A Flexible Shared Library Profiler for Early Estimation of Performance Gains in Heterogeneous Systems

Adrian Matoga, Ricardo Chaves, Pedro Tomás and Nuno Roma
INESC-ID / IST TULisbon
Lisbon, Portugal
Email: {Adrian.Matoga,Ricardo.Chaves,Pedro.Tomas,Nuno.Roma}@inesc-id.pt

***Abstract*—The effective acceleration of computationally demanding applications in heterogeneous systems often requires significant optimization efforts. Although such task typically starts with a thorough profiling stage, a special attention must be given to the migration procedure of each application kernel: apart from the actual computation time, the cost of the data transfers between the main processor memory and the accelerator plays a significant role, which often limits the actual resulting speedup. In some cases, no performance gain is actually achieved, given the excessively high communication to computation ratio. To ease the system designer effort, this paper proposes a framework that transparently collects extensive profile information, including, but not limited to, the values of the processor performance counters, as well as an estimation of the amounts of data to be transferred to and from the accelerator. The framework focuses on transparent acceleration of kernels implemented as library functions and is based on the shared library interposing technique. By further processing of the obtained execution profiles, together with the proper communication and computation models, the attainable global speedup of the accelerated application is predicted. The presented methods were validated experimentally for a set of existing applications. The measured global speedup estimation error typically ranged between 1 and 4%.**

***Keywords*—heterogeneous systems, performance prediction, software instrumentation**

I. Introduction

The combined processing power of multi-core CPUs and Graphics Processing Units (GPUs) is now available in most off-the-shelf personal computers. Such type of heterogeneous architectures has found a significant interest for High-Performance Computing (HPC) applications, where computationally demanding algorithms are developed with massively parallel processing units and vector instructions in mind, in order to fully exploit the available computational resources. Specialized accelerator chips and reconfigurable technologies, such as FPGAs, are also often used on HPC or embedded systems to meet more restricted requirements of performance, real-time constraint, or energy consumption.

Meanwhile, the rapid growth of the market share of heterogeneous machines has been followed by an increasing interest of system developers in simplifying and automatizing the programming and use of these architectures. Standards such as OpenCL [1] aim at unifying the programming model for different heterogeneous architectures.

Many HPC applications rely on library implementations of computationally intensive kernels. Some popular examples include: BLAS [2] and LAPACK [3] for linear algebra, GSL [4] for solving numerical problems, FFTW [5] for computing the Discrete Fourier Transforms, OpenCV [6] for image and video processing and feature extraction, and the Insight Toolkit [7] for medical image analysis. Most often, the code of these libraries is not included in the application binary. Instead, it is dynamically loaded at run time from *shared libraries* that were previously and separately installed in the system.

Apart from other advantages, this decoupling provides the opportunity to selectively replace the original kernel implementations with alternative ones, without modifying the application code or the default libraries. This technique is known as *shared library interposing* and its basic principle is depicted in Figure 1. In Unix-like operating systems, the user can configure the dynamic linker to load additional shared libraries before, or instead of, the default ones. The original shared library functions may either be completely replaced with different implementations, or a *wrapper library* may be introduced between the application and the original library, in order to redirect the library function calls to one of possible existing alternatives based on various static or dynamic policies. In both cases, the wrapper library must have the same Application Binary Interface (ABI) as the original one.

The shared library interposing technique has already shown to be a promising approach for transparent application acceleration [8]. The ability to accelerate applications without the need to modify either the application or the libraries makes

This work was partially supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) under project "HELIX: Heterogeneous Multi-Core Architecture for Biological Sequence Analysis" (reference number PTDC/EEA-ELC/113999/2009), project "Threads: Multitask System Framework with Transparent Hardware Reconfiguration" (reference number PTDC/EEA-ELC/117329/2010), and project PEst-OE/EEI/LA0021/2011.

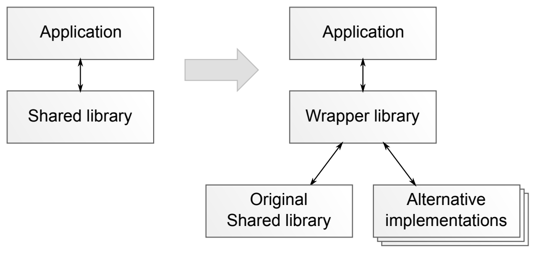


Figure 1. Illustration of the shared library interposing technique. A wrapper library is inserted in place of the library originally used by the application, in order to provide alternative implementations of the library functions.

this technique especially useful for closed-source applications, but also enables the users and higher-level programmers with little or no knowledge on the actual heterogeneous architecture to utilize the capabilities it offers.

However, development of efficient accelerated libraries requires the detailed knowledge of how the libraries are used by real applications, including the patterns of communication between the software and the accelerated library. In fact, offloading the computation from the CPU to an external resource often requires transferring large amounts of data. Depending on the interconnection interface, the communication scheme and the amount of data that is manipulated by the application, the data transfers in the considered system may become a serious bottleneck, considerably decreasing the overall performance. Thus, an accurate estimate of the attainable performance gains must take the actual communication patterns into account.

The framework that is herein presented aims at accurately predicting the performance gains of newly designed accelerators, by integrating the detailed execution traces of the target applications with the communication and computation models of the underlying accelerator architectures. Specifically, the following original contributions are presented in this paper: *i*) a method for automatic estimation of the amount of data exchanged between the application and the called library functions, together with its evaluation on several different case studies; *ii*) a flexible wrapper library generator for obtaining detailed performance data, which also implements the proposed method; *iii*) a post-processing tool for accurate prediction of the global speedup on different accelerator architectures.

The remaining of this paper is organized as follows. In the next section, the related work is presented and discussed. Section III describes the performance prediction approach adopted in the proposed framework. Section IV gives a more detailed insight into the framework architecture. Section V discusses the obtained experimental results, including the accuracy of the obtained estimates, and the overheads imposed by the instrumentation. Section VI concludes this paper with some final remarks and future work directions.

