Transparent application acceleration by intelligent scheduling of shared library calls on heterogeneous systems

João Colaço, Adrian Matoga, Aleksandar Ilic, Nuno Roma, Pedro Tomás, and Ricardo Chaves

INESC-ID / IST,
Rua Alves Redol, 9, 1000-029 Lisboa - Portugal

Abstract. Transparent application acceleration in heterogeneous systems can be performed by automatically intercepting shared libraries calls and by efficiently orchestrating the execution across all processing devices. To fully exploit the available computing power, the intercepted calls must be replaced with faster accelerator-based implementations and intelligent scheduling algorithms must be incorporated. When compared with previous approaches, the framework proposed herein does not only transparently intercepts and redirects the library calls, but it also incorporates state-of-art scheduling algorithms, for both divisible and indivisible applications. When compared with highly optimized implementations for multi-core CPUs (e.g., MKL and FFTW), the obtained experimental results demonstrate that, by applying appropriate light-weight scheduling and load-balancing mechanisms, performance speedups as high as 7.86 (matrix multiplication) and 4.6 (FFT) can be obtained.

Keywords: Transparent acceleration, heterogeneous computing, automatic scheduling, load balancing

1 Introduction

Over the past decade, Graphics Processing Units (GPUs) have evolved into general-purpose computing devices, offering a substantial performance boost for highly parallel applications. Together with other types of accelerators, including high-end Field-Programmable Gate Arrays (FPGAs) or parallel coprocessors (such as Intel Xeon Phi), GPUs have become widely used in the High-Performance Computing (HPC) domain, and their share is still growing.

However, the adoption of these powerful accelerators in heterogeneous environments often requires a substantial amount of work by skilled programmers to identify the parallelizable kernels and optimize them for specific architectures. In contrast, many application developers usually use different software packages or computing environments (such as Matlab, Octave, R, etc.) to perform the computations, mainly focusing on the algorithm correctness and efficiency, and not

on optimizing the code for any specific device architecture. In fact, highly optimized computational libraries are often adopted to attain the high performance on different devices, such as BLAS, LAPACK and FFTW (for general-purpose CPUs), or cuBLAS, CULA and cuFFT (for GPUs). However, additional speed-ups can be achieved by effectively selecting the most efficient library implementation on a per device basis or even by dividing the work load and simultaneously executing it across multiple device.

Beisel, et al. [1], proposed an interposition scheme to intercept calls to shared libraries and delegate them to one of the existent library implementations, allowing to transparently accelerate applications in heterogeneous systems without changing the original code. To effectively select which library is called at each time, a static scheduling algorithm was used, based on pre-defined performance models. However, this approach does not take into account the performance differences in software implementations, hardware devices or even real-time system usage. Furthermore, to fully exploit the computational power of modern heterogeneous systems, it is absolutely crucial to provide the means for efficient cross-device execution.

In this paper, the idea of transparently accelerating existing applications by replacing the kernels that are implemented as library functions is further investigated. The original idea is extended and further improved by adding intelligence to the system. Two approaches are considered: (a) when the problem cannot be divided due to data/control dependencies, the system selects the best available accelerator; (b) when the problem can be divided into multiple parallel computations, the system automatically assigns each device with different computation portions, thus achieving a collaborative execution. To perform this task, a dynamic load balancing algorithm is adopted that relies on partial estimations of performance models for multiple devices in the heterogeneous system, which are built and updated in real-time. By relying on the proposed framework, it is possible to exploit the full computing capacity of the system, without requiring any special intervention on the system or on the original code by the end-user.

The remaining of this paper is organized as follows. The next section presents the architecture of the proposed framework to transparently accelerate existing applications. Section 3 describes the adaptive scheduling algorithms for both indivisible and divisible load problems. Section 4 presents the experimental results that were obtained with the adopted transparent acceleration approach and discusses the achieved speed-ups for two case-study applications. The last section concludes and addresses some future work directions.

