higher levels comparing to the CPU-only heterogeneous environments.

3 Environment for Collaborative Execution on Heterogeneous Desktop Systems

With the aim of employing the available architecturally diverse set of devices to cooperatively execute a single application, we present herein the CHPS collaborative execution environment. The proposed execution environment provides the means for synergetic cross-device execution in a desktop platform consisting of at least a multi-core CPU and a GPU, by exploiting both data and task parallelism.

As previously referred, the collaborative execution across a set of heterogeneous devices requires to address the platform programmability issues at the very beginning. It is worth to emphasize that our major goal is not to ease the programming individually, at the level of the device, but to explore the ways of describing the parallelism and to provide the means for collaborative execution *between* the heterogeneous devices in a desktop system. This means that the approach presented in this section is aimed on addressing the overall platform's programmability from the perspective of accommodating the collaborative cross-device execution at the system level, thus tackles the problems which are beyond the problems of the low-level on-device programming. Nevertheless, the synergetic execution across a set of diverse devices still requires to coalesce substantially different and vendor-specific programming models on a per device type basis.

In this paper, we advocate redefining the basic units of programming by introducing the task abstraction. The tasks in the proposed system do not only carry the application implementation,

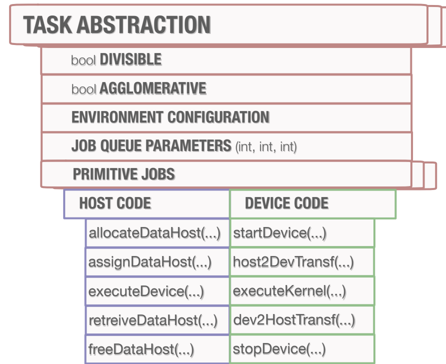


Figure 3: Abstract structure of the task in the unified execution model.

but they also provide the additional information substantial for their further execution, such as the type of devices required or the environment configuration for efficient execution. The task abstractions are capable of encapsulating the implementations on a per device type basis, which can be expressed using the natively supported programming models and constructs. In detail, we encourage the use of vendor-specific models as they are usually the fruitful ground to completely explore the computational capabilities of the devices and to employ the optimization techniques directly provided by the device manufacturers. In contrast to other approaches that propose the construction of the new, common languages across the devices, we are rather focusing on building the *sinergizer* for several reasons. First of all, when defining the common language for architecturally different devices, only a subset of available, vendor-specific techniques is usually considered, thus accommodating mainly the basic use of a certain device, while not being concerned with achieving the peak performance. Moreover, the language constructs must also be constantly updated to sustain the inclusion of emergent specialized accelerators, or to incorporate the newly available features arising from the architectural changes in an already supported device type, e.g., features provided by Fermi [16] regarding to the GT200 [15] GPU architecture.

Figure 3 depicts the structure of a coarse-grained task, redefined to include a set of parameters sustaining the collaborative heterogeneous execution. At first, the tasks are extended to encapsulate the information required to configure the execution environment regarding the available system devices. Practically, the programmer is allowed to specify the environmental constraints of the task implementation, e.g., when the task execution requires more than one device at the time or when it supports only certain device types. In the proposed environment, different types of tasks can be specified, which allows the collaborative environment to configure the execution in the most suitable form. The tasks in the CHPS environment can be defined as *Divisible* and/or *Agglomerative*, and the CHPS environment sustains different execution modes depending on the combination of those two parameters. A *Divisible* task incorporates a set of fine-grained program portions called *Primitive Jobs*. Each *Primitive Job* may consist of a set of functions, marked as Host and Device Code, as shown in Figure 3. The Device Code refers to a set of functions to drive direct low level on-device execution, whereas the Host Code embraces the necessary operations executed on the host processor prior/after the actual device kernel execution. *Primitive Jobs* are practically the only computation carriers in the system, and their concurrent execution on several devices require separate implementations on a per device type basis. Therefore, by designating the task as *Divisible*, the collaborative environment is instructed to guarantee the availability of the structures to sustain the cooperative execution of *Primitive Jobs*. Those execution structures, such as *Job Queues* in Figure 4, allow to organize and plan the execution of the pool of *Primitive Jobs*. Depending on the implementation, the *Primitive Job* can be perceived as an instance of the provided application implementation that corresponds to a single entry in the sustaining execution structure, or the separate implementations may be provided for different structure entries. Furthermore, the task might also be *Agglomerative*, if several *Primitive Jobs* can be grouped into one coarse-grained job,

thus allowing groups of *Primitive Jobs* to be scheduled for execution on a device.

In detail, the following combinations of *Divisible* and *Agglomerative* parameters are permitted when specifying the task type in the CHPS environment:

- the task designated as neither *Divisible* nor *Agglomerative* represents a single coarse-grained task consisting of exactly one *Primitive Job* and is forwarded to the direct on-device execution in its entirety according to the specified configuration parameters;
- the task which is *Divisible*, but not *Agglomerative*, consists of a set of *Primitive Jobs*, whose execution requires the use of specialized execution structures, such as a pool of *Primitive Jobs*; the execution structures are used to drive the data-parallel execution of the *Primitive Jobs* across as set of heterogeneous devices, where exactly one pool entry corresponds to a single *Primitive Job* to be executed on the available device regarding to the configuration parameters;
- *Divisible* and *Agglomerative* tasks are sharing the same structure and similar execution principle as *Divisible*-only tasks, except that for this task type the several *Primitive Jobs* from the specialized execution structure can be grouped together, thus assembling a single coarse-grained job to be executed on the requested device.