II. Related Work

Although performance prediction and estimation in heterogeneous systems has been investigated with increasing depth, with a few exceptions the existing approaches often only consider the performance of certain applications on a single architecture. In [9] it is proposed a new heuristic for implementation planning in heterogeneous systems, in order to help the selection of the right implementation for a specific work size. Although the presented results are promising, the relation between the work size and the function arguments must be specified by the user, in the form of a class that implements the expected interface. Sato et al. [10] presented a runtime performance prediction model based on the classification of the parameters to OpenCL kernel calls depending on their correlation with past execution times on a particular resource.

Hence, a profiling tool capable of automatically estimating the work size should be able to examine the data structures passed to and from the called functions in the context of the previous and future activity of the traced application.

There are tools for tracing the shared library function calls, such as *latrace* [11] and *ltrace* [12] However, in order to be able to display the arguments passed to the called functions, they rely on ABI-specific stack inspection, as well as custom-formatted configuration files, which redefine the prototypes of known library functions.

DTrace [13] is a dynamic instrumentation framework to be integrated in the operating system kernel and allows for tracing different aspects of a running system by means of user-defined actions taken at specified execution points, called *probes*. It focuses on minimizing the tracing overhead by only executing the probes which are explicitly enabled for a specific session, as well as reducing the amount of produced data. Tracing user-level function calls is also possible, but it depends on the ABI of the specific platform on which the application and the framework are executed.

Although very flexible and powerful, instrumentation tools based on runtime binary interpretation, such as Pin [14] or Valgrind [15], impose noticeable overheads even without profiling, and their use is limited to the supported Instruction Set Architectures (ISA).

Hence, when compared with the above approaches, the shared library interposing technique inherently enables the combination of the static information of all the function arguments with run-time access to the application state. Furthermore, it does not require any support from the operating system kernel and does not depend on any architecture-specific implementation. Since the function arguments, as well as the complete state of the application at the called function entry and exit points are available for the wrapper library, any kind of information about the application execution can be extracted.

The major difficulty presented by shared library interposing, which is the need to implement the wrapper library before it can be actually used, has been already addressed by several code generation approaches, the following three being notable examples.

Curry [16] presented the first profiling framework which used automatically generated wrapper libraries to transparently collect performance data. Adding function-specific code to the generated libraries required that the generated sources were manually edited.

In [8], automatically generated wrapper libraries are used to redirect the BLAS subroutine calls to different heterogeneous accelerators, based on their previously obtained performance characteristics. These characteristics, however, were obtained using a different approach, namely, the specifically crafted Pintools [14].

The most flexible shared library generation tool to date was presented by Fetzer and Xiao in [17], as a part of their framework for increasing the *robustness* of shared libraries. The architecture of their framework allows building the wrapper libraries from small code pieces inserted at different scopes in the generated source. A similar approach was adopted in the work described herein, and will be described in the following sections.

III. Accurate Speedup Prediction in Heterogeneous Systems

The maximum attainable speedup when accelerating a given portion of an application is typically computed by applying the Amdahl's law:

$$S_{\mathrm{G}} = \frac{1}{(1-P) + \frac{P}{S_{\mathrm{P}}}}, \qquad (1)$$

where S_{G} is the attained global speedup, P is the execution time fraction corresponding to the portion to be optimized, and S_{P} is the speedup for the accelerated portion alone.

However, the main difficulty in predicting the actual speedup in real applications lies in the fact that applications often call their computationally intensive kernels multiple times for different inputs, so that each call corresponds to a different fraction of the total execution time. Moreover, the speedups offered by the accelerated kernel implementations in a given system depend upon the problem size. This means that the accelerated fraction and the partial speedups will differ from one application to another, as well as between several runs of the same application with different inputs, affecting the global speedup accordingly. The situation is further aggravated by the fact that the performance of the existing accelerated kernels is usually announced without considering the communication overhead, which depends on the characteristics of the communication interface and the amounts of transferred data.

Hence, the main aim of the proposed framework is to provide the means for an accurate measure of the execution time of each kernel invocation, and a consequent substitution of the measured value with the time the accelerated implementation of the same kernel is expected to require for the same problem size.

Let K be the set of kernels that are to be accelerated, and N_k the number of times a particular kernel $k \in K$ is called during a single run of an application. Let t_{ki} be the execution time (without acceleration) for each of these calls, $i \in \{1, 2, \ldots, N_k\}$. The fraction to be accelerated, including all accelerated kernels, can thus be written as:

$$P = \frac{\sum_{k \in K} \sum_{i=1}^{N_k} t_{ki}}{T}, \qquad (2)$$

where T is the total execution time of the application before acceleration.

In order to obtain the predicted speedup for the considered portion of the application (S_{P}), let us consider w_{ki} as the *work size* associated with the i-th call to kernel k. The work size is a metric that quantifies the computational effort in a kernel-specific way. For example, the work size for matrix multiplication can be defined as the matrix dimension.

By introducing a performance model function $t'_{ki} = f_k(w_{ki})$, which maps the work size w_{ki} to the execution time of the accelerated kernel k, including the time needed to copy the data between the CPU and the accelerator, the speedup for the accelerated portion can be computed using the following formula:

$$S_{\mathrm{P}} = \frac{\sum_{k \in K} \sum_{i=1}^{N_k} t_{ki}}{\sum_{k \in K} \sum_{i=1}^{N_k} f_k(w_{ki})}. \qquad (3)$$

The global speedup S_{G} can now be computed by substituting (2) and (3) into (1), provided that the execution times t_{ki} and work sizes w_{ki} for each kernel invocation are available. Optionally, the prediction procedure may even consider only those calls for which $t'_{ki} < t_{ki}$. This will allow an implementation of an advanced scheduling mechanism that will always choose the fastest available implementation.