2 Framework Architecture

The architecture of the developed framework, illustrated in Fig. 1, is based on the model proposed in [1]. Accordingly, the LD_PRELOAD environment variable is used to specify the *wrapper library* which transparently redirects the function

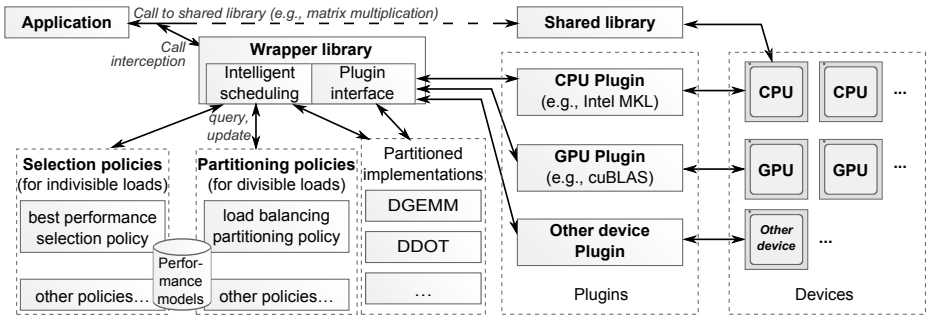

Fig. 1. Architecture of the proposed framework.

execution to one or more of the available *plugins*. An one-to-one mapping between plugins and the underlying devices is assumed.

In the execution environment illustrated in Fig. 1, an application natively using the BLAS library is accelerated using more efficient implementations, such as MKL for CPU or cuBLAS for GPU (see *Plugins* module). Other similar examples could also be devised using other widely used libraries, such as the FFTW/cuFFT or the LAPACK/CULA libraries.

2.1 Selection and Partitioning Policies

Upon a call to a library function, the wrapper library first determines the problem size based on the function arguments and classifies the requested computation as divisible or indivisible in the *Intelligent scheduling* module. If the considered function has an implementation that allows work load partitioning across multiple devices, the configured divisible load *partitioning policy* is queried in order to find a balanced distribution of the loads given to each device. If the call is recognized as indivisible, the fastest plugin for the detected problem size is selected using the configured *selection policy*.

Subsequently, the appropriate function is executed by the selected plugin(s) via *Plugin Interface*. For indivisible work loads, the plugin usually only calls an equivalent function of the accelerated library, such as MKL or cuBLAS, translating the arguments and results where appropriate. Divisible work loads require partitioning of the computation according to the previously computed distribution. The partitioning depends on the actual function that was called and must be implemented separately for individual functions. All available plugins are called with a different portion of the computation and the results are combined accordingly. After the execution, the wrapper requests the used policy to update its *performance model* according to the actual measured performance.

Section 3 presents two policies based on run-time updated performance models. However, the proposed framework is not limited to these two strategies, as it allows integrating different selection and partitioning policies.

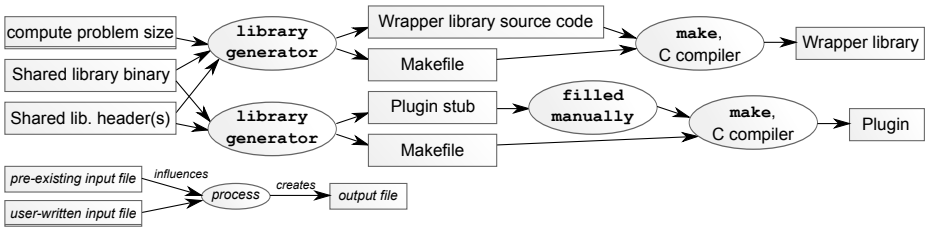

Fig. 2. Generation procedure of the wrapper library and plugins.

2.2 Library Generation

The adopted library generation procedure is described in more detail in [2] along with an application profiling framework that transparently collects extensive profile information (e.g., processor performance counter values and the estimation of the amounts of data to be transferred to and from the accelerator). The approach differs from [1] in what concerns the *micro-generators* concept [3]. Each micro-generator is defined as a class that generates pieces of code related to a specific feature, which are inserted at several distinct scopes in the generated source, such as: global declarations, global definitions, library initialization, library finalization, wrapper function prefix, and wrapper function postfix. Multiple micro-generators can be combined and their options adjusted by the framework user, in order to produce a library dedicated and optimized for a specific purpose. A special kind of micro-generator was considered to allow the programmer to specify external definition files containing portions of code to be inserted at different scopes of the generated library, without modifying the generator itself. This feature is used to insert the code required to estimate the problem size.

