Automatically Harnessing Sparse Acceleration

Philip Ginsbach  
School of Informatics  
The University of Edinburgh  
United Kingdom  
philip.ginsbach@ed.ac.uk

Bruce Collie  
School of Informatics  
The University of Edinburgh  
United Kingdom  
bruce.collie@ed.ac.uk

Michael F. P. O’Boyle  
School of Informatics  
The University of Edinburgh  
United Kingdom  
mob@ed.ac.uk

Abstract

Sparse linear algebra is central to many scientific programs, yet compilers fail to optimize it well. High-performance libraries are available, but adoption costs are significant. Moreover, libraries tie programs into vendor-specific software and hardware ecosystems, creating non-portable code.

In this paper, we develop a new approach based on our specification Language for implementers of Linear Algebra Computations (LiLAC). Rather than requiring the application developer to (re)write every program for a given library, the burden is shifted to a one-off description by the library implementer. The LiLAC-enabled compiler uses this to insert appropriate library routines without source code changes.

LiLAC provides automatic data marshaling, maintaining state between calls and minimizing data transfers. Appropriate places for library insertion are detected in compiler intermediate representation, independent of source languages.

We evaluated on large-scale scientific applications written in FORTRAN; standard C/C++ and FORTRAN benchmarks; and C++ graph analytics kernels. Across heterogeneous platforms, applications and data sets we show speedups of 1.1× to over 10× without user intervention.


Keywords  sparse linear algebra, domain specific languages, library integration, declarative languages, data marshalling

1 Introduction

Linear algebra is an important component of many applications and a prime candidate for hardware acceleration. While there has been significant compiler effort in accelerating dense algebra [23, 36, 40], there has been less success with sparse codes. This is largely due to indirect memory access, which challenges compiler analysis [32]. Sparse-based algorithms are, however, increasingly important as the basis of graph algorithms and data analytics [28].

We currently see the wide-scale provision of fast sparse libraries [2, 3, 5, 55]. They deliver excellent performance, but require significant programmer intervention and are rarely portable across platforms. Alternatives, such as the SLinGen/LGen system [45, 46], provide specialized code generators for linear algebra, but again require code modification by the programmer and focus only on dense computations.

Program modification is particularly problematic when the targets are hardware accelerators that require careful data marshaling. Such modifications are often program-wide and severely reduce the portability of the program. Furthermore, they require a commitment to specific hardware vendors, resulting in codebases that quickly become obsolete. In order to mitigate this, many projects have to keep multiple execution paths, resulting in arcane build systems and un-maintainable code. In this time of rapid hardware innovation, such a vendor lock-in is undesirable. In fact, the difficulty of efficient portable integration is a key impediment to the wider use of accelerator libraries and hardware.

In this paper, we reexamine how compilers and libraries can be used to achieve performance without programmer effort. Highly tuned and platform specific libraries invariably remain the fastest implementations available. However, we show that we can automatically integrate these libraries without polluting the source code. This is performed as a compiler transformation step, leaving the original source code intact and portable.

To achieve this, we develop a new specification language for implementers of libraries, the specification Language for implementers of Linear Algebra Computations (LiLAC). Using LiLAC, library implementers specify with a few lines of code, what a library does and how it is invoked. Our compiler then determines where the library specification matches user code and automatically transforms it to utilize the library. The language has two complementing parts.
LiLAC-What is a high-level language to describe sparse and dense linear algebra computations. The LiLAC compiler uses it to detect such functionality in user applications at compiler intermediate representation level. It is powerful enough to formulate linear algebra routines, yet remains independent of compiler internals and is easy to understand and program. LiLAC-How specifies how libraries can be used to perform a LiLAC-What-specified computation. Besides generating setup code and handling hardware context management, it crucially enables efficient memory synchronization. It uses memory protection mechanisms to automatically track data changes and transfers memory only when necessary.

