Using Shared Library Interposing for Transparent Application Acceleration in Systems with Heterogeneous Hardware Accelerators

Tobias Beisel, Manuel Niekamp and Christian Plessl
Paderborn Center for Parallel Computing, University of Paderborn
{tbeisel|niekma|christian.plessl}@uni-paderborn.de

***Abstract*—Todays computer systems increasingly comprise heterogenous computing elements like multi-core processors, graphics processing units, and specialized co-processors, which allow parallel processing. Programming applications to utilize such systems is a complex process and needs good knowledge about the hardware architecture. Automatic and transparent use of these resources is a major concern of domain specific software developers and users. We present a new approach of using shared library interposing to replace libraries in binary applications with highly optimized accelerated versions. A plugin-based framework was developed, which allows interposing shared library calls, delegating them to accelerator specific libraries and adapting them to the library specific interface. Accelerator specific plugins can be added with a high degree of automatism. First steps were taken to develop a fast and intelligent selection component, choosing the best possible accelerator for a shared library call. It was shown, that such a framework may be efficiently used to apply shared library interposing to transparently speedup existing applications. The BLAS library for linear algebra was used as an example to develop plugins for an acceleratable library. Runtimes of BLAS functions were measured on different architectures and expose significant differences depending on the used implementation and hardware, showing the potentially high speedups of the approach.**

I. Introduction

Current computing systems experience a trend towards increasingly heterogeneous architectures. While today even standard configurations of personal computers provide a combination of multi-core CPUs and general-purpose graphics processing units (GPUs), there is also a trend in compute centers to provide diversification within their systems. Experimental cluster systems use multi-core processors, GPUs, and specialized co-processors, such as ClearSpeed CSX or FPGAs, to accelerate scientific applications. Using a combination of several accelerators within a single workstation is a powerful and cost-effective way to solve medium sized scientific computing problems without accessing large and expensive computing systems. This not only provides exclusive access to computing power to nearly every scientific and industrial institution, but also allows to customize the architecture to the targeted application domain.

However, a significant barrier in terms of usability of such systems with heterogeneous hardware accelerators is the inevitable knowledge about the used hard- and software components, which is required to develop efficient solutions. The required knowledge includes the specific characteristics of the hardware architecture, such as the architecture of the compute elements, the bandwidth of communication connections, or the size and architecture of different memory layers. Further it is required to have detailed knowledge in implementing the interaction between the host processor and the accelerators, managing the memory efficiently, and partitioning the workload to match the smallest compute entities of the used hardware using several different programming languages. In addition, scientific users frequently do not only program and run their own custom applications, but use proprietary domain-specific software or simulation frameworks like MATLAB for which no source code is available. This prevents modifying the applications to target different accelerators. As a result, missing hardware and programming knowledge and the unavailability of source code keep many scientific users outside the field of computer science from exploiting the possible acceleration of heterogenous computer systems.

In this paper, we present a novel approach to make the performance of heterogeneous computers transparently available without requiring any programming from the user's side by using a technique called *shared library interposing*. Shared library interposing provides a mechanism for modifying existing applications without requiring access to the original source code by intercepting and replacing calls to shared library functions with different functions. For leveraging shared library interposing for transparent acceleration we exploit the fact that many scientific codes make extensive re-use of highly optimized libraries for computationally intensive operations. Examples of frequently used libraries are the BLAS library for linear algebra [1], the GNU scientific library GSL for numerical calculations [2], or the Vector Signal Processing Library VSIPL for embedded signal processing platforms [3]. These numeric libraries are typically installed as shared libraries, which are dynamically loaded by the applications at runtime. One approach for using a particular hardware accelerator without the need to recompile the application is to replace the shared library with a compatible library that implements the same interface but executes the code on the hardware accelerator. For example, [4] uses an OpenGL based API for automatic acceleration of graphics applications on FPGAs as a static replacement for the original OpenGL. This simplistic approach has however a number of drawbacks, e. g.,

Published in: *Proc. Int. Conf. on Application-Specific Systems, Architectures, and Processors (ASAP)*
 DOI: 10.1109/ASAP.2010.5540798

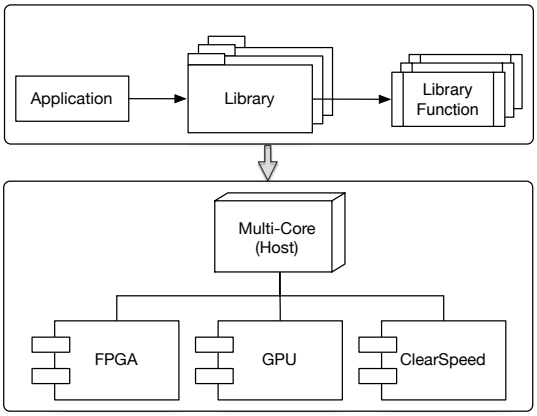

Figure 1: General scenario: library functions are mapped transparently to hardware accelerators in a heterogeneous system.