The adopted procedure to generate the described wrapper library is presented in Fig. 2. The library generator tool finds and parses the library binary and header files installed in the system and produces the wrapper library incorporating the basic logic that manages the available scheduling policies and plugins. The generated plugin stubs can be used to add support for new accelerators to the system.

Once the above configuration is complete, the programmers and the application users can automatically benefit from the heterogeneous computing resources, without even being aware of their existence.

3 Adaptive Multi-Device Task Scheduling

A selection policy and a partitioning policy are proposed, which use performance models for the available plugins, built during run time. An individual performance model is constructed for each function of each plugin. Each model is stored as an ordered map which associates the problem size N with the performance s expressed as the ratio between the size of computation assigned to

a plugin and the time taken to perform computation and communication. The definition of the problem size depends on the function. For example, for the multiplication of two square matrices, the problem size N may be defined as the matrix dimension M or the number of scalar multiplications involved, i.e. M^3.

3.1 Best Performance Selection Policy

In the most general case, each function call is considered as an indivisible work that must be executed by a single plugin. The goal is therefore to choose the fastest plugin for a particular call, based on the problem size. Assuming that the performance characteristics of the available plugins are not known a-priori, the proposed algorithm dynamically builds the performance models during run time. If, for a given problem size and model, the fastest plugin is not yet known, all plugins are simultaneously executed and their performance is recorded. As soon as the collected information is sufficient to determine the best choice for a particular call, only the fastest plugin is executed, and the model is updated with its last achieved performance. This way, the scheduling scheme is not only self-learning, but is also adaptable to the changing characteristics of the system, such as concurrent application execution that might compete for shared resources.

Given the problem size of N, the detailed procedure to determine the fastest plugin for a particular call uses the two existing neighboring points (left and right) in the performance model (n_{L} and n_{R}), such that $n_{\mathrm{L}} \leq N \leq n_{\mathrm{R}}$:

1. INITIALIZE maximum performance variables for n_{L} and n_{R}: $s_{\mathrm{L}}:=0$, $s_{\mathrm{R}}:=0$.
2. LOOP: for the performance model of each plugin i, $1 \leq i \leq p$:
   (a) Find n_{L} and n_{R} defining the narrowest range, such that $n_{\mathrm{L}} \leq N \leq n_{\mathrm{R}}$;
   (b) If n_{L} is found and performance at n_{L} is greater than s_{L}, then $best_{\mathrm{L}}:=i$;
   (c) If n_{R} is found and performance at n_{R} is greater than s_{R}, then $best_{\mathrm{R}}:=i$.
3. IF both $best_{\mathrm{L}}$ and $best_{\mathrm{R}}$ are assigned and hold the same value, this value determines which plugin should be used to execute the function.
   ELSE, the optimal selection is not known and all plugins execute the function simultaneously. The execution time for each of them is measured and the obtained speed values used to update the models.

Additional overheads are imposed by the need to pass the function arguments to the several threads in which the plugins are concurrently run, as well as to allocate temporary buffers for the results (in order to isolate the plugins from each other) and to copy the results back to the application buffer upon the fastest plugin completes its execution.

While the overheads mentioned above may be significant, generally they only play a role during the first few executions of a given function within the application run time, and further calls will choose the optimal implementation with a minimum overhead. The actual costs related to this approach will be precisely measured and discussed in the experimental evaluation section of this paper.

3.2 Load Balancing Policy

Whenever the intercepted function call allows work load partitioning across multiple devices, a divisible load partitioning policy is used to partition and balance the load given to each plugin. The problem that arises here is how to partition the problem size N across several heterogeneous devices i ($1 \leq i \leq p$), such that the overall cross-device execution (computation and communication) is finished in the shortest possible time. In detail, each device i should process a certain number of independent parcels n_i of the problem size, such that load balancing is achieved. In contrast to other usual approaches in heterogeneous systems, where the speed s_i of each device is described with constants, a more realistic Functional Performance Modeling (FPM) principle [4] is used. In this model the performance of each device i is modeled as a continuous function of the assigned fraction of the problem size n_i, and it is defined at the interval $[0, N]$.