It is worth to note that the combination of parameters where the task is *Agglomerative*, but not *Divisible*, is not defined by the current implementation, as it does not make any practical sense. The rationale behind introducing these task descriptions, using *Divisible* and *Agglomerative* parameters, lies in a fact that all three enumerated combinations demand different execution patterns. The non-*Divisible* tasks can be immediately forwarded to the on-device execution and do not require any specialized execution structure. However, the *Divisible* and not *Agglomerative* tasks are relying on the pool of *Primitive Jobs*, where each *Primitive Job* is executed separately. This means that the employed devices request a single *Primitive Job* at the time, as soon as they have finished processing the previously assigned *Primitive Job*. As a result, the complete execution requires many single *Primitive Job* executions to be performed by each employed device, thus the clear benefit of using this approach lies in its adoption of an inherent dynamic load balancing scheme. On the other hand, *Divisible* and *Agglomerative* tasks are aimed at collaborative execution of a complete set of *Primitive Jobs* at once. This means that each device is assigned with a single coarse-grain job assembled from the requested number of *Primitive Jobs*, defined *a priori*. The benefits of using this approach lie in the fact that there are no scheduling overheads imposed for identifying the end of the execution, and for assigning the next *Primitive Job*. Moreover, in practice, the efficient utilization of both interconnection channels and device's computation resources highly depend on the amount of the work assigned to the execution. Hence, by agglomerating the *Primitive Jobs* into one coarse-grained job, more efficient execution is expected using this execution set-up, in comparison with the execution of many fine-grained *Primitive Jobs*. However, the major drawback when using this execution mode lies in the fact that the workload distribution between the devices can not be know *a priori*, thus requires careful performance modeling of the devices and interconnection channels to decide on the number of *Primitive Jobs* to accommodate as load balanced execution as possible.

Focusing on the execution environment, Figure 4 presents the high-level structural model of the CHPS collaborative environment consisting of four main functional modules, namely: the **Task Scheduler**, the **Job Queue**, the **Job Dispatcher**, and the **Device Query**.

The execution begins with the group of **tasks** entering the system. At this point it is worth to emphasize the ability to express the task dependencies using the above-mentioned task description scheme in the proposed environment. It respects the order in which the tasks have to be performed in order to produce the correct overall result. Namely, the current implementation is based on the list dependency scheme where the tasks are organized in a list and the dependencies are specified by enumerating the preceding tasks on which the current one depends on. However, the independent tasks, or the tasks from different applications, are free to be enqueued without any preceding constraint thus accommodating their independent parallel execution.

The **Task Scheduler** is responsible for examining the list of the tasks in order to detect the independent task(s) which can be executed next in the system according to several criterions. First of all, the specified conditions from the task configuration parameters must be satisfied, including

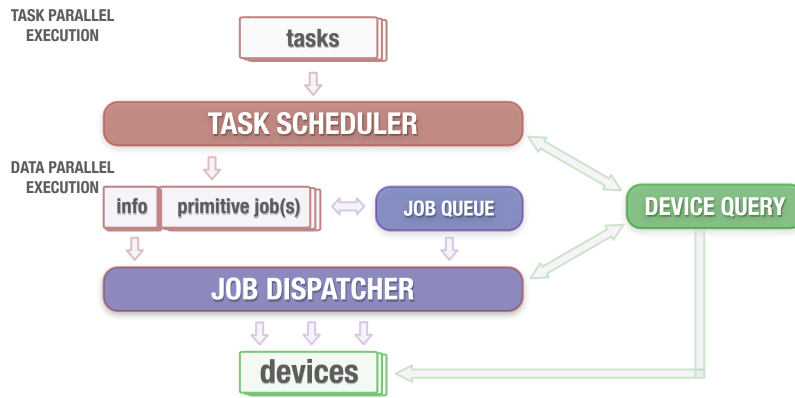


Figure 4: Structure of the unified execution model.

the availability of the requested devices. Secondly, the execution of all previous tasks on which the current task is dependent must be finished, i.e., the task dependencies are satisfied.

As previously referred, depending on the specified type of the selected task, the actual execution is performed using different execution principles.

- For **not** ***Divisible*** tasks:

  The selected not *Divisible* task is simply forwarded to the **Job Dispatcher** which assigns a requested device to the task and initiates on device execution by launching the kernel calls. Moreover, the Job Dispatcher is also responsible to track the status and control the overall execution of all started kernels. Practically, not *Divisible* tasks require the execution of a coarse-grain *Primitive Job* represented by a single entry **Job Queue**.

  At the same time, the *Task Scheduler* continues to examine the list of tasks, and in case that all the prerequisites are satisfied, it is free to forward to the *Job Dispatcher* all independent tasks. In this case, a large number of independent tasks can be running in the system simultaneously, thus accommodating the ***task level parallelism***.

- For ***Divisible*** tasks:

  If the selected task is *Divisible*, its *Primitive Jobs* are arranged into the **Job Queue** structure according to the parameters specified in the task properties. The current implementation permits the grid organization of the *Primitive Jobs* inside the *Job Queue* with grid size spanning to up to three dimensions. However, the functionality of the **Job Dispatcher** differs according to the type of the selected task, i.e., if the *Divisible* task is *Agglomerative* or not.

  – For ***Divisible***, but **not** ***Agglomerative*** tasks:
    The *Job Dispatcher* sends for execution, to each requested and available device, a single *Primitive Job* from the *Job Queue*. Upon detecting the completion of the processing on any of the employed devices, the *Job Dispatcher* examines the *Job Queue* to find the next not processed *Primitive Job* and forwards it to be executed on the device. This single *Primitive Job* assigning scheme is continuing until all *Primitive Jobs* inside the *Job Queue* are processed.

  – For ***Divisible*** **and** ***Agglomerative*** tasks:
    The *Job Dispatcher* is responsible to agglomerate the *Primitive Jobs* from the *Job Queue* into the coarse-grained workloads to be dispatched for execution on each requested device. The agglomeration process is instructed by the workload sizes specified in the task configuration parameters on a per device basis. Generally, this execution scheme is aimed on processing a complete set of *Primitive Jobs* at once and on all specified devices. However, in case that there are still not processed *Primitive Jobs* in the *Job Queue* after

the first run, the *Job Dispatcher* repeats the agglomeration procedure on the rest of the unexamined *Primitive Jobs* and forwards the coarse-grained tasks to each of the devices once again, until the all *Primitive Jobs* are processed.

As one can notice, in the proposed environment, the different portions of a single task can also be executed on several devices simultaneously, thus accommodating ***data level parallelism***. However, it is worth to emphasize that the current implementation also allows the *Primitive Jobs* to include the parallel sections inside the Device Code, thus allowing a single *Primitive Job* to be executed using *nested parallel* paradigm.