1) replacing a library at the file system affects all applications which may expect a specific version of library, 2) a purely static mapping of all library functions to a specific accelerator is introduced, and 3) no selective functions or fine-granular binary extracts can be replaced.

In contrast, our approach presented in this paper leverages shared library interposing which does not require replacing libraries at the file system level and allows for a fully flexible and even dynamic mapping of library functions to hardware accelerators. The flexibility allows us to exploit the fact that the optimal accelerator for a particular library function depends not only on the function, but also on other factors, such as the size of the input data.

Specifically, our paper makes the following contributions:

- we introduce *shared library interposing* as a *mechanism* for transparent application acceleration
- we present an extensible framework that allows for implementing different *policies* for delegating library functions to hardware accelerators at runtime
- we present a case study, which allows us to evaluate the benefits and the overheads of our approach with the example of the BLAS library for linear algebra

The remainder of this paper is structured as follows. In Section II we present an overview of related work in transparent acceleration and shared library interposing. In Section III we present an overview of different approaches for shared library interposing and discuss their suitability for the purpose of transparent application acceleration. In Section IV we present our framework that implements one specific interposing mechanism and can be customized with different policies to select hardware accelerators at runtime. Finally, we evaluate the benefits and overheads of our framework in Section V and present conclusions and further work in Section VI.

II. Related Work

Using library interposing to map shared library functions to heterogeneous accelerators is a new approach and has hardly been investigated. Binary instrumentation and library interposing have been used for other intentions though. The general focus is on profiling applications to collect runtime statistics and workload profiles. One example of how instrumentation can be used to attach to running process and redirect execution is shown in [5]. Control is redirected to a user-space emulator to dynamically instrument the binary to analyze taints in a safe mode execution. In our work we concentrate on library interposing to intercept library calls. This has been done in several previous works. Curry [6] proposed an approach of profiling and tracing dynamic library usage using library interposing. Library functions are resolved to special wrapper functions at run-time that collect statistics before and after calling the real library. The application and library remain unaltered. This allows to trace multiple processes and analyze applications to get useful performance data. On GNU/Linux systems the LD_PRELOAD environment variable stores a list of libraries to be loaded before all others and is used to delegate the library calls to specific libraries in this work. [7] presents a library interposing approach to generate audit data without recompiling the application. The ability to detect and prevent unsafe programming practices, race conditions and buffer overflow attacks was shown by surrounding original function with analysis code using LD_PRELOAD. González et al. present a tracing mechanism based on dynamic code interposition [8]. They add monitoring code to unmodified parallel binaries and shared libraries at run-time to produce input traces to feed an analysis tool.

A very interesting approach for seamlessly using heterogeneous hardware systems is presented by Moore et. al. [9]. The VSIPL++ for Reconfigurable Computing (VForce) library allows VSIPL++ applications to use FPGAs and GPUs as co-processors by defining an interface that decouples the software from the hardware development. The approach mainly differs from our approach in the need to modify the existing code by including a VForce header and thus being able to access the original code and the lack of a runtime selection support for the best performing function implementation. These restrictions are also true for the approach provided in [10]. IGOL (Imaging and Graphics Operator Libraries) presents a compile- and run-time framework for reconfigurable data processing applications. The main goal of IGOL is to simplify the software-hardware interface of reconfigurable systems by providing abstraction layers. IGOL is based on the Microsoft Component Object Model (COM) and is limited to the use of few FPGA based platforms.

Our work emphasizes the use of library interposing for dynamic accelerator utilization. This approach differs from the works stated before in the main goal of using library interposing to transparently accelerate existing applications on heterogeneous systems. A new approach is shown to facilitate the use of such systems and is demonstrated with a case study

for the BLAS library.

To the authors' best knowledge only ClearSpeed uses a library interposing mechanism to intercept the use of the standard BLAS library in favor of an accelerated one. LD_PRELOAD is used to preload the CSXL library [11], which uses CSX BLAS code replacements to accelerate specific BLAS functions with the ClearSpeed floating-point accelerator card. The host BLAS library is used as a fallback for unsupported function calls. Currently only one BLAS function (DGEMM) and two LAPACK functions are supported within CSXL. CSXL offers an additional environment variable to set the percentage of host BLAS usage. Also, only input sizes above a certain threshold will be computed on the ClearSpeed board.

AMD Core Math Library for Graphic Processors (ACML-GPU) [12] provides an accelerated version of ACML for GPUs. It provides a dynamic decision whether or not to offload functions to the GPU based on the parameters received from the calling function (the problem size) or statically uses an environment variable. This smooth migration is only supported for two functions (SGEMM and DGEMM) and limited to GPUs supported by the ATI Stream SDK.

We use the same mechanism, but present a more general infrastructure offering several accelerator technologies and arbitrary policies to select an adequate accelerator.