Hence, the main contribution of the proposed framework is to fully automate the process of measuring the execution times and estimating the work sizes for multiple and possibly different kernel invocations. Assuming that a performance model is available for a given accelerated kernel implementation, the collected characteristics are combined with this model in order to predict the speedup that can be achieved if the current kernel implementation is replaced with an accelerated one.

IV. A Performance Prediction Framework Using Shared Library Interposing

This section describes the developed framework for transparent profiling of shared libraries and accurate prediction of attainable performance gains in heterogeneous systems. The main characteristics of this framework are:

- It does not require any modification to the traced application or library. The wrapper libraries are automatically generated based on the shared library binary and corresponding header files installed in the system.

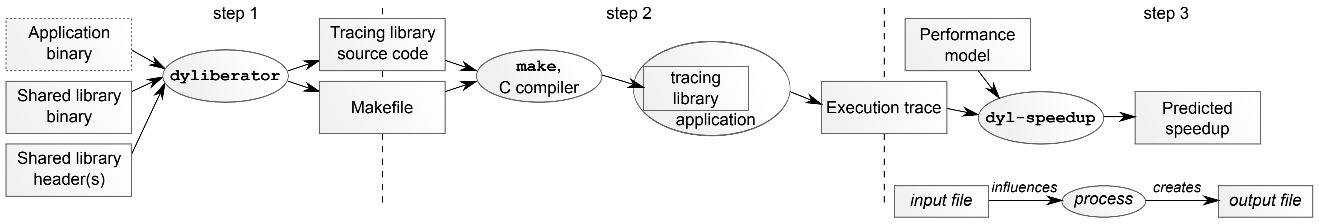


Figure 2. Flow diagram depicting the speedup prediction process by using the proposed framework.

- Arbitrary actions can be executed upon each function call. A set of default actions for tracing and profiling tasks is provided in the form of *micro-generators* (as in [17]), which can be enabled separately and combined according to the user's needs. User-defined actions can be supplied via additional definition files.
- Entirely automatic use of the framework for performance prediction provides meaningful estimates. Further refinement is still possible, by using the customization options mentioned above.
- It is fully portable to different instruction set architectures. The only requirements are an available C compiler and the operating system support for shared library interposing by means of the LD_PRELOAD environment variable.

The following subsections provide a detailed description of the key features of the developed framework.

A. Framework Architecture

The usage of the framework as a performance prediction tool is depicted in Figure 2. The procedure consists of three steps, which can be invoked either manually, in order to adjust the available option settings, or by using a script, to automate the task and avoid typing repetitive commands.

The first step is the generation of the actual tracing library for a particular application and the targeted set of functions to be traced, in order to collect the execution characteristics. This is done by using the developed tool called *dyliberator*. It reads the application executable file and the shared library files in order to find all library functions that the application may call. It then searches for the function prototypes in the C header files for the relevant libraries. Based on these prototypes, it generates new implementations for the set of functions selected by the user, filling them with the required C code that: *i*) calls the original function implementations and *ii*) traces all calls to these functions, collecting the necessary profiling data. For the purpose of performance prediction, at least the execution time t_{ki} and the work size w_{ki} must be collected for each traced call.

The second step is the compilation of the generated library and the preload of it for the time of the application execution. The execution traces should be collected for a sufficiently large representative set of operating conditions for which the accelerated application is intended to be used.

In the final step, the traces are combined with the performance models for different implementations of the specific function (or functions), in order to estimate the speedups that can be obtained with each kernel. This last step is performed using the developed *dyl-speedup* tool.

B. Wrapper Library Generation

The wrapper library generation (step 1 in Figure 2) is implemented similarly to the framework presented in [17]. The architecture is based on *micro-generators*. Each micro-generator is a class that inserts small portions of code related to a specific feature in certain locations in the generated library source. The user can compose multiple micro-generators, thus combining different actions (such as profiling, tracing or call delegation) in one wrapper library.

Compared to its predecessor, *dyliberator* offers a number of improvements which make it more robust and flexible. The headers are parsed using the Clang C compiler front-end [18], which ensures correct parsing of headers written using recent ANSI C standard and GCC-specific extensions.

Instead of fixed priorities (like in [17]) the micro-generators of *dyliberator* may specify which other micro-generators provide their prerequisites, thus allowing for easier usage and more flexible dependency relationships. The user can also specify parametrization options for the micro-generators, in order to fine-tune and adjust the code they generate.

In addition to the C source for the wrapper library, *dyliberator* also generates an appropriate Makefile, ensuring that the library will be built with the correct set of compiler and linker options, and all the required external libraries will be linked.

An example of a wrapper function generated by *dyliberator* is shown in Figure 3. It combines the following micro-generators: *std.FuncPtr*, providing the pointer to the original function; *std.Loader*, which is responsible for loading the original function implementation; and *std.Caller*, which inserts the call (through the function pointer) to the original implementation.

C. Determining the Work Size

In the next step (see Figure 2), the application is executed and the calls to library functions are traced. The execution times and work sizes for each call are saved to a trace file. The execution time can be measured precisely using the

```
/* From header file "/usr/local/include/blas.h" */
double ddot_(const int* n, const double* dx,
  const int* incx, const double* dy, const int* incy)
{
  double __result;
  /* generators.std.FuncPtr () */
  static double (*__funcPtr)(const int* n,
    const double* dx, const int* incx,
    const double* dy, const int* incy) = NULL;
  /* generators.std.Loader () */
  if (!__funcPtr)
    __funcPtr = __Dyl_loadFunction(__DYL_FID_ddot_);
  /* generators.std.Caller () */
  __result = (*__funcPtr)(n, dx, incx, dy, incy);
  return __result;
}
```

Figure 3. Example of a wrapper function generated using a set of three micro-generators: std.FuncPtr, std.Loader, and std.Caller.

system-wide real-time clock or hardware performance counters via PAPI [19]. There are standard micro-generators for both methods. The estimation of the work size deserves a more detailed explanation.