The research contribution of this paper is a combination of three techniques for the acceleration of sparse linear algebra:

- Accelerate unchanged source code by identifying sparse linear algebra computations with backtracking search.
- Avoid vendor lock-in with an extensible specification language that adapts to new accelerator libraries.
- Achieve program-wide memory synchronization with only local transformations using memory protection.

Together, these techniques result in a system that works on existing and novel software. It offers the full performance of fast libraries, avoids vendor lock-in, and keeps the source code easy to maintain and free from pollution.

2 Overview

Figure 1 shows the LiLAC-enabled compiler from the user perspective. In the top left corner (1), we see unmodified application source code. This is conjugate gradient from the NAS-PB suite. To achieve good performance on Intel processors, the compiler (2) has been configured to offload native sparse code to Intel MKL. Using a specification of What computations MKL supports, it recognizes the highlighted loop as a suitable sparse matrix-vector product. Instead of passing it on to the compiler backend for code generation, it inserts a call to a harness function. This is performed on intermediate code (3) and results in a program (4). In the bottom left (5) is an equivalent source-level representation.

LiLAC also generates the corresponding harness code (6), which gets compiled into a shared library (7) that is linked with the application binary. This harness interfaces with the underlying library implementation, Intel MKL (8).

2.1 Implementation Overview

Figure 2 shows the internals of the LiLAC system. It is fully integrated into the build system of the established LLVM compiler framework, extending the clang compiler.

On the left is the LiLAC specification - just 16 lines of code. It is independent of the user application and can be provided by the library implementer. It consists of a What and a How part. These two parts are processed by the LiLAC system and result in a runtime library and a generated detection function, which is incorporated into the clang compiler.

LiLAC-What specifies the functionality that is provided by a library, in this example sspmv-csr (cf. Figure 2). From this, a function that detects the computation in normalized LLVM IR code is generated and the harness interface is determined. The detection functions are based on a backtracking search algorithm, as elaborated in section 4. The detection function is linked directly into the LiLAC-compiler, either statically or dynamically at (compiler) run time.

LiLAC-How specifies how the library, Intel MKL in this case, is invoked to perform the specified calculation. This involves boilerplate code, but also advanced features. These include efficient data synchronization and the caching of invariants. In the given example, the columns variable is such an invariant. It is required for the library call, but not
3.1 LiLAC-What: Functional Description

At the heart of our approach is a simple language to specify sparse and dense linear algebra operations. This serves two purposes in our LiLAC system: Firstly, it is used to generate a detection program for finding the computation in user code. Secondly, it identifies the variables that are arguments to the library, thus defining the harness interface.
3.2 Sparse Matrix Variations in LiLAC-What
Sparse matrices can be stored in different formats. We introduce two of them explicitly, but others are supported in the same way by LiLAC-What.

Compressed Sparse Row (CSR) [44] All non-zero entries are stored in a flat array \( \text{val} \). The \( \text{col\_ind} \) array stores the column position for each value. Finally, the \( \text{row\_ptr} \) array stores the beginning of each row of the matrix as an offset into the other two arrays. The number of rows in the matrix is given directly by the length of the \( \text{row\_ptr} \) array minus one, however, the number of columns is not explicitly stored. In Figure 4, a 5x5 matrix is shown represented in this format, the LiLAC-What code is in the top left of Figure 2.

Jagged Diagonal Storage (JDS) [43] The matrix rows are reordered such that the number of non-zeros per row is decreasing. The permutation is stored in a vector \( \text{perm} \), the number of nonzeros in \( \text{nzcnt} \). The non-zero entries are then stored in an array \( \text{val} \) in the following order: The first nonzero entry in each row, then the second nonzero entry in each row etc. The array \( \text{col\_ind} \) stores the column for each of the values and \( \text{jd\_ptr} \) stores offsets into \( \text{val} \) and \( \text{col\_idx} \). The product of a sparse matrix in JDS format with a dense vector is specified in LiLAC-What at the bottom of Figure 5.