III. Shared Library Interposing

In Linux systems the *dlopen()* system call [13] implements the interface to the dynamic linking loader. By either getting passed a library name or a path to a library, *dlopen()* loads a shared library and returns a handle to it. If an absolute path is given (begins with "/") *dlopen()* refers to exactly that path and returns an error, if the shared library is not found. If no absolute path to a library name is given, the requested library is searched at the search path specified in the LD_LIBRARY_PATH environment variable. If the library cannot be found there, the system's default library folders are searched (/lib, /lib64, etc.). The handle returned by *dlopen()* can subsequently be used to load a symbol from the library into memory. This is done by *dlsym()*, which uses the handle and a symbol name to return the address of the memory location to which the symbol is loaded.

In the remainder of this section, we introduce and evaluate different approaches allowing to substitute specific shared library calls within binaries, that base on the following ideas:

- Customize the behavior of the default dynamic linker with environment variables (Sec. III-A)
- Use a different, custom dynamic linker (Sec. III-B)
- Statically patch application binaries to use different library calls (Sec. III-C)
- Dynamically modify the code of applications after they have been loaded to memory (Sec. III-D).

A. Configure dynamic linker

The simplest method to intercept library calls in GNU/Linux systems is to point the LD_PRELOAD environment variable to a list of substituting shared libraries. LD_PRELOAD affects the runtime linker and defines one or several shared libraries which are loaded and searched for matching symbols first and thus overload the original functions. This capability allows for selectively replacing symbols to be favored over the regular symbols resolved with *dlsym()*.

This method can be used to exchange a standard library with an accelerator-specific parallel library. The parameters passed to the called shared library function have to be adapted to the interface of the replacing library. This will be discussed later.To avoid the interception of library calls in every application the LD_PRELOAD environment variable is typically not defined globally, but is selectively set in the environment of the target application only, e.g., using a wrapper shell-script.

The main advantage of the LD_PRELOAD method is that it is very convenient to use, as it is supported by the standard dynamic linker of the operating system and no dependencies to other tools are introduced. However, problems may arise when the process environment is not completely controlled by the user, for example if the application itself configures environment variables affecting the linker.

B. Exchanging the dynamic linker

Another approach to intercept the library calls is to redirect them using a different dynamic linker. The ELF binary format, which today is used on most Unix-derived systems, explicitly supports to specify which dynamic linker is to be used to resolve dynamic linking requests. To this end, the PT_INTERP ELF header entry points to the executable of the dynamic linker. On Linux systems this field generally points to the default dynamic loader (/lib/ld-linux.so.2).

We can use this custom dynamic linker mechanism for our purposes of shared library interposing by modifying this header to point to our own, custom dynamic linker. If the source code of the program is available, we can pass the option *–Wl –dynamic-linker=/path/to/our/linker* to the GNU linker *ld* when linking the final executable. Without access to the source code, we can use the *readelf* tool to locate the PT_INTERP header and replace the linker name in the application's binary. The drawback of this method is that it requires to patch the application's binary, which is not always desirable. Also, implementing a custom dynamic linker that can act as a full replacement for the standard dynamic linker is a non-trivial endeavor, requiring substantial knowledge about binary formats and linking.

An application of this technique is described in the work of Reiser [14] on *rtldi*, which is an alternative dynamic loader that enables each application to use its own runtime environment with different library search paths.

C. Static binary infection

The previously introduced methods use different ways of intercepting and subsequently delegating library calls of existing applications. A different approach is to modify the binary itself, thus not only replacing the dynamic loader, but replacing the library call itself within the binary.

The techniques required for this binary modification depends strongly on the operating system and the binary format. One particular approach that can be used for this purpose is named *ELF PLT infection* and is applicable to binaries and shared libraries in the ELF object format, which is used for example by Linux. ELF shared libraries use position independent code and can thus be loaded at runtime to any address without modification. Dynamic linking is done by the ELF linker using essentially two process-specific tables, the Procedure Link Table (PLT) and the Global Offset Table (GOT). The GOT stores the static absolute offsets of the position independent symbols, such as functions and global variables, in the position independent shared library. When calling a function in the shared library, the program is not calling the addresses specified in the GOT directly, but calls a corresponding symbol in the PLT, providing the address of the corresponding GOT entry. This level of indirection allows to postpone linking to runtime and to use the same memory image for a shared library from several applications.

To delegate a call to a different shared library function, the PLT entry of a called symbol can be modified to point to the address of the replacing shared library function directly. By modifying the program entry point in the application binary, it is possible to wrap the actual application execution with code, that loads the needed library and maps it to the address space of the application. Cesare [15] presents a detailed discussion of how PLT infection can be used to redirect library calls.

