Automatically Harnessing Sparse Acceleration

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

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

Michael F.P. O'Boyle
School of Informatics
University of Edinburgh
Edinburgh, United Kingdom
mob@inf.ed.ac.uk

Abstract

Sparse linear algebra is central to many scientific codes, yet compilers fail to optimize it well. High performance acceleration libraries are available, but adoption costs are significant. Furthermore, 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. Using this, the LiLAC-enabled compiler then inserts appropriate library routines automatically, without source code changes.

LiLAC provides automatic data marshaling, maintaining state between calls as needed and minimizing data transfers. Appropriate places for library replacement are detected at compiler intermediate representation level, independent of source languages.

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

ACM Reference Format:
Philip Ginsbach, Bruce Collie, and Michael F.P. O'Boyle. 2020. Automatically Harnessing Sparse Acceleration. In *Proceedings of ACM SIGPLAN International Conference on Compiler Construction (CC'20).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 [22, 34, 38], there has been less success with sparse codes. This is largely due to indirect memory access, which challenges compiler analysis [30]. Sparse-based algorithms are, however, increasingly important as the basis of graph algorithms and data analytics [27].

CC'20, February 22–23, 2020, San Diego, CA, USA
2020. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

We currently see the wide-scale provision of fast sparse libraries [2, 3, 5, 52]. They deliver excellent performance, but require significant programmer intervention and are rarely portable across platforms. Alternatives, such as the SLinGen/LGen system [43, 44], 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 code bases that quickly become obsolete. In order to mitigate this, many projects have to keep multiple execution paths, resulting in arcane build systems and unmaintainable 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 are 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. The program is written in a straightforward manner and a naively compiled executable would fail to exploit the full potential of modern hardware.

In order to achieve good performance on Intel processors, the compiler (2) has been configured to offload native sparse code to Intel MKL. Using an internal 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 the auto-generated *harness* function and captures the parameters of the computation as function call arguments. This is performed on optimized intermediate representation 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, implementing the previously discussed behavior. The functionality 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 *spmv-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, in this case Intel MKL, 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 statically available. Therefore, it has to be computed at runtime. Using *Marshaling*, LiLAC automatically generates the harness such that this is only recomputed if the values in *row_ptr* change. Such changes are captured with generated memory protection code using *mprotect*, managed by LiLAC.

On the right of the figure we can see how the components generated from the LiLAC specification are used to build the LiLAC-compiler. The detection function is compiled and used directly by the LiLAC-Compiler, linked either statically or dynamically. Interacting with the internals of LLVM, it implements a transformation pass that is executed after the normal optimization pipeline. Using the generated detection function, it finds instances of the computation and replaces them with calls to the specified harness interface.

The harness, on the other hand, is compiled into a shared library. The LiLAC-compiler dynamically links applications to this shared library whenever it inserts harness calls. When multiple LiLAC-How programs are provided, the generated harnesses are compatible and linking the user program to a different harness library at runtime is sufficient.

3 What and How

This section describes in more detail the two components of the LiLAC language. LiLAC-What specifies the computations that a library performs, LiLAC-How describes how exactly the library should be invoked to perform these computations.

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.

The key design challenge was to stay simple enough to automatically generate robust detection functionality, yet to be able to capture operations in all relevant data formats. Most importantly, this includes the CSR/CSC, JDS and COO formats. CSR and JDS are part of our evaluation. The control flow on the other hand is rigid and easy to express. This is reflected in the grammar as shown in Figure 3.

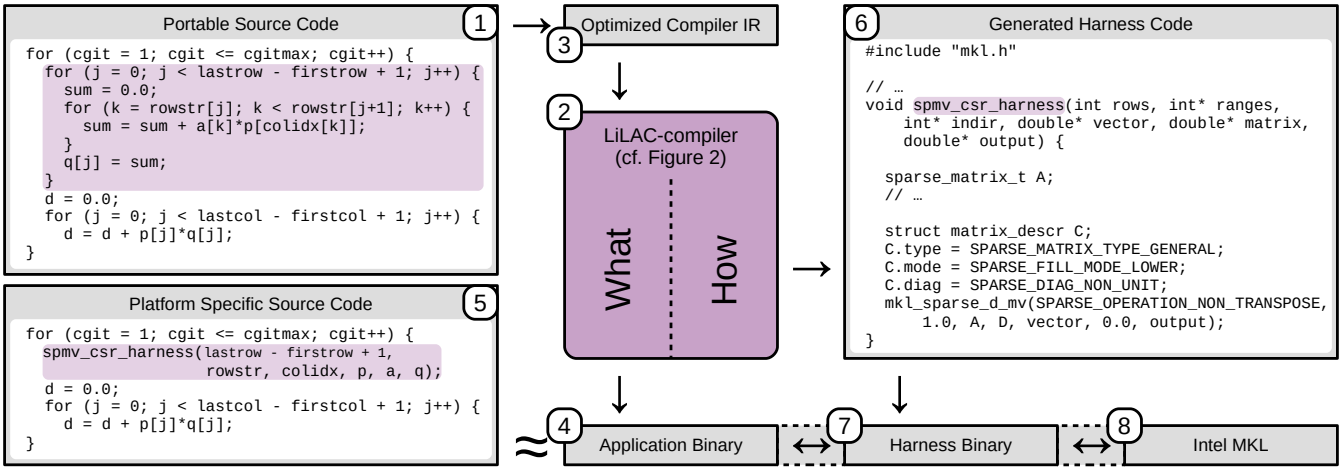


Figure 1. LiLAC applied to NPB Conjugate Gradient: Code (1) that matches the LiLAC-What specification (cf. Figure 2) is replaced by calls to a harness (5) during compilation (2), resulting in an application binary (6) that corresponds to (hypothetical) platform specific source code (4). The harness is generated from the LiLAC-How specification (cf. Figure 2) to utilize Intel MKL.