Regardless of the taken execution path, the **Device Query** module interacts with all other modules in the system and provides a mechanism to examine and identify all underlying **devices** which are currently available in the system. It also maintains a specific data structure to hold the relevant information related to each device, such as resource type, status, device memory management and performance history.

Furthermore, as the developed platform is capable of storing the task performance history for each device, it can be re-used to extend the *Job Dispatcher* to perform the agglomerations of the *Primitive Jobs* to maximize the performance benefits of the collaborative execution. The platform can also be configured to run in an exhaustive search mode, where the *Parallel Jobs* are executed on the requested devices with varying workloads, in order to obtain the full performance models of employed devices. The obtained performance models can be used to find the best mapping and decide on optimal load balancing scheme in the systems where multiple runs of the same kernels are requested. Finally, for devices that support concurrency between the memory transfers and kernel execution, such as GPU devices with streaming capabilities, the *Job Dispatcher* module is extended to support the overlapping of the communication with the actual on-device computation by carefully scheduling the *Primitive Jobs* for the *Divisible* tasks using the streaming technique.

4 Programming the CHPS Environment for Matrix Multiplication and 3D FFT

In order to demonstrate the practical usability of the presented approach, we have implemented two of the most commonly used applications in linear algebra and digital signal processing, namely dense matrix multiplication and 3D complex fast Fourier transform. Both applications encompass data parallelism which makes them perfectly suited for providing a detailed insight on the attainable performance in heterogeneous commodity desktop systems. Applications are implemented following the fundamental principles of the CHPS environment, presented in Section 3.

4.1 Matrix Multiplication

The general method for performing multiplication of two dense matrices, A and B producing matrix C, is based on a block decomposition, where $M \times K$, $K \times N$, and $M \times N$ matrices are divided into sub-matrices of an $P \times R$, $R \times Q$, and $P \times Q$ size, respectively. As shown in Figure 5, each sub-matrix $C_{i,j}$ requires the following computation:

$$C_{i,j} = \sum_{l=0}^{K/R} A_{i,l} \times B_{l,j} \tag{1}$$

where $0 \leq i \leq M/P$ and $0 \leq j \leq N/Q$.

In order to perform the matrix multiplication using this method in the CHPS environment, a three dimensional *Job Queue* has to be created of a size $M/P \times N/Q \times K/R$. Therefore, each (i, j, l) tuple from the *Job Queue* will direct the execution of the corresponding *Primitive Job* in order to produce the requested $C_{i,j}$ sub-matrix. As it can be noticed, direct implementation of the general multiplication in current heterogeneous desktop systems embraces huge drawbacks concerning the

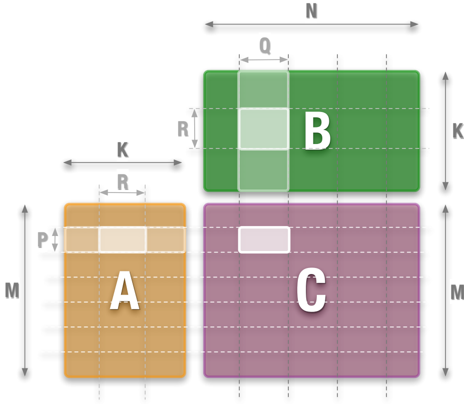


Figure 5: Parallelization of the general matrix multiplication.

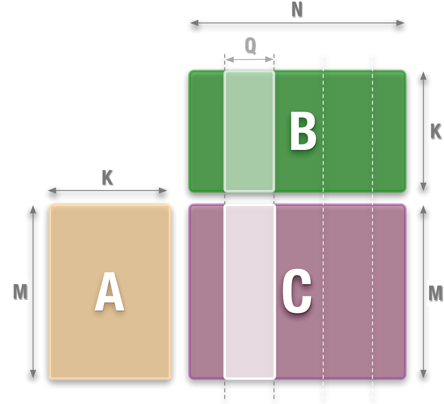


Figure 6: Communication aware matrix multiplication.

number of data transfers needed to be preformed. Nevertheless, the number of data transfers can be reduced by declaring a matrix multiplication task as *Divisible* and *Agglomerative*, thus supplying a single device with more than one sub-matrix multiplication.

However, careful selection of the P, Q, and R ($P = M$, $K = R$) parameters allows derivation of the algorithm which accommodates further memory transfer reduction to only N/Q transfers, thus providing the possibility to attain the system's peak performance (see Figure 6). In order to perform optimized matrix multiplication in the CHPS environment, a single matrix multiplication task is declared as *Divisible* and *Agglomerative*, and the *Primitive Jobs* are grouped into one dimensional *Job Queue* of a size N/Q. Prior to actual kernel execution, each computational device is supplied with the A matrix, which is followed by the *Primitive Jobs'* agglomeration and distribution according to the specified scheme in the task configuration parameters. Nevertheless, this implementation is bound to the memory capacities of the requested devices, which means that the device with the smallest amount of global memory sets the algorithm's upper bound.

To lessen those restrictions, we have also implemented the block matrix multiplication using the Horowitz scheme [9], which we have modified to include the awareness of the memory limitations on a per-device basis. The memory aware Horowitz algorithm is built for the cases where the sizes of A, B and C matrices surpass the memory capacities of certain devices. Then the overall matrix multiplication can be performed by subdividing the problem into several multiplications of the block matrices, with sizes that respect the global memory limitations of each employed device, as depicted in Figure 7. The recursive HorowitzMA Function presents the basic functionality of the algorithm. The matrices A, B and C are recursively subdivided into matrix blocks until reaching the size to perform matrix multiplication on each requested device (*mem_size* represents the minimum global memory size between all employed devices). In particular, partitioning of all three matrices requires to reorganize the overall execution, such that eight block matrix multiplications are performed after the optimal sub-block size is determined, followed by four sub-matrix additions for each level of recursion applied, as shown in HorowitzMA Function. After the partitioning of the matrices is finished and the sub-matrix size is decided, the actual collaborative execution is performed using the previously mentioned optimized matrix multiplication algorithm, designated as `CHPS_mat_mul` in HorowitzMA Function.