ELF PLT infection is a powerful method of implementing library interposing. As no environment variable is used, the interposing process is completely hidden from the application and does not influence other processes. However, controlling static instrumentation of ELF binaries at this low level is error prone and requires in-depth knowledge about the binary format and the interceptable calls to avoid possible damage. This makes ELF infection hard to use for software engineers. An alternative to this static binary modification approach is to alter the loaded binary image of the application at runtime. This can be achieved for example with the powerful (*Pin*) tool for binary analysis and binary instrumentation, which will be discussed in the next subsection.

D. Binary instrumentation with Pin

Pin [16] is a binary instrumentation tool. It provides a framework for writing customized program analysis tools (*Pintools*). Pin can be thought of as a compiler that recompiles executables with new code segments inserted to the binaries to do fine-grained analysis. Pintools are plugins modifying the code generation within Pin. Instrumentation within Pin is composed of a mechanism to decide which code to insert at which point (*instrumentation*) and the *analysis* code itself.

Dynamic instrumentation in Pin can be done in two ways: using the Just-In-Time (JIT)- or the Probe-Mode. The Probe-Mode replaces instructions in the original program with so called trampolines, that branch to instrumentation code. Thus, the application can be executed natively and no slow down arises. This method can be used to exchange methods or calls to shared libraries. The signature of the replacing function must match the original function. JIT uses a Just-in-Time compiler to dynamically insert instrumentation code into the application binary at compile time. JIT provides access to low-level functions and enables modifications at the assembler level, however executing applications in JIT mode is about one order of magnitude slower as in probe mode. While this execution overhead excludes JIT mode from being used for library interposing, JIT mode has still proven to be very useful for analyzing applications.

The advantage of using Pin for the purpose of library interposing is its modular design and the availability of high-level functions for binary analysis and modifications. This allows to conveniently performing binary instrumentation at runtime without requiring low-level knowledge on dynamic linking, binary formats, or calling conventions. A major drawback in the current implementation of Pin is that Pintools are not allowed to link against libpthread or use pthread functions, limiting the code inserted to the binary to single-threaded code.

E. Summary

Summarizing the approaches, all methods come with certain drawbacks of different importance. With the goal of making library interposing available to many developers, complexity has to be kept low and the approach has to work for most possible use cases. This is currently only given using LD_PRELOAD. Pin would be a reasonable alternative, but suffers from the incompatibility with libpthread. Table I gives an overview of the different approaches and their characteristics.

Table I: Library Interposing Approaches

	Using linker	Exch. linker	ELF infect.	Pin
Static/dynamic	dyn.	dyn.	stat.	dyn.
Environment var.	yes	no	no	no
Binary modified	no	yes	yes	yes
Complexity	low	med.	high	med.

IV. A Framework for Transparent Application Acceleration Using Shared Library Interposing

In this section we describe our framework for transparent application acceleration which is based on shared library interposing. We introduce a plugin-based library approach that separates the concerns of implementing accelerator-specific functions from the mechanism of library interposing and from the policy for selecting a particular function implementation at runtime. As a case study, we have chosen the widely used BLAS library for linear algebra, for which optimized implementations for CPUs and hardware accelerators exist.

A. Accelerating BLAS Libraries

The original specification of the BLAS library was published in 1979 [17] and was made available as an open-source Fortran implementation on netlib.org [1]. This standard was extended several times, the last time in 2002. Many optimized implementations of the BLAS library have been released, the

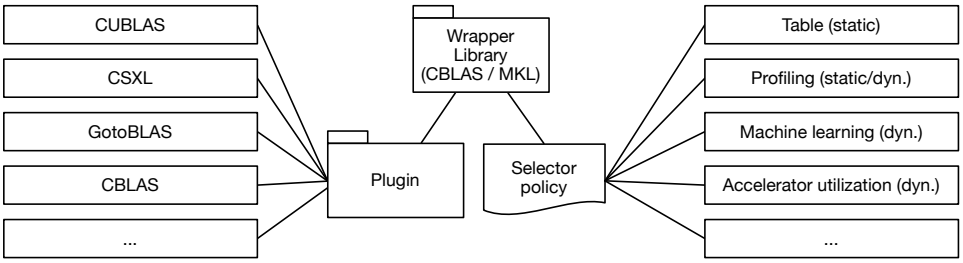

Figure 2: Library interposing framework using a wrapper library for intelligent dynamic linking of accelerator libraries.

most widely known being CBLAS (Appendix B of [18]), Intel Math Kernel Library (MKL) [19], AMD Core Math Library (ACML) [12], Automatically Tuned Linear Algebra Software (ATLAS) [20] and GotoBlas [21]. Additionally, there exist implementations for specific hardware accelerators, such as CUBLAS [22] for GPUs. All of these implementations date back to the original netlib BLAS implementation. While the functions of the implementations are basically the same, there exist certain variations between different implementations, leading to incompatibilities in parameter types or argument order for certain functions. In addition some BLAS libraries only offer a subset of the original BLAS interface, which is particularly the case for accelerator-specific BLAS libraries.

