Data-Parallel Language for Correct and Efficient Sparse Matrix Codes

Gilad Arnold

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2011-142
http://www.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-142.html

December 16, 2011

Data-Parallel Language for Correct and Efficient Sparse Matrix Codes

by

Gilad Arnold

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Rastislav Bodík, Chair
Professor Koushik Sen
Professor Leo A. Harrington

Fall 2011

Data-Parallel Language for Correct and Efficient Sparse Matrix Codes

Abstract

Data-Parallel Language for Correct and Efficient Sparse Matrix Codes

by

Gilad Arnold

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Rastislav Bodík, Chair

Sparse matrix formats encode very large numerical matrices with relatively few nonzeros. They are typically implemented using imperative languages, with emphasis on low-level optimization. Such implementations are far removed from the conceptual organization of the sparse format, which obscures the representation invariants. They are further specialized by hand for particular applications, machines and workloads. Consequently, they are difficult to write, maintain, and verify.

In this dissertation we present LL, a small functional language suitable for implementing operations on sparse matrices. LL supports nested list and pair types, which naturally represent compressed matrices. It provides a few built-in operators and has a unique compositional dataflow model. As a result, LL programs are often direct analogs of high-level dataflow diagrams. This correspondence is useful when implementing sparse format construction and algebraic kernels such as sparse matrix-vector multiplication (SpMV). Despite its limited expressiveness, LL can specify interesting operations on real-world sparse formats.

Next, we present a full functional verifier for sparse codes written in LL. We use a higher-order logic theorem prover, deploying standard simplification and introduction techniques. We use novel parametric predicates for tracking relationships between mathematical objects and their concrete representation. A simple heuristic tactic is used for automating proofs of different sparse formats. A qualitative analysis shows that our rule base exhibits good reusability and that the system easily extends to hierarchical formats.

Finally, we describe a compiler for LL programs that generates efficient, parallel C code. We systematically map high-level nested datatypes onto compact and efficient low-level types. This facilitates a straightforward, syntax-directed translation of code. Local optimizations improve the performance of nested loops and eliminate redundant memory accesses. A coarse-grained heuristic loop parallelization scheme produces data-parallel code for execution on shared-memory multicore machines. Empirical evaluation shows that the sequential throughput of generated SpMV code is within 0.94–1.09 of a handwritten implementation. Parallel execution is 1.82–1.98 faster on two cores and 2.44–3.59 faster on four cores.

To Merav, my partner in this journey

To Erel, Roie and Nohar, who travel with us

The solutions all are simple—after you have arrived at them. But they're simple only when you know already what they are.

–Robert M. Pirsig

Contents

List of Figures

List of Tables

Acknowledgments

They say that thesis writing is the loneliest phase in a graduate student's life. This is generally true. Still, I'd like to thank a number of people who made the solitude bearable, and the research that preceded it fruitful.

I am deeply grateful to Ras Bodík, who advised, supported and (sometimes) pulled me through my graduate school endeavor. At times, advising can be tough and unrewarding, and students can be stubborn and unwieldy. My case was no exception. Ras took it all in good spirit and insisted on making the best of every situation, in his pleasant and persistent style. His sharp judgment, kindness and willingness to provide guidance are second to none, and I am truly appreciative of all I've learned from him during this time.

My graduate career owes a lot to Mooly Sagiv, my former advisor and mentor ever since. People as knowledgeable and eager to collaborate as Mooly are rare to find. The fact that he is such a great person makes it even better.

Special thanks goes to Johannes Hölzl, whose expertise in theorem proving and intellectual generosity turned an important part of this work into a reality. Our relatively brief collaboration led to interesting and surprising results, and I enjoyed it quite a bit. I'd also like to thank Ali Sinan Köksal, whose tremendous help in the early implementation stages of the LL compiler yielded our initial empirical results.

Several people provided invaluable feedback that helped me steer the work on this project. The list includes Jim Demmel, Mark Hoemmen, Marghoob Mohiyuddin, Michelle Strout, Bryan Catanzaro, Sam Williams, George Necula, Alex Aiken, Arthur Whitney, Lindsay Errington and Richard Waldinger. I also wish to thank Koushik Sen and Leo Harrington for serving on my dissertation committee and for their thoughtful advice.

While at Berkeley, I was fortunate to work with a number of brilliant researchers, who helped me focus my research interests and develop the necessary skills. Notable among them are Armando Solar-Lezama, Eran Yahav, Trey Cain, Jong-Deok Choi, Bill McCloskey, Adam Chlipala and Bor-Yuh Evan Chang. I had a great time sharing an office with Amir Kamil, Carol Hurwitz, Kaushik Datta and Andrew Gearhart, among others. Thanks to them it has been way more than plain research.

Berkeley is an amazing place and no words can express my gratitude to this community, which made it such an enjoyable ride. I am sending a huge, collegial "thank you" to all these wonderful individuals.

Finally, and most importantly, I wish to thank my family: my parents, Ora and Danny, for their love and guidance, and for always being there for me; my parents-in-law, Lea and Yossi, for their ongoing encouragement and support; my wife Merav and my children Erel, Roie and Nohar, for walking with me down this road, with absolute faith and endless love. I love you very much.

Chapter 1

Introduction

1.1 Overview

Sparse matrix formats encode very large numerical matrices with a low nonzero density in a way that makes them usable in later calculations. Computations with sparse matrices are at the core of many applications in science and engineering. Due to an inherent emphasis on efficiency and the way in which programming languages and computer systems have evolved, sparse codes have been developed using low-level programming languages such as Fortran and C. A number of techniques were developed for improving the throughput of sparse matrix kernels [34, 33]. This trend, together with an increasing number of base formats and an increasingly diverse set of target architectures and machine configurations, makes the implementation of new formats, as well as modification of existing ones, dauntingly complex.

This dissertation addresses the problem of writing correct and efficient sparse matrix programs. We introduce LL, a small functional language that is suitable for implementing operations on matrices, naturally and succinctly. We show how sparse codes written in LL can be fully and automatically verified, thanks to LL's limited expressive power and purely functional semantics. We present a compiler for LL program, which generates low-level code that is competitive with handwritten C programs and scales well when executed in parallel on multicore machines.

1.2 Challenges with sparse matrix implementation

Implementing efficient operations on sparse matrices (a.k.a., kernels) that are tuned for particular machines and workloads is a complex task. Even elementary compression schemes—such as the compressed sparse rows (CSR), jagged diagonals (JAD) and coordinate (COO) formats described in Section 2.4—are based on intricate representation invariants. Simple operations such as sparse matrix-vector multiplication (SpMV) require nested looping and

indirect dereferencing of arrays. This work proposes a new programming paradigm for sparse matrix kernels, which mitigates the implementation burden. It is our hope that more complex kernels—such as recent communication-avoiding iterative solvers [25, 45] that amount to hundreds of lines of C code—can be implemented naturally and succinctly via this approach.

The emergence of dedicated libraries of automatically tunable sparse matrix implementations [59, 60] has tremendously improved the availability of kernels to the end-user. At the same time, it delegated the development effort to seasoned experts, further imposing an artificial distinction between two kinds of experts: on the one hand, *domain experts* experiment with ideas pertaining to a particular application, and use powerful dynamic languages such as Python in creating unrefined prototypes. On the other hand, *programming experts* implement solid, carefully tuned codes with emphasis on reliability and performance, which dictates a choice of low-level, "bare bone" language such as Fortran or C. Our work aims to unify these seemingly contradicting trends by providing a clear mapping from high-level programs to low-level code with good performance characteristics. It also addresses parallelism, which adds another degree of complexity to traditional sparse matrix implementations [36].

While formal correctness used to get little attention by the scientific computing community, more work has been done in recent years on generating correct-by-construction sparse codes. This is in large part associated with the emergence of synthesis frameworks for sparse matrices—such as the sparse restructuring compiler by Bik et al. [6, 7] and Bernoulli [39, 38]—whose transformational approach is guaranteed to be semantics preserving (at least semi-formally). More recently, inductive synthesis approaches—such as the Sketch synthesizer [55, 54]—have relied on combinatorial decision procedures for fully verifying candidate programs. We observe that a tractable prover for sparse matrix programs is a prerequisite for building a rigorous synthesis framework for this domain: in the inductive case, it may allow one to scale beyond the capacity of combinatorial verifiers like SAT; in the deductive (i.e., transformational) case, a formal domain theory is a necessary foundation for discharging proof obligations during the synthesis process.[1] This work presents new results in automated formal verification of sparse matrix programs, building on the aforementioned high-level programming model.

The goal of this research is to increase the productivity of programmers of sparse matrix formats, allowing them to write concise, correct and efficient sparse matrix codes with ease. We wish to show that a carefully designed high-level language is suitable for achieving this goal.

[1] For an example of such framework designed for simpler domains, such as synthesizing divide-and-conquer algorithms, see [53]. The Denali superoptimizer [37] uses a first-order theorem prover [26] in an analogous fashion.

1.3 LL: language, verifier, compiler

In this thesis we present LL, a small functional language for implementing sparse matrix formats. Sparse matrix programs written in LL can be verified for full functional correctness using an automated prover. They can be compiled into parallel low-level C code, which is further compiled and executed efficiently on modern multicore architectures.

1.3.1 Implementing sparse formats using LL

We was designed to be powerful enough for describing interesting operations on a variety of known sparse formats, naturally and succinctly. The implementation of operations on sparse matrices in LL is distinguishable from a low-level, imperative implementation in several ways.

High-level datatypes and operators. LL has built-in pair and list types. We use lists for representing vectors, and nested lists for matrices. For sparse matrices, we use a combination of nested lists and pairs. For example, the compressed sparse rows (CSR) format is a list of compressed matrix rows, each of which is a list of pairs of column indexes and floating point values corresponding to nonzeros in the represented dense matrix. The following is an LL representation of a small CSR matrix with four rows, three of which are nonempty and contain a total of five values (denoted by letters) at different column offsets:

$$\begin{bmatrix} [\,(0,a)\,] \\ [\,(0,b)\ (1,c)\,] \\ [\,] \\ [\,(1,d)\ (3,e)\,] \end{bmatrix}$$

The tools for manipulating lists include maps, filters and reductions. The latter include only a small set of associative and commutative operations on sets, such as summation (arithmetic) and conjunction (Boolean). General reduction and recursion cannot be expressed in LL, which is key to enabling automated verification. We show, however, that the few supported operators are sufficient for expressing useful operations on a wide variety of formats. For example, the multiplication of a CSR matrix by a dense vector involves two maps—one traversing each compressed row, another for multiplying each nonzero value in a row with the vector elements that corresponds to its column position—and a summation of the products in each row.

Syntax-oriented dataflow model. The implementation of sparse matrix operations in LL closely resembles their high-level dataflow view. This simplifies mapping from a high-level diagram of a desired operation to a program that performs it. For example, the multiplication of a CSR matrix such as the above can be naturally described using

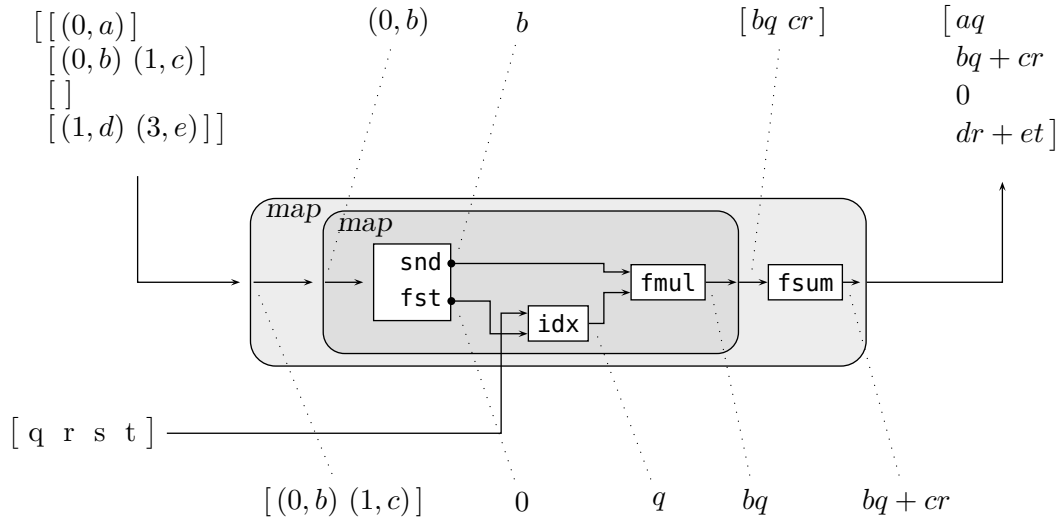


Figure 1.1: Dataflow view of high-level CSR SpMV

the high-level diagram in Figure 1.1. The functions `fst` and `snd` extract, respectively, the first and second components from a pair; `fmul` multiplies a pair of floating point values; and `fsum` adds a sequence of floats. It maps directly to the following LL code:

```
A -> [[(snd, (x, fst) -> idx) -> fmul] -> fsum]
```

The LL language is presented in Chapter 2, where we show how operations on complex sparse formats can be expressed easily and effectively.