The implementation of the memory aware Horowitz algorithm in the CHPS environment starts by forming the task list consisting of matrix multiplication kernels. In case when the level of recursion is equal to one, the implementation requires to enqueue exactly eight multiplication tasks. Each task is defined as *Divisible* and *Agglomerative* with assigned one dimensional *Job Queue*. As a result, all eight matrix multiplications are performed using the full potential of an heterogeneous desktop system, and scheduled by the *Task Scheduler* according to the specified requirements and device availability. It is worth noting that the overall architectural principle of the heterogeneous desktop

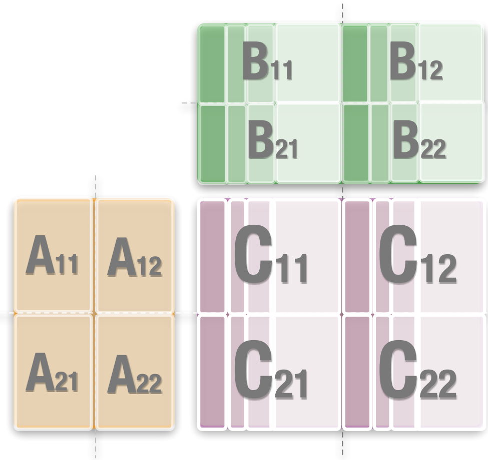


Figure 7: Parallelization of matrix multiplication for large matrices.

Function HorowitzMA(in A, in B, out C)

```
if
(size(A) + size(B) + size(C)) < mem_size
then
    C = CHPS_mat_mul(A, B)
else
    HorowitzMA (A11, B11, P0)
    HorowitzMA (A12, B21, P1)
    HorowitzMA (A11, B12, P2)
    HorowitzMA (A12, B22, P3)
    HorowitzMA (A21, B11, P4)
    HorowitzMA (A22, B21, P5)
    HorowitzMA (A21, B12, P6)
    HorowitzMA (A22, B22, P7)
    C11 = P0 + P1
    C12 = P2 + P3
    C21 = P4 + P5
    C22 = P6 + P7
end
```

systems demands to store the produced C matrix only into the host's memory space. Therefore, the final additions are performed only by the host processor.

4.2 3D FFT

Starting from a general definition of a 3D discrete Fourier transform as a complex function $H(n_1, n_2, n_3)$ for a given function of the same type $h(k_1, k_2, k_3)$, both defined over the tree-dimensional grid $0 \leq k_1 \leq N_1 - 1$, $0 \leq k_2 \leq N_2 - 1$, $0 \leq k_3 \leq N_3 - 1$,

$$W_j = e^{-2\pi i \frac{k_j n_j}{N_j}} \tag{2}$$

$$H(n_1, n_2, n_3) = \sum_{k_3=0}^{N_3-1} \sum_{k_2=0}^{N_2-1} \sum_{k_1=0}^{N_1-1} W_1 W_2 W_3 h(k_1, k_2, k_3) \tag{3}$$

one can notice three distinct implementation possibilities in parallel environments, namely:

$$H = FFT_{1D}(FFT_{1D}(FFT_{1D}[h])) \tag{4}$$
$$= FFT_{1D}(FFT_{2D}[h]) \tag{5}$$
$$= FFT_{2D}(FFT_{1D}[h]). \tag{6}$$

Method 4 requires 1D fast Fourier transforms to be applied on each dimension of the original function, whereas methods 5 and 6 demand 2D FFT application on two dimensions accompanied by 1D FFT transform along the remaining dimension of the original function. The major drawback in parallel implementation of 3D FFT is induced by the inevitable transpositions of the input data between FFTs applied on different dimensions. Moreover, after executing the final FFT, an additional transposition is required to restore the original data layout. Figure 8 depicts the complete process of performing 3D FFT when method 5 is employed, as the best suited for parallelization due to its characteristics.

Implementing method 5 for 3D FFT in the CHPS execution environment requires to form a list of three different and dependent tasks to be scheduled for execution in the system. First task is assigned with 2D FFT batch execution and declared as *Divisible* and/or *Agglomerative* with the 1D *Job Queue* of a maximum size N_1. Once the first task has finished its execution, the 3D matrix needs to be

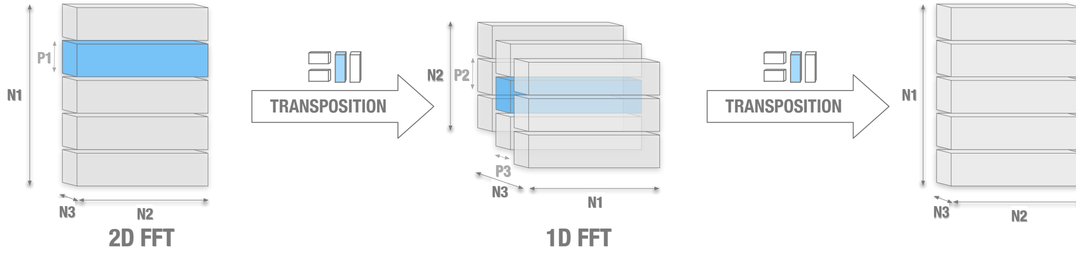


Figure 8: 3D FFT procedure.

transposed to allow the execution of the 1D FFT batches over the third dimension. Depending of the matrix storage scheme, and used transposition method (in-situ/out-of-place, parallel/sequential), second task can be declared with a suitable combination of *Divisible* and *Agglomerative* flags. As soon as transposition is performed, the third task is allowed to proceed to the *Job Dispatcher*. As in the case of 2D FFTs, 1D fast Fourier transform over the third dimension requires the task to be set as *Divisible* and/or *Agglomerative* with the 1D *Job Queue* of a maximum size $N_2 \times N_3$. If needed, a fourth transposition task can be also added in order to bring back the original data layout.

5 Experimental Results

The performance of the proposed CHPS environment for heterogeneous desktop systems is evaluated in the CPU+GPU system consisting of an Intel Core 2 Quad Q9550 processor, 12 MB L2 cache, running at 2.83 GHz, and 4 GB of DDR2 RAM as the CPU, and an NVIDIA GeForce 285 GTX with 1.476 GHz of core frequency and 1 GB of global memory as the GPU. Devices are interconnected via Memory Controller Hub with 1.33 GHz Front Side Bus to the CPU, whereas PCI Express 2.0 16x is used at the GPU side.