By tracing the calls to BLAS functions of an application during runtime, we can study the potential for acceleration by redirecting BLAS calls to optimized libraries. Functions that might be accelerated by specific BLAS libraries can be searched by their names. The parameters of these functions are not easily readable from the binary, hence we need additional information about the BLAS interface which is expected by the application. We chose CBLAS as the fallback implementation for all BLAS functions that are not implemented in the library for a particular hardware accelerator, as it is the the most widely used, complete reference implementation of the original netlib specification and is freely available. As many BLAS libraries (e. g., MATLAB, CSXL, CUBLAS) use the MKL BLAS library as reference implementation, we also support the MKL BLAS interface in our framework.

B. Using Shared Library Interposing

Using one of the shared library interposing approaches for transparent acceleration requires to adapt the technique to the additional needs of a heterogeneous system which delegates the call to optimized libraries targeting hardware accelerators or the host CPUs. A corresponding shared library interposing solution needs to satisfy the following requirements: It

- initializes each accelerator automatically
- performs automatic name translation to matching functions
- adjusts the original call to the library-specific demands
- chooses the best possible hardware accelerator for a specific call.

Figure 2 depicts how our framework implements these requirements. To support a broad range of available accelerators, we have implemented a plugin-based wrapper library, which allows us to insert additional support for supplemental hardware easily. It delegates the applications library calls to the hardware accelerators. The plugins implement the gateway from the wrapper library to the accelerator-specific libraries. As each supported accelerator needs to be initialized in a different way, this is done by the specific plugin. Furthermore, the calls to the library are adjusted to the accelerator-specific library call by modifying the function name and the parameters. The plugin finally performs the actual library call and returns the results calculated by the accelerator libraries. The wrapper library as well as the plugin stubs are generated automatically, as described in subsequent sections. Up to now, we have built plugins for CUBLAS and ClearSpeeds CSXL, which both are based on MKL. A plugin for the ATLAS library is available as well as a GotoBlas2 (GotoBlas version 2, short goto2) library plugin, which spreads the BLAS computation to several CPU cores in an SMP machine using OpenMP. The wrapper library supports interfaces to the CBLAS library and the MKL.

Additionally, the wrapper library may use different selector policies to decide which plugin is used best for a specific call. These policies rely on static information (e.g., performance data collected during profiling of BLAS functions) or dynamic information, such as the utilization of accelerators. We will discuss the selector policy used in this work in the next section.

In the current implementation of the framework we use the method introduced in section III-A for library interposing, which uses the LD_PRELOAD mechanism to inject shared libraries. We found this method to provide sufficient performance and flexibility for our current use case.

C. Selector Policy

The selector policy is the component which decides to which plugin an incoming library call is delegated. Currently, we have implemented two policies *Library priority* and *Lookup table*, both relying on static information.

The *Library priority* approach assigns each plugin a priority value. For each intercepted library call, all plugins are searched in the order of priorities for the availability of a function matching the request. The library call is delegated to the first plugin providing a matching implementation. The lowest

priority is assigned to the non-accelerated CBLAS library, which is used as the fallback library, if no other library offers a matching function.

The *Lookup table* policy improves on the library priority method by considering not only the requested function but also a problem size metric for selecting the plugin to which a call is delegated. Additionally, caching in the form of a pre-computed lookup table (LUT) is used to perform this lookup quickly. The LUT contains $s \times f$ library function pointers, where each entry represents the best possible accelerator library function based on the problem size s and the used BLAS function f. A mapping function is used to translate the input dimension to a value s, defining regions of input data sizes. During the initialization phase the LUT is populated with the optimal mapping from functions and problem sizes to a particular plugin, which can be determined experimentally using profiling. Once initialized, this method is significantly faster than the library priority strategy since only a single table lookup is required, while the library priorities approach in the worst case requires the selector component to send a request to all available libraries whether the function is supported. The Lookup table implementation also allows for dynamically configuring and adapting the preferred library mapping by modifying the LUT contents, e. g., based on measured execution times of the plugin functions.

D. Profiling support

To compute and estimate for the amount of acceleration that can be obtained when accelerating library calls of a specific application, we have to know, how frequently any BLAS function is called by the application and how fast each library call executes on a particular accelerator. This information can be used as a decision input to the wrapper library, which currently has to be determined manually by using profilers as stated on the left hand side of Figure 3.

We use Pin (cf. Section III-D) for this profiling task. Pin provides a tool to trace function calls of binaries. Having knowledge about how BLAS function calls are named, we can use a substring search to identify the library calls. We can extract this information from the library header files. Using CBLAS as an example, every call to a routine begins with *"cblas_"*. This procedure is transferable to other BLAS libraries. The signatures of the BLAS functions have to be known and recognizable by an unambiguous substring. In addition we are using the instruction counter tool of *Pin* to evaluate how often the functions are called. Unfortunately, Pin does not allow to get accurate function timings. Hence, we use custom micro-benchmarks for accurately determining the execution time for each function for varying problem sizes. The results from these benchmarks are shown in Section V-A.