1.3.2 Verifying high-level sparse matrix codes

We developed a framework for full functional verification of sparse matrix formats written in LL using a higher-order logic proof assistant, Isabelle/HOL [47]. We embed LL programs into Isabelle's typed lambda calculus, which allows us to leverage HOL's rich library of theories and tactics in proving their correctness. For the example CSR format, we take two LL functions—`csr` and `csrmv`—that implement, respectively, the construction of a CSR representation from a dense representation and multiplication of a CSR matrix by a dense vector; we then pose the following verification goal: "for all matrices A and vectors x, `csrmv`(`csr`$(A), x$) implements the dense multiplication Ax." Our framework is based on the following foundations:

Parametric representation relations. We use parametric predicates for tracking how a function (e.g., `csrmv`(`csr`$(A), x$)) represents a mathematical object (e.g., Ax) through

the deductive steps of the proof. These predicates can (i) express nested representation relations conveniently and concisely; and (ii) encapsulate intricate representation invariants, separate from the proof rules which use them. For example, *ilist* specifies a dense vector representation and *alist* a compressed vector representation using index-value pairs. We can then assert a CSR representation of some matrix as an "*ilist* representation, the elements of which are an *alist* representation whose elements are represented by equality (i.e., scalars)." Each predicate encodes a set of integrity constraints, such as ensuring that the length of data structures matches the dimensions of represented vectors and that all nonzero values are represented exactly once in a compressed vector representation.

Proof by simplification and introduction. In our sparse matrix domain theory, simplification rules are used for rewriting subterms in proof goals into a simpler form. For example, we simplify two consecutive maps, **`map`** f **`->`** **`map`** g into a single map with a composite inner functions, **`map`** **`(`**f **`->`** g**`)`**. However, simplification alone cannot encompass the full variety of compound functions. We therefore complement it with introduction rules, which break a given goal into a set of smaller subgoals. For example, we substitute a goal of the form "**`map`** f applied to A yields Ax" by "for all i, $f(A_i)$ yields the i-th element of Ax."

Specifically, the use of parametric predicates for capturing nested representation relations is the main novelty of our solution. We use automated tactics in discharging proof goals for even the most complex formats described in Chapter 2. The verification framework itself is presented in Chapter 3, where we evaluate its effectiveness and extensibility.

1.3.3 Compiling LL programs

We developed a compiler that translates high-level sparse matrix functions written in LL into efficient data-parallel C code. While we share common goals and techniques with previous efforts in this area—most notably, the NESL project [13, 12, 21]—we diverge from the preexisting paradigm in several ways:

Simple translation model. We present a systematic approach for mapping high-level LL types, such as compound objects with nested lists and pairs, into compact, reusable and efficient C types. This allows us to do most of the compilation work in a syntax-directed fashion, making our code generator straightforward while still producing C code with very low constant overhead.

Sequential optimization. We place great emphasis on eliminating unnecessary memory accesses along sequential execution paths and facilitating low-level loop optimizations. The techniques used include exploiting rich type information such as constant lengths of nested lists, fusion of reduction operators into preceding loops, and flattening of nested

list indexing. Our benchmarks indicate that the throughput of LL-generated sequential SpMV kernels is within 0.94–1.09 of that of a handcrafted reference implementation.

Coarse-grained parallelism. Unlike NESL, we refrain from vectorizing programs and opt for simpler, heuristics-guided loop parallelization. This approach appears to be well-suited for shared memory architectures such as multicore. Our benchmarks indicate that the speedup due to parallel execution on two cores is within 1.82–1.98 of the sequential case, and it is within 2.44–3.59 when executed on four cores.[2]

The LL compiler is presented in Chapter 4, along with benchmarks evaluating the effectiveness of generated sparse matrix kernels.

1.4 Related work

1.4.1 Implementing sparse matrix formats

The problem of generating sparse matrix codes has been addressed is several lines of work. Bik and Wijshoff [6, 7, 4, 5] pioneered research in synthesis of sparse matrix programs from dense specifications. Their compiler replaced dense access patterns such as A`[`i`][`j`]` with a representation function that maps to the corresponding value in a sparse representation. It then applied restructuring code transformations—such as iteration space reordering—to coalesce the code with a *compressed hyperplane storage* (CHS) format, which generalizes a number of simpler formats including CSR and CSC (see Section 2.4). The compiler supported a small set of simple built-in formats and could not extend to new and more complex ones.

The Bernoulli project [39, 38, 43, 1] is a system for synthesizing efficient low-level implementations of matrix operations given a description of the sparse format using relational algebra [56]. This permits rapid development of fast low level implementations, but requires understanding of the underlying theory, which limited it impact.

Siek et al. have developed a C++ library [52, 41] that provides a convenient abstraction for sparse matrix operations via generic programming. Using this library, and thanks to feature languages such as operator overloading and template instantiation, programmers can use idioms such as `y += A * x` in their code, where `A` is a sparse matrix and `x` and `y` vectors, which contain numerical values of some parametric type. Their implementation is part of the Boost C++ library and relies on generic graph algorithms for performing matrix reordering optimizations [42].

The Sparse Polyhedral Framework (SPF) [40] enables sophisticated iteration space and data access reordering. It is based on automatic generation of inspector/executor code during program compilation, which can then adjust the order of data access based on the input to the

[2]These speedups were observed after scaling down the native CPU frequency, to reduce the saturation of memory bandwidth.

program at runtime. These adjustments may increase data locality and, therefore, improve performance.

In LL, programmers express operations on sparse matrices directly, using a high-level functional programming language. The framework does not perform automatic transformations other than local sequential optimizations and parallelization of loops. The dataflow structure of the code generated by the LL compiler closely resembles that of the high-level program. This gives the user a clear and straightforward execution model for LL programs.

1.4.2 Language design

The design of LL was inspired by several predecessor languages. Compared to other functional languages, LL is pure, strongly typed, and has call-by-value semantics. Like SETL [51], APL [35] and Python, it has built-in support for operations on lists (vectors). Like FP/FL [2, 3], it constitutes a unique function-level programming methodology, also known as a point-free style [8, 31]. As in NESL [11, 12], programs are implicitly parallel and parallelized automatically. The programming methodology entailed by LL's dataflow model is reminiscent of visual programming environments such as LabVIEW, Max/MSP and others. That said, LL provides no visual interface per se, and can be used as any other text-based language.

1.4.3 Implicit data parallelism

The NESL line of work [13, 9, 9, 10, 12] pioneered the field of implicit data-parallel languages in the early 90's, demonstrating how complex algorithms on nested data structures can be expressed at a high-level of abstraction. Programs were then compiled to fully exploit the parallelism available on massively parallel vector machines of the era. We find many of the ideas embodied by NESL—such as the use of high-level constructs to express implicit parallel operations, and the flattening of nested data strucutres—to be extremely useful for high-level implementation of sparse kernels. However, our approach to parallel code generation diverges from that of NESL, and appears to work better on modern day multicore architectures. An in-depth analysis of the NESL parallelization model can be found in Section 4.2.2.

Data Parallel Haskell [18] is an effort to embed and augments the techniques introduced by NESL in a modern functional language. It requires language extensions—including extensions to the type system of both the source language [17, 16] and the intermediate representation [58]—in order to support vectorization. DPH is implemented as an extension to the widely used Glasgow Haskell Compiler. However, its development appears have stalled and it is not clear how applicable it is to the domain of sparse matrix formats.

The Intel Array Building Block (ArBB), formerly known as Ct [29, 30], is a C++ extension that provides an abstraction for parallel operations on segmented vectors. It is implemented as a template library and uses a standard C++ compiler for generating calls to a library of data parallel primitives. ArBB differs from LL (and other NESL-like languages) in the level of abstraction it provides to programmers: it does not support operations on

arbitrarily nested vectors, and it is up to the programmer to map nested list structures onto flat or segmented arrays. Hence, ArBB may serve as an intermediate representation for higher-level languages, including LL.

Copperhead [15] is a recent effort to embed a data-parallel code specializer in Python. The Copperhead compiler uses program annotations to identify parts of a Python program that can be compiled in a just-in-time fashion. It generates C++ code that is then executed by the Python runtime. Copperhead is currently targeting emerging GPGPU environments such as Nvidia's CUDA. While Copperhead can be used for implementing sparse matrix formats, it is a general purpose parallel language and not specifically designed for that purpose. Present limitations of the Copperhead language and compiler prevent it from achieving performance results that are comparable to hand-tuned low-level code (see Section 4.9).

1.4.4 Automated verification

We are not aware of previous work in which sparse matrix programs were successfully formally verified, let alone an automated full functional verification. General principles underlying our solution—in particular, the use of parametric representation relations—appear central to harnessing the power of higher-order verification for proofs on nested data representations. In one case, Duan et al. [27] verified a set of block ciphers using the interactive theorem prover HOL-4. They proved that the decoding of an encoded text results in the original data, relying heavily on inversion rules of the form $f\ (f^{-1}\ x) = x$. This approach does not seem applicable to the significantly more complex domain of sparse matrix programs.

1.5 Conclusion

This thesis shows how productivity of programmers in a particular domain can benefit from high-level programming models that were designed specifically for it. We show that a careful selection of language features, along with identification of important abstractions underpinning the representation of data in the particular domain of interest, are crucial to achieving strong correctness guarantees and generating efficient and scalable code. It is our hope that this work will break new grounds and lead to further developments in this direction.

Chapter 2

The LL Language

2.1 Overview

This chapter describes LL—short for *"little language"*—a strongly typed, point-free [8, 31] functional programming language designed for manipulating sparse matrix formats. LL borrows influences from FP/FL [2, 3], NESL [13], SETL [51] and APL [35], but favors simplicity and ease of programming over generality and terseness. LL provides several built-in functions and combinators for operating on vectors and matrices. The language is restricted by design, lacking custom higher-order functions, recursive definitions, and a generic reduction operator. These limitations of LL, as well as its purely functional semantics, facilitate automatic verification of sparse codes and allows rigorous optimization of low-level code that is generated from LL programs.

2.2 Introduction to LL

LL is a small, restricted language designed to easily describe operations on compound data objects, with emphasis on nested vectors and tuples. The design of LL aims to strike a balance between ease of use and applicability to the problem domain on the one hand, and amenability to verification and generation of efficient code on the other hand. We introduce the basics of LL by constructing a simple matrix-vector multiplication example.

2.2.1 Types

LL is statically and strongly typed. Natively supported primitive types include integer and floating-point numbers, and Boolean values. Compound types include pairs (or tuples, in their more general form) and lists (a.k.a. vectors, as they have built-in random access capability). The distinction between pairs and lists is an important one from the typing standpoint: while pairs can be composed of different types, a list is homogeneous. This

is similar to the distinction between product types and recursive types in strongly typed languages like ML and its derivatives. It deviates from the approach taken by FP, where a single heterogeneous aggregate is used for both purposes. This choice has proved essential to verification using a HOL-based theorem prover (see Chapter 3). It is also essential for the generation of efficient low-level code (see Chapter 4).

In the following, we denote primitive types by int, float and bool. Pair and tuple types are denoted by (τ_1, τ_2) and $(\tau_1, \ldots, \tau_k)$, respectively. List types are denoted by $[\tau]$. For example, a list of type [float] can be used for representing a vector of floating-point numbers. Note that compound types such as pairs and lists can contain arbitrary inner types, including pairs and lists. For example, a matrix can be represented by [[float]], namely a list of rows, where each row is a list of floating-point numbers. In general, there is no restriction on the length of a list, so nested lists can have varying lengths.

2.2.2 Syntax and dataflow

LL supports a point-free programming model in which variables are not being used. Instead, every function in the language assumes a single (possibly compound) input value and returns a single output value. Means for passing values between functions boil down to a handful of fixed higher-order operators (also called *combinators*):

Pair/tuple constructor. Denoted by `(`f_1`, ..., `f_k`)`, it feeds its single input to all functions f_i and constructs the tuple consisting of their respective outputs.

For example, suppose that `Ai` is a function that returns a row of a matrix of type [float] and `x` returns a vector of the same type. Then the following function returns a pair of vectors, ([float], [float]):

```
(Ai, x)
```

Function composition. Denoted by f `->` g, it passes the output of f as an input to g. An alternative notation g `(`f_1`, ..., `f_k`)` resembles an ordinary application syntax in many languages but is in fact shorthand for `(`f_1`, ..., `f_k`)` `->` g.

The `zip` built-in function takes a pair of lists and returns a single list containing pairs of corresponding elements. Composed with the pair constructor above, it returns a list of pairs of corresponding values in the two vectors, [(float, float)]:

```
(Ai, x) -> zip
```

Alternatively, the following form yields the exact same operation:

```
zip (Ai, x)
```

List operators. Denoted by **`map`** f and **`filter`** g. The former applies f to each of the elements of an input list and returns the list of outputs. The latter applies g of type $\tau \to$ bool to each of the input list's elements and returns the list of elements for which g yields *true*.