For a large class of algorithms, the work size is directly related to the size of the data passed to the called function and returned from it. Even though the data structures may be arbitrarily complex, many useful functions use plain arrays. In C and Fortran libraries, arrays and matrices are usually passed to functions by a pointer to the data under processing together with one or more integer values defining the array dimensions and strides. Although this scheme is very common, there is no universal convention of specifying this information. The input size and the expected output size may not be directly specified in the function arguments, but rather hidden in the data structures, for private use of the library. Thus, extracting this information would require the understanding of at least some of the library/function implementation details. Hence, an autonomous algorithm might not be able to correctly derive the data size from statically available information, such as the function prototypes or even the complete source code. A different approach is thus considered, in which the data size is estimated based on the observed dynamic behavior of the application.

1) Pointer Tracking Method for Automatic Work Size Estimation: The method herein proposed is based on dynamic pointer inspection. It is assumed that whenever a pointer is passed to a function, it is not by itself a parameter to the kernel. Instead, the pointer usually points to a buffer where the actual input data is placed before the function is called, or where the output data is to be written by the function.

Most applications use heap-allocated buffers or memory-mapped files to pass large amounts of data. Smaller buffers are occasionally allocated on the stack. This method aims at estimating the actual size of the data by monitoring the structure of the memory areas used by the application and by mapping the pointers passed as function arguments to these regions.

The implementation of the proposed method is included in the generated wrapper library by enabling the relevant micro-generator upon *dyliberator* invocation. When the application executes, the wrapper records all heap management activity of the traced application, namely, the calls to the following C standard library functions: *malloc()*, *free()*, *calloc()*, *realloc()*, and *posix_mem_align()*. It also examines the process memory map, in order to find the stack and static data regions.

The recorded information regarding identified data memory regions consists of pairs $R_i = (s_i, e_i)$, where s_i and e_i denote the first address of the region, and the first address past the end of the region, respectively. Since the regions do not overlap, any pointer p points into at most one region in the linear virtual address space of the traced process.

Locating a pointer within one of the known regions is not sufficient to determine the actual size of the data within the buffer pointed to by this pointer. Applications often allocate more memory than they immediately need, because the buffers may be reused for varying amounts of data. Furthermore, one memory region may be allocated for multiple buffers, or, alternatively, subvectors of one matrix may be passed by different pointers. Deeper pointer hierarchies are also possible (e.g. an array of pointers to multiple buffers), in which case the algorithm continues to dereference successive pointers in the potential array, until the first pointer that does not fall into any known region is encountered.

In order to detect multiple buffers located within one allocated memory region, the following solution was applied. Upon each traced function call, the pointer arguments are grouped by the region into which they point. Given a subset P of pointers $p_1, \ldots, p_n$ falling into the same region R_i (i.e. $\forall p_j : s_i \leq p_j < e_i$) in a single function call, the upper bound for the total amount of data passed via all n pointers is $l = e_i - \min(P)$.

This rough, but rather useful estimate is the basis of the *fast* variant of the proposed method. If any pointer within R_i is *const*-qualified, l is added to the size of the input in the current call. Otherwise, it is added to the size of the output. In-out buffers are not detected by this variant.

2) Buffer Content Inspection: In an attempt to automatically determine the size of the partially used buffers, another variant of the proposed method was also implemented, by considering only the memory contents that are actually modified. In this approach, a copy of the previous content of each memory region is kept, so that only the modified content is included in the resulting estimate. Upon each call to the wrapper function, the memory content is examined twice to identify possible changes: before and after the original function is called. If the buffer content is determined to have changed at the entry point, the changed content is assumed to be the input data to the function. The changes found at the exit point contribute to the size of the function output. This variant is referred to as the *full* inspection variant.

In both variants, arguments passed by value contribute to the amount of input data with the result of the *sizeof* operator applied to them.

3) Manual Work Size Estimation: In case the automatic work size estimation methods are unable to give sufficiently accurate results, the proposed framework provides alternative means for the user to manually specify the expression used to compute the amount of data based on the function arguments.

This expression is defined in an external file, consisting of a series of definitions in the following format:

library_name::*function_name*::*location* { *C expressions* }

Arbitrary C language code may be inserted. The function arguments can be accessed by the same names specified in the prototype contained in the library header file. The library and function names may also contain wildcard symbols, which allow the insertion of the same code for a group of functions. The *location* is either "prefix" or "postfix" and tells the tool to insert the code before or after the call to the original function, respectively.

As an example, to compute the number of bytes corresponding to the input and output matrices for the general matrix multiply functions of BLAS, the user can specify the expressions for the input and output size estimation as:

```
// void dgemm_(char *transa, char *transb,
//  int *m, int *n, int *k, double *alpha,
//  double *a, int *lda, double *b, int *ldb,
//  double *beta, double *c, int *ldc);
libblas*::*gemm_::postfix {
  output_size = *m * *n * sizeof(*c);
  input_size  = *m * *k * sizeof(*a)
              + *k * *n * sizeof(*b)
              + *beta == 0 ? 0 : output_size; }
```

D. Computing the Global Speedup

In the last step (step 3 in Figure 2), the collected trace files are combined with the performance models of the considered kernels in order to compute the predicted total execution time and the speedup that is obtained when the acceleration is enabled. It is assumed that the performance models were obtained beforehand, by measuring or estimating the kernel performance on an existing or simulated accelerator.

A performance model, as defined in Section III, is a function that maps the work size w to the execution time t' of each kernel for a given work size. This function is defined by a set of samples (w_i, t'_i). Execution times for work sizes not included in this set are estimated by linear interpolation. Thus, the samples should be denser in those regions where a small change of work size induces a significant difference in the resulting execution time.

For existing kernel implementations, these performance models can be obtained by measuring the time of many identical calls to the modeled kernel and computing the mean. This procedure is repeated for a wide range of work sizes with reasonable density. In order to have a reliable model, it is important that the measured execution time includes all the necessary communication overheads between the CPU and the accelerator.