E. Automatic plugin generation using XSLT

For automating the process of generating the plugins we have developed a tool flow which is shown in Figure 3. As input specification for the tool flow we use an XML specification

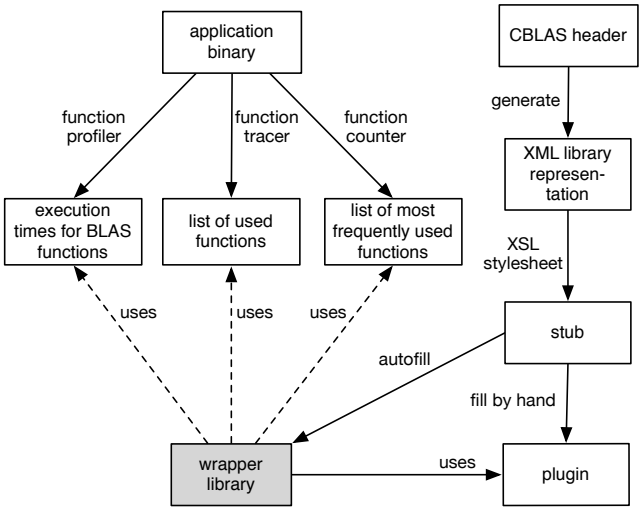

Figure 3: Work- and tool flow to construct a plugin for the interposing wrapper library.

of the CBLAS library which describes all available functions with the respective parameter names and types. This XML specification is generated directly from the CBLAS header using a Python script.

For generating the wrapper library, we automatically derive an interface supporting the most commonly used CBLAS and MKL library functions using an XSL stylesheet and a standard XSLT processor. The CBLAS interface is generated directly from the XML description. Function stubs for all BLAS functions are generated and automatically filled with the corresponding call to the selector component and the subsequent call to the pluginThe MKL interface is known as well and can be mapped from the CBLAS interface by a set of custom rules, such as mapping *call-by-reference* to *call-by-value* or changing the order of parameters. The plugin interfaces are also generated from the CBLAS XML description. Within the generated function bodies, the CBLAS interface has to be mapped to the specific library interface.

Currently filling the plugins stubs has to be performed by hand, based on the known interface of the specific accelerated BLAS library. Generally this requires simple conversion of data types, such as converting enumeration values to a set of single parameters. For CUBLAS function calls the function arguments have to be marshalled with Set/GetMatrix or Set/GetVector operations to copy the needed data to the GPU, as the library functions expect device memory pointers as parameters. CSXL by contrast automatically transfers the data to the ClearSpeed board when a library function is called. As not all CBLAS functions are supported by the accelerator libraries, the corresponding function bodies have to be deleted from the plugin interface to avoid calls to not existing functions. All of these adjustments have to be done only once for every accelerator and may be used without any changes afterwards. In addition, the specific accelerator

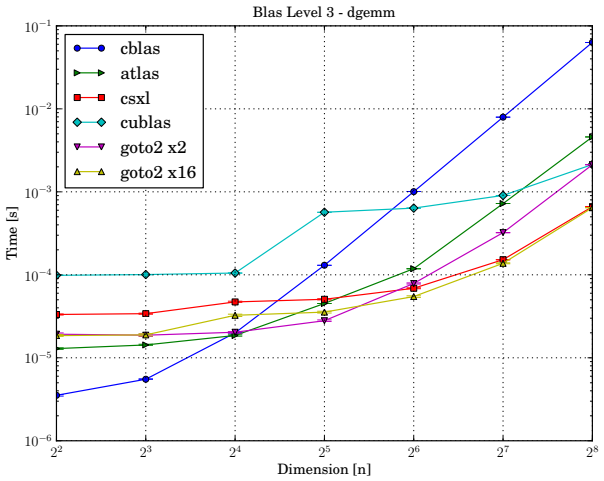

Figure 4: Runtimes of the DGEMM function using different BLAS implementations and hardware architectures.

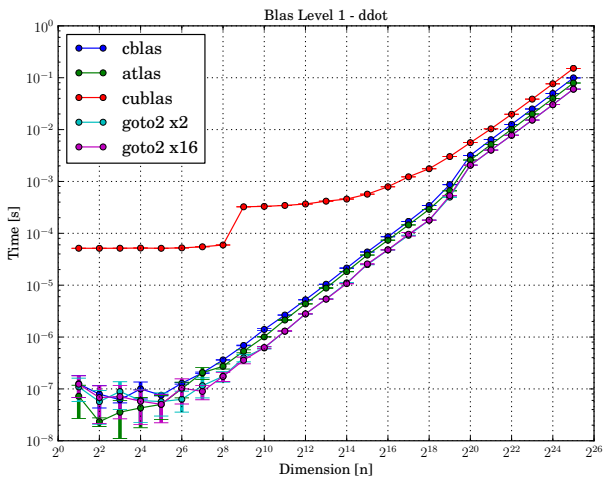

Figure 5: Runtimes of DDOT function using different BLAS implementations and hardware architectures.

initialization code has to be implemented for each plugin.