We can now turn our forming example into a complete vector inner product: having zipped the two vectors, we compute the product of each pair of values in the list by mapping with `fmul`, whose type is $(\text{float}, \text{float}) \to \text{float}$. We then sum the products using `fsum`, whose type is $[\text{float}] \to \text{float}$:

```
zip (Ai, x) -> map fmul -> fsum
```

Name binding. Denoted by l_1`, ...,` l_k`:` f. Provided an input tuple $(v_1, \ldots, v_k)$, this construct binds the label l_i to a constant function that returns v_1, and executes f in the new context. For example, `x, y: (y, x)` binds `x` and `y` to the first an second components of the input pair, respectively, and returns a swapped pair. The LL binding construct resembles a λ-abstraction in ordinary functional languages. Although it is a deviation from the strict pipeline-style dataflow model, we found it very useful when an input value is used multiple times in a sequence of computations, and when using values inside list comprehensions. From a practical standpoint, name binding also improves the efficiency of generated code compared to alternatives such as value replication and distribution over lists.

We can use name binding to wrap our inner product example with another map, which iterates over the rows of an input matrix `A`. Here `Ai` is bound to return the current row that is being iterated on by the outer map:

```
A -> map (Ai: (Ai, x) -> zip -> map fmul -> fsum)
```

While name binding can improve code clarity, in this case we can achieve the same functionality by using the `id` built-in function, which merely returns its input:

```
A -> map ((id, x) -> zip -> map fmul -> fsum)
```

However, name binding turns very useful if we do not assume that our input values—a matrix and a vector—are named. In this case, we can use a binding construct to assign the names `A` and `x` to the two components of a single input pair. Then we can use `x` inside the map that iterates over rows in `A`. This gives us a complete matrix-vector multiplication function.

```
A, x: A -> map ((id, x) -> zip -> map fmul -> fsum)
```

The dataflow model of LL programs tightly corresponds to their syntactic structure. With the exception of name binding, all dataflow edges are made explicit by the use of composition, pair construction and comprehension. All named values are defined prior to being used. As shown in Section 2.4, we rely on this correspondence when implementing sparse codes in LL.

2.2.3 Productivity

Several measures were taken to make it easy for programmers to use LL, without compromising the clean dataflow model.

Flexible syntactic convention. LL comes equipped with an alternative syntax for infix arithmetic and Boolean logic. For example, f `*.` g stands for scalar multiplication of two floating point values, and is shorthand for `(`f`, `g`) -> fmul`. LL also supports vector arithmetic operators, such as `fvmul` whose type is $([\text{float}], [\text{float}]) \to [\text{float}]$ and which computes a cross product of two vectors of floats. This function also has an infix version in the form of f `**.` g. In fact, it is sugar for `(`f`, `g`) -> zip -> map fmul`. We can rewrite the above matrix-vector multiplication as:

```
A, x: A -> map (id **. x -> fsum)
```

Or, equivalently:

```
A, x: A -> map (fsum (id **. x))
```

LL is also equipped with list comprehension sugar, which eases the processing of lists. The form `[`l_1`, ..., `l_k`: `f` ? `g`]` is shorthand for

```
filter (l1, ..., lk: f) -> map (l1, ..., lk: g)
```

Any of the two functions, f and g, can be omitted; in this case the comprehension will translate to either a filter or a map. Clearly, name binding is optional.

LL also supports a more verbose Python-style comprehension syntax of the form `[`g **for** l_1`, ..., `l_k **in** h **if** f`]`. This is equivalent to h `-> [`l_1`, ..., `l_k`: `f` ? `g`]`. Like Python, multiple bindings and conditionals are allowed in a single comprehension. This version of comprehension syntax slightly deviates from the LL design guidelines: the input h, the binding $l_1, \ldots, l_k$, the filtering predicate f and the map function g are

intermixed and appear out of order. Still, it provides a familiar construct for programmers adapting to LL's syntax. We believe that newcomers to LL will gradually appreciate the clarity and succinctness of LL's native constructs.

We can revise our matrix-vector multiplication example to use comprehension, as follows:

```
A, x: A -> [id **. x -> fsum]
```

Or, using Python-style comprehension and application-style composition:

```
A, x: [fsum (Ai **. x) for Ai in A]
```

Function and type definitions. LL supports user-defined functions and type definitions. Definitions are only permitted in the global scope. An exception to this rule is the definition of ad hoc constant functions via name binding (see above), whose scope is limited to their syntactically associated function.

LL has no support for higher-order functions, except for a fixed set of combinators. This limits the expressive power of LL—in particular prohibiting arbitrary recursion—but enables automated verification of programs. Although we were able to express a wide variety of sparse matrix formats, we may extend LL with additional higher-order constructs in order to support more operations on matrices in the future.

Implicit typing. LL deploys a Hindley-Milner [44] style global type inference for statically typing programs without relying on user annotations. This frees programmers from specifying the type of each and every syntactic construct in their code, while still providing the safety guarantees of a statically typed language.

LL permits specifying optional type constraints for bound names, including named function arguments. This is useful for imposing stricter type constraints that what the compiler would have otherwise inferred. An example for that is the annotation of fixed-length list inputs, covered in Section 4.6.1. It may also improve code readability and clarify programmers' intent.

Implicit parallelism. LL does not expose parallelism to the programmer. Instead, parallelism is implied by syntactic constructs such as list comprehensions and pair constructors. This paradigm follows other high-level languages like APL and NESL. It allows programmers to design algorithms with sequential semantics and have them later parallelized by the compiler.

We may extend LL with explicit parallel constructs in the future, to allow programmers more control on workload distribution and synchronization. This may be necessary as we apply LL to a larger variety of matrix computations.

2.3 Specification of LL semantics

The LL language constructs and their intended meaning are summarized in Table 2.1 and Table 2.2 (functions), and in Table 2.3 (actions). Every regular function (i.e., not a combinator) takes a single input and produces a single output. Some functions have an alternative syntactic form, shown in parentheses. For example, f `==` g is shorthand for `(`f`, `g`) -> eq`, and f `*.` g is the same as `(`f`, `g`) -> fmul`.

To keep the language and its type system simple, we avoid overloading of arithmetic functions for different types. Instead, we use distinct functions that operate on floating point and vector types, as well as corresponding infix notation. For example, `add` (or `+`) is used for integers, `fadd` (or `+.`) for floats, `vadd` (or `++`) for vectors of integers and `fvadd` (or `++.`) for vectors of floats. While it is generally possible to implement overloaded operations in a functional language—e.g., via type classes [61]—we defer this to later phase in the language implementation.

Note that tuples of labels in binding constructs are in fact generalized into arbitrarily nested structures. This allows greater flexibility in decomposing compound inputs, similarly to pattern matching constructs found in other functional languages like ML and Haskell. The following decomposes a pair of nested pairs, the first of which containing two floats, with a single binding construct:

```
((x :: float, y :: float), (h, w)): ...
```

In this section we suffice for an informal definition of the semantics of LL constructs. An unambiguous alternative definition is provided in Section 3.3 via translation of LL constructs into Isabelle/HOL's typed lambda calculus.

---

[1] `::`τi are optional annotations.

[2] n-ary tuples are treated as right-associated nested pairs.

[3] Floating-point and vector-level arithmetics are provided as `fadd` (`+.`), `vadd` (`++`), `fvadd` (`++.`), etc.

[4] Floating-point comparators are provided as `fleq` (`<=.`), etc.

[5] Floating-point and vector-level arithmetics are provided as `fsum` (`/+.`), `vsum` (`/++`), `fvsum` (`/++.`), etc.

[6] Sorting by floating-point keys provided as `fsort` and `frsort`, respectively.

[7] Tiling of two-dimensional lists provided as vblock : $(\text{int}, \text{int}, [[\tau]]) \to [[[[\tau]]]]$.

[8] Value naming is optional, f and h default to `id`, g defaults to `true`.

[9] **`if`** clause is optional; comprehension clauses can be chained as in Python.

[10] τ is optional annotation.

[11] τ_i are optional annotations.

Function	Type	Meaning
`id`	$\tau \to \tau$	The identity function
`eq` (`==`), `neq` (`!=`)	$(\tau, \tau) \to$ bool	Equality/inequality
3	$\tau \to$ int	Integer constant
3.1415	$\tau \to$ float	Floating-point constant
`true`, `false`	$\tau \to$ bool	Boolean constant (*true* or *false*)
f `?` g `|` h	$\tau \to \tau'$ where $f : \tau \to$ bool, $g : \tau \to \tau'$, $h : \tau \to \tau'$	Return output of g or h depending on output of f
l_1`::`τ_1`, ...,` l_k`::`τ_k`:` g [1]	$(\tau_1, \ldots, \tau_k) \to \tau'$ where $g : (\tau_1, \ldots, \tau_k) \to_{l_i:\tau_i' \to \tau_i} \tau'$	Bind each l_i to input tuple component of type τ_i, then apply g
l_1`, ...,` l_k `=` f`:` g	Same as: f `->` l_1`, ...,` l_k`:` g	
`(`f`)`	Same as: f	
`(`f_1`,` f_2`, ...,`f_k`)` [2]	$\tau \to (\tau_1, \ldots, \tau_k)$ where $f_i : \tau \to \tau_i$	Construct a tuple of values from the outputs of f_i
`fst`, `snd`	$(\tau_1, \tau_2) \to \tau_1$, $(\tau_1, \tau_2) \to \tau_2$	Extract the first/second component of a pair
f `->` g	$\tau \to \tau'$ where $f : \tau \to \tau''$, $g : \tau'' \to \tau'$	Evaluate g on the output of f
g `(`f_1`,...,`f_k`)`	Same as: `(`f_1`, ...,` f_k`)` `->` g	
g `'` f	Same as: `(`f`, id)` `->` g	
`add` (`+`), `sub` (`-`), `mul` (`*`), `div` (`/`), `mod` (`%`)	(int, int) $\to$ int	Arithmetic operators[3]
`leq` (`<=`), `lt` (`<`), `geq` (`>=`), `gt` (`>`)	(int, int) $\to$ bool	Comparators[4]
`sum` (`/+`), `prod` (`/*`)	[int] $\to$ int	Arithmetic reduction operators[5]
`and` (`&&`), `or` (`||`)	(bool, bool) $\to$ bool	Boolean operators
`neg` (`!`)	bool $\to$ bool	Boolean negation
`conj` (`/&&`), `disj` (`/||`)	[bool] $\to$ bool	Boolean reduction operators

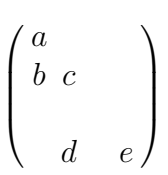
Table 2.1: LL functions: general, tuples, composition, arithmetic, Boolean logic

Function	Type	Meaning
`len`	$[\tau] \to \text{int}$	List length
`rev`	$[\tau] \to [\tau]$	Reverse a list
`idx` (`[]`)	$([\tau], \text{int}) \to \tau$	Extract an element
`vidx` (`{}`)	$([\tau], [\text{int}]) \to [\tau]$	Extract a list of elements
`distl` (`<<`), `distr` (`>>`)	$(\tau_1, [\tau_2]) \to [(\tau_1, \tau_2)]$, $(\tau_1, [\tau_2]) \to [(\tau_2, \tau_1)]$	Distribute a value from the left/right over a list
`zip`	$([\tau_1], [\tau_2]) \to [(\tau_1, \tau_2)]$	Merge lists into a list of pairs
`unzip`	$[(\tau_1, \tau_2)] \to ([\tau_1], [\tau_2])$	Break a list of pairs
`enum`	$[\tau] \to [(\text{int}, \tau)]$	Enumerate list elements
`concat`	$[[\tau]] \to [\tau]$	Concatenate lists
`infl`	$(\tau, \text{int}, [(\text{int}, \tau)]) \to [\tau]$	Inflate an associative list
`aggr`	$[(\tau_1, \tau_2)] \to [(\tau_1, [\tau_2])]$	Aggregate elements by key
`sort`, `rsort`	$[(\text{int}, \tau)] \to [(\text{int}, \tau)]$	Sort elements by ascending/descending key order[6]
`trans`	$[[\tau]] \to [[\tau]]$	Transpose a list of lists
`block` [7]	$(\text{int}, [\tau]) \to [[\tau]]$	Break a list into fixed-length sublists
`map` f	$[\tau] \to [\tau']$ where $f : \tau \to \tau'$	Apply f to each element
`filter` f	$[\tau] \to [\tau]$ where $f : \tau \to \text{bool}$	Filter elements by predicate f
`num` (`#`)	$\tau \to \text{int}$	Index of the current element
`[`l_1`, ..., `l_k `= `f`: `g ` ? `h`]` [8]	Same as: **`filter`** `(...:` g`)` `->` **`map`** `(...:` h`)`	
`[`f **`for`** l_1`, ..., `l_k **`in`** g **`if`** h`]` [9]	Same as: g `->` `[`l_1`, ..., `l_k`:` h `?` f`]`	

Table 2.2: LL functions: lists

Action	Meaning
`def` l`::`τ `=` f [10]	Bind l to function f with input type τ
`def` l `(`l_1`::`τ_1`, ..., `l_k`::`τ_k`) =` f [11]	Same as: **`def`** l `=` `(`l_1`::`τ_1`, ..., `l_k`::`τ_k`):` f
`type` l `=` τ	Bind l to type τ

Table 2.3: LL actions



(a) Dense matrix

R: [0 1 3 3 5]
J: [0 0 1 1 3]
V: [a b c d e]

(b) CSR sparse format

d = 2
P: [1 3 0 2]
D: [0 3 5]
J: [0 1 0 1 3]
V: [b d a c e]

(c) JAD sparse format

Figure 2.1: Low-level CSR and JAD sparse formats

2.4 Implementing sparse matrix codes using LL

We survey a breadth of sparse matrix formats and describe how they can be created and used in LL. Our study focuses on two routines: *construction* of the sparse format from a dense representation of two-dimensional list; and *multiplication* of the sparse matrix by a dense vector (SpMV for short). While the former is mostly used for illustrative purposes and for framing the verification problem (see Chapter 3), the latter is a pivotal computational kernel in numerous application ranging from scientific and high-performance computing, through data mining and information retrieval, to physical simulations.