Focusing on the collaborative environment implementation, the system is built using the OpenMP [20], and CUDA [17] programing model for NVIDIA GPUs. Vendor-provided high performance libraries are used for assessing the peak performance of computational units, namely the Intel Math Kernel Library (MKL) 10.2 [11] for the CPU, and CUBLAS 3.1 [17] and CUFFT 3.1 [17] libraries for the GPU with CUDA 3.1 beta driver. The enlisted libraries serve as the kernel providers for task's *Primitive Jobs* in their Device Code modules, namely for dense matrix multiplication and complex 2D and 1D batch fast Fourier transformations. Both implementations involve double precision floating point arithmetic and employ offered optimization techniques without altering the library sources, such as: vector intrinsics and thread affinity for MKL kernels, and pinned memory allocation in combination with streaming for GPUs.

All experiments are conducted on Linux Open Suse 11.1, using task, data or nested parallelism if permitted by the implementation. For data parallel executions, the CHPS environment was set to operate in the exhaustive search mode over the number of *Primitive Jobs* in order to achieve load balanced execution. Correspondingly, the obtained results provide a real insight on the high-performance computing potentials of the tested desktop system. The experiments are conducted to exploit collaborative capabilities of the platform by using different execution set-ups, and double precision floating point operations per second (FLOPS) is adapted as a performance metric, to provide a fair comparison between heterogeneous devices' performance.

5.1 Matrix Multiplication

In order to evaluate the performance of dense matrix multiplication in the tested desktop environment, several tests were conducted using double precision floating-point arithmetic on square matrices with varying sizes ($M = K = N$) and using the different execution set-ups. The results obtained are presented in Figures 9 and 10. For the matrix sizes which were not limited by the global memory of the GPU, the optimized implementation from Section 4 was used, whereas for the

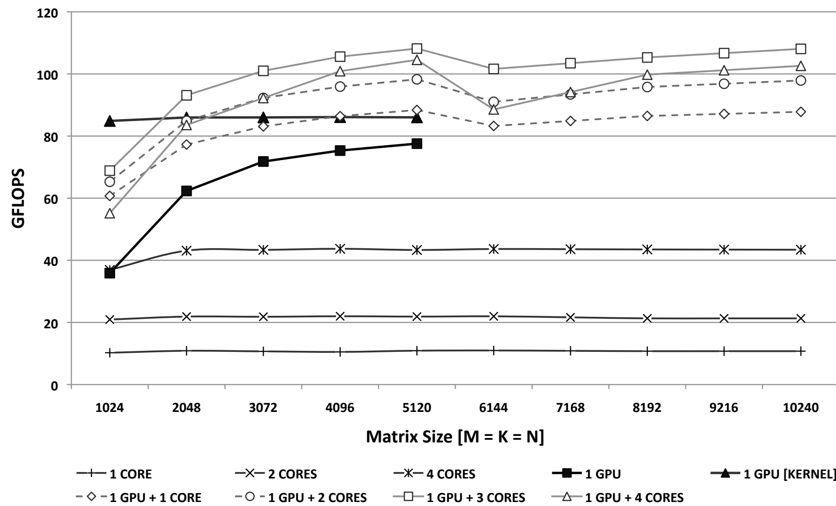


Figure 9: Matrix multiplication performance without nested parallelism.

cases when matrices could not fit into the GPU global memory, the implementation was based on the modified Horowitz scheme. In this case, the largest executable problem on the GPU was for the matrices with less than 5120 elements in each dimension.

For matrices that fit into GPU global memory, the results are obtained using an exhaustive search over the combinations of workload distribution for both devices in order to find the optimal load balancing scheme. The acquired schemes are then re-used to perform the memory-aware Horowitz multiplications, which is evidenced by a slight performance drop for the 6144 test case (eight matrix multiplications were performed using the 3072 matrix kernels, thus retaining their performance). The results presented in Figure 9 are obtained by combining the execution in the GPU with the execution in different numbers of CPU cores, where each core independently executes the assigned problem portions. As it can be seen, the proposed CHPS environment is capable of outperforming the GPU only execution and the multicore execution using the full parallel CPU capacities (four cores) for all tested cases. Moreover, with the proposed collaborative execution model and careful parallelization methods, we were able to retain the high performance in our environment even for the test cases which are generally not executable on the GPU device, thus demonstrating the full platform's synergetic potential. As expected, the best results are obtained using the GPU and three CPU cores, where the fourth core is completely devoted to control the execution on the GPU. The rationale behind this behavior is not related with the CHPS environment, but with the architecture of the employed CPU cores which do not support the Intel Hyper-Threading Technology [10]. In detail, executing the application with the number of threads larger than the number of cores does not usually provide performance benefits on this type of multi-core architecture. This is mainly due to the fact that the unsupported thread-level parallelism at the core level usually implies the execution serialization between the multiple threads in a single core. Correspondingly, the collaborative execution in the tested environment when employing all four CPU cores and the GPU requires to conduct the execution using the five threads on the four cores, and thus does not outperform the collaborative execution with three CPU cores and the GPU for the above-mentioned reasons.

The same set of tests was repeated when employing the MKL's parallel implementations for the CPU execution, as shown in Figure 10. The number of nested threads was varied according to the number of available threads, thus the execution in one thread was expanded to two (1x2) or four (1x4) threads, or when two cores were working in parallel with the remaining two cores (2x2). The obtained results show the possibility to outperform both the GPU only and the CPU only executions even for nested parallelism. However, we do not encourage neither execution which combines GPU and nested parallel execution in the CPU, nor overloading the CPU with the number of threads greater than the number of cores. The practical evaluations show very high performance fluctuations for memory