Dense Detecting dense is easier than sparse, and existing literature covers it well. We fully support dense but evaluate it only briefly for completeness.

3.3 LiLAC-How
Where LiLAC-What specifies the computations implemented by a library, LiLAC-How describes how precisely library calls can be used to perform them. The language was designed to support important existing libraries such as cuSPARSE, dBLAS, and Intel MKL. The idiosyncrasies of these libraries require LiLAC-How to capture some boilerplate C++ code that manages the construction of parameter structures, calling conventions etc. Aside from this aspect, we designed it as high-level as possible without compromising performance. In particular, LiLAC-How abstracts away memory transfers. These considerations result in two interacting components. Firstly, a harness describes the boilerplate code for individual library invocations. Secondly, data marshaling between the core program and the library is specified, which is crucial for heterogeneous compute environments. Figure 6 shows the grammar specification of LiLAC-How.

3.3.1 Individual Library Invocations
We need to encapsulate the boilerplate code that any given library requires, such as setup code, filling of parameter structures etc. This part of the language is straightforward.

Harness The harness construct is the central way of telling the LiLAC system how a library can be used to perform a computation that was specified in LiLAC-What. As we can see at the top of Figure 6, a harness refers to a LiLAC-What program by name and also has a name itself. It is built around some C++ code, which can use all the variables from the LiLAC-What program to connect with the surrounding program. It also needs to specify the relevant C++ header files that the underlying library requires. Lastly, the harness can incorporate persistent state and utilize data marshaling.

Persistence Many libraries need setup and cleanup code, which is specified with the keywords BeforeFirstExecution and AfterLastExecution. These are used in combination with PersistentVariables, allowing state to persist between harness invocations, e.g. to retain handlers to hardware accelerators.

Example In Figure 7, we see a trivial LiLAC-What program for implementing \( \text{spmv\_csr} \) with the Intel MKL library. The actual call to the relevant library function is in line 16. To prepare for that call, there is boilerplate code in lines 7–14 to fill parameter structures.

Critically, there is an additional parameter required by the library that is data-dependent: the number of columns, \( \text{cols} \), in the sparse matrix. It is determined at runtime, in lines 2–5, leading to reduced performance. We will avoid this with the data marshaling constructs in the next section.
3.3.2 Data Marshaling

Heterogeneous accelerators require data transfers to keep memory consistent between host device and accelerator. To achieve the best performance, these have to be minimized.

Importantly, unchanged data should never be copied again. This requires program-wide analysis that is not available statically. LiLAC-How uses memory protection to implement this at runtime with minimal overhead by capturing read and write accesses to memory ranges. The same mechanism is used to cache data-dependent invariants across several invocations, such as cols in Figure 7.

Data marshaling routines are bound to ranges of memory in the harness. In the specification, the underlying array is available using the identifiers in, size, and out.

3.3.3 Detailed Example

In Figure 8, the cudaMemcpy function from NVIDIA CUDA is integrated with LiLAC-How. It is used to copy data from the host to the accelerator. For this to work, it first needs to allocate memory of the device using cudaMemcpy, which is later freed with cudaFree. Minimal memory transfers are obtained by executing cudaMemcpy only when a value in the array changes.

We can use the same construct to efficiently compute values such as the cols variable in Figure 7, as shown in Figure 9. The optimized implementation is derived from Figure 7 lines 2–5. However, instead of the concrete variable names, the reserved identifiers in, size, and out are used.

Figure 10 shows an spmv_csr LiLAC-How program for the cuSPARSE library. A number of data marshaling variables are introduced in lines 12–17, that automatically optimize both memory transfers and the computation of the cols variable. The core of the harness in lines 2–10 is again nothing more than library-specific boilerplate C++ code.