In this section we contrast LL sparse format implementations with their imperative world equivalents. For efficiency reasons, low-level implementations insist on using flat, contiguous arrays of primitive values when representing compressed matrices. It is up to the programmer to handle the details of segment layout and dereferencing. Instead, in LL we can use nested lists with row-major semantics. Since nested lists can have varying lengths, they are equally suitable for representing both dense (i.e., cube) and sparse data. We exploit LL's native support for arbitrary nesting of lists and pairs in capturing hierarchical compression, naturally and concisely. In Section 4.4 we show how these high-level representations are translated to low-level ones that closely resemble hand-written, efficient data structures.

2.4.1 Compressed sparse rows (CSR)

This format compresses each row by storing nonzero values together with their column indexes. The resulting sequence of compressed rows is not further compressed, so empty (all zero) rows are retained. This enables random access to the beginning of each row, but requires linear traversal to extract a particular element out of a row. CSR is widely used because it is relatively simple and entails good memory locality for row-wise computations such as SpMV.

The low-level CSR representation of the example matrix in Figure 2.1(a) is shown in Figure 2.1(b). The arrays `J` and `V` are used for storing, respectively, the column indexes and values of non-zero matrix cells. The array `R` determines the beginning and ending position

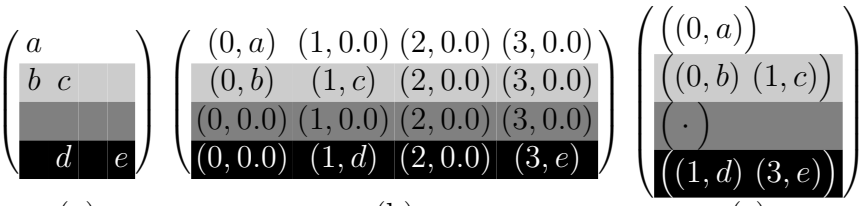
```
/* CSR construction. */
for (i = k = 0; i < m; i++) {
  for (j = 0; j < n; j++)
    if (M[i][j] != 0.0) {
      J[k] = j;
      V[k] = M[i][j];
      k++;
    }
  R[i] = k;
}
```

(a)

```
/* CSR SpMV. */
zero (y, m);
for (i = k = 0; i < m; i++) {
  for (y = 0; k < R[i]; k++)
    y += V[k] * x[J[k]];
  y[i] = y;
}
```

(b)

Figure 2.2: CSR construction and SpMV implementation in C



(a) (b) (c)

Figure 2.3: Conceptual phases in CSR construction

of each compressed row. For example, `R[0]` determines the beginning of the first row; `R[1]` determines the position right past the end of the first row, which is also the beginning of the second row; `R[4]` determines the end of the last row. Note that `R` pertains to segments in both `J` and `V`, as the two buffers contain corresponding components in a sequence of logical pairs.

Implementing CSR construction and multiplication in C—shown in Figure 2.2(a) and (b)—is not trivial. Nested loops are used for traversing either the dense matrix (construction) or the compressed index-value pairs (SpMV). Individual array cells are being dereferenced via single or nested indirection. Sparse row boundaries are stored and used explicitly in the code. For brevity, we omit memory allocation and initialization and assume that matrix dimensions are known at compile-time. On the positive side, we note that the resulting SpMV code is rather efficient, as the inner product of each row is computed incrementally in registers.

Figure 2.3 shows the high-level stages in CSR compression mentioned above. Given (a), each row is enumerated with column indexes, resulting in (b). Pairs containing a zero value

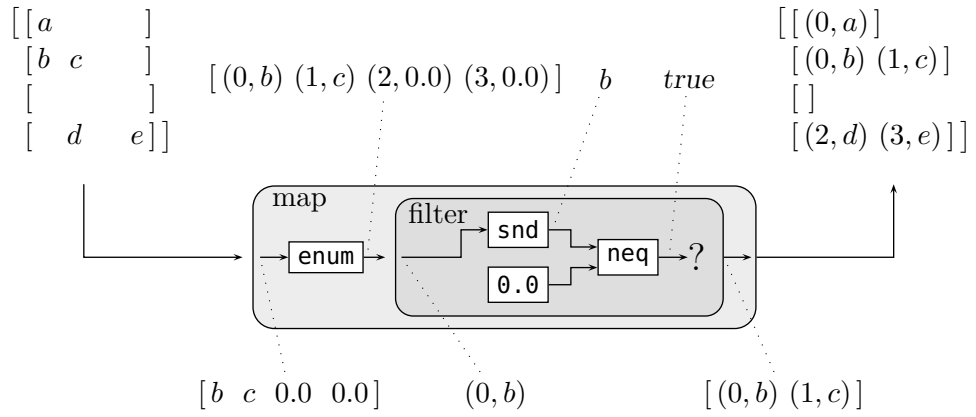


Figure 2.4: Dataflow view of high-level CSR construction

are then filtered, yielding (c). Figure 2.4 shows a dataflow view of such a process. This maps directly to the following LL function:

```
def csr = [enum -> [(snd, 0.0) -> neq ? ]]
```

Using name binding and infix notation may improve clarity, resulting in the following variant:

```
[enum -> [j, v: v != 0.0 ? ]]
```

Alternatively, one can use an explicit enumeration operator inside comprehensions. The following variant performs the index enumeration after filtering out zero values, but in fact entails the exact same semantics:

```
[[id != 0.0 ? (#, v)]]
```

Finally, a programmer may choose to use the more verbose Python-style comprehensions. The following variant is equivalent to the first definition above:

```
def csr (A) = [[(j, v) for j, v in enum(r) if v != 0.0] for r in A]
```

Figure 2.5 gives a high-level view of the stages in CSR multiplication by vector. We process each compressed row in isolation, as shown in (b). Within each row, each pair of index and nonzero is processed as in (c). We multiply the nonzero by the value of x corresponding to the column index of the nonzero, as in (d). The products of all nonzeros in a given row is summed, shown in (e). The final result is the output vector shown in (f). The high-level dataflow view of this computation is shown in Figure 1.1. It maps directly to the following LL function:

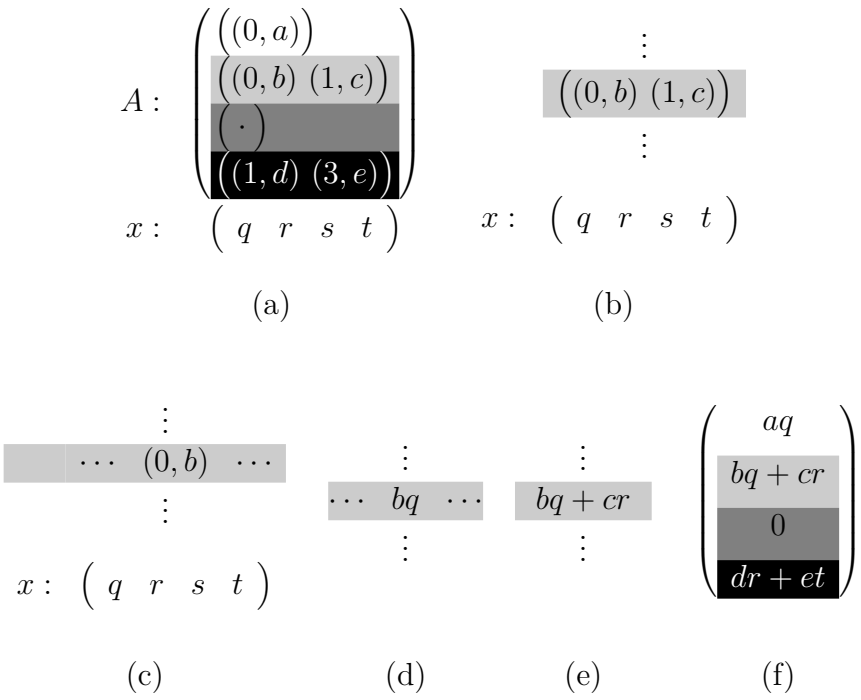


Figure 2.5: Conceptual phases in CSR SpMV: scalar operations

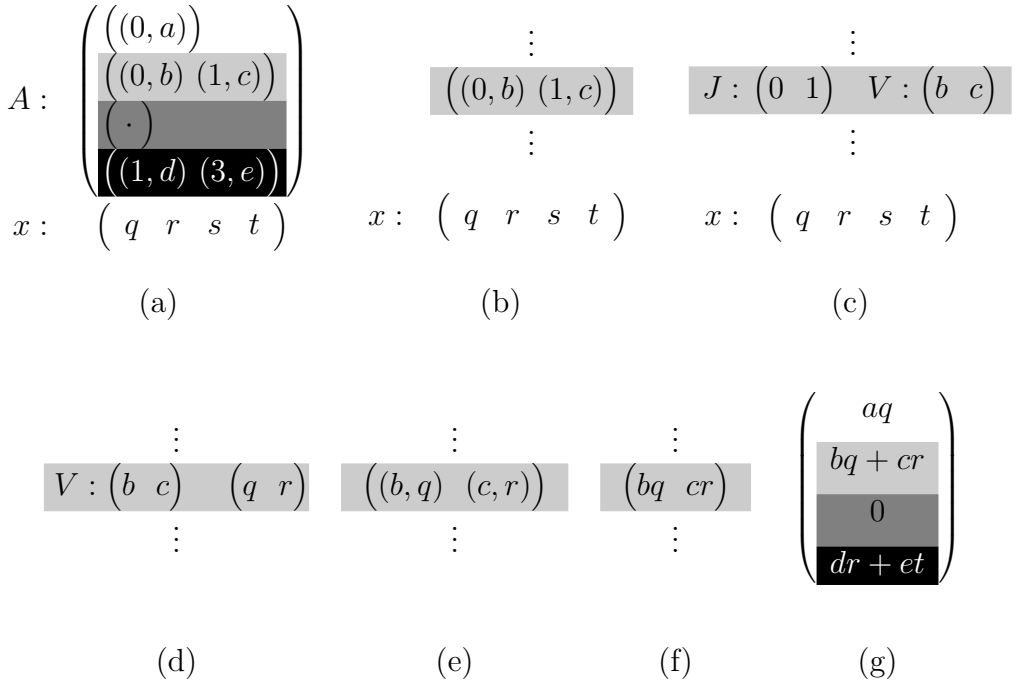


Figure 2.6: Conceptual phases in CSR SpMV: vector operations

```
def csrmv (A, x) = A -> [[(snd, (x, fst) -> idx) -> fmul] -> fsum]
```

Using an alternative operator notation and name binding for improved code clarity:

```
A -> [[j, v: v *. x[j]] -> fsum]
```

And using a more verbose Python-style comprehension:

```
[fsum ([v *. x[j] for j, v in r]) for r in A]
```

Figure 2.6 shows a different way to compute CSR SpMV, this time using more powerful operations on vectors. Each compressed row is multiplied separately, as shown in (b). First, column indexes are separated from nonzero values as in (c). They are used to retrieve corresponding values from x, shown in (d). The nonzeros and the retrieved x values are paired, as in (e). It is left to multiply pairs, as in (f), and sum the products, resulting in the inner product shown in (g). Such a computation can be expressed in LL as follows.

```
def csrmv (A, x) =
  A -> [J, V = unzip: (V, x{J}) -> zip -> [fmul] -> fsum]
```

More concisely, one can omit the name binding, and also deploy a vector cross product operator in lieu of the `zip` and `[fmul]` operations.

```
A -> [unzip -> snd **. x{fst} -> fsum]
```

Although semantically equivalent, this code may be more amenable to vectorization due to the use of vector-level operations such as gather and cross product.