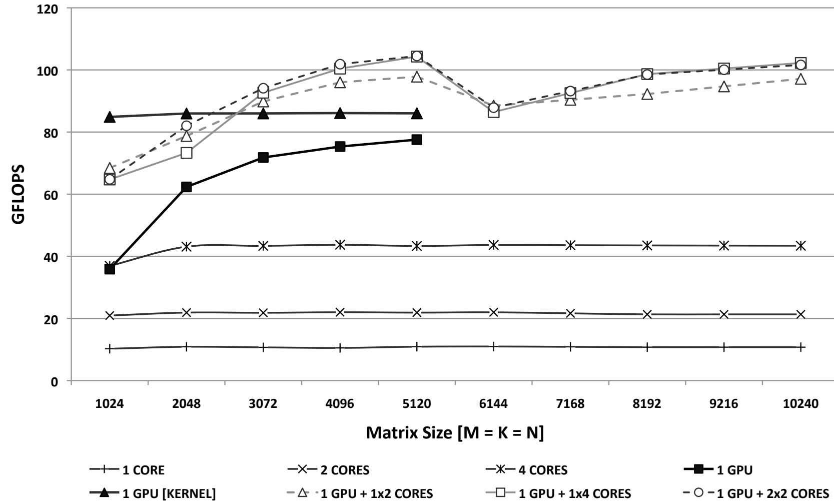


Figure 10: Matrix multiplication performance using nested parallelism.

transfers between the host and the device, when the CPU operates in one of the mentioned modes. This is due to the limited ability of the memory subsystem to serve both Front Side Bus (FSB) and PCI Express requests at the same time, thus making the execution on the device unpredictable. In this particular case, this impact is not significant due to the high computation to communication ratio of the matrix multiplication. This can be evidenced by the narrowing gap between the GPU kernel performance (*1 GPU [Kernel]*), and the GPU performance when the bidirectional transfers are included (*1 GPU*). For all test cases, the GPU kernel performance is predominantly constant, which demonstrates the capability of CUBLAS implementations to efficiently employ the GPU architecture for matrix multiplication. On the other hand, the overall GPU performance (*1 GPU*) increases with the problem size and approaches the performance of the *1 GPU [Kernel]*, which clearly shows that the impact of memory transfers becomes less significant when compared to the overall computational complexity of the matrix multiplication for bigger problem sizes. It is also worth to mention that the reduced communication impact is related to the size of memory transfers capable to efficiently use the bandwidth of the interconnection PCI Express channel. In this particular case, all the transfers above 4KB were capable of providing good performance, which proves that the concept of grouping the memory transfers generally turns into performance benefits. Still, deriving the general performance metric for optimal transfer size is not really possible due to the transfer's high per-system and per-application dependencies.

Furthermore, it is worth to emphasize that the high cost of finding the optimal load balancing using the exhaustive search over the different number of *Primitive Jobs* restricts its application to an extensive range of the matrix problem sizes. However, we have provided an extension of the *Job Dispatcher* module to reuse the information from the performance history in order to make better scheduling decisions of the *Primitive Jobs* for devices that allow concurrency between the memory transfers and kernel execution, such as GPU devices. Namely, the performance history can provide an insight on the exact application communication and execution requirements, which are further used by the *Job Dispatcher* to make decisions on the number of *Primitive Jobs* to be scheduled in each stream, such that the communication is completely overlapped by the computation in the GPU. This approach is tested for the multiplication of matrices with $M = N = K = 4096$, and the results obtained show the average collaborative performance improvement of about 3.9% when compared to the results depicted in Figure 9.

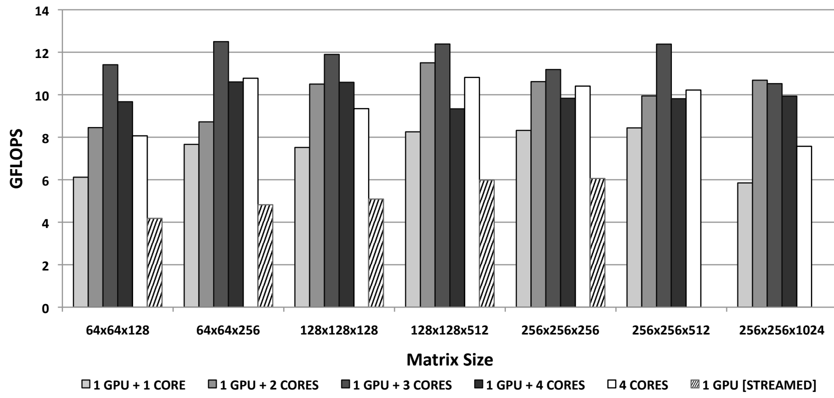


Figure 11: 2D FFT batch results without nested parallelism.

5.2 3D FFT

According to the remarks given in Section 4, we implemented double-precision complex 2D and 1D batch FFTs using the proposed CHPS environment. The tests were mainly conducted to evaluate the benefits of performing 3D FFT in the collaborative environment, through the evaluation of those batch executions, as major steps to calculate the compete 3D transform.

In contrast to matrix multiplication, the FFT operation embraces significantly higher communication to computation ratio. In order to reduce the impact of memory transfers to the overall execution time, several optimization techniques are supported by the current implementation. Firstly, data is allocated in special memory regions, referred as page-locked or pinned memory, to reduce the pre-transfer overheads at the CPU side. Secondly, the CUDA streaming technology is employed, at the GPU side, to sustain the concurrency between the memory transfers and kernel execution, which is preceded by an additional exhaustive search to determine the number of streams capable of achieving the efficient overlap of the communication with computation.

The results obtained for the 2D FFT batches are presented in Figures 11 and 12. The 2D FFT execution without nested parallelism shows that performance gains can be achieved in the collaborative environment regarded to the execution in both four CPU cores and the GPU. However, the results obtained for nested parallel executions in Figure 12 are highly unstable. This is mainly due to the previously referred incapability of memory system to serve both FSB and PCI Express requests, but also due to the volatile performance of the MKL kernels and the overall system load.

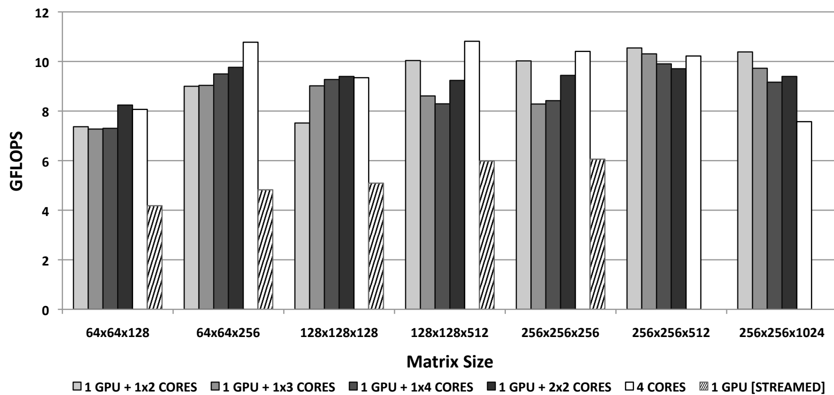


Figure 12: 2D FFT batch results with nested parallelism.