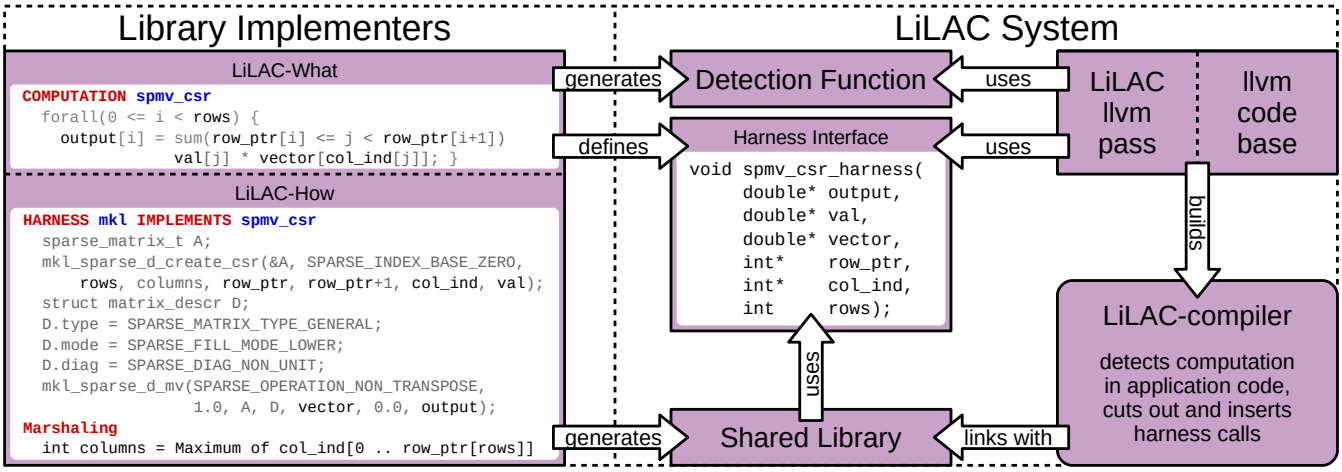


Figure 2. Overview of LiLAC internals: On the left is the entire LiLAC program that the library implementer has to provide. At build time of LLVM, this program is parsed and incorporated into a modified clang C++ compiler, behaving as in Figure 1.

3.2 Sparse Matrix Variations in LiLAC-What

Sparse matrices can be stored in different formats. In this section we introduce two of them and show how LiLAC-What can express the corresponding computations.

Compressed Sparse Row (CSR) [42] All non-zero entries are stored in a flat array **val**. The **col_ind** array stores the column position for each value. Finally, the **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 **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) [41] The matrix rows are reordered such that the number of non-zeros per row is decreasing. The permutation is stored in a vector **perm**, the number of nonzeros in **nzcnt**. The nonzero entries are then stored in an array **val** in the following order: The first nonzero entry in each row, then the second nonzero entry in each row etc. The array **col_ind** stores the column for each of the values and **jd_ptr** stores offsets into **val** and **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. fully support dense, but evaluate on it only 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 the data formats used in sparse linear algebra routines from important existing libraries such as cuSPARSE, clBLAS and Intel MKL. The idiosyncrasies of these libraries require LiLAC-How specifications to be built around boilerplate C++ code that manages the construction of parameter structures, calling conventions etc. However, we also wanted to make it as high-level as possible, without sacrificing any performance. In particular, LiLAC-How abstracts away memory transfers.

The result is a language that can be further divided into two interacting components. Firstly, it describes boilerplate code that is required for individual library call invocations in a *harness*. Secondly, it is used for data *marshaling* between the core program and the library, which is particularly 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 `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, *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. 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. The LiLAC-generated code then uses memory protection to detect read and write accesses to these memory ranges. In the specification, the underlying array is available using the identifiers `in`, `size`, and `out`.

3.3.3 Detailed Example

In Figure 8, LiLAC-How functionality is specified using the `cudaMemcpy` function from NVIDIA CUDA. It is used to copy data from the host to the accelerator. Note that for this to work, it first needs to allocate memory of the device using `cudaMalloc`, which is later freed with `cudaFree`. `cudaMemcpy` is executed only when a value in the array changes, resulting in minimal memory transfers.

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 nearly identical to the code in Figure 7 lines 2-5, however, instead of the concrete variable names, the reserved identifiers `in`, `size`, and `out` are used.

We can now use these in Figure 10, showing a CSR SPMV 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

Once the LiLAC-What sections have been parsed, they are turned into C++ functions that are used by the LiLAC LLVM pass to detect appropriate places for harness calls. Instead of using syntax-level pattern matching, detection is done on optimized compiler intermediate representation. Standard optimizations from `-O2`, excluding loop unrolling and vectorization, normalize the intermediate code. This removes many programming language specific artifacts and minimizes 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 exact detection algorithm works in two steps. Firstly, the control flow skeleton is recognized. This is simple, as the control flow that can be expressed in LiLAC-What is limited, and we only need to detect 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 that requires some careful consideration in order to get robust results on implementation details of LLVM, particularly for different ways of array indexing, pointer casts and integer conversions.

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 ⟨*exp*⟩ 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 ⟨*exp*⟩ expressions that are used as limits or iterators in ⟨*range*⟩ 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 `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. When a complete solution cannot be determined, the control flow candidate is discarded.

4.1.2 Code Replacement

For each identified loop nest that matches a LiLAC-What specification, the code 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

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

Consider Figure 14, generated from LiLAC-How code in Figure 8. We see the correspondence between generated C++ and LiLAC-How code in the three functions that contain the source code specified in LiLAC-How. These functions are used to specialize the *ReadObject* class template. It guarantees the following properties using hardware memory protection: ***construct*** is called before the first invocation and when `in` or `size` change for consecutive harness invocations. ***update*** is called after *construct* and if any of the data in the array is changed between consecutive harness invocations. ***destruct*** is called in between consecutive *construct* calls and before the program terminates.

4.3 FORTRAN

Compatibility with FORTRAN was a key implementation hurdle. The LLVM frontend under active development, flang, is in an unfinished state and produces unconventional LLVM IR code by default. Significant additional work was required to normalize the IR code. We developed normalization passes in LLVM to overcome the specific shortcomings, enabling FORTRAN programs to be managed as easily as C/C++.

The problems that we encountered included: differing indexing conventions requiring offsetting pointer variables on a byte granularity with untyped pointers; incompatible intermediate representation types where all parameters are passed in as `i64` pointers, frequently necessitating a pointer type conversion followed by a load from memory; obfuscated loops with additional induction variable that counts down instead of up such that the standard LLVM **indvars** pass is unable to merge the loop iterators.