V. Experimental Validation

We have evaluated the framework using the following system setup:

- SMP system with 2-way 4-Core Intel Xeon CPU E5520 @ 2.27GHz with hyperthreading enabled (looks to the operating system like 16 cores), 12 GB main memory
- NVIDIA Geforce GTX 295, 2*240 thread-processors, 2*896 MB GDDR3
- ClearSpeed Advance e710 accelerator, 2 GB DDR2-533 SDRAM, ECC support
- CentOS 5.4, 2.6.18-164.11.1.el5 kernel version
- CUDA 3.0, OpenMP 3.0, GCC 4.4.0

A. Runtimes and Overheads

To justify the use of our proposed framework, we have first measured the runtimes of commonly used BLAS functions on different architectures. DGEMM is one of the most used BLAS functions and most promising towards an acceleration. DGEMM performs one of the matrix-matrix operations $C := \alpha\mathbf{A} \times \mathbf{B} + \beta\mathbf{C}$, where the matrices $\mathbf{A}$ and $\mathbf{B}$ can optionally be transposed. Figure 4 shows the results as a function of the matrix dimension n (problem size) for different library implementations. The runtimes include the overhead of copying data to the accelerator and back.

It is easy to observe, that the runtimes differ significantly between different implementations and architectures. The problem sizes are crucial for the decision which BLAS library is most efficient for a library call. For very small n the unaccelerated CBLAS is fastest, from 2^4 it is the goto2 library (using 2 cores) and with increasing n, CSXL and the goto2 library (running on 16 cores) are the fastest. The CUBLAS library is inefficient for small n, performs better on larger n and is best at $n = 2^{12}$, which is the largest input size that fits into the device's memory. These observations motivate our approach: Depending on function, input data sizes and additional constraints, the framework may choose to delegate the call to the optimal hardware architecture.

Table II: Library interposing overheads

Step	Count	Library priority	Lookup table
LUT init	1	0 ms	586.49 ms
Wrapper library init	1	9.68 s	9.71 s
Cublas plugin init	1	2.01 s	2.01 s
Goto2 plugin init	1	7.64 s	7.64 s
Normal call	n	2.93 ns	4.73 ns
Call through lib	n	7079.66 ns	37.43 ns
Select priority based	n	4148.72 ns	9.96 ns

Other BLAS functions (e.g., *DDOT*, the dot product of two vectors, see Figure 5) not only show a different, but also a more constant order of the library runtimes. In contrast to DGEMM, a static decision table saving only the library priority is sufficient to decide which library to choose. Thus, different approaches are needed as described in section IV-C and implemented in the framework.

To verify whether the necessary overheads arising from the framework use are worth the time, we have to contrast it with the gained speedups. We distinguish between necessary overheads of the framework setup (creation of selection policies), framework initialization (library and LUT initialization) and runtime (library interposing and selection policy).

When implementing and studying new delegation policies for the selector component, basic overheads are inevitable. Profiling the performance of library functions has to be performed to measure runtimes for different input data sizes. These are used as basis for decision making in the selector component. These overheads are one-time overheads and do not affect the runtime of the application. More important are the overheads that occur at runtime, as listed in Table II.

The one-time initialization overhead for the wrapper library includes the times for initializing all plugins and for the wrapper library itself. As can be seen from the measurements, the library initialization cost is dominated by the times for initializing the goto2 and the CUBLAS library, which we cannot affect. Still, these overheads have to be be taken into account by the selector component. If only few library calls are made, the overhead might not be worth it. Using the much faster Lookup tables for the library selection instead of static library priorities, results in an initial overhead of roughly 0.58 seconds for a tested LUT configuration 13×143 (problem size regions $\times$ number of BLAS functions). Depending on the amount of used library calls, this overhead will be amortized by the much faster library calls (roughly 7000ns faster per call) and the function based selection of the accelerator. Interposing and delegating a shared library call involves additional overheads that arise for every call. Selecting the used library makes up the major part of a dynamic library call. Interposing on binary files results in the need to transfer the input data for each call, such that data reuse is impossible.

VI. Conclusions and Future Work

In this work we have introduced our novel approach of using shared library interposing for transparently accelerating applications. We have introduced different methods for interposing library calls and have discussed their suitability for our purposes. Additionally, we have shown that the approach is practicable by a proof-of-concept implementation targeting the BLAS library. Once implemented for a set of libraries, the developed wrapper library can be used consistently to accelerate applications with just a small effort. Preparing the wrapper library for that purpose is widely automated, although some work still has to be done by hand in this stage of the prototype. This implementation is easily portable to other libraries with an initial adaptation overhead. The measured runtimes of cost-intensive BLAS functions have shown, that the one-time overhead of the accelerator initialization and the recurring call overhead pay off when using sufficiently large input data and using the framework in long running processes that make extensive use of BLAS functions.