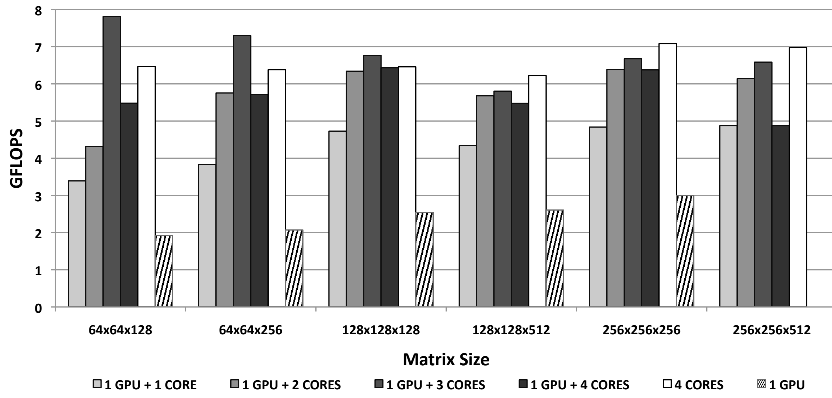


Figure 13: 1D FFT batch results without nested parallelism.

On the other hand, results obtained for the 1D FFT batch execution, presented in Figure 13, show that collaborative execution is favorable for certain problem sizes, whereas for others it will be beneficial to retain the execution in the CPU-only environment.

It is worth to emphasize that the achieved performance for FFT batches does not depend neither on the implementation of the proposed CHPS environment nor the limited computational capabilities of the GPU device. The limiting factor for the overall FFT batch performance lies in the communication-bound nature of the batch FFT algorithm, where the time taken to perform the memory transfers dominates the overall execution time. Figure 14 depicts the performance when calculating 2D FFT batches in the GPU without taking into account the memory transfers (*GPU Kernel*), and when the memory transfers are included into the overall GPU performance for standard (*1 GPU*) and optimized (*1 GPU [streamed]*) implementations. In contrast to matrix multiplication, the performance of 2D FFT batches when the data transfer time is included, does not follow the trends of the GPU kernel performance, which clearly revels that the communication has a higher impact to the overall preference in comparison to the computation. Moreover, a slight overall performance improvement can also be seen for larger problem sizes, which is mainly due to the efficient utilization of the PCI Express bandwidth by larger transfers. Therefore, the results reveal that the current bottleneck for executing the FFT lies in a limited interconnection bandwidth between host and device, and not in the computational capabilities of the GPU device.

Finally, taking into account that the obtained execution values must be extended to include the

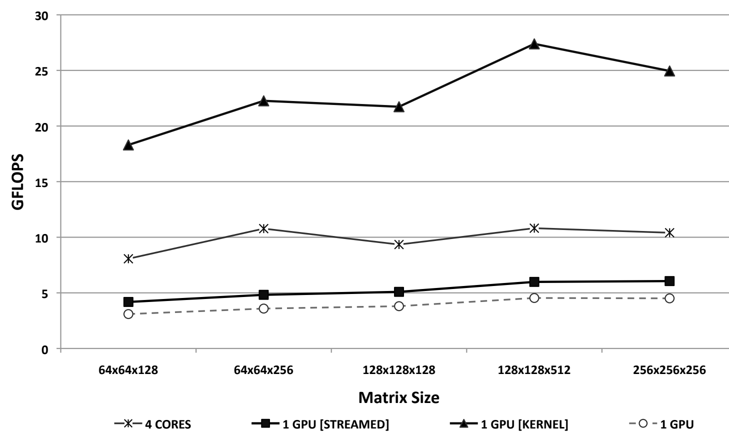


Figure 14: 2D FFT memory transfer impact.

time needed to perform the matrix transpositions, the collaborative execution of the complete 3D FFT in the tested system, and for the evaluated problem sizes, is not capable of providing significant performance benefits.

6 Related Work

Recent research efforts related to the heterogenous commodity desktop systems are mainly focused on exploiting the computational capabilities of the devices, but from the perspective of using a single device at a time to perform domain-specific computations. Occasional attempts to unify the functional resources into a collaborative execution environment are generally twofold. One approach to accomplish the execution unification is to introduce high-level hybrid programming models and languages, such as OpenCL [13], RapidMind [22], Stanford's Brook [5], and Google's PeakStream [21], or to develop the compiler frameworks to sustain the collaboration, e.g. HMPP [7], MCUDA [23], StarPU [1], GPUSs [2] or OpenMP to GPGPU [14].

Another strategy, encouraged by our current implementation, anticipates the low-level integration of highly optimized and vendor-specific libraries to accommodate not only high performance, but also finest level of application tuning and control. Recent works adopting this strategy [19, 3, 18, 8] are predominantly focused on performance and memory transfer modeling in order to find an optimal mapping for the underlying computing devices. Although a large set of obtained experimental results allows us to derive a model of this kind, this model will certainly have a questionable practical importance due to its high per-system and per-application nature.

Studies on matrix multiplication in the CPU+GPU environments [19, 3, 8], are mainly oriented on reaching the peak performance, thus restricting the implementation to the optimized algorithm for small matrix sizes as presented in Section 4. Although the potential to attain high performance using this approach is undeniable and confirmed by the results in Section 5, its practical use is limited to a finite set of problem sizes due to high per-device memory demands. In order to provide a support for larger problem sizes, in this paper we propose more demanding approaches, i.e., the memory-aware Horowitz matrix multiplication algorithm which lessens the memory requirements by appropriate selection of the execution parameters.

Focusing on 3D FFT algorithm, to the best of our knowledge there are no present studies dealing with its implementation and evaluation in heterogeneous desktop systems. Also, Ogata et. al. [18] agree on the lack of the studies in this area, when implementing 2D FFT algorithm in a CPU+GPU environment.