5 Experimental Setup

We wrote short LiLAC programs for a collection of linear algebra libraries and applied our approach to a chemical simulation application, two graph analytics applications and a collection of standard benchmark suites.

Name	Hardware	Libraries
Intel-0	2× Intel Xeon E5-2620 Nvidia Tesla K20 GPU	MKL cuSPARSE
Intel-1	Intel Core i7-8700K Nvidia GTX 1080 GPU	clSPARSE SparseX
AMD	AMD A10-7850K AMD Radeon R7 iGPU Nvidia Titan X GPU	cuSPARSE clSPARSE ×2 SparseX

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

Libraries We selected four different libraries for sparse linear algebra functions. These were: Intel MKL [3], Nvidia cuSPARSE [5], clSPARSE [2] and SparseX [18]. MKL is a general-purpose mathematical library designed to provide easy-to-integrate performance primitives, 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.

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 [53] 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 [50], and PATHSAMPLE provides a useful example of this. We also evaluated two modern C++ graph analytics kernels (BFS and PageRank [10, 14]). PATHSAMPLE was run in two different modes and three different levels of pruning, in each case using a system of 38 atoms [17] 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 [13], 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 codes from standard benchmark suites: CG from the NAS parallel benchmarks [8], spmv from Parboil [46] and the Netlib sparse benchmark suites [16]. 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 show that across a variety of hardware and software platforms, LiLAC can speed up real-world applications. 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. Different platforms and applications profit from different libraries, we discuss in more detail in subsection 6.2. Speedup ranges from 1.1–3× 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 are able to 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 Implementation NPB and Parboil contain expert-written alternative versions with GPU acceleration. This allows evaluation of LiLAC against heterogeneous code reaching close to peak performance. Furthermore, we can count the lines of code that need to be modified in order to add heterogeneous acceleration manually.

Figure 16 shows the results. 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. 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. Parboil SPMV

Platform	Implementation	PFold			NGT			PageRank			BFS		
		L0	L1	L2	L0	L1	L2	Erdos	LJ-2008	Road	Erdos	LJ-2008	Road
AMD	cuSPARSE	1.38	1.18	0.67	0.69	0.69	0.70	**3.44**	**1.18**	**9.97**	1.62	6.55	1.96
	clSPARSE (eGPU)	**2.17**	**1.82**	**1.22**	**1.16**	**1.16**	**1.16**	3.08	1.24	6.06	0.50	11.03	0.24
	clSPARSE (iGPU)	2.03	1.78	1.03	0.90	0.90	0.90	3.26	1.31	4.05	0.14	4.17	0.05
	SparseX	-	-	-	-	-	-	-	-	-	**1.93**	-	-
Intel-0	MKL	**2.88**	**2.46**	**1.00**	**1.18**	**1.18**	**1.18**	**1.25**	**2.93**	**1.72**	**2.50**	**1.06**	**1.05**
	cuSPARSE	0.75	0.60	0.45	0.66	0.66	0.66	1.39	1.00	3.32	0.87	1.74	1.28
	clSPARSE	0.90	0.75	0.46	0.81	0.79	0.78	1.24	0.95	2.24	0.13	1.45	0.07
	SparseX	-	-	-	-	-	-	-	-	-	1.19	-	-
Intel-1	MKL	**2.70**	**2.43**	**1.01**	**1.20**	**1.20**	**1.19**	1.63	1.03	2.26	**1.06**	**2.09**	**1.27**
	cuSPARSE	0.48	0.41	0.30	0.68	0.69	0.68	**1.59**	**0.87**	**4.44**	1.01	1.83	1.63
	clSPARSE	1.00	1.00	1.00	1.00	1.02	1.00	1.50	0.87	3.46	0.23	1.81	0.13
	SparseX	-	-	-	-	-	-	-	-	-	1.25	-	-

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.

$$
\begin{aligned}
\mathit{program} &::= \textbf{COMPUTATION}\ \langle \mathit{name}\rangle\ \langle \mathit{body}\rangle \\
\mathit{body} &::= \langle \mathit{forall}\rangle \mid \langle \mathit{dotop}\rangle \\
\mathit{range} &::= \textbf{(}\ \langle \mathit{exp}\rangle <= \langle \mathit{name}\rangle < \langle \mathit{exp}\rangle\ \textbf{)} \\
\mathit{forall} &::= \textbf{forall}\ \langle \mathit{range}\rangle\ \textbf{\{}\ \langle \mathit{body}\rangle\ \textbf{\}} \\
\mathit{dotp} &::= \langle \mathit{addr}\rangle = \textbf{dot}\ \langle \mathit{range}\rangle\ \langle \mathit{addr}\rangle * \langle \mathit{addr}\rangle\ \textbf{;} \\
\mathit{addr} &::= \langle \mathit{name}\rangle\ \{\ \textbf{[}\ \langle \mathit{exp}\rangle\ \textbf{]}\ \} \\
\mathit{add} &::= \langle \mathit{exp}\rangle + \langle \mathit{exp}\rangle \\
\mathit{mul} &::= \langle \mathit{exp}\rangle * \langle \mathit{exp}\rangle \\
\mathit{exp} &::= \langle \mathit{name}\rangle \mid \langle \mathit{cnst}\rangle \mid \langle \mathit{addr}\rangle \mid \langle \mathit{add}\rangle \mid \langle \mathit{mul}\rangle
\end{aligned}
$$

Figure 3. Grammar of the LiLAC-What language

$$
\begin{bmatrix} 1 & 0 & 1 & 0 & 0 \\ 0 & 2 & 0 & 2 & 0 \\ 0 & -1 & 3 & 2 & 0 \\ 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & -1 & 0 & 1 \end{bmatrix}
$$

$$
\begin{aligned}
\textbf{val} &= \begin{bmatrix} 1 & 1 & 2 & 2 & -1 & 3 & 2 & 2 & -1 & 1 \end{bmatrix} \\
\textbf{col\_ind} &= \begin{bmatrix} 0 & 2 & 1 & 3 & 1 & 2 & 3 & 3 & 2 & 4 \end{bmatrix} \\
\textbf{row\_ptr} &= \begin{bmatrix} 0 & 2 & 4 & 7 & 8 & 10 \end{bmatrix}
\end{aligned}
$$