CSR has been widely implemented in many sparse matrix frameworks. Since CSR SpMV is naturally data parallel, it is frequently used as a running example in parallel language research projects such as Bernoulli [39, 43], NESL [12], Data Parallel Haskell [18] and Copperhead [15]. Therefore, in subsequent chapters we will be paying great attention to the handling of CSR and its variants by the LL verifier and compiler framework.

2.4.2 Jagged diagonals (JAD)

This format deploys a clever compression scheme that allows handling of sequences of nonzeros from multiple rows, taking advantage of vector instructions. The ith nonzero values from all rows are laid out consecutively in the compressed format, constituting a "jagged diagonal." Since nonzeros are distributed differently in each row, column indexes need to be stored as well. However, packing ith elements in a predetermined order—e.g., from the first to the last row—induces a problem because some rows may contain less nonzeros than others. To compensate for this fact, rows are sorted by decreasing number of nonzeros prior to being stored as diagonals. The sorting permutation is stored together with the matrix, so the correct order of rows can subsequently be reconstructed.

The JAD representation of the example matrix in Figure 2.1(a) is shown in Figure 2.1(c). Here, too, the arrays `J` and `V` are used for storing corresponding column index and nonzero content values along the jagged diagonals. The array `D` determines the beginning and end boundaries of the two diagonals. The array `P` tracks the order of rows by which the diagonals are stored. The scalar `d` stores the number of diagonals stored.

Low-level imperative implementations of JAD fuse the aforementioned stages into a single loop nest, for better efficiency. This, however, complicates code comprehension, making it hard to analyze and maintain. For example, Figure 2.7(a) shows the code for compressing a dense matrix `M` into a JAD format, represented by `P`, `D`, `J` and `V`. It reads and writes a single word at a time, relies heavily on array indirections (i.e., array accesses whose index expressions are themselves array accesses), and explicitly spells out loop boundaries. There is no distinguishing between the three construction steps, thus it provides little insight into the nature of the JAD compression.

The aforementioned stages in JAD compression can be thought of as (i) compressing each row of the dense matrix (as in CSR); (ii) sorting compressed rows by decreasing length; and (iii) transposing the sorted compressed rows. These conceptual steps are visualized in Figure 2.8. From this diagram, it is easy to derive an LL function the looks as follows:

```
/* JAD construction. */
lenperm (M, P);
for (d = k = 0; d < n; d++) {
  kk = k;
  for (i = 0; i < n; i++) {
    for (j = nz = 0; j < m; j++)
      if (M[P[i]][j])
        if (++nz > d) break;
    if (j < m) {
      J[k] = j;
      V[k] = M[P[i]][j];
      k++;
    }
  }
  if (k == kk)
      break;
  D[d] = k;
}
```

(a)

```
/* JAD SpMV. */
zero (y, m);
for (dd = k = 0; dd < d; dd++)
  for (i = 0; k < D[dd]; i++, k++)
    y[P[i]] += V[k] * x[J[k]];
```

(b)

Figure 2.7: JAD construction and SpMV implementation in C

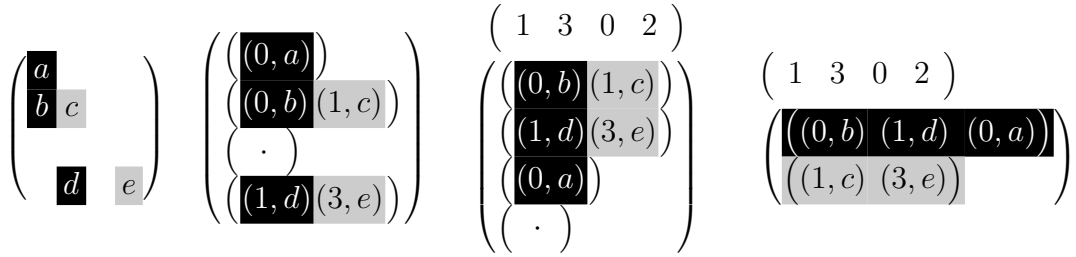


Figure 2.8: Conceptual phases in JAD construction

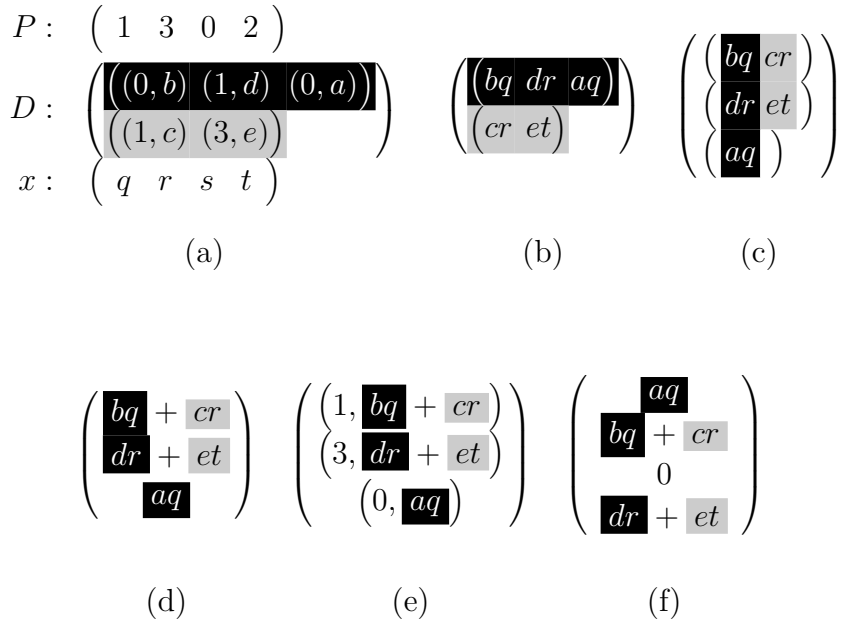


Figure 2.9: Conceptual phases in JAD SpMV

```
def jad = csr -> lenperm -> (fst, snd -> trans)
```

The first step compresses rows by invoking the constructor for CSR sparse format (see Section 2.4.1). In the second step, the function `lenperm` sorts the compressed rows by decreasing length, returning both the row permutation and the sorted rows:

```
def lenperm = [(len, (#, id))] -> rsort -> [snd] -> unzip
```

Note that the output of this compression, which is the same as the resulting object in Figure 2.8(d), is more abstract than the JAD format used by the C version of JAD in Figure 2.1(c). Whereas LL formats rely on nesting of lists and pairs, C formats linearize the outer list and create explicit indexing structures for accessing the inner lists. LL thus frees the programmer from reasoning about these optimized data structure layouts, eliminating dependence on explicit array indirection.

Figure 2.9 shows the high-level steps in JAD SpMV. (b) is obtained by computing, for each diagonal, the cross-product of its induced vector of values with the elements of x corresponding to their column indexes. These are transposed to obtain the lists of products in each (nonzero) row as in (c). Products along rows are summed, obtaining (d). In (e), each inner product is paired with its row index, which are then "inflated" to obtain the dense result vector in (f). The following LL function implements these steps:

```
def jadmv((P, D), x) =
  D -> [unzip -> snd **. x{fst}] -> trans -> [sum] ->
  zip (P, id) -> infl (0.0, m, id)
```

In comparison, consider the C version of JAD SpMV shown in Figure 2.7(b). Although it is arguably concise, it is also quite cryptic: the clean separation between phases is violated in favor of fusing the login: summation and permutation of the rows is done directly onto the output vector `y` via indirection; iteration along diagonals is done using a single, continuous index `k`. One hidden assumption is that the length of each diagonal—i.e., $\texttt{D[}j+1\texttt{]}-\texttt{D[}j\texttt{]}$—is not greater than the length of the permutation vector `P`; this assumption is made explicit in the LL version by the semantics of `zip`.

While JAD is of less practical interest due to the diminishing presence of large vector machines, it is still of interest from the specification and correctness point of view. We consider it particularly challenging due to the intricate use of multiple data layout transformations in the compression and decompression scheme, which has deemed brute force attempts at verifying it intractable. Thanks to posing it as a purely functional transformation, and using a special abstraction of various vector representations in HOL, we are able to automatically verify the correctness of JAD compression and SpMV (see Chapter 3).

2.4.3 Coordinate (COO)

COO is a portable compression scheme where nonzeros are stored together with both their row and column indexes in a single, arbitrarily ordered sequence. Its construction can be implemented by (i) enumerating rows with their indexes; (ii) for each row, enumerating its element with column indexes, then (iii) filtering out zeros while attaching row indexes to each tuple; and finally (iv) concatenating the resulting lists into a single one. In LL, this can be expressed as follows:

```
def coo =
  enum -> [i, r: enum (r) -> [j, v: v != 0.0 ? (i, j, v)]] -> concat
```

Explicit enumeration can be replaced for an integrated use of comprehension indexes, as follows:

```
[i = #: [v: v != 0.0 ? (i, #, v)]] -> concat
```

Alternatively, one can use a cascaded Python comprehension, as follows:

```
def coo (A) = [(i, j, v) for i, r in enum (A)
                         for j, v in enum (r) if v != 0.0]
```

COO multiplication is less straightforward: one needs to account for the fact that nonzeros of a particular row might be scattered along the compressed list. It is necessary to aggregate those values prior to computing the inner-product. This can be expressed as follows:

```
def coomv (A, x) =
  A -> aggr -> [i, r: (i, r -> [j, v: v *. x[j]] -> sum)] ->
  infl (0.0, m, id)
```

This is one case where a C implementation is relatively straightforward, attributed to the fact that COO has little structure and that it is a good fit for word-level operations. In the following, the arrays `I`, `J` and `V` store the row/column indexes and nonzero values, respectively:

```
/* COO SpMV. */
zero (y, m);
for (k = 0; k < nz; k++)
  y[I[k]] += V[k] * x[J[K]];
```

While COO is widely used as a portable storage format, it is rarely deployed directly by sparse kernels for most computational purposes due to its loose memory locality and excessive use of indexes. We therefore concentrate on its specification and correctness aspects only. As a future line of work, it may be interesting to consider COO as the starting point for various format constructions, instead of a dense matrix—such an approach will model more closely the actual use pattern that is implemented by various sparse matrix packages [57, 59].

2.4.4 Compressed sparse columns (CSC)

This is another standard format that is referred to in the literature [43] A CSC representation is obtained by compressing the nonzero values in the column direction, instead of row direction as in CSR. In C, it is done by swapping the order of the loops iterating over the dense matrix, and storing the row index with the nonzero values. In LL, it amounts to prepending a transposition to CSR construction.

```
def csc = trans -> csr
```

Like COO, CSC SpMV calls for aggregation prior to summing row cross-products. The following function (i) zips compressed columns with their corresponding vector elements; (ii) for each pair of column and vector element, multiply nonzeros by this element, a list of pairs of row index and a product value; (iii) concatenate all pairs, then (iv) aggregate all products belonging to the same row; finally, (v) inflate into a dense vector form.

```
def cscmv =
  zip -> [cj, xj: cj -> [i, v: (i, v *. xj)]] ->
  concat -> aggr -> [(fst, snd -> fsum)] -> infl (0.0, m, id)
```

Here, too, the fact that data layout is not in line with the computation entailed by matrix-vector multiplication calls for additional steps to massage the result into a proper vector form. A typical C implementation takes advantage of direct assignment to array elements for incrementing the output vector, and is omitted here.

2.5 Hierarchical formats

This section evaluates the usability of LL for expressing sparse formats with hierarchical compression schemes, providing insights on its applicability to complex real-world sparse kernels. We first present SCSR, which further removes empty rows from a CSR representation. We then present two blocking schemes—cache blocking and register blocking—that improve performance of SpMV kernels by improving temporal locality at two different levels of the memory hierarchy. Locality is improved by reorganizing the computation to operate on smaller segments of the input matrix, which in turn allows the reuse of memory segments containing the vector for multiplication.

2.5.1 Sparse CSR (SCSR)

The SCSR format extends CSR with another layer of compression. It further compresses the list of compressed rows by filtering out empty rows, namely rows that only contain zeros. The remaining rows are associated with their row index. SCSR is beneficial in cases where the nonzeros in the matrix are not uniformly distributed across rows.