4 Implementation

The LiLAC system, as shown in Figure 2 is entirely integrated into the LLVM build system. When LLVM is compiled, the LiLAC specification is parsed using a Python program. Based on the LiLAC-What and LiLAC-How sections, C++ code is generated that is automatically incorporated into LLVM in further stages of the build process.

The result is an LLVM optimization pass that is available when linking LLVM with the clang C/C++ compiler. This pass performs the discovery of linear algebra code and the insertion of harness calls. Furthermore, the harness libraries themselves are built at compile time of LLVM, using C++ code emitted from the LiLAC-How sections.

The two crucial implementation details are therefore the following: Firstly, how automatic detection functionality in C++ is generated from the LiLAC-What specifications. Secondly, how the LiLAC-How sections are used to generate fast C++ implementations of the specified library harnesses.
4.1 LiLAC-What
The parsed LiLAC-What sections are turned into C++ functions that recognize places for harness call insertions in an LLVM pass. This builds on previous work via a formulation in CAnDL [21]. Detection is done on optimized compiler intermediate representation. Standard -O2 optimizations, excluding loop unrolling and vectorization, normalize the intermediate code. Optimizations minimize programming language-specific artifacts and the impact of syntax-level programmer decisions.

The effect is demonstrated in Figure 11, which shows three implementations of a dot product in different languages: C, C++, and FORTRAN. After translating to LLVM IR and performing optimizations, the dot product is recognized in the LiLAC system using the same LiLAC-What specification.

The detection comprises two steps, as demonstrated in Figure 13. Firstly, the control flow skeleton is recognized. This is simple, as LiLAC-What can only express control flow in the form of loop nests of a certain depth. After candidate loop nests have been identified, the index and loop range calculations from LiLAC-What are mapped onto the LLVM IR nodes. This is done via a backtracking search procedure and allows robust detection across many syntactically different input programs, as described in [21, 22].

4.1.1 Backtracking Search Algorithm
For detecting instances of LiLAC-What specifications in user programs, LLVM IR segments that match the control flow skeleton are identified. These control flow candidates are then processed with a backtracking search algorithm.

All \( \langle exp \rangle \) expressions in the LiLAC-What program are identified. These have to be assigned instructions or other values from the LLVM IR segment. Those top-level \( \langle exp \rangle \) expressions that are used as limits or iterators in \( \langle range \rangle \) expressions are easily connected with the corresponding loop boundaries in the control flow candidates.

The remaining expressions are successively assigned by backtracking. Consider the example in Figure 12, which shows a candidate loop from the LLVM IR generated from the C++ dot product code in Figure 11. The iteration space is determined by loop analysis and this immediately allows us to assign the iterator and range in Figure 13 on the left. The LLVM IR values that correspond to \( a[i], a, b[i], b, a[i] * b[i] \) and \( \text{result} \) are then searched for. When a partial solution fails, the algorithm backtracks. This happens in the example once, when no suitable multiplication can be found in step 5. If no complete solution can be determined, the control flow candidate is discarded.

4.1.2 Code Replacement
Each loop nest that matches a LiLAC-What specification is replaced with a harness call. To minimize the invasiveness of our pass, this is performed as follows: Firstly, a harness call is inserted directly before the loop. The function call arguments are selected from the backtracking result and passed to the harness. Secondly, the LLVM instruction that stores the result of the computation or passes it out of the loop as a phi node is removed. The remainder of the loop nest is removed automatically by dead code elimination.

4.2 LiLAC-How
LiLAC-How syntax elements that take C++ code generate generic functions, and template parameter deduction inserts concrete types during the compilation process.

In Figure 14, we see the correspondence between generated C++ template functions and the specification in Figure 8. The three function bodies are directly inserted. The functions are used to specialize the ReadObject class template, which guarantees the following properties via memory protection: construct is called before the first invocation and when in or size change for consecutive harness invocations.
We wrote short LiLAC programs for a collection of linear algebra designed to be executed on the GPU, and cuSPARSE is a general-purpose mathematical library, while clSPARSE and cuSPARSE are OpenCL and CUDA implementations of sparse linear algebra designed to be executed on the GPU, and SparseX uses an auto-tuning model and code generation to optimize sparse operations on particular matrices.