Figure 4. Compressed Sparse Row representation as used by the LiLAC-What example in Figure 1

$$
\begin{bmatrix} 0 & -1 & 3 & 2 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 2 & 0 & 2 & 0 \\ 0 & 0 & -1 & 0 & 1 \\ 0 & 0 & 0 & 2 & 0 \end{bmatrix}
$$

$$
\begin{aligned}
\textbf{perm} &= \begin{bmatrix} 1 & 2 & 0 & 4 & 3 \end{bmatrix} \\
\textbf{val} &= \begin{bmatrix} -1 & 1 & 2 & -1 & 2 & 3 & 1 & 2 & 1 & 2 \end{bmatrix} \\
\textbf{col\_ind} &= \begin{bmatrix} 1 & 0 & 1 & 2 & 3 & 2 & 2 & 3 & 4 & 3 \end{bmatrix} \\
\textbf{jd\_ptr} &= \begin{bmatrix} 0 & 5 & 9 & 10 \end{bmatrix} \\
\textbf{nzcnt} &= \begin{bmatrix} 3 & 2 & 2 & 2 & 1 \end{bmatrix}
\end{aligned}
$$

```
COMPUTATION spmv_jds
  forall(0 <= i < rows) {
    output[perm[i]] = sum(0 <= j < nzcnt[i])
      val[jd_ptr[j]+i] * vector[col_ind[jd_ptr[j]+i]]; }
```

Figure 5. Jagged Diagonal Storage in LiLAC-What

$$
\begin{aligned}
\mathit{harness} &::= \textbf{HARNESS}\ \langle \mathit{name}\rangle\ \textbf{IMPLEMENTS}\ \langle \mathit{name}\rangle \\
&\quad \langle C++\rangle \\
&\quad [\ \langle \mathit{marshaling}\rangle\ ]\ [\ \langle \mathit{persistence}\rangle] \\
&\quad [\ \textbf{CppHeaderFiles}\ \{\ \langle \mathit{name}\rangle\ \}\ ] \\
\mathit{persistence} &::= \textbf{PersistentVariables}\ \{\ \langle \mathit{name}\rangle\ \langle \mathit{name}\rangle\ \} \\
&\quad [\ \textbf{BeforeFirstExecution}\ \langle C++\rangle\ ] \\
&\quad [\ \textbf{AfterLastExecution}\ \langle C++\rangle\ ] \\
\hline
\mathit{marshalling} &::= \textbf{Marshaling} \\
&\quad \{\ \langle \mathit{type}\rangle\ \langle \mathit{name}\rangle = \langle \mathit{name}\rangle\ \textbf{of} \\
&\quad \langle \mathit{name}\rangle\ \textbf{[0 ..}\ \langle \mathit{exp}\rangle\ \textbf{]}\ \} \\
\mathit{input} &::= \textbf{INPUT}\ \langle \mathit{name}\rangle\ \langle C++\rangle \\
&\quad [\ \textbf{BeforeFirstExecution}\ \langle C++\rangle\ ] \\
&\quad [\ \textbf{AfterLastExecution}\ \langle C++\rangle\ ] \\
\mathit{output} &::= \textbf{OUTPUT}\ \langle \mathit{name}\rangle\ \langle C++\rangle \\
&\quad [\ \textbf{BeforeFirstExecution}\ \langle C++\rangle\ ] \\
&\quad [\ \textbf{AfterLastExecution}\ \langle C++\rangle\ ]
\end{aligned}
$$

Figure 6. Grammar of LiLAC-How

is a more realistic comparison where the expert version focuses on improved sparse linear algebra. Here the difference between an expert and LiLAC is only 1.07×.

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. Not committing to any single implementation in the source code is therefore important in order to reach full performance.

Table 2 has more detailed data. The best-performing implementation varies considerably, depending on characteristics of the problem in question. No accelerator library performs well reliably, each harness outperforms any other harness on some combination of data set and platform. For some small problem sizes, hardware acceleration is not profitable. Those slowdowns are due to inherent overheads, not LiLAC.

```
1  HARNESS mkl IMPLEMENTS spmv_csr
2    int cols = 0;
3    for(int i = 1; i < rowstr[rows]; i++)
4      cols = colidx[i]>cols?colidx[i]:cols;
5    cols = cols+1;
6
7    sparse_matrix_t A;
8    mkl_sparse_d_create_csr(&A, SPARSE_INDEX_BASE_ZERO,
9                            rows, cols, rowstr,
10                           rowstr+1, colidx, a);
11   struct matrix_descr dscr;
12   dscr.type = SPARSE_MATRIX_TYPE_GENERAL;
13   dscr.mode = SPARSE_FILL_MODE_LOWER;
14   dscr.diag = SPARSE_DIAG_NON_UNIT;
15
16   mkl_sparse_d_mv(SPARSE_OPERATION_NON_TRANSPOSE,
17                   1.0, A, dscr, iv, 0.0, ov);
18 PersistentVariables
19   "mkl.h"
```

Figure 7. This LiLAC-What code implements CSR SPMV naïvely with Intel MKL. Performance is degraded because of lines 2–5. Figure 9 will present a solution to this bottleneck.

```
1  INPUT CudaRead
2    cudaMemcpy(out, in, sizeof(type_in)*size,
3               cudaMemcpyHostToDevice);
4  BeforeFirstExecution
5    cudaMalloc(&out, sizeof(type_in)*size);
6  BeforeFirstExecution
7    cudaFree(out);
```

Figure 8. LiLAC-How code to provide efficient automatic data marshaling between host and CUDA accelerator.

```
1  INPUT Maximum
2    out = in[0];
3    for(int i = 1; i < size; i++)
4      out = in[i]>out?in[i]:out;
5    out = out+1;
```