The shortest parallel processing time is attained when all devices finish their execution and communicate the results back at the same time (load balancing condition), such that:

$$\frac{n_1}{s_1(n_1)} = \frac{n_2}{s_2(n_2)} = \cdots = \frac{n_p}{s_p(n_p)}; \quad \sum_{i=1}^{p} n_i = N \tag{1}$$

Since n_i must be integers, Eq. (1) is solved by using a two step algorithm. The first step starts by defining the upper and lower bounds of the solution search space, converging towards the optimal distribution by bisecting the angle between these two boundaries, assigning such bisection to one of the search limits in each iteration. Then, it approximates the load distributions by rounding down to the nearest integers. With such preliminary distribution, the algorithm proceeds to the second step (*refinement*), which iteratively redistributes the remaining load to the processing devices according to the devices speed s_i, until assigning the total problem size N to all processors. This results in an algorithm with $\mathcal{O}(p \log N)$ complexity, whose formal proof can be found in [4].

However, the process of building the full performance models for each application and device in the system (i.e., the model for a full range of problem sizes) might be very time consuming. Hence, the adaptive load balancing approach proposed in [5] is adopted, which builds the partial estimations of the full FPMs during the application run-time. For each device, the partial FPMs are built by applying piecewise linear approximations on a set of points, which are obtained from previous application runs according to the number of performed loads and the time taken to process them [5]. Since the performance models are not known *a-priori*, the adaptive load balancing starts by assigning each device with an equal amount of loads to process. Hence, in the first iteration, the total problem size N is evenly partitioned, such that each device is assigned with $n_i = N/p$. This part of the load balancing is referred to as the *initial phase*. The subsequent *iterative phase*, used to obtain the new load distributions, consists of two steps: *i)* update of the partially built FPMs; and *ii)* determination of the new load distributions, by applying the previously described algorithm to the newly approximated FPMs [5].

4 Experimental Results

Computationally intensive mathematical applications are particularly suitable to evaluate the proposed framework. For such purpose, GNU Octave was naturally selected since it relies on several libraries to execute different types of mathematical operations. From all the Octave required libraries we selected two of the most commonly used: BLAS, for double-precision dense matrix multiplication (DGEMM), and double-precision complex to complex Fast-Fourier Transforms (FFTs). The following subsections describe the experimental setup and present the obtained application acceleration when using these two example libraries.

4.1 Experimental Setup

For the purpose of evaluating the proposed framework, a machine with a Quad-core Intel i7-950 CPU (3.07 GHz), with 12 GB DDR3-1033, and two NVIDIA GTX 580 GPUs was used. Given the limited performance of the native GNU Octave BLAS library (cBLAS), the conducted analyses uses the highly optimized MKL multi-threaded library for the BLAS baseline and the FFTW library [6] as the reference for the FFT performance.

To improve Octave performance, the mechanisms provided by the proposed framework were used to automatically create the plug-in wrappers [2] for the accelerated libraries, namely MKL, cuBLAS, FFTW and cuFFT [7, 8]. It should be noticed that while all the considered libraries adapt the algorithms to the work load and underlying architecture, for the FFT plugins, the micro-generation is more challenging. In detail, the computation of the FFT is split into two phases, *planning* and *execution*, which were incorporated into the framework.

4.2 Performance Characterization

In order to evaluate the efficiency of the proposed **Best Performance** Selection policy (BPS), we executed multiple calls and with different problem sizes for the DGEMM and FFT libraries within Octave. The obtained experimental results for the DGEMM function, depicted in Figure 3, show the execution time obtained when using the BPS policy. As it can be observed BPS was capable of selecting in all cases the plugin with the best performance. In the specific DGEMM case, the MKL library delivers better performance for matrix sizes of up to 350×350, after which the cuBLAS library becomes faster. This is mainly due to the fact that for smaller problem sizes the data transfer overheads are too large and the GPU computing potential is under-used. Naturally, for other libraries the selection point will be different, according to the characteristics of the algorithm. For example, for the considered 1D FFT problem sizes (65k to 4.2M) the best performance is always achieved with cuFFT library (see Fig. 4).