<table>
<thead>
<tr>
<th>Name</th>
<th>Hardware</th>
<th>Libraries</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel-0</td>
<td>2x Intel Xeon E5-2620</td>
<td>MKL</td>
</tr>
<tr>
<td></td>
<td>Nvidia Tesla K20 GPU</td>
<td>cuSPARSE</td>
</tr>
<tr>
<td>Intel-1</td>
<td>Intel Core i7-8700K</td>
<td>clSPARSE</td>
</tr>
<tr>
<td></td>
<td>Nvidia GTX 1080 GPU</td>
<td>SparseX</td>
</tr>
<tr>
<td>AMD</td>
<td>AMD A10-7850K</td>
<td>cuSPARSE</td>
</tr>
<tr>
<td></td>
<td>AMD Radeon R7 iGPU</td>
<td>clSPARSE ×2</td>
</tr>
<tr>
<td></td>
<td>Nvidia Titan X GPU</td>
<td>SparseX</td>
</tr>
</tbody>
</table>

Table 1. Evaluated platforms and library harnesses; AMD-0 supports clSPARSE on both its internal and its external GPU.

Applications To evaluate the impact of LiLAC in a real-world context, we used the PATHSAMPLE physical chemistry simulation suite, a large FORTRAN legacy application [56] consisting of over 40,000 lines of code. Recent work shows that applications in this area are amenable to acceleration using sparse linear algebra techniques [53], and PATHSAMPLE provides a useful example of this. We also evaluated two modern C++ graph analytics kernels (BFS and PageRank [11, 15]). PATHSAMPLE was run in two different modes and three different levels of pruning, in each case using a system of 38 atoms [18] commonly used to evaluate applications in this domain. The graph kernels were run against 10 matrices from the University of Florida’s sparse matrix collection [14], with sizes between 300K and 80M non-zero elements.

For completeness and validation that our LiLAC-generated implementations were correct, we also applied our technique to sparse programs from standard benchmark suites: CG from the NAS parallel benchmarks [9], spmv from Parboil [48] and the Netlib sparse benchmark suites [17]. Each benchmark suite was run using their supplied inputs.

Platforms We evaluated our approach across 3 different machines with varying hardware performance and software availability. Each one was only compatible with a subset of our LiLAC-generated implementations—a summary of these machines is given in Table 1.

6 Results
We first present raw performance impact, then we analyze two intermediate metrics: reliability of linear algebra discovery and effectiveness of memory transfer optimizations.

6.1 Performance
LiLAC achieves significant speedups on real applications as well as benchmarks, as shown in Figure 15. Baselines were compiled with -O2 using the same version of clang without LiLAC extensions. Higher optimization levels (-O3) had a negligible impact on performance. Different platforms and applications profit from different libraries (subsection 6.2). Speedup ranges from 1.1–3.3x on the scientific application codes to 12× on well-known sparse benchmark programs.
Applications  On the PATHSAMPLE applications (PFold and NGT), we measured consistent speedups of approximately 50% and 10% respectively across all 3 platforms. For large applications, Amdahl’s law is a severe limitation for approaches like ours – other parts of the applications dominate execution times when linear algebra is accelerated.

Graph kernels  PageRank requires a large number of SPMV calls using the same input matrix to iterate until convergence. The GPU implementations running on AMD and Intel-1 take advantage of data remaining in memory. The larger number of CPU cores and slower GPU available on Intel-0 make MKL its best-performing implementation. CPU implementations perform best on BFS by avoiding memory copies entirely – on AMD, SparseX outperforms GPU implementations.

Benchmarks  LiLAC achieves speedups of up to 12× on standard sparse linear algebra benchmarks. The impact is independent of the source language, as the C and FORTRAN versions of the Netlib benchmark demonstrate. LiLAC is able to achieve consistent, useful speedups across a variety of hardware configurations.