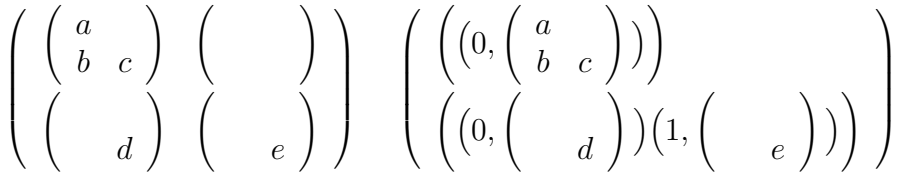
Implementing SCSR in LL amounts to obtaining the CSR format, which compresses individual rows, followed by compression of the resulting list of compressed rows. Again, LL manages to express format construction as a pipeline of stages.

```
def scsr = csr -> [len != 0 ? (#, id)]
```

The corresponding SpMV implementation needs to account for the row indexes. It must also inflate the resulting sparse vector into dense format:

```
def scsrmv(A, x) =
  A -> [(fst, snd -> unzip -> snd **. x{fst} -> fsum)] ->
  infl (0.0, m, id)
```

Alternatively, we can reuse SpMV for CSR:



(a) (b)

Figure 2.10: Dense and compressed 2×2 block matrix representation

```
A -> unzip -> (fst, csrmv (snd, x)) -> zip -> infl (0.0, m, id)
```

SCSR is a good example of LL's advantage when composing compression schemes. Compare the above with what a similar implementation effort would look like if done in a language like C. This is true even when a programmer merely extends an existing CSR implementation: (i) they would need to figure out how to extend the low-level representation in Figure 2.1(b) to accommodate the additional index, and how the filtering of empty rows affects the content stored in the existing data structure; (ii) they would need to intrusively extend a CSR construction to implement storing of row indexes and the elimination of empty rows; (iii) similarly, they would need to deploy an additional indirection when updating the output vector with the products of a particular row during SpMV. All this amounts to a non-trivial implementation effort. For the most part, it is the realization of how the low-level data layout and its semantics change that poses a burden to programmers' productivity. LL frees programmers from such consideration, letting them focus on data transformation instead.

2.5.2 Register blocking

This optimization was introduced in [60]. It is particularly useful when the nonzero values in the matrix appear in small clusters (e.g., tridiagonal). The idea is to treat it as a matrix of small dense matrices by partitioning it into uniformly sized rectangular blocks. These blocks are the elements that are then being compressed—a zero block is one whose values are all zeros. The size of these blocks is chosen so that the corresponding portion of the vector for multiplication can reside in registers during processing of a block. Register-blocked formats such as RBCSR have shown to significantly speed up throughput of SpMV kernels when tuned for proper block sizes [34, 59].

Register blocking can be applied to a variety of sparse formats described in Section 2.4. The 2×2 blocked representation of Figure 2.1(a) can be seen in Figure 2.10(a). Applying CSR compression to this blocked matrix results in the register-blocked CSR format in Figure 2.10(b).

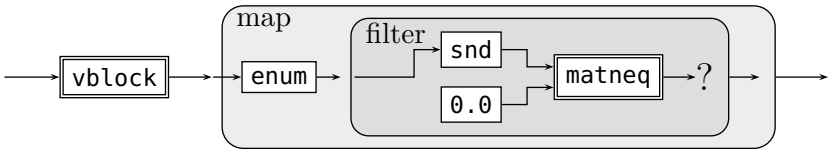


Figure 2.11: Dataflow view of RBCSR construction

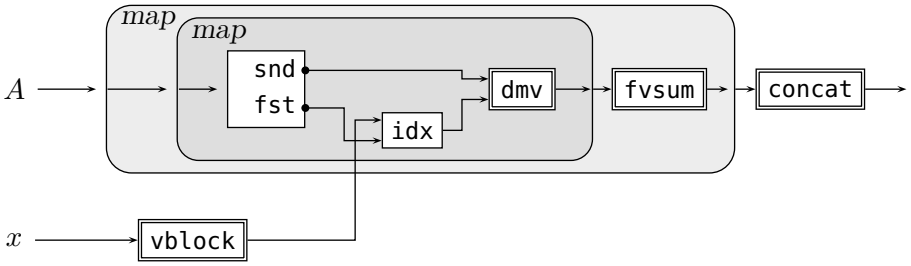


Figure 2.12: Dataflow view of RBCSR SpMV

Constructing a RBCSR representation in LL is just a slight deviation from the ordinary CSR construction shown in Section 2.4.1. It requires (i) partitioning the matrix into fixed size blocks; and (ii) replacing the scalar zero test with a variant for a dense matrix (nested list of values). The block dimensions $\texttt{r} \times \texttt{c}$ are assumed to be globally bound values. The relative change compared to the dataflow view shown in Figure 2.4 is shown in Figure 2.11 with new or different operations marked by a double line. The corresponding LL implementation looks as follows.

```
def rbcsr (A) = vblock (r, c, A) -> [enum -> [(snd, 0.0) -> matneq ? ]]
```

The function `matneq` is implemented as follows.

```
def matneq (B, x) = B -> [[id != x] -> disj] -> disj
```

Multiplication of an RBCSR format is also quite similar to that of ordinary CSR, with the following exceptions: (i) the vector for multiplication needs to be partitioned into blocks of size c; (ii) scalar multiplication is replaced by dense matrix-vector multiplication, and scalar summation by vector summation; and (iii) the overall blocked result needs to be concatenated into a single, flat vector. The dataflow view of this computation and the relative difference from the CSR SpMV shown is Figure 1.1 are shown in Figure 2.12 with new or different operations marked by a double line. A corresponding LL implementation looks as follows.

Rb: [0 1 3]
Jb: [0 0 1]
Vb: [a 0.0 b c 0.0 0.0 0.0 d 0.0 0.0 0.0 e]

Figure 2.13: Low-level RBCSR representation

```
zero (y, m);
for (i = 0; i < m / 2; i++, y += 2) {
  double y0 = 0, y1 = 0;
  for (k = Rb[i]; k < Rb[i + 1]; k++, Jb++, Vb += 4) {
    int j = 2 * Jb[0];
    double x0 = x[j], x1 = x[j + 1];
    y0 += Vb[0] * x0;
    y1 += Vb[2] * x0;
    y0 += Vb[1] * x1;
    y1 += Vb[3] * x1;
  }
  y[0] = y0; y[1] = y1;
}
```

Figure 2.14: 2×2 RBCSR SpMV implementation in C

```
def rbcsrmv (A, x) =
  xb = block (c, x):
    A -> [[(snd, xb[fst]) -> dmv] -> fvsum] -> concat
```

The function `dmv`, which computes a dense matrix-vector multiplication, was introduced in Section 2.2, and looks as follows:

```
def dmv (A, x) = A -> [id **. x -> fsum]
```

It is interesting to contrast this high-level approach with the how it is implemented in C. A first challenge faced by the programmer is figuring out how data will be laid out in flat arrays and what are the invariants that characterize the data structure. Contingent with the continuous buffer approach, an array `Vb` contains the nonzero dense blocks laid out consecutively in a row-major fashion. A separate array `Jb` holds the position indexes of *blocks*. A third array `Rb` contains the beginning and end boundaries of the compressed rows of blocks. A low-level representation of the abstract RBCSR matrix in Figure 2.10(b) is shown in Figure 2.13. Note that it encapsulates the implicit assumption about the block

size: since it is known to the programmer at compile time, no explicit boundary markers are necessary to determine where each dense block in `Vb` starts and ends. This specialization needs to be matched by the code that traverses this data structure.

Figure 2.14 shows a C implementation of RBCSR SpMV. It was adapted with minimal alteration from the OSKI [59] autotuning framework, also described in [60]. It is interesting to note the degree of specialization that this code exhibits: there are only two loops, but in fact the innermost loop body represents two nested loops computing the dense block multiplication, which have been fully unrolled by the programmer. The aforementioned assumptions about block size are embedded as constants, in the form of fixed strides used in advancing pointers and computing offsets.

We will further examine RBCSR in the context of verification and (in particular) compilation and generation of efficient code. The implications of dealing with nested dense matrices to verification are discussed in Section 3.7.5. The challenges in generating efficient data structures for block types, and how code generation can take advantage of such specialization, are discussed in Section 4.4 and Section 4.6.

2.5.3 Cache blocking

The idea in cache blocking is to reduce cache misses for the source vector x when it is too large to entirely fit in cache during SpMV. We consider *static cache blocking* [33]. Here, too, the matrix is partitioned into rectangular sub-matrices of size $r \times c$. However, unlike register blocking, these sub-matrices are themselves compressed. The lengths r and c are chosen such that a portion of x of length c can reside in cache while it is multiplied by a matrix block.

Our cache blocking scheme differs from the one in [33] in that blocks start columns are multiples of c. This restriction is due to LL's limited expressiveness, and might induce sub-optimal space utilization. It is possible to relax it by augmenting LL so that it is expressive enough to accommodate optimal block placement (e.g., by adding a built-in function). This is beyond the scope of this work and has not been attempted.

The construction of a cache-blocked matrix is very similar to construction with register blocking. The only difference is the additional compression of each block. The LL code for constructing a cache-blocked CSR representation whose blocks are stored in CSR format is shown below.

```
def cbcsr (A) =
  vblock (r, c, A) ->
  [enum -> [(snd, 0.0) -> matneq ? (fst, snd -> csr)]]
```

We again notice the similarity of cache-blocked SpMV with the register-blocked version. The difference is in the function used for multiplying a block by a vector. The LL implementation of cache-blocked SpMV follows.

```
def cbcsrmv (A, x) =
  xb = block (c, x):
    A -> [[(snd, xb[fst]) -> csrmv] -> fvsum] -> concat
```

Cache-blocked formats are not yet covered by our verification framework. Based on our experience with other formats and the analysis in Section 3.8, we expect that it is doable with relatively little effort.

2.6 Discussion

As we saw in this chapter, LL lifts tedious low-level data representations and intricate imperative computation into a cleaner functional form, exposing coarser-grained computation stages and the flow of data from one stage to another. This encourages the programmer to think about intermediate results as cohesive and consistent objects, and to associate them with semantic invariants. LL is also well suited for composing transformations, as shown with the hierarchical formats. It is generally possible to implement even more deeply nested compression schemes—e.g., nest register blocking within a cache-blocked matrix—although we did not attempt to do so.

These benefits are not merely due to the use of functional programming. We believe that they are equally attributed to our careful selection of a very simple subset of functional language features, designed with the sparse matrix domain in mind. In particular, the absence of a direct notion of a "value" encourages expressing computations as composition of functions. Unlike strict point-free languages such as FP [2], we support name binding as an alternative means for distributing values over lists. This is important for convenience of expression, as well as generation of efficient code (see Chapter 4). Other approaches for generating sparse matrix implementations are discussed in Section 1.4 (p. 6).

There are limitations to what can be done in LL. LL excludes definitions of recursive functions and a general fold operator, both of which are compensated for by a versatile set of built-ins (e.g., `zip` and `sum`) and combinators for handling lists (e.g., **`map`** and **`filter`**). These restrictions contribute to our ability to automatically verify LL programs because they sidestep the need to infer induction hypotheses, which is a hard task for automated tools (and humans).

It may be desirable to be able to use a restricted form of higher-order functions when implementing sparse matrix codes. For example, the implementation of RBCSR and CBCSR SpMV are identical except for the operators used for multiplying blocks—it makes sense to replace these variants (`dmv` and `csrmv`) by a function argument and unify the two implementations. We have refrained from enriching LL with first-order functions for now because this simplifies automatic verification and gives us broader verification coverage. It may be beneficial to extend LL with a template mechanism as a substitute for fully fledged higher-order functions, which will allow better code reuse in similar situations.

A compiler from LL to C is presented in Chapter 4. Its implementation is yet experimental and currently supports only a handful of sparse kernels, including SpMV of CSR and register-blocked CSR formats. Further implementation effort is necessary for supporting other useful formats, such as cache-blocked CSR, JAD and SCSR. This includes broadening the implementation of the built-in LL function library, and extending coverage of the different analyses described in Chapter 4 to these functions as well.

Chapter 3

Verifying High-Level Sparse Codes

3.1 Overview

In this chapter we present a verification framework for sparse matrix kernels written in LL. Having framed sparse format construction and SpMV as functional programs (see Chapter 2), we use Isabelle/HOL [47] to verify their full functional correctness. Our solution is based on (i) parametric representation predicates that are able to capture structural and arithmetic invariants of nested sparse formats; (ii) a sparse matrix theory consisting of simplification and introduction rules for reasoning about the various operations used in sparse matrix kernels; (iii) a tactic for applying rules that can automatically prove the correctness of different formats. We demonstrate this approach with the CSR format. We then shown how to extend it to other formats, including JAD, COO, and hierarchical formats like SCSR and register blocking. Finally, we evaluate the generality and extensibility of our domain theory, projecting on the feasibility of verifying new formats.

3.1.1 Motivation

There are at least two arguments for full functional verification of sparse matrix codes. First, classical static typing is insufficient for static bug detection because these programs contain array indirection, whose memory safety would be typically guaranteed only with run-time safety checks. Dependent type systems may be able to prove memory safety but, in our experience, the necessary dependent-type predicates would need to capture invariants nearly as complex as those that we encountered during full functional verification. For example, to prove full functional correctness, one may need to show that a list is some permutation of a subset of values in another list; to prove memory safety, one may need to show that the values in a list are smaller than the length of another list. Therefore, it seems that with little extra effort, we can use theorem proving to extend safety to full functional correctness.