Figure 9. *INPUT* can also be used to specify data dependent computations that are only recalculated when required.

```
1  HARNESS cuda IMPLEMENTS spmv_csr
2    double alpha = 1.0;
3    double beta  = 0.0;
4    cusparseMatDescr_t descrA;
5    cusparseCreateMatDescr(&descrA);
6    cusparseDcsrmv(handle,
7                   CUSPARSE_OPERATION_NON_TRANSPOSE,
8                   rows, cols, ranges[rows], &alpha,
9                   descrA, d_mat, d_ranges, d_indir,
10                  d_vec, &beta, d_out);
11 Marshaling
12   int      cols = Maximum of indir[0..ranges[rows]]
13   double* d_mat = CudaRead of matrix[0..ranges[rows]]
14   double* d_vec = CudaRead of vector[0..cols]
15   int* d_ranges = CudaRead of ranges[0..rows+1]
16   Int*  d_indir = CudaRead of indir[0..rowstr[rows]]
17   double* d_out = CudaWrite of output[0..rows]
18 PersistentVariables
19   cusparseHandle_t handle
20 BeforeFirstExecution
21   cusparseCreate(&handle);
22 AfterLastExecution
23   cusparseDestroy(handle);
24 CppHeaderFiles
25   <cuda_runtime.h> "cusparse_v2.h"
```

Figure 10. This LiLAC-What specification implements an efficient SPMV harness using cuSPARSE in 25 lines of code.

```
COMPUTATION dotproduct
    result = sum(0 <= i < length) a[i] * b[i];
```

```
int i = 0;
while(i < N) {
    x += (*(A+i))*(*(B+i));
    i+=1; }
```

```
for(int i = 0; i < vec_a.size(); i++)
    x += vec_a[i]*vec_b[i];
```

```
DO I = 1, N, 1
    X = X + A(i)*B(i)
END DO
```

Figure 11. Syntactically different computations in C, C++ or FORTRAN are captured by one LiLAC-What specification.

```
; <label>:17:
  %18 = phi i64 [ 0, %10 ], [ %26, %17 ]
  %19 = phi double [ 0.0, %10 ], [ %25, %17 ]
  %20 = getelementptr double, double* %9, i64 %18
  %21 = load double, double* %20
  %22 = getelementptr double, double* %12
  %23 = load double, double* %22
  %24 = fmul double %21, %23
  %25 = fadd double %19, %24
  %26 = add nuw i64 %18, 1
  %27 = icmp ugt i64 %14, %26
  br i1 %27, label %17, label %15
```

Figure 12. LiLAC intercepts LLVM IR after optimizations. This ensures normalized and language independent features.

			a[i]	←	**1:** %21	
			a	←	**2:** %9	
i	←	%18	b[i]	←	**3:** %21	**6:** %23
length	←	%14	b	←	**4:** %9	**7:** %12
			a[i] * b[i]	←	**5:** fail!	**8:** %24
			result	←		**9:** %25

Figure 13. After finding a candidate loop and receiving some variables from loop analysis (left), the backtracking solver attempts to assign the remaining variables one by one (right).

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.

```
template<typename type_in, typename type_out>
void CudaRead_update(type_in* in, int size,
                     type_out& out) {
    cudaMemcpy(out, in, sizeof(type_in)*size,
               cudaMemcpyHostToDevice);
}
template<typename type_in, typename type_out>
void CudaRead_construct(int size, type_out& out) {
    cudaMalloc(&out, sizeof(type_in)*size);
}
template<typename type_in, typename type_out>
void CudaRead_destruct(int size, type_out& out) {
    cudaFree(out);
}
template<typename type_in, typename type_out>
using CudaRead = ReadObject<type_in, type_out,
    CudaRead_update<type_in,type_out>,
    CudaRead_construct<type_in,type_out>,
    CudaRead_destruct<type_in,type_out>>;
```

Figure 14. LiLAC uses code from Figure 8 to define three functions that specialize the ReadObject template, which uses `mprotect` for memory protection internally.

6.4 Reliability of Discovery

To achieve performance impact, LiLAC needs to first reliably detect linear algebra computations, independent of coding style and source programming language. Previous results already demonstrated that this works reliably and Table 3 reiterates this. Established approaches, like the polyhedral model, are unable to model sparse linear algebra. We verified with the Polly compiler. Similarly, the Intel icc and ifort compilers fail to auto-parallelize, as they cannot reason about sparsity and have to assume additional dependencies where sparse accesses occur.

7 Related Work

Compiler centric linear algebra optimization Compiler management of indirect memory accesses was first examined using an inspector-executor model for distributed-memory machines [9]. 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 [19, 36, 39, 47]. 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 [30].

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

Benchmark	LiLAC	Polly	Intel icc/ifort
PFold	CSR	-	parallel dependence
NGT	CSR	-	parallel dependence
Parboil-SPMV	JDS	no SCoP	parallel dependence
BFS	CSR	no SCoP	parallel dependence
NPB-CG	CSR	-	parallel dependence
PageRank	CSR	no SCoP	parallel dependence
Netlib C	CSR	no SCoP	parallel dependence
Netlib Fortran	CSR	-	parallel dependence

The polyhedral model is an established compiler approach for modeling data dependencies [11, 23, 25, 40, 45]. Such an approach has been implemented in optimizing compilers, such as the Polly extensions to LLVM [15]. Recent work has extended the polyhedral model beyond affine programs to some forms of sparsity with the PENCIL extensions [7]. These can be used to model important features of sparse linear algebra, such as counted loops [55], i.e. loops with dynamic, memory dependent bounds but statically known strides. Such loops are central to sparse linear algebra. The PPCG compiler [51] 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.

Compiler detection Previous work has detected code structures in compilers using constraint programming. Early work was based on abstract computation graphs [35], but more recent approaches have used compiler intermediate code and made connections to the polyhedral model [20].

In [21] 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, data marshaling is manually inserted. Recent work [12] uses type-guided program synthesis to model library routines, which are then detected by a constraint 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 [26, 29]. These approaches tend to focus on a more restricted set of computations (dense memory access). While this allows formal reasoning about correctness, 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 [48]. Halide [37] was designed for image processing. [49]. 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 [33] but in general the computational burden is put on the application programmer.

The SLinGen [43] 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 [44]. The approach outperforms libraries focused on large data sizes, but is unable to utilize heterogeneous compute and requires program rewrites.