For the specific cases of divisible problems, additional speed-ups can still be achieved by relying on the **Load Balancing** (LB) policy. When appling the LB policy to the DGEMM function, we consider the basic column based matrix partitioning algorithm, where each plugin computes a different set of columns of the

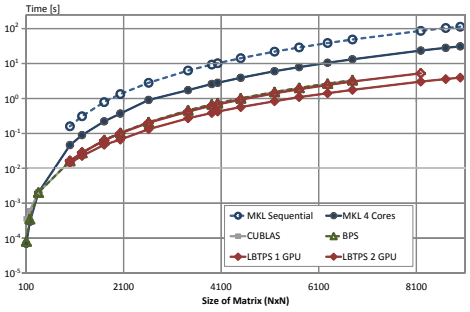

Fig. 3. DGEMM Execution time.

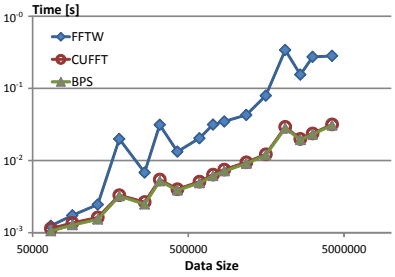

Fig. 4. 1D FFT Execution time.

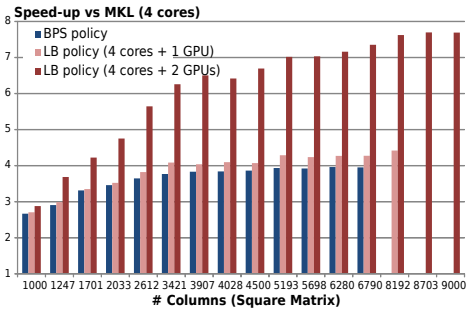

Fig. 5. DGEMM Speed-up.

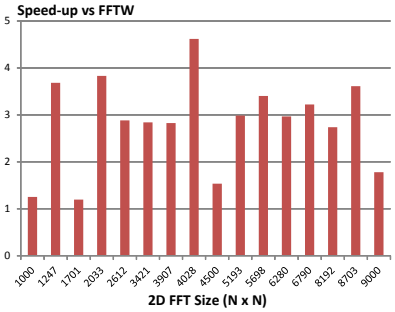

Fig. 6. 2D FFT Speed-up

resulting matrix. In this scope, the source matrix B is divided into column-sets according to the partial estimations of the full performance model of the available plugins (which are constructed in run-time). Figure 5 presents the speed-up values regarding the highly optimized, multi-threaded MKL implementation, running on all four cores. It can be concluded that the adaptive LB policy not only delivers a performance which greatly surpasses a single device, but it also allows executing the DGEMM function on problem sizes that do not fit on the GPU memory (see right side of Figure 5). Figure 6 presents the speed-ups obtained when collaboratively performing 2D FFT across all four CPU cores and a single GPU. The results reported herein reflect the performance of parallel 2D FFT CPU+GPU implementation which performs 1D FFT on different matrix dimensions, each of them followed by a matrix transposition (performed with Eigen linear algebra library). Experimental results for the 2D FFT, show that despite the matrix transposition overhead, speed-up of up to 4.6 can be achieved, regarding the original FFTW 2D implementation.

A temporal diagram of DGEMM execution is presented in Figure 7, where not only it can be observed the acheived load balancing between the devices, but also the insignificant overhead, as discussed in the following subsection.

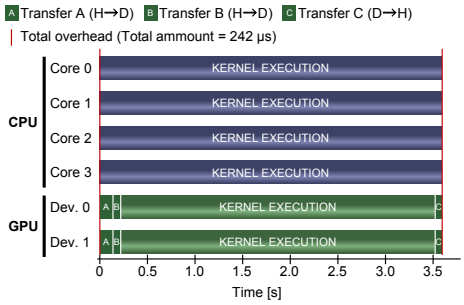

Fig. 7. Temporal diagram of BLAS dgemm execution after load-balancing for a 8703×8703 matrix multiplication case.

Overhead	Amount	Time
Library Interception, Redirection and Return	C	0.16 μs
BPS: Model Update	1	0.34 μs
Thread Dispatch	D	36 μs
Fastest Selection	$C-1$	3.16 μs
LBTPS: Model Update	C	0.42 μs
Partition Distribution	C	25.05 μs
Thread Dispatch	$C \times D$	36 μs
GPU INIT: cuFFT	1	1.3 s
cuBLAS	1	0.273 s

Table 1. Framework overheads, considering C function calls of a given work size, using D devices.