The second reason for full functional verification is synthesis of sparse matrix programs, including the discovery of new formats. This is applicable to deductive synthesis—

e.g., [53, 37]—where a new implementation is derived from a reference implementation by repeated application of transformations rules in a way that preserve the overall semantic of the program. The rules we develop as part of our domain theory can serve for this purpose, subject to finding proper ways to effectively explore the space of provably correct implementation variants. It is also applicable to inductive synthesis—e.g., [55]—where a new implementation is found by searching a space of (potentially incorrect) candidates and attempting to find one that fully complies with the specification. Here, a full functional verifier is a prerequisite for synthesis because it is an arbiter of correctness of the selected implementation. The application of our domain theory to synthesis of sparse kernels is outside the scope of this work. We discuss ideas for further work in this direction in Chapter 5.

3.1.2 Preliminary assumptions

In proving the correctness of LL programs in Isabelle/HOL we are making the following assumptions as a baseline for the verification problem.

Correctness criteria. For each of the formats we consider with construction function f_{const} and multiplication by vector f_{spmv}, we frame the following verification goal: for all dense matrices A and vectors x, $f_{\text{spmv}}(f_{\text{const}}(A), x)$ represents the mathematical value $A \cdot x$. Namely, we prove that f_{spmv} computes the correct result with respect to the compression performed by f_{const}. In effect, our prover uses the user-provided constructor f_{const} to automatically infer the representation invariant of the format at hand, freeing the user from formalizing this invariant in HOL. That said, framing the problem as such can be somewhat restrictive, especially since the compression of dense matrices is rarely done in practice: since dense matrices are often prohibitively large to store and process, most real-world applications of sparse kernels use the coordinate format (COO) as the basis for the construction of other formats. We believe that our framework can be adapted to such a use case scenario by encoding (or inferring) the representation invariant of a COO representation, and requiring from the user a function f'_{const} that constructs the desired format from the COO representation. This is left for future work.

Natural number arithmetic. Our proof rules are currently limited to reasoning on natural number. The reason is that there is built-in support for it in Isabelle/HOL, and that for the purpose of matrix multiplication it is an acceptable proxy for other algebraic domains (e.g., integers and real numbers). Clearly, verifying correctness of true floating point computations calls for a separate set of methodologies and tools, and is beyond the scope of this work.

Sequential semantics. Isabelle/HOL models sequential execution of functions written in typed λ-calculus programs. Although we consider LL programs to represent parallel

computations, this is a reasonable assumption to make because the implicit parallelism in LL has no effect on the overall semantics of programs.

3.2 Introduction to sparse format verification

Let us use the simple CSR format to give the rationale for the design of our proof system. Suppose that A and x are concrete language objects that, respectively, contain dense representations of a mathematical matrix B and a vector y. We want to prove that the product of the CSR-compressed A with x produces an object that is a valid (dense) representation of the vector $B \cdot y$. Note that the multiplication is CSR-specific. Formally, our verification goal is

$$\texttt{csrmv}(\texttt{csr}(A), x) \overset{m}{\triangleright} B \cdot y$$

The goal expresses the relationship between a mathematical object and its concrete counterpart with the *representation relation* $a \overset{k}{\triangleright} b$, which states that the concrete object a represents the mathematical vector b: the lengths of a and b are k and for all $i < k$, $a\texttt{[}i\texttt{]}$ equals b_i. In the course of the proof, we may need to track relationships on various kinds of concrete objects. A key contribution of our verification framework is the definition of suitable representation relations for the objects that arise in sparse matrix programs.

We use Isabelle/HOL, in interactive theorem prover for higher-order logic, as our underlying prover. It is an LCF-style theorem prover in which all parts of a proof are checked from first principles by a small trusted kernel. We embed LL functions in Isabelle using typed λ-calculus and Isabelle libraries. Our proofs deploy two techniques: (a) *term simplification*, which rewrites subterms in functions into simpler, equivalent ones; and (b) *introduction*, which substitutes a proof goal with a certain term for alternative goal(s) that do not contain the term, and whose validity implies the validity of the original goal. In our example, term simplification unfolds the definitions of `csrmv` and `csr` and applies standard rewrite rules for simplifying function application and composition, map and filter operations on lists, and extraction of elements from pairs. This results in the following goal:

$$\texttt{[enum -> [snd != 0 ?] -> [snd} * x\texttt{[fst]] -> sum]}(A) \overset{m}{\triangleright} B \cdot y \tag{3.1}$$

The LL function on the left enumerates each row of A into a list of column index and value pairs, then filters out pairs whose second element is zero (`[snd != 0 ? ]`). For the remaining pairs, it multiplies the second (nonzero) component with the value of x at the index given by the first component (`[snd` $*$ x`[fst]]`). Finally, it sums the resulting products (`sum`). So far, simplification has done a good job.

To carry out the next step of the proof, we observe that the missing zeros do not affect the result of the computation, so we would like to simplify the left-hand-side by rewriting away the filter (`[snd != 0 ? ]`); this would effectively "de-sparsify" the left-hand side, moving it closer to the mathematical right-hand-side. Unfortunately, standard simplification available

in prover libraries cannot perform the rewrite; we would need to add a rule tailored to this format. The hypothetical rule, shown below, would match p with `snd != 0` and f with `snd *` x`[fst]`:

$$\frac{\forall y \;.\; \neg p(y) \longrightarrow f(y) = 0}{\texttt{[}p\texttt{ ?] -> [}f\texttt{] -> sum} = \texttt{[}f\texttt{] -> sum}}$$

The rule would achieve the desired simplification but we refrain from adding such a rule because it would take a considerable effort to prove it. Additionally, the rule would be of little use in cases where the LL operations appear in just a slightly syntactically different way.

We instead resort to introduction, which—by substituting the current goal with a new set of goals—isolates independent pieces of reasoning. In this case, an introduction rule is more useful and more general than a simplification rule because it is concerned with a smaller term in the current goal. Consequently, its validity is easier to establish.

Our first introduction rule substitutes the goal in Eq. (3.1), which assert a property of the *whole result vector*, with an assertion about a *single element* of that vector. In effect, this removes the outermost map from the LL function on the left-hand side. Semi-formally, the rule for map can be stated as follows:

$$\frac{\text{length of } A \text{ is } m \qquad \forall i < m \;.\; f(A\texttt{[}i\texttt{]}) = B_i}{\texttt{[}f\texttt{]}(A) \overset{m}{\triangleright} B} \tag{3.2}$$

In the goal in Eq. (3.1), f matches the entire chain of `enum -> ... -> sum` and the new subgoals are

(i) length of A is m

(ii) $\forall i < m$.

```
enum -> [snd != 0 ?] ->
  [snd * x[fst]] -> sum
```
$(A\texttt{[}i\texttt{]}) = \sum_{j<n} B_{i,j} \cdot y_j$

We now need a second introduction step to remove the summation on both sides of the equality: instead of requiring equivalence between *sums of sequences* of numbers, we will require equivalence between the *values in the sequences* themselves. In order for such a rule to be general enough, we need to permit arbitrary permutations of the values in a sequence to prove programs that exploit associativity and commutativity of addition. A hypothetical rule may look as follows, where $[x_i|p(x_i)]_{i=a,\ldots,a+\delta}$ denotes a construction of an ordered list

of elements out of $x_a, \ldots, x_{a+\delta}$ that satisfy p:

$$\frac{\begin{array}{l}\exists n' \leq n,\ \text{permutation } P\ .\\ \quad f(A\texttt{[}i\texttt{]}) \overset{n'}{\triangleright} [B_{i,j} \mid B_{i,j} \neq 0]_{j=P_0,\ldots,P_{n-1}}\end{array}}{\texttt{sum}(f(A\texttt{[}i\texttt{]})) = \sum_{j<n} B_{i,j}}$$

This rule is problematic for two reasons. First, it is more complex than what we may want to prove. For example, the premise constructs a filtered and permuted mathematical vector on the right-hand side (via list comprehension), rather than keeping the mathematical object untouched. This might hinder our ability to link our proof goal to the original input matrix in the assumptions of the theorem. Second, the rule is not as general as we would like because a concrete representation *may* contain zeros.

Our approach is to enrich the representation relation ($a \overset{k}{\triangleright} b$). This relation uses plain equality to relate single elements from the two vector objects, which limits its applicability to more subtle mappings. To express a relation where, say, each element in a concrete representation equals the corresponding vector element multiplied by some value, we parameterize the representation relation with an *inner relation* that describes how individual elements represent their mathematical counterparts. Individual elements need not be scalars; they could be, recursively, lists. Therefore, inner relations could be parameterized by further inner relations.

Our domain proof theory for sparse matrices is novel in two ways. First, we define common representation relations that occur in our domain. Our infrastructure is powerful because we (i) insist on relaxing invariants as much as possible (e.g., zeros may still be present in a compressed representation); (ii) encapsulate many quantifications and implications in the representation relations (e.g., universal quantification on all indexes of a vector, existence of a permutation); (iii) include necessary integrity constraints in the representation relations (e.g., lengths must match). The representation relations we define include indexed list (*ilist*), where the element at position i represents the ith vector element; value list (*vlist*), in which all nonzero values are represented; and associative list (*alist*), which contains index-value pairs. These representation relations raise the level of abstraction and focus theory development on these prevalent data representation. It is also crucial to scaling and automating proofs: introduction depends on unification of goal terms with higher-order variables found in rules. The use of representation predicates effectively canonicalizes terms in intermediate proof goals, thereby preventing unrestrained branching due to unification of custom, fine-grained terms. The use of representation relations also prevents oversimplification of proof terms by concealing their internal conjuncts from Isabelle's simplifier.

The second novelty is parameterizing the inner predicate, which describes how the vector elements represent their mathematical counterparts. In the case of a vector of numbers, we

use equality. For matrices, the inner relation relates a single row to its concrete indexed-list representation (*ilist*); technically, the inner relation predicate is a parameter to the (outer) representation predicate for the whole matrix. In addition to reducing the number of rules, parameterization helps with syntactic matching and substitution of inner comparators during introduction. For example, with a parameterized relation, an introduction rule for map similar to that in Eq. (3.2) can be written more generally and concisely: the conclusion of the rule contains an indexed-list representation relation where the concrete object is the term $\texttt{[}f\texttt{]}(x)$ (i.e., map with an arbitrary function f over x) and the inner representation relation is some arbitrary predicate P—our parameter. The premise of the rule is again an indexed-list representation relation where the concrete object is x and the inner representation relation is $\lambda i\ a\ b.\ P(i, a, f(b))$. Fortunately, Isabelle can match and substitute terms that contain parameters such as P (as well as f and x); these rules can thus be applied automatically.

3.3 Translating LL to Isabelle/HOL

We formalize an LL function f in Isabelle by applying to it a syntax-directed translation function $\mathcal{I}[\![f]\!]$, shown in Table 3.1. It relies heavily on the theory of lists in Isabelle's standard library, as well as other standard features like product types, function composition, and so on. This translation constitutes a *shallow embedding* [63] of LL in Isabelle/HOL. It is a standard technique for formalizing languages when the goal is to verify the correctness of programs written in those languages. In this approach, the functions and types of an object language (LL) are written directly in the language of the theorem prover (typed λ-calculus). Subsequent logical formulas relate to these translated programs as first-class HOL objects, which allows to leverage existing support for proving properties of them.

We translate two implementations of CSR construction and SpMV from Section 2.4.1 into the following definitions (mildly simplified for better readability):

$$\begin{aligned} \mathit{csr} &= \mathit{map}\ \Big((\mathit{filter}(\lambda(j,v).v \neq 0)) \circ \mathit{enum}\Big) \\ \mathit{csrmv}\ (A,x) &= \mathit{map}\ \Big(\mathit{listsum} \circ\ \mathit{map}\ (\lambda(x,y).\ x * y) \circ \\ &\qquad \mathit{unsplit}\ \mathit{zip} \circ \\ &\qquad (\lambda(J,V).\ (V, \mathit{map}\ (\lambda i.\ x\ !\ i)\ J)) \circ \\ &\qquad \mathit{unzip}\Big)\ A \end{aligned} \tag{3.3}$$

We now pose the verification theorem: when A index-represents the $m \times n$-matrix A' and x the n-vector x', the result of CSR SpMV applied to a CSR version of A and to x represents

[1] When used with blocked formats, this function translates to *concat_vectors* (see Section 3.7.5).

[2] See Section 3.7.5.

[3] We rewrite `[ ... num ... ]` into `enum -> [`l_i`,` l_v`:` l_v `-> ...` l_i `...]`, where l_i and l_v are fresh labels used for the position index and value of the current element, respectively.