V. Experimental Evaluation

In order to evaluate the proposed performance prediction framework, the accuracy of the resulting estimates was measured for different application/library pairs. First, the accuracy of the global speedup prediction is discussed based on the comparison of the predicted values with the actual performance gains for several benchmarks covering different work sizes. Second, the results of the automatic data transfer size estimation are compared with the expected values. Finally, in order to evaluate the impact of the used wrapper libraries on the execution of the traced applications, the overheads imposed by the wrappers were measured.

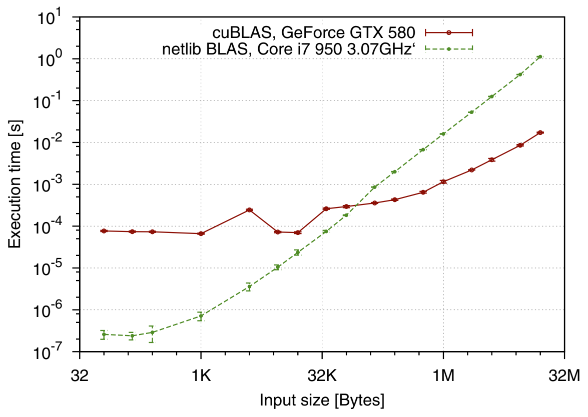
A. Global Speedup Prediction

To accurately analyze the global speedup prediction, the actual achievable speedup was measured for two distinct applications, running on different heterogeneous architectures, namely:

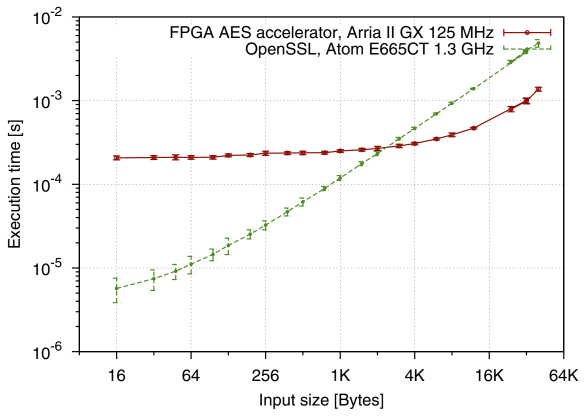
1) GNU Octave [20] for matrix multiplication, originally using the reference BLAS implementation [21], and subsequently accelerated using the equivalent function from the CUBLAS library [22]. The considered setup is denoted as *GPU* in Table I.
2) EncFS encrypted file system [23], using the Advanced Encryption Standard (AES) implementation provided by the OpenSSL library [24], and subsequently accelerated using a state of the art equivalent implementation provided by a custom built cryptographic IP core [25]. The configuration used for this test is denoted *FPGA* in Table I.

The performance charts for these accelerated implementations were obtained by measuring the kernel execution time on the relevant accelerators for different input sizes. As it can be seen in Figure 4, in both cases the difference in the execution time between the accelerated and the non-accelerated implementations depends on the size of the input data. Moreover, it is also observed that for certain input sizes the accelerated implementations are actually slower.

In order to evaluate the accuracy of the proposed speedup prediction framework, a set of profiling runs was first carried out for each application using the original shared library (i.e. no acceleration). The execution time and the work size for the relevant function calls were collected. The execution times were measured with a resolution of 1 ns, while the work sizes were computed using user-supplied expressions. The obtained execution traces were then processed together with the performance models of the considered accelerators, in order to derive the predicted execution time and speedup values. Finally, the wrapper libraries, which redirect the library function calls to hardware-accelerated kernels were built,



(a)



(b)

Figure 4. Performance models of two hardware-accelerated functions used to predict the attainable speedup values. Performance of equivalent software implementations is included for comparison. (a) matrix multiplication. (b) AES (CBC, 256-bit key) encryption.

TABLE I
TEST SETUP CONFIGURATIONS.

	GPU	FPGA
CPU	4-core (8-thread) Intel Core i7-950 3.06 GHz, 8 MB cache	Intel Atom E665CT 1.3GHz, 512KB cache
Main mem.	12 GB DDR3-1033	1 GB DDR2-800
Accelerator hardware	GPU: NVIDIA GeForce GTX 680, PCIe 3.0 x16	FPGA: Arria II GX, PCIe 1.1 x1
Operating system	CentOS 6.3, kernel version 2.6.32-279.14.1.el6.x86_64	Fedora 14, kernel version: 2.6.38-1.MSMST.fc14.i686
Software	GCC 4.4.6, Octave 3.6.3, CUDA 5.0	GCC 4.5.1, EncFS 1.7.4, OpenSSL 1.0.0e

TABLE II
PREDICTED AND ACTUAL SPEEDUPS FOR MATRIX MULTIPLICATION.

i	Time on CPU only (s)	# of replaced calls	Replaced portion (%)	Predicted time (s)	Predicted speedup	Actual[a] time (s)	Actual[a] speedup	Relative error (%)
1	4.102	80000	27.8	12.35	0.332	13.52	0.303	9.5
2	18.15	90	96.4	0.998	18.19	0.986	18.38	-1
3	22.26	80090	83.6	13.37	1.67	14.49	1.54	8.4
4	22.26	90	78.5	5.13	4.34	5.17	4.3	0.9

a. Not including CUBLAS initialization

TABLE III
PREDICTED AND ACTUAL SPEEDUPS FOR FILE ENCRYPTION.

i	Time on CPU only (s)	# of replaced calls	Replaced portion (%)	Predicted time (s)	Predicted speedup	Actual time (s)	Actual speedup	Relative error (%)
1	0.945	787	40.1	0.808	1.169	0.831	1.137	2.8
2	0.945	1229	40.5	0.896	1.055	0.916	1.031	2.3
3	49.145	65536	42.2	37.944	1.295	39.474	1.245	4
4	49.145	73732	47.8	39.573	1.242	40.745	1.206	2.9

and the benchmarks were rerun with transparent acceleration enabled. The predicted and actual speedups for 4 considered execution scenarios are presented in Tables II and III and discussed in the following analysis.