Dense  We evaluated on some dense benchmarks as well. In line with the literature, dense is very amenable to heterogeneous acceleration. We achieve 20× speedup on Parboil sgemm by inserting LiLAC-harnessed calls into sequential baseline. However, impressive heterogeneous speedups on dense are well explored in the literature, we focus on sparse.

Comparison to Expert  NPB and Parboil contain expert-written alternative versions with GPU acceleration. This allowed the evaluation of LiLAC against heterogeneous code reaching close to peak performance, shown in Figure 16.

While the expert version of NPB-CG is ~3× faster, this is not due to an improved sparse linear algebra operation, but a complete parallelization and rewrite of the program for the GPU. In Parboil SPMV, the expert version focuses on improved sparse linear algebra. Here the difference between an expert and LiLAC is only 1.07×.

Productivity  The bottom of Figure 16 shows the amount of code modified in order to add heterogeneous acceleration manually vs with LiLAC. This demonstrates the productivity improvements for application programmers. No lines of user code need to be modified using LiLAC, while both expert versions require significant application rewrites. Only 44 lines of application-independent LiLAC code is required.

6.2 Necessity of Flexible Backends  The relative performance of different accelerator libraries is highly dependent on the application, problem size, and platform, as Figure 17 shows.
Table 2. LiLAC speedups on each platform, across different applications and problem sizes. SparseX demonstrated promising performance on some applications, but we were unable to evaluate on every relevant instance due to instability. Implementation with best geomean speedup per benchmark and platform is bold.

6.3 Effectiveness of Data Marshaling
Our implementation of LiLAC relies on a non-trivial data marshaling system that prevents redundant computations and memory transfers. We present performance results that show the importance and effectiveness of this system.

We repeated our experiments, using the best-performing implementations from Figure 15. Instead of using the data marshaling scheme, we recompute and transfer memory naively for each invocation. The results are in Figure 18. Across the best AMD versions of PFold, NGT, PageRank and BFS – where accelerators are profitable with marshaling – only PageRank achieves a significant speedup naively.

For BFS, the naive approach leads to drastic performance degradation, the marshaling version is 25× faster. This is because it performs an internal matrix tuning phase that is far more expensive than a memory copy. For the other three programs, there is a factor of 1.4–3.5× between the naive and the smart version.

6.4 Reliability of Discovery
For performance impact, LiLAC needs to first detect linear algebra computations. Previous results already implied that this works reliably, and Table 3 reiterates this. All relevant sparse matrix-vector multiplications were recognized.

Established approaches, like the polyhedral model, are unable to model sparse linear algebra, as verified with the Polly compiler. Similarly, the Intel C/C++ and FORTRAN compilers fail to auto-parallelize, as they cannot reason about sparsity and have to assume additional dependencies.

These results show the novelty of the abilities of LiLAC rather than implementation weaknesses of Polly and ICC, as neither were designed for accelerating sparse computations.

Table 3. Sparsity does not fit the polyhedral model; Polly is not available for FORTRAN; Intel compilers fail to parallelize sparse. Only LiLAC detects sparse linear algebra reliably.
7 Related Work

**Compiler centric linear algebra optimization** Compiler management of indirect memory model accesses was first examined using an inspector-executor model for distributed-memory machines [10]. The location of read data was discovered at runtime and appropriate communication inserted. Later work was focused on efficient runtime dependence analysis and the parallelization of more general programs [20, 38, 41, 49]. However, the performance achieved is modest due to runtime overhead and falls well short of library performance. More recent work developed equality constraints and subset relations that help reduce the runtime overhead [32].