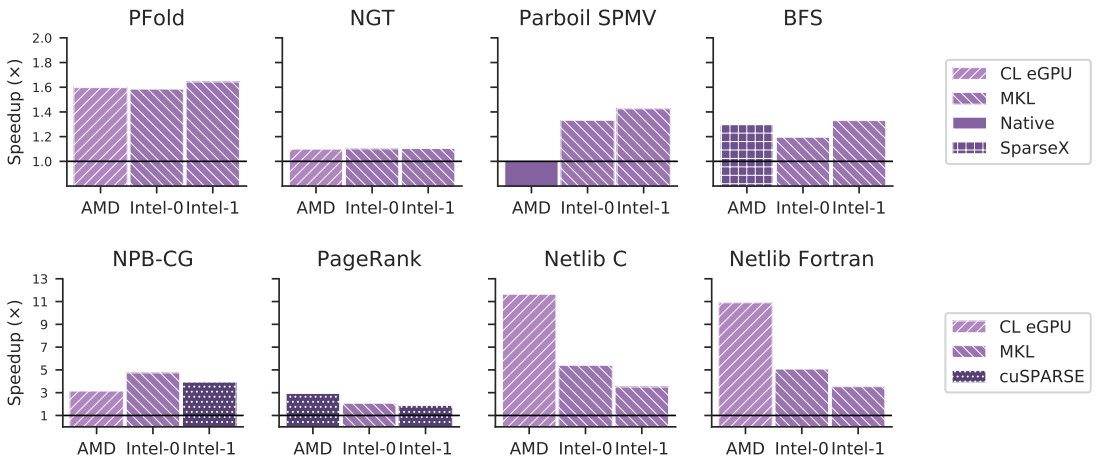
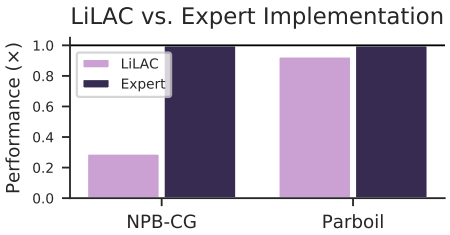


Figure 15. Evaluation on real-world applications and well-known benchmarks: Bars show geomean speedup of best-performing LiLAC harness across the set of input examples for each program and platform. Hatchings encode the selected implementations. The baseline is identical source code compiled with clang optimization level O2, resulting in sequential CPU code.



Benchmark	Modified LoC	
	LiLAC	Expert
NPB-CG	0 (44)	1948
Parboil SPMV	0 (44)	261

Figure 16. LiLAC performance as fraction of expert version performance. We achieve good speedup with no application programmer effort (measured as required LoC change). The LiLAC required code – identical across programs – is in parentheses. Amdahl's Law limits our impact on NPB-CG.

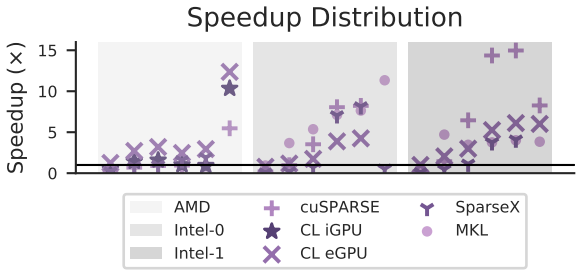


Figure 17. Distribution of speedups on NPB-CG. The stacks within each of the three columns are sorted by problem size, each point shows the speedup of a specific implementation.

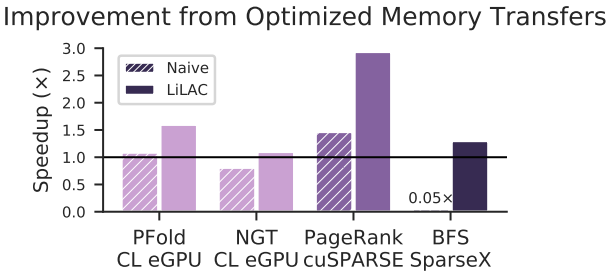


Figure 18. LiLAC vs. naïve library calls without marshaling optimizations, speedup over sequential baseline: Advanced marshaling features of LiLAC are critical for performance.

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 [31, 32, 54].

CPU-GPU data transfer optimizations Data transfers between CPU and GPU have been studied extensively as an important bottleneck for parallelization efforts. Previous work [24] established a system for automatic management of CPU-GPU communication. The authors of [28] 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 has presented LiLAC, a language and compiler that enables existing code bases to exploit sparse (and dense) linear algebra accelerator libraries. No effort is required from the application programmer. Instead, the library implementer provides a few lines of specification, which LiLAC uses to automatically and efficiently match user code to underlying 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. Many language features of the LiLAC language could be repurposed for libraries beyond linear algebra. 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

[1] [n. d.]. clMathLibraries clBLAS. https://github.com/clMathLibraries/clBLAS. ([n. d.]).

[2] [n. d.]. clMathLibraries clSPARSE. https://github.com/clMathLibraries/clSPARSE. ([n. d.]).

[3] [n. d.]. IntelÂő Math Kernel Library (MKL). https://software.intel.com/mkl. ([n. d.]).

[4] [n. d.]. NVIDIA cuBLAS. https://developer.nvidia.com/cublas. ([n. d.]).

[5] [n. d.]. NVIDIA CUDA Sparse Matrix library (cuSPARSE). https://developer.nvidia.com/cusparse. ([n. d.]).

[6] 2002. An Updated Set of Basic Linear Algebra Subprograms (BLAS). *ACM Trans. Math. Softw.* 28, 2 (June 2002), 135–151. https://doi.org/10.1145/567806.567807

[7] R. Baghdadi, U. Beaugnon, A. Cohen, T. Grosser, M. Kruse, C. Reddy, S. Verdoolaege, A. Betts, A. F. Donaldson, J. Ketema, J. Absar, S. v. Haastregt, A. Kravets, A. Lokhmotov, R. David, and E. Hajiyev. 2015. PENCIL: A Platform-Neutral Compute Intermediate Language for Accelerator Programming. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*. 138–149. https://doi.org/10.1109/PACT.2015.17

[8] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. 1991. The NAS Parallel Benchmarks&Mdash;Summary and Preliminary Results. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing (Supercomputing '91)*. ACM, New York, NY, USA, 158–165. https://doi.org/10.1145/125826.125925

[9] D. Baxter, R. Mirchandaney, and J. H. Saltz. 1989. Run-time Parallelization and Scheduling of Loops. In *Proceedings of the First Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '89)*. ACM, New York, NY, USA, 303–312. https://doi.org/10.1145/72935.72967

[10] Scott Beamer, Krste AsanoviÄĞ, and David Patterson. 2015. The GAP Benchmark Suite. (08 2015).