The second column of both tables shows the total execution time without acceleration. The next column shows the number of times the kernel to be accelerated was called during each scenario. The fourth column contains the percentage of the total execution time that was taken by the execution of the non-accelerated kernels. The next columns contain the predicted and actual execution time, as well as the corresponding speedup values with the acceleration enabled. Finally, the last column shows the relative error of the predicted speedup.

The results suggest that the predicted values are consistent with the actual performance gains. However, the prediction accuracy slightly drops with the increase of the number of kernel calls (see rows 1 and 3 in Table II, as well as rows 3 and 4 in Table III). This deviation is mainly caused by the fact that small inaccuracies that may be introduced in the modeled execution times accumulate over multiple calls (see eq. (3)) and their effect on the resulting total may become more noticeable.

In the presented results, the predicted speedups are generally slightly higher than the measured values. This suggests that the overheads involved in the transparent acceleration should also be considered during the acquisition of the kernel performance models. In fact, although usually negligible, the cost of the additional call indirection may also become noticeable, especially if many short calls are considered.

B. Automatic Estimation of Transferred Data Size

The pointer inspection methods for the automatic estimation of the transferred data size were evaluated by comparing the results they produce to the expected values computed using user-supplied expressions. In the following discussion, three applications are considered, namely: 1) matrix multiplication using BLAS, 2) data encryption using OpenSSL, and 3) Discrete Cosine Transform (DCT) using the libjpeg library by the Independent JPEG Group. Each of them uses a different scheme for passing the data to the library functions. The matrix multiplication functions employ the Fortran calling convention, thus even scalar values are passed by pointers. In the data encryption application, the data are passed using a pointer to the buffer and an integer value for the buffer length, plus a pointer to the context structure containing the encryption key and additional temporary buffers. The last application uses the most elaborate scheme of the three: the size of the data is computed based on the information in the context structure and the directly passed arguments. The actual data is passed using an array of pointers to consecutive elements, thus it may not be stored in a contiguous buffer. By analyzing such diverse set of schemes, the applicability of the methods can be projected to a broader spectrum of possible scenarios.

The obtained results are presented in the form of scatter plots, where each function call is represented by two points, representing the input and the output size. The x coordinate represents the expected work size value, whereas the y coordinate represents the automatic estimate. The closer these points are to the identity line, the better the estimation.

The obtained results are depicted in Figures 5, 6 and 7. As it can be seen, the accuracy of the results differs greatly between the *fast* and the *full* estimation methods.

Three distinct regions can be identified in the graphs on the right hand side, which correspond to the *full* method. For large data sizes, starting from a few kilobytes, the accuracy is very high. This consistence between the estimated model and the measured values is particularly important when large data sizes are involved, since the imposed communication overhead is more significant in this cases and affects more severely the resulting global speedup. In the middle region (for tens or hundreds of bytes), some overestimation becomes noticeable, comparing to the absolute values. The main reason for this deviation arises from the the scalar values on the stack being passed by pointers, making the estimation algorithm treat all recent stack activity as data passed to the called function. The current implementation is unable to detect stack frame boundaries (frame pointers are eliminated in optimized code), so the entire stack must be treated as a single region. Finally, for very small data sizes, the visible overestimation results from the fact that not all scalar arguments were considered in the user-supplied formulas. These arguments usually correspond to control and status information transferred aside of the actual data, and their real impact on the performance depends on the particular implementation.

An interesting anomaly can be observed in Figure 7. For larger data sizes, the estimated values are sometimes significantly lower than expected. The reason for this deviation is that the performed tests include rescaling static images, which results in certain pixel information being repeated in consecutive scanlines for large zoom factors. The unchanged buffer contents are not considered as new data to be transferred, hence the underestimation. Detecting such situations can help in the optimization of the communication between the CPU and the accelerator in cases where the gain from the suppression of unnecessary transfers overcomes the cost of checking the data for duplicates.

In general, the estimation error does not vary greatly with the actual size of the data, and oscillates around a few hundred bytes above the expected value over the entire examined range. The root-mean-square error of the obtained estimates in relation to the expected values was 284 bytes for BLAS, 335 bytes for OpenSSL and 482 bytes for libjpeg. Hence, this method is particularly useful for estimating data sizes starting from a few kilobytes, where such differences typically cause a relatively small change in the data transfer and kernel execution time.

At a first glance, the results obtained using the *fast* method (left hand side graphs) contrast with the accuracy obtained for the *full* method and do not seem useful in terms of work size estimation. However, this method can still be used to reveal the memory usage patterns of applications and compare them to the actual needs. For example, if an application frequently allocates significantly more memory than it actually needs, as is the case for the Octave example (see Figure 5), revising the allocation scheme could greatly reduce the overall memory demand of the application. Furthermore, a potential situation where the allocated buffer is too small can also be easily identified, thus helping to solve a serious security problem.

C. Overheads

Shared library interposition obviously causes an increase in the program execution time and in its memory footprint, since the additional code is loaded into the process memory and executed. The resulting overheads differ according to the purpose and implementation of each particular wrapper library. The overheads for a set of wrapper libraries which implement the features described earlier were measured to evaluate their impact on the execution of the profiled applications.

The overheads, measured using the test setup that was denoted in Table I as *GPU*, are summarized in Table IV. The presented execution time overhead resulting from the wrapper library activity is normalized to one function call. The constant memory overhead parcel includes the code and data structures which reside in memory regardless of any wrapper library activity. Additional memory may still be allocated during the execution for any bookkeeping structures used by a given wrapper library.