The polyhedral model is an established compiler approach for modeling data dependencies [12, 24, 26, 42, 47]. Such an approach has been implemented in optimizing compilers, such as the Polly extensions to LLVM [16]. Recent work has extended the polyhedral model beyond affine programs to some forms of sparsity with the PENCIL extensions [8]. These can be used to model important features of sparse linear algebra, such as counted loops [58], i.e., loops with dynamic, memory dependent bounds but statically known strides. Such loops are central to sparse linear algebra. The PPCG compiler [54] can detect relevant code regions, but it relies on well behaved C code with all arrays declared in variable-length C99 array syntax. This excludes most real-world programs; nothing in our evaluation fits this structure.

The Appollo system [51] integrates thread level speculation with the polyhedral model,allowing its application to sparse linear algebra. However, it requires sub-parts of the computation to perform dense accesses at runtime. Similar approaches [7] also require regular sub-computations.

**Compiler detection** Previous work has detected code structures in compilers using constraint programming. Early work was based on abstract computation graphs [37], but more recent approaches have used compiler intermediate code and made connections to the polyhedral model [21].

In [22] they implement a method that operates on SSA intermediate representation. It uses a general-purpose low-level constraint programming language aimed at compiler engineers. The paper focuses on code detection, with manual data marshaling. Recent work [13] uses type-guided program synthesis to model library routines, which are then detected by a solver. Again, data marshaling is not taken into account.

Other advanced approaches to extracting higher-level structures from assembly and well-structured FORTRAN code involve temporal logic [27, 31]. These approaches tend to focus on a more restricted set of computations (dense memory access). While this allows formal reasoning about correctness, it is too restrictive to model sparse linear algebra.

**Domain-Specific Languages** There have been multiple domain-specific libraries proposed to formulate linear algebra computations. Many of these contain some degree of autotuning functionality to achieve good performance across different platforms [50]. Halide [39] was designed for image processing. [52]. Its core design decision is the scheduling model that allows the separation of the computation schedule and the actual computation. There has been work on automatically tuning the schedules [35] but in general, the computational burden is put on the application programmer.

The SLinGen [45] compiler takes a program expressed in the custom LA language, inspired by standard mathematical notation. It then implements custom code generation for the expressed calculations, with a focus on small, fixed-size operands. This is built on top of building blocks provided by previous work on LGen [46]. The approach outperforms libraries focused on large data sizes but is unable to utilize heterogeneous compute and requires program rewrites.

**Libraries** The most established way of encapsulating fast linear algebra routines is via numeric libraries, generally based on the BLAS interface [6]. These are generally very fast on specific hardware platforms, but require application programmer effort and offer little performance portability. Implementations of dense linear algebra are available for most suitable hardware platforms, such as cuBLAS [4] for NVIDIA GPUs, clBLAS [1] for AMD GPUs and the Intel MKL library [3] for Intel CPUs and accelerators.

Fast implementations of sparse linear algebra are fewer, but they exist for the most important platforms, including cuSPARSE [5] and clSPARSE [2]. There have been several BLAS implementations that attempt platform independent acceleration and heterogeneous compute [33, 34, 57].

**CPU-GPU data transfer optimizations** Data transfers between CPU and GPU have been studied extensively as an important bottleneck for parallelization efforts. Previous work [25, 30] established systems for automatic management of CPU-GPU communication. The authors of [29] implemented a system to move OpenMP code to GPUs, optimizing data transfers using data flow analysis. However, this approach performs a direct translation, not optimizing the code for the specific performance characteristics of GPUs.

8 Conclusion

This paper presented LiLAC, a language and compiler that enables existing codebases to exploit sparse (and dense) linear algebra accelerators. No effort is required from the application programmer. Instead, the library implemencer provides a specification, which LiLAC uses to automatically and efficiently match user code to high-performance libraries.

We demonstrated this approach on C, C++, and FORTRAN benchmarks as well as legacy applications, and shown significant performance improvement across platforms and data sets. In future work, we will investigate how our framework can be adapted to other application domains, enabling effort-free access to an even larger set of accelerator libraries.
References