Moreover, the proposed CHPS environment introduces several principles which are not present in the current state-of-the-art approaches, such as flexible task description scheme, which allows to configure the execution environment according to the task type. The task abstractions and user defined per-task configuration parameters allow different task granularities to be executed in the system, thus providing the flexible computation partitioning schemes. The developed platform is also capable of storing the task performance history for each device, which can be re-used to extend the *Job Dispatcher* to perform the agglomerations of the *Primitive Jobs* in a manner to maximize the performance benefits of the collaborative execution. Furthermore, the platform can also be configured to run in an exhaustive search mode, where the *Parallel Jobs* are executed on the requested devices with varying workloads, in order to obtain the full performance models of employed devices. The obtained performance models can be used to find the best mapping and decide on optimal load balancing scheme in the systems where multiple runs of the same kernels are required. Finally, for devices that support concurrency between the memory transfers and kernel execution, such as GPU devices with streaming capabilities, the *Job Dispatcher* module is extended to support the overlapping of the communication with the actual on-device computation by carefully scheduling the *Primitive Jobs* for the *Divisible* tasks.

7 Conclusions

Nowadays commodity computers are powerful heterogeneous systems that are capable of sustaining a significant computational power when employing the full set of available processing units. In this work we propose the CHPS environment which allows to perform the collaborative execution of parallel applications for heterogeneous desktop systems, by exploiting both task and data parallelism. The environment encapsulate different methodologies for collaborative execution depending on the type of the tasks to execute, which are defined by the proposed task description scheme. At the device level, the CHPS environment allows to express parallelism via native programming models, such as OpenMP for the multi-core CPUs or CUDA for GPU devices.

To demonstrate its usability, the proposed framework was programmed to cooperatively compute matrix multiplication, and the main steps for calculating 3D fast Fourier transform, i.e., 2D and 1D FFT batches. The obtained experimental results show that, when using the proposed approach, significant performance improvements can be achieved for matrix multiplication in quad-core CPU and GPU environment, whereas the available interconnection bandwidth between the devices limits the performance of FFT batches.

Future developments are aimed on heterogeneous desktop systems with a higher degree of heterogeneity, such as multi-GPU and/or Cell-based environments. Moreover, our future research interests will be focused on reducing the cost of building the full performance models via exhaustive search to find the optimal load balancing scheme. Namely, we are currently investigating the possibilities of extending the CHPS to make dynamic load balancing decisions with a given accuracy by relying on the partial performance models of heterogeneous devices.

References

[1] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience, Euro-Par 2009 best papers issue*, 2010.

[2] E. Ayguadé, R. M. Badia, F. D. Igual, J. Labarta, R. Mayo, and E. S. Quintana-Ortí. An extension of the StarSs programming model for platforms with multiple GPUs. In *Euro-Par*, pages 851–862, 2009.

[3] S. Barrachina, M. Castillo, F. D. Igual, R. Mayo, and E. S. Quintana-Orti. Evaluation and tuning of the level 3 CUBLAS for graphics processors. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–8, April 2008.

[4] A. Bayoumi, M. Chu, Y. Hanafy, P. Harrell, and G. Refai-Ahmed. Scientific and engineering computing using ATI Stream Technology. *Computing in Science Engineering*, 11(6):92 –97, nov.-dec. 2009.

[5] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream computing on graphics hardware. *ACM TRANSACTIONS ON GRAPHICS*, 23:777–786, 2004.

[6] D. R. Butenhof. *Programming with POSIX threads.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[7] R. Dolbeau, S. Bihan, and F. Bodin. HMPP: A hybrid multi-core parallel programming environment. In *Workshop on General Processing Using GPUs*, 2006.

[8] M. Fatica. Accelerating linpack with CUDA on heterogenous clusters. In *GPGPU-2: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, volume 383, pages 46–51, New York, NY, USA, 2009. ACM.

[9] E. Horowitz and A. Zorat. Divide-and-conquer for parallel processing. *Computers, IEEE Transactions on*, C-32(6):582 –585, jun. 1983.

[10] Intel. Intel Hyper-Threading Technology, October 2010. http://www.intel.com/technology/platform-technology/hyper-threading/index.htm.

[11] Intel. *Intel Math Kernel Library Reference Manual*, March 2010. Version 10.2.

[12] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *IBM Journal of Research and Development*, 49(4.5):589 –604, jul. 2005.

[13] Khronos OpenCL Working Group. *OpenCL Specification*, October 2009. Version 1.0.

[14] S. Lee, S.-J. Min, and R. Eigenmann. OpenMP to GPGPU: A compiler framework for automatic translation and optimization. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 101–110, New York, NY, USA, 2009. ACM.

[15] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *Micro, IEEE*, 28(2):39–55, May 2008.

[16] NVIDIA Corporation. NVIDIA's Next Generation CUDA Compute Architecture: Fermi. White paper, Version 1.1, 2009. Available online (22 pages), *http://www.nvidia.com/content/ /PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf.*

[17] NVIDIA Corporation. *NVIDIA CUDA - Compute Unified Device Architecture Programming Guide*, October 2010. Version 3.1.

[18] Y. Ogata, T. Endo, N. Maruyama, and S. Matsuoka. An efficient, model-based CPU-GPU heterogeneous FFT library. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–10, 2008.

[19] S. Ohshima, K. Kise, T. Katagiri, and T. Yuba. Parallel processing of matrix multiplication in a CPU and GPU heterogeneous environment. In *In 7th International Meeting on High Performance Computing for Computational Science (VECPAR06)*, pages 41–50, 2006.

[20] OpenMP Architecture Review Board. *OpenMP Application Program Interface*, May 2008. Version 3.0.

[21] PeakStream. *PeakStream Stream Platform API: C++ Programming Guide*, 2007. Version 1.0.

[22] Rapidmind. Rapidmind, August 2009. http://www.rapidmind.net.

[23] J. Stratton, S. Stone, and W.-M. Hwu. MCUDA: An efficient implementation of CUDA kernels for multi-core CPUs. In Jos Amaral, editor, *Languages and Compilers for Parallel Computing*, volume 5335 of *Lecture Notes in Computer Science*, pages 16–30. Springer Berlin / Heidelberg, 2008.