The “Trace” wrapper library writes the function name and the arguments to the trace file upon each call, which makes its impact noticeable, but not significant. The “Trace + fast” wrapper library additionally performs the data transfer size estimation using the faster variant of the automatic method presented earlier. Finally, the “Trace + full” wrapper maintains complete copies of all data that the traced program stores on the heap and on the stack, effectively doubling the requirements of the process for the allocated memory. At this respect, it is important to note that those automatic estimation methods

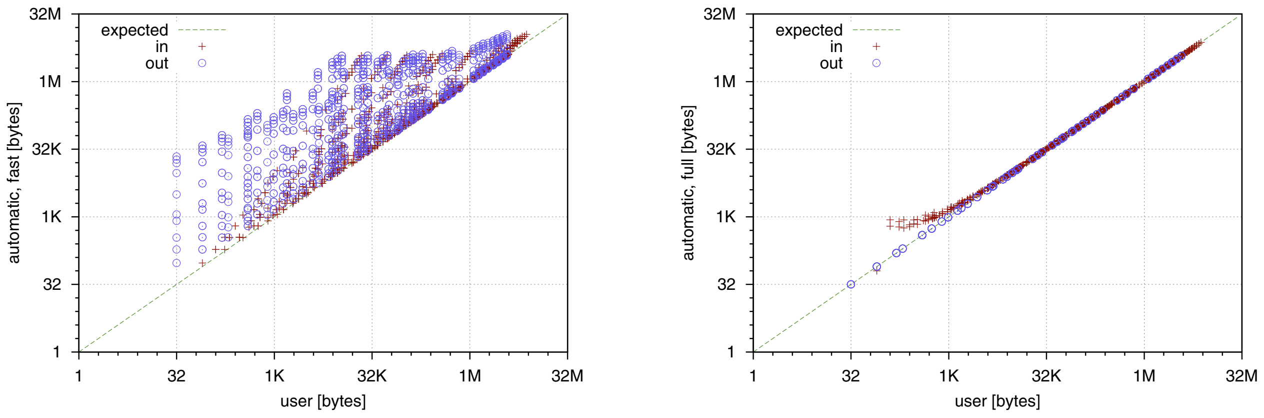


Figure 5. Data transfer size estimates obtained by memory inspection compared to user-defined estimates; application: Octave, library: BLAS, function: dgemm; 1000 points

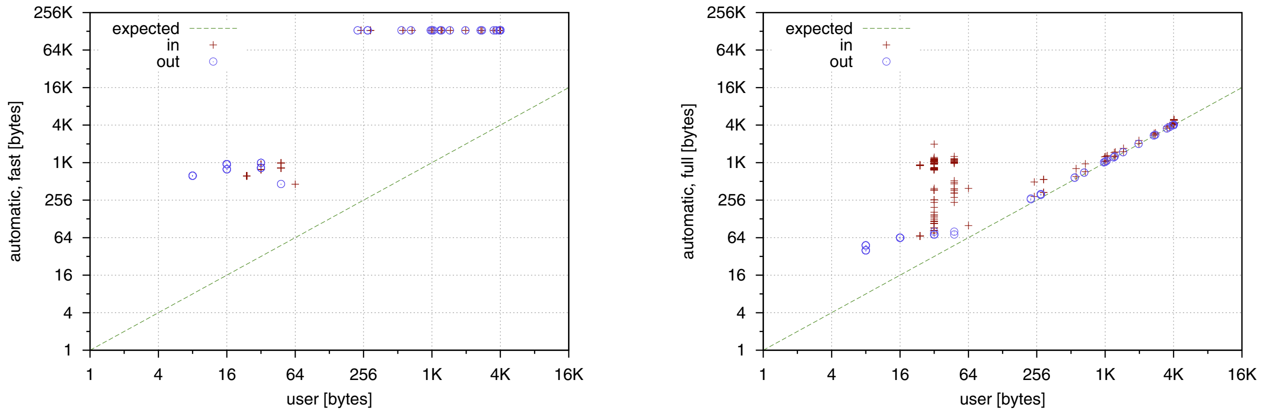


Figure 6. Data transfer size estimates obtained by memory inspection compared to user-defined estimates. 1188 points. Application: EncFS, library: OpenSSL, functions: EVP_EncryptUpdate, EVP_DecryptUpdate.

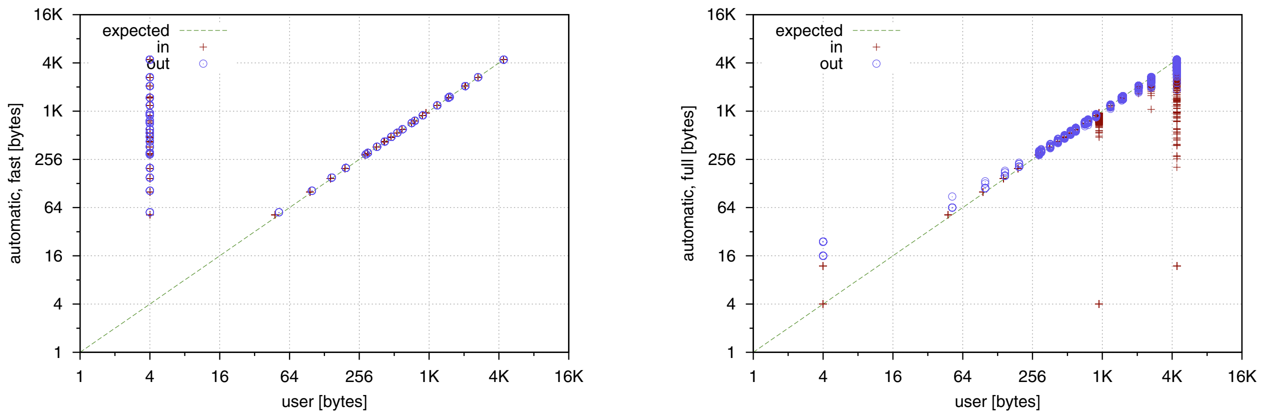


Figure 7. Data transfer size estimates obtained by memory inspection compared to user-defined estimates. 29500 points. Application: convert (ImageMagick), library: jpeglib, functions: jpeg_read_scanlines, jpeg_write_scanlines.

are not intended for execution in production systems, and thus the imposed overheads are only important in the context of the accelerator development process.