Future work will concentrate on developing more sophisticated algorithms to select an appropriate accelerator for each library call. The current static approach may be replaced by intelligent dynamic hardware selection and scheduling. This can be achieved by using static and dynamic profiling results as decision input. A decision based on input data sizes is most likely practicable and very promising. Also machine learning approaches might be applicable as well as monitoring functions on the accelerators to take the current utilization into account. Complexity of decision algorithms is unbound, but has to be set into relation with the emerging overheads.

In addition we plan to extend the framework to support a wider variety of accelerators and libraries. Using the framework with multithreaded applications will also be investigated. Currently the initial mapping of the standard BLAS interface to the accelerator-specific library interfaces is done by hand when used for the first time. To achieve fully automated plugin generation, rules have to be identified and formalized to automatically map the calls to different libraries. Furthermore we intend to test the solution with different applications using computationally more intensive libraries to evaluate the merit as a comprehensive solution. The framework was developed in Linux, but the method is portable to other Unix derivates.

To promote the uptake of our work by other researchers and end users, we will make our framework available as open-source software at `http://github.com/pc2/liftracc`.

References

[1] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, "Basic Linear Algebra Subprograms for Fortran Usage," *ACM Trans. Math. Softw.*, vol. 5, no. 3, pp. 308–323, 1979.

[2] B. Gough, *GNU Scientific Library Reference Manual - 3rd Edition*, 2009.

[3] R. Janka, R. Judd, J. Lebak, R. M., and D. Campbell, "VSIPL: An object-based open standard API for vector, signal, and image processing," in *Proc. of Intl. Conf. on Acoustics, Speech, and Signal Processing*, vol. 2, 2001, pp. 949–952.

[4] H. Styles and W. Luk, "Customizing Graphics Applications: Techniques and Programming Interface," *Symp. on Field-Programmable Custom Computing Machines*, p. 77, 2000.

[5] G. Portokalidis and H. Bos, "Eudaemon: Involuntary and On-Demand Emulation Against Zero-Day Exploits," in *EuroSys 08*, 2008, pp. 287–299.

[6] T. Curry, "Profiling and Tracing Dynamic Library Usage Via Interposition," in *Proc. Summer USENIX'94*, 1994.

[7] B. A. Kuperman and E. Spafford, "Generation of application level audit data via library interposition," COAST Laboratory, Purdue University, West, Tech. Rep., 1999.

[8] M. Gonzalez, A. Serra, X. Martorell, J. Oliver, E. Ayguade, J. Labarta, and N. Navarro, "Applying Interposition Techniques for Performance Analysis of OpenMP Parallel Applications," *Intl. Symp. on Parallel and Distributed Processing*, p. 235, 2000.

[9] N. Moore, A. Conti, M. Leeser, and L. S. King, "Vforce: An Extensible Framework for Reconfigurable Supercomputing," *Computer*, vol. 40, pp. 39–49, 2007.

[10] D. B. Thomas and W. Luk, "Framework for Development and Distribution of Hardware Acceleration," in *Proc. of SPIE*, vol. 4867, 2002, pp. 60–70.

[11] ClearSpeed Technology, Inc, *CSXL User Guide*, 2007.

[12] AMD Core Math Library for Graphic Processors (ACML-GPU). [Online]. Available: http://developer.amd.com/GPU/ACMLGPU/Pages/default.aspx

[13] dlopen(3) - Linux man page.

[14] J. Reiser. rtldi – indirect runtime loader. [Online]. Available: http://www.bitwagon.com/rtldi/rtldi.html

[15] S. Cesare, "Shared library call redirection using ELF PLT infection," *Phrack Magazine*, vol. 10, 1999.

[16] C.-K. Luk *et al.*, "Pin: building customized program analysis tools with dynamic instrumentation," in *Proc. of the ACM SIGPLAN Conf. on Programming language design and implementation*, 2005, pp. 190–200.

[17] BLAS (Basic Linear Algebra Subprograms). [Online]. Available: http://www.netlib.org/blas/index.html

[18] S. Blackford *et al.*, "Basic Linear Algebra Subprograms Technical (BLAST) Forum Standard," *Int. J. High Perform. Comput.*, vol. 15, 2001.

[19] Intel Corporation. Intel Math Kernel Library (Intel MKL). [Online]. Available: http://software.intel.com/en-us/intel-mkl

[20] Automatically tuned linear algebra software (ATLAS). [Online]. Available: http://math-atlas.sourceforge.net/

[21] K. Goto. GotoBLAS. [Online]. Available: http://www.tacc.utexas.edu/tacc-projects/#blas

[22] NVIDIA Corporation, *CUDA CUBLAS Library*, 2008.