[11] Lam Chi-Chung, P Sadayappan, and Rephael Wenger. 1997. On optimizing a class of multi-dimensional loops with reduction for parallel execution. *Parallel Processing Letters* 7, 02 (1997), 157–168.

[12] B. Collie, P. Ginsbach, and M. F. P. O'Boyle. 2019. Type-Directed Program Synthesis and Constraint Generation for Library Portability. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 55–67. https://doi.org/10.1109/PACT.2019.00013

[13] Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.* 38, 1, Article 1 (Dec. 2011), 25 pages. https://doi.org/10.1145/2049662.2049663

[14] Camil Demetrescu, Andrew V Goldberg, and David S Johnson. [n. d.]. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*. Vol. 74. American Mathematical Soc.

[15] Johannes Doerfert, Kevin Streit, Sebastian Hack, and Zino Benaissa. 2015. Polly's Polyhedral Scheduling in the Presence of Reductions. *CoRR* abs/1505.07716 (2015). http://arxiv.org/abs/1505.07716

[16] Jack Dongarra, Victor Eijkhout, and Henk van der Vorst. 2001. An Iterative Solver Benchmark. *Sci. Program.* 9, 4 (Dec. 2001), 223–231. https://doi.org/10.1155/2001/527931

[17] Jonathan P. K. Doye, Mark A. Miller, and David J. Wales. 1999. The double-funnel energy landscape of the 38-atom Lennard-Jones cluster. *The Journal of Chemical Physics* 110, 14 (1999), 6896–6906. https://doi.org/10.1063/1.478595 arXiv:https://doi.org/10.1063/1.478595

[18] Athena Elafrou, Vasileios Karakasis, Theodoros Gkountouvas, Kornilios Kourtis, Georgios Goumas, and Nectarios Koziris. 2018. SparseX: A Library for High-Performance Sparse Matrix-Vector Multiplication on Multicore Platforms. *ACM Trans. Math. Softw.* 44, 3 (Jan. 2018), 26:1–26:32. https://doi.org/10.1145/3134442

[19] Allan L Fisher and Anwar M Ghuloum. 1994. Parallelizing complex scans and reductions. In *ACM SIGPLAN Notices*, Vol. 29. ACM, 135–146.

[20] Philip Ginsbach, Lewis Crawford, and Michael F. P. O'Boyle. 2018. CAnDL: A Domain Specific Language for Compiler Analysis. In *Proceedings of the 27th International Conference on Compiler Construction (CC 2018)*. ACM, New York, NY, USA, 151–162. https://doi.org/10.1145/3178372.3179515

[21] Philip Ginsbach, Toomas Remmelg, Michel Steuwer, Bruno Bodin, Christophe Dubach, and Michael F. P. O'Boyle. 2018. Automatic Matching of Legacy Code to Heterogeneous APIs: An Idiomatic Approach. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*. ACM, New York, NY, USA, 139–153. https://doi.org/10.1145/3173162.3173182

[22] Tobias Grosser, Armin Größlinger, and Christian Lengauer. 2012. Polly - Performing Polyhedral Optimizations on a Low-Level Intermediate Representation. *Parallel Processing Letters* 22, 4 (2012). https://doi.org/10.1142/S0129626412500107

[23] Gautam Gupta and Sanjay V Rajopadhye. 2006. Simplifying reductions.. In *POPL*, Vol. 6. 30–41.

[24] Thomas B. Jablin, Prakash Prabhu, James A. Jablin, Nick P. Johnson, Stephen R. Beard, and David I. August. 2011. Automatic CPU-GPU Communication Management and Optimization. *SIGPLAN Not.* 46, 6 (June 2011), 142–151. https://doi.org/10.1145/1993316.1993516

[25] Pierre Jouvelot and Babak Dehbonei. 1989. A unified semantic approach for the vectorization and parallelization of generalized reductions. In *Proceedings of the 3rd international conference on Supercomputing*. ACM, 186–194.

[26] Shoaib Kamil, Alvin Cheung, Shachar Itzhaky, and Armando Solar-Lezama. 2016. Verified Lifting of Stencil Computations. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, New York, NY, USA, 711–726. https://doi.org/10.1145/2908080.2908117

[27] Jeremy Kepner, David A. Bader, Aydin BuluÃğ, John R. Gilbert, Timothy G. Mattson, and Henning Meyerhenke. 2015. Graphs, Matrices, and the GraphBLAS: Seven Good Reasons. In *ICCS*.

[28] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. 2009. OpenMP to GPGPU: A Compiler Framework for Automatic Translation and Optimization. *SIGPLAN Not.* 44, 4 (Feb. 2009), 101–110. https://doi.org/10.1145/1594835.1504194

[29] Charith Mendis, Jeffrey Bosboom, Kevin Wu, Shoaib Kamil, Jonathan Ragan-Kelley, Sylvain Paris, Qin Zhao, and Saman Amarasinghe. 2015. Helium: Lifting High-performance Stencil Kernels from Stripped x86 Binaries to Halide DSL Code. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, New York, NY, USA, 391–402. https://doi.org/10.1145/2737924.2737974

[30] Mahdi Soltan Mohammadi, Tomofumi Yuki, Kazem Cheshmi, Eddie C. Davis, Mary Hall, Maryam Mehri Dehnavi, Payal Nandy, Catherine Olschanowsky, Anand Venkat, and Michelle Mills Strout. 2019. Sparse Computation Data Dependence Simplification for Efficient Compiler-generated Inspectors. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. ACM, New York, NY, USA, 594–609. https://doi.org/10.1145/3314221.3314646

[31] Ana Moreton-Fernandez, Arturo Gonzalez-Escribano, and Diego Ferraris. 2017. Multi-device Controllers: A Library to Simplify Parallel Heterogeneous Programming. (12 2017).