Overhead To evaluate the scheduling overhead that is introduced by the proposed framework, the amount of time required by the several steps of the implemented algorithm were properly characterized, as presented in Table 1.

The first component corresponds to the function call *interception, redirection and return*. As it can be observed, it represents a rather insignificant amount of time, independent of the problem size. This overhead must be considered once per function call.

Then, when the *Best Performance* selection policy (BPS) is used and it is not possible to find the fastest plugin for a given problem size, all implementations are run at the same time. This implies an added overhead to dispatch a thread to each of the D devices and a consequent model update. As soon as the information regarding a given problem size is collected and stored in the model, the only overhead for consequent calls in the scheduler will be the plugin selection. Therefore, in a best case scenario, the total overhead of BPS is just 3.327 μs. In the case of the *Load-Balancing* policy (LB) the incurred overhead for each function call is related to partition distribution and model update. Although the model update overhead is stable, the partition distribution depends on the information already gathered in the model (hence, the presented values represent average values).

Finally, since several implementations run in parallel, there is an average of 36 μs per plugin overhead for spawning threads. This parcel occurs only once with BPS (for each device), while LB imposes this overhead at every execution.

The last two entries in this table represent *library initialization* phases. They are independent of the scheduling framework and have to be considered only once, before the actual scheduling takes place. In the considered experimental procedure, both cuFFT and cuBLAS imply large overheads. To mask these CUDA libraries initialization times, they can be executed asynchronously. As such, provided that the target application does not use them at start-up time, this overhead component can be completely hidden. As a consequence, these initialization times were ignored in the presented benchmark tests.

5 Conclusions

A new application acceleration framework based on a transparent redirection of shared-library function calls to the several existing devices in a heterogeneous system was proposed in this paper. The adoption of dynamically constructed performance models allowed this framework to reduce the overheads and to quickly adapt to the user application behavior, without the need to modify the program source code. By recording the actual performance of the available devices for different library functions and problem sizes, inteligent scheduling mechanisms were implemented in order to allow divisible work loads to be partitioned across all devices in the system, achieving the best balance and attaining the maximum performance. Indivisible work loads are redirected to the fastest single-device implementation that is available, based on the corresponding problem size.

The framework was evaluated by comparing the attained performance with state-of-the art single-device implementations, i.e. Intel MKL and FFTW. The obtained results have shown that speedups as high as 7.86 for matrix multiplication and 4.6 for FFT can be obtained, with negligible overheads imposed by the proposed call interception and scheduling mechanisms.

Acknowledgments

This work was partially supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) under projects "HELIX" (ref. PTDC/EEA-ELC/113999/2009), "Threads" (ref. PTDC/EEA-ELC/117329/2010), "TAGS" (PTDC/EIA-EIA/112283/2009) and project PEst-OE/EEI/LA0021/2013.

References

1. Beisel, T., Niekamp, M., Plessl, C.: Using shared library interposing for transparent acceleration in systems with heterogeneous hardware accelerators. In: Proceedings of the ASAP. (2010)
2. Matoga, A., Chaves, R., Tomás, P., Roma, N.: A flexible shared library profiler for early estimation of performance gains in heterogeneous systems. In: Proceedings of the HPCS. (2013)
3. Fetzer, C., Xiao, Z.: A flexible generator architecture for improving software dependability. In: Proceedings of the ISSRE. (2002) 102 – 113
4. Lastovetsky, A., Reddy, R.: Data partitioning with a functional performance model of heterogeneous processors. Int. Jrnl. of High Perf. Comp. App. **21**(1) (2007) 76–90
5. Clarke, D., Lastovetsky, A., Rychkov, V.: Dynamic load balancing of parallel computational iterative routines on highly heterogeneous HPC platforms. Parallel Processing Letters **21**(02) (2011) 195–217
6. Frigo, M., Johnson, S.G.: The design and implementation of FFTW3. Proceedings of the IEEE **93**(2) (2005) 216–231
7. NVIDIA: CUBLAS Llibrary : User Manual. (2012) `http://docs.nvidia.com/cuda/pdf/CUDA_CUBLAS_Users_Guide.pdf`.
8. NVIDIA: CUFFT Llibrary : User Manual. (2012) `http://docs.nvidia.com/cuda/pdf/CUDA_CUFFT_Users_Guide.pdf`.