VI. Conclusions and Future Work

A novel approach for automatic performance prediction in heterogeneous systems was presented. The proposed framework allows a transparent and automatic estimation of the work size and its relation to the execution time of the library functions during application execution. It then combines the collected execution traces with the accelerator performance models in order to predict the overall speedup that can be obtained for the considered application with a given accelerated kernel implementation.

TABLE IV
OVERHEADS IMPOSED BY THE DIFFERENT WRAPPER LIBRARIES.

Wrapper library	Mean time ov. per call (μs)	Const. memory overhead (kB)	Extra mem ov. per region (B)
Trace	8.2	3884	-
Trace + fast	9.3	5632	24
Trace + full	12.8 + 0.29/KB[b]	8208	24 + region size

a. i.e. $0.29\mu s$ per each kilobyte of inspected memory

The proposed prediction methods have been evaluated for a set of existing applications without introducing any modifications to them. The obtained experimental results showed that accurate work size estimation can be successfully performed for diverse library interfaces. The estimated work size figures are consistent with the actual values for all examined applications within the range from hundreds of bytes up to tens of megabytes. The estimation error does not vary significantly with the absolute work size. The root-mean-square error of the estimated values was measured between 284 and 482 bytes for the whole examined range.

Using the correctly estimated work sizes, the proposed global speedup prediction method demonstrated the ability to produce accurate results, with the relative error not exceeding 10%, and in most cases ranging between 1 and 4%.

Possible future improvements may concern the accuracy of the automatic data transfer size estimation methods, as well as the overheads they impose. Another interesting future direction is the investigation of kernels which operate on data structures that are more complex than arrays or matrices. Reliable identification and quantification of structures such as linked lists, trees or graphs without a priori knowledge is not trivial, but may be feasible if certain patterns are properly identified in a large set of samples.

REFERENCES

[1] K. O. W. Group *et al.*, "The OpenCL specification," *A. Munshi, Ed*, 2008.

[2] "An updated set of basic linear algebra subprograms (BLAS)," *ACM Trans. Math. Softw.*, vol. 28, no. 2, pp. 135–151, Jun. 2002.

[3] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide*, 3rd ed. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1999.

[4] M. Galassi, J. Davies, J. Theiler, B. Gough, G. Jungman, P. Alken, M. Booth, and F. Rossi, *GNU Scientific Library Reference Manual*, 3rd ed., B. Gough, Ed. Network Theory Ltd., 2009.

[5] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005, special issue on "Program Generation, Optimization, and Platform Adaptation".

[6] G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, 2000.

[7] L. Ibanez, W. Schroeder, L. Ng, and J. Cates, *The ITK Software Guide*, 2nd ed., Kitware, Inc. ISBN 1-930934-15-7, 2005.

[8] T. Beisel, M. Niekamp, and C. Plessl, "Using shared library interposing for transparent acceleration in systems with heterogeneous hardware accelerators," in *Proc. IEEE Int. Conf. on Application-Specific Systems, Architectures, and Processors (ASAP).* IEEE Computer Society, Jul. 2010.

[9] J. Wernsing and G. Stitt, "A scalable performance prediction heuristic for implementation planning on heterogeneous systems," in *Embedded Systems for Real-Time Multimedia (ESTIMedia), 2010 8th IEEE Workshop on*, oct. 2010, pp. 71 –80.

[10] K. Sato, K. Komatsu, H. Takizawa, and H. Kobayashi, "A history-based performance prediction model with profile data classification for automatic task allocation in heterogeneous computing systems," in *Parallel and Distributed Processing with Applications (ISPA), 2011 IEEE 9th International Symposium on*, may 2011, pp. 135 –142.

[11] J. Olsa, "latrace(1)," Linux man page.

[12] R. Branco, "Ltrace internals," in *Ottawa Linux Symposium*, 2007.

[13] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal, "Dynamic instrumentation of production systems," in *Usenix '04 Annual Technical Conference*, 2004.

[14] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," *SIGPLAN Not.*, vol. 40, no. 6, pp. 190–200, Jun. 2005.

[15] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," *SIGPLAN Not.*, vol. 42, no. 6, pp. 89–100, Jun. 2007.

[16] T. W. Curry, "Profiling and tracing dynamic library usage via interposition," in *USENIX Summer 1994 Technical Conference*, 1994.

[17] C. Fetzer and Z. Xiao, "A flexible generator architecture for improving software dependability," in *Software Reliability Engineering, 2002. ISSRE 2003. Proceedings. 13th International Symposium on*, 2002, pp. 102 – 113.

[18] "clang: a C language family frontend for LLVM." [Online]. Available: http://clang.llvm.org/

[19] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci, "A portable programming interface for performance evaluation on modern processors," *International Journal of High Performance Computing Applications*, vol. 14, no. 3, pp. 189–204, Fall 2000.

[20] J. W. Eaton, D. Bateman, and S. Hauberg, *GNU Octave Manual Version 3*. Network Theory Limited, 2008.

[21] "Basic Linear Algebra Subprograms Technical Forum Standard," *International Journal of High Performance Applications and Supercomputing*, vol. 16, no. 1, pp. 1–111, 2002.

[22] *CUBLAS Library User Guide*, NVIDIA, Oct. 2012.

[23] "EncFS – Encrypted FileSystem." [Online]. Available: http://www.arg0.net/encfs

[24] J. Viega, P. Chandra, and M. Messier, *Network Security with OpenSSL*, 1st ed. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2002.

[25] A. Matoga, R. Chaves, P. Tomás, and N. Roma, "Accelerating user-space applications with FPGA cores: profiling and evaluation of the PCIe interface," in *Proceedings of the 9th Portuguese Meeting on Reconfigurable Systems*, 2013.