[32] Ana Moreton-Fernandez, Eduardo Rodriguez-Gutiez, Arturo Gonzalez-Escribano, and Diego R. Llanos. 2017. Supporting the Xeon Phi Coprocessor in a Heterogeneous Programming Model. In *Euro-Par 2017: Parallel Processing*, Francisco F. Rivera, Tomás F. Pena, and José C. Cabaleiro (Eds.). Springer International Publishing, Cham, 457–469.

[33] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. 2016. Automatically Scheduling Halide Image Processing Pipelines. *ACM Trans. Graph.* 35, 4, Article 83 (July 2016), 11 pages. https://doi.org/10.1145/2897824.2925952

[34] Dorit Nuzman, Sergei Dyshel, Erven Rohou, Ira Rosen, Kevin Williams, David Yuste, Albert Cohen, and Ayal Zaks. 2011. Vapor SIMD: Auto-vectorize Once, Run Everywhere. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '11)*. IEEE Computer Society, Washington, DC, USA, 151–160. http://dl.acm.org/citation.cfm?id=2190025.2190062

[35] Shlomit S Pinter and Ron Y Pinter. 1994. Program optimization and parallelization using idioms. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16, 3 (1994), 305–327.

[36] Bill Pottenger and Rudolf Eigenmann. 1995. Idiom recognition in the Polaris parallelizing compiler. In *Proceedings of the 9th international conference on Supercomputing*. ACM, 444–448.

[37] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. *SIGPLAN Not.* 48, 6 (June 2013), 519–530. https://doi.org/10.1145/2499370.2462176

[38] Gil Rapaport, Ayal Zaks, and Yosi Ben-Asher. 2015. Streamlining Whole Function Vectorization in C Using Higher Order Vector Semantics. In *Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium Workshop (IPDPSW '15)*. IEEE Computer Society, Washington, DC, USA, 718–727. https://doi.org/10.1109/IPDPSW.2015.37

[39] Lawrence Rauchwerger and David A Padua. 1999. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. *IEEE Transactions on Parallel and Distributed Systems* 10, 2 (1999), 160–180.

[40] Xavier Redon and Paul Feautrier. 1994. Scheduling reductions. In *Proceedings of the 8th international conference on Supercomputing*. ACM, 117–125.

[41] Y. Saad. 1989. Krylov Subspace Methods on Supercomputers. *SIAM J. Sci. Statist. Comput.* 10, 6 (1989), 1200–1232. https://doi.org/10.1137/0910073 arXiv:https://doi.org/10.1137/0910073

[42] Y. Saad. 2003. *Iterative Methods for Sparse Linear Systems* (second ed.). Society for Industrial and Applied Mathematics. https://doi.org/10.1137/1.9780898718003 arXiv:https://epubs.siam.org/doi/pdf/10.1137/1.9780898718003

[43] Daniele G. Spampinato, Diego Fabregat-Traver, Paolo Bientinesi, and Markus Püschel. 2018. Program Generation for Small-Scale Linear Algebra Applications. In *International Symposium on Code Generation and Optimization (CGO)*. 327–339.

[44] Daniele G. Spampinato and Markus Püschel. 2014. A Basic Linear Algebra Compiler. In *International Symposium on Code Generation and Optimization (CGO)*. 23–32.

[45] Kevin Stock, Martin Kong, Tobias Grosser, Louis-Noël Pouchet, Fabrice Rastello, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. 2014. A framework for enhancing data reuse via associative reordering. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 65–76.

[46] John Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Daniel Geng, Wen-Mei Liu, and Wen-mei Hwu. 2018. Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing. (Nov. 2018).

[47] Toshio Suganuma, Hideaki Komatsu, and Toshio Nakatani. 1996. Detection and global optimization of reduction operations for distributed parallel machines. In *Proceedings of the 10th international conference on Supercomputing*. ACM, 18–25.

[48] Arvind K. Sujeeth, Kevin J. Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. 2014. Delite: A Compiler Architecture for Performance-Oriented Embedded Domain-Specific Languages. *ACM Trans. Embed. Comput. Syst.* 13, 4s, Article 134 (April 2014), 25 pages. https://doi.org/10.1145/2584665

[49] Patricia Suriana, Andrew Adams, and Shoaib Kamil. 2017. Parallel Associative Reductions in Halide. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization (CGO '17)*. IEEE Press, Piscataway, NJ, USA, 281–291. http://dl.acm.org/citation.cfm?id=3049832.3049863

[50] Kyle H. Sutherland-Cash, Rosemary G. Mantell, and David J. Wales. 2017. Exploiting sparsity in free energy basin-hopping. *Chemical Physics Letters* 685 (2017), 288 – 293. https://doi.org/10.1016/j.cplett.2017.07.081

[51] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral Parallel Code Generation for CUDA. *ACM Trans. Archit. Code Optim.* 9, 4, Article 54 (Jan. 2013), 23 pages. https://doi.org/10.1145/2400682.2400713

[52] Richard Vuduc, James W Demmel, and Katherine A Yelick. 2005. OSKI: A library of automatically tuned sparse matrix kernels. *Journal of Physics: Conference Series* 16 (jan 2005), 521–530. https://doi.org/10.1088/1742-6596/16/1/071

[53] David J. Wales. 2002. Discrete path sampling. *Molecular Physics* 100, 20 (2002), 3285–3305. https://doi.org/10.1080/00268970210162691 arXiv:https://doi.org/10.1080/00268970210162691

[54] Linnan Wang, Wei Wu, Zenglin Xu, Jianxiong Xiao, and Yi Yang. 2016. BLASX: A High Performance Level-3 BLAS Library for Heterogeneous Multi-GPU Computing. In *Proceedings of the 2016 International Conference on Supercomputing (ICS '16)*. ACM, New York, NY, USA, Article 20, 11 pages. https://doi.org/10.1145/2925426.2926256

[55] Jie Zhao, Michael Kruse, and Albert Cohen. 2018. A Polyhedral Compilation Framework for Loops with Dynamic Data-dependent Bounds. In *Proceedings of the 27th International Conference on Compiler Construction (CC 2018)*. ACM, New York, NY, USA, 14–24. https://doi.org/10.1145/3178372.3179509