LL function (f)	Isabelle/HOL function ($\mathcal{I}[\![f]\!]$)
`id`	$\lambda x.\ x$
`eq`, `neq`	$\lambda(x,y).\ x = y$, $\lambda(x,y).\ x \neq y$
n, `true`, `false`	$\lambda y.\ n$, $\lambda y.\ true$, $\lambda y.\ false$
f `?` g `|` h	$\lambda x.\ if\ \mathcal{I}[\![f]\!]\ x\ then\ \mathcal{I}[\![g]\!]\ x\ else\ \mathcal{I}[\![h]\!]\ x$
l_1`,...,`l_k `=` f`:` g †	$\Big(\lambda(x_1,\ldots,x_k).\ \mathcal{I}[\![g[l_i/\lambda y.\ x_i]]\!]\ (x_1,\ldots,x_k)\Big) \circ \mathcal{I}[\![f]\!]$
`(`f_1`,`f_2`,...,`f_k`)`	$\lambda x.\ (\mathcal{I}[\![f_1]\!]\ x, \mathcal{I}[\![f_2]\!]\ x, \ldots, \mathcal{I}[\![f_k]\!]\ x)$
`fst`, `snd`	$\lambda(x,y).\ x$, $\lambda(x,y).\ y$
f `->` g	$\mathcal{I}[\![g]\!] \circ \mathcal{I}[\![f]\!]$
`add`, `sub`, `mul`, `div`, `mod`	$\lambda(x,y).\ x + y$, $\lambda(x,y).\ x - y$, ...
`vadd`, `vsub`, ...	$\mathcal{I}[\![$`zip -> [add]`$]\!]$, $\mathcal{I}[\![$`zip -> [sub]`$]\!]$, ...
`leq`, `lt`, `geq`, `gt`	$\lambda(x,y).\ x \leq y$, $\lambda(x,y).\ x < y$, ...
`sum`, `prod`	$foldl\ (op +)\ 0$, $foldl\ (op *)\ 1$
`and`, `or`	$\lambda(x,y).\ x \wedge y$, $\lambda(x,y).\ x \vee y$
`neg`	$\lambda x.\ \neg x$
`conj`, `disj`	$foldl\ (op \wedge)\ True$, $foldl\ (op \vee)\ False$
`len`	$length$
`rev`	rev
`idx`	$\lambda(v,i).\ v\ !\ i$
`vidx`	$\lambda(v,s).\ map\ (\lambda i.\ v\ !\ i)\ s$
`distl`, `distr`	$\lambda(x,v).\ map\ (\lambda y.\ (x,y))\ v$, $\lambda(x,v).\ map\ (\lambda y.\ (y,x))\ v$
`zip`	$unsplit\ zip$
`unzip`	$\lambda l.\ (map\ fst, map\ snd)$
`enum`	$\lambda v.\ zip\ [0\ ..<\ length\ v]\ v$
`concat`	$concat$[1]
`infl`	$\lambda(d,n,v).\ foldr\ (\lambda(i,x)\ v.\ v[i := x])\ v\ (replicate\ n\ d)$
`aggr`	$\lambda xs.\ map\ (\lambda k.\ (k, map\ snd\ (filter\ (\lambda(k',v).\ k = k')\ xs)))$ $(remdups\ (map\ fst\ xs))$
`sort`, `rsort`	$sort\_key\ fst$, $\mathcal{I}[\![$`sort -> rev`$]\!]$
`trans`	$\lambda v.\ [map\ (\lambda v.\ v\ !\ i)\ (takeWhile\ (\lambda v.\ i < length\ v)\ v)\ .$ $i \leftarrow [0\ ..<\ if\ v = []\ then\ 0\ else\ length\ (v\ !\ 0)]]$
`block, vblock`	$block\_vector$, $block\_matrix$[2]
`map` f	$map\ \mathcal{I}[\![f]\!]$
`filter` f	$filter\ \mathcal{I}[\![f]\!]$
`num`	Eliminated via rewriting[3]

Table 3.1: LL embedding in Isabelle/HOL

the m-vector that is equal to $A' \cdot x'$. More formally:

$$\begin{aligned} &\mathit{ilist}_M\ m\ n\ A'\ A \wedge \mathit{ilist}_v\ n\ x'\ x \\ &\longrightarrow \mathit{ilist}_v\ m\ (\lambda i.\ \Sigma j < n.\ A'\ i\ j * x'\ j)\ (\mathit{csrmv}\ (\mathit{csr}\ A,\ x)) \end{aligned} \tag{3.4}$$

In the following we present the formalism and reasoning used in proving this goal.

3.4 Formalizing vector and matrix representations

We begin by formalizing vectors and matrices in HOL. Mathematical vectors and matrices are formalized as functions from indexes to values, namely nat $\rightarrow \alpha$ and nat $\rightarrow$ nat $\rightarrow \alpha$, respectively; note that the $\rightarrow$ type constructor is right-associative, hence a matrix is a vector of vectors. Dimensions are not encoded in the type itself, and values returned for indexes exceeding the dimensions can be arbitrary, which means that many functions can represent the same mathematical entity. Concrete representations of dense and sparse vectors/matrices are derived from the LL implementation and consist of lists and pairs. Commonly used representations include indexed lists, value lists and associative lists, all of which are explained below.

We introduce *representation relations* (defined as predicates in HOL) to link mathematical vectors and matrices with different concrete representations, for three reasons. First, in proving correctness of functions we map operations on concrete objects to their mathematical counterparts. This is easy to do for indexed list representations but gets unwieldy with others. We encapsulate this complexity inside the definitions of the relations. Second, a relation predicate can enforce integrity constraints of the representation. For example, an associative list representation requires that index values are unique; or the lengths of a list of indexed list representations need to be fixed. Third, for some representations (e.g., value list) there exists no injective mapping from concrete objects to abstract ones, forcing us to use relations rather than *representation functions*. Using relations across the board yields a more consistent and logically lightweight framework.

3.4.1 Indexed list representation

An *indexed list* representation of an n-vector x' by a list x is captured by the *ilist* predicate. Note that we refrain from fixing vector elements to a specific type (e.g., integers) and instead use type parameters α and β to denote the types of inner elements of the mathematical and concrete vectors, respectively:

$$\begin{aligned} &\mathit{ilist} :: \text{nat} \rightarrow (\text{nat} \rightarrow \alpha \rightarrow \beta \rightarrow \text{bool}) \rightarrow \\ &\qquad (\text{nat} \rightarrow \alpha) \rightarrow [\beta] \rightarrow \text{bool} \\ &\mathit{ilist}\ n\ P\ x'\ x \iff \\ &\qquad (\mathit{length}\ x = n) \wedge (\forall i < n.\ P\ i\ (x'\ i)\ (x\ !\ i)) \end{aligned}$$

The parameter P is a relation that specifies the representation of each element in the vector. For ordinary vectors, it is equality of elements. However, P turns useful for matrix representation, as we can use arbitrary relations to determine the representation of inner vectors. We introduce abbreviations for the common cases of indexed list representations:

$$\begin{aligned} \mathit{ilist}_v\ n\ x'\ x &\iff \mathit{ilist}\ n\ (\lambda j.\ \mathit{op} =)\ x'\ x \\ \mathit{ilist}_M\ m\ n\ A'\ A &\iff \mathit{ilist}\ m\ (\lambda i.\ \mathit{ilist}_v\ n)\ A'\ A \end{aligned}$$

3.4.2 Associative list representation

An *associative list* representation is central to sparse matrix codes as it is often used in vector compression. It is captured by the *alist* predicate:

$$\begin{aligned} &\mathit{alist} :: \text{nat} \to (\text{nat} \to \alpha \to \beta \to \text{bool}) \to \\ &\quad (\alpha\ \text{set}) \to (\text{nat} \to \alpha) \to [(\text{nat}, \beta)] \to \text{bool} \\ &\mathit{alist}\ n\ P\ D\ x'\ x \iff \\ &\quad \mathit{distinct}\ (\mathit{map}\ \mathit{fst}\ x) \wedge \\ &\quad (\forall (i, v) \in \mathit{set}\ x.\ P\ i\ (x'\ i)\ v \wedge i < n) \wedge \\ &\quad (\forall i < n.\ x'\ i \notin D \longrightarrow \exists v.\ (i, v) \in \mathit{set}\ x) \end{aligned}$$

Here, *distinct* is a predicate stating the uniqueness of indexes (i.e., keys) in x. Each element in an associative list must relate to the respective vector element, also requiring that index values are within the vector length. Finally, each element in the vector that is not a default value (specified by the set of values D) must appear in the representing list. Note that a *set* of default values accounts for cases where more than one such value exists, as in the case of nested vectors where each function mapping the valid dimensions to zero is a default value. Also note that *alist* does not enforce a particular order on elements in the compressed representation, nor does it insist that all default values are omitted.

3.4.3 Value list representation

Sometimes concrete objects contain only the values of the elements in a given vector, without mention of their indexes. This *value list* representation often occurs prior to computing a cross- or dot-product. It is captured by the *vlist* predicate, which states that the list of values can be zipped with some list of indexes p to form a proper associative list representation. The length restriction ensures that no elements are dropped from the tail of x:

$$\begin{aligned} &\mathit{vlist} :: \text{nat} \to (\text{nat} \to \alpha \to \beta \to \text{bool}) \to \\ &\quad (\alpha\ \text{set}) \to (\text{nat} \to \alpha) \to [\beta] \to \text{bool} \\ &\mathit{vlist}\ n\ P\ D\ x'\ x \iff \\ &\quad \exists p.\ \mathit{length}\ p = \mathit{length}\ x \wedge \\ &\quad\quad \mathit{alist}\ n\ P\ D\ x'\ (\mathit{zip}\ p\ x) \end{aligned}$$

Additional representations can be incorporated into our theory. For example, when a matrix is compressed into an associative list, a dual-index representation relation can be defined similarly to *alist*.

3.5 Verifying CSR

We prove Eq. (3.4) using term rewriting and introduction rules. Introduction rules are used whenever further rewriting cannot be applied. An introduction rule is applied by resolution: applying the rule $G\ x \wedge H\ y \longrightarrow F\ x\ y$ to the goal $F\ a\ b$ yields two new subgoals, $G\ a$ and $H\ b$.

The theorem in Eq. (3.4) makes the following two assumptions:

$$\mathit{ilist}_M\ m\ n\ A'\ A \tag{3.5}$$

$$\mathit{ilist}_v\ n\ x'\ x \tag{3.6}$$

which are added to the set of available introduction rules as $\mathit{true} \longrightarrow \ldots$. The conclusion of Eq. (3.4) is our initial proof goal:

$$\mathit{ilist}_v\ m\ (\lambda i.\ \Sigma j < n.\ A'\ i\ j * x'\ j)\ (\mathit{csrmv}\ (\mathit{csr}\ A)\ x) \tag{3.7}$$

3.5.1 Simplifying the goal

We begin by applying Isabelle's simplifier using Eq. (3.3) and standard rules for pairs, lists, arithmetic and Boolean operators. This removes most of the function abstractions, compositions and pair formations due to the translation from LL. Our new goal is analogous to Eq. (3.1):

$$\begin{aligned}&\mathit{ilist}_v\ m\ (\lambda i.\ \Sigma j < n.\ A'\ i\ j * x'\ j)\\ &\quad(\mathit{map}\ (\lambda r.\ \mathit{listsum}\ (\mathit{map}\ (\lambda v.\ \mathit{snd}\ v * x\ !\ \mathit{fst}\ v)\\ &\qquad\qquad(\mathit{filter}\ (\lambda v.\ \mathit{snd}\ v \neq 0)\ (\mathit{enum}\ r))))\ A)\end{aligned} \tag{3.8}$$

Solving the entire goal using rewriting alone calls for simplification rules that are too algorithm-specific. For example, the rule

$$\begin{aligned}&(\forall x \in \mathit{set}\ xs.\ \neg\ P\ x \longrightarrow f\ x = 0)\\ &\longrightarrow \mathit{listsum}\ (\mathit{map}\ f\ (\mathit{filter}\ P\ xs)) = \mathit{listsum}\ (\mathit{map}\ f\ xs)\end{aligned} \tag{3.9}$$

allows further simplification of Eq. (3.8), but fails for all formats that introduce more complex operations between *map* and *filter*.

3.5.2 Introduction rules on representation relations

Consider the equation in the conclusion of Eq. (3.9). We know that it holds when the two lists, xs and *filter* P xs, value-represent the same vector. By introducing rules, describing when it is allowed to apply *map*, *filter* and *enum* operations to value list representations, we prove that the result of *listsum* in Eq. (3.8) equals the mathematical dot-product.

Figure 3.1 shows the introduction rules used in proving Eq. (3.4). Application of introduction rules is syntax directed, choosing rules whose conclusion matches the current goal. Given Eq. (3.8), the prover applies ILIST-MAP, which moves the map from the representing object into the inner representation relation, followed by ILIST-LISTSUM, which substitutes *listsum* with an equivalent notion of value-represented rows. This results in

$$\begin{aligned} &\mathit{ilist}\ m\ \Big(\lambda i\ r'\ r.\ \mathit{vlist}\ n\ (\lambda j.\ \mathit{op} =)\ \{0\}\ r' \\ &\qquad\qquad (\mathit{map}\ (\lambda v.\ \mathit{snd}\ v * x\ !\ \mathit{fst}\ v) \\ &\qquad\qquad\quad (\mathit{filter}\ (\lambda v.\ \mathit{snd}\ v \neq 0)\ (\mathit{enum}\ r)))\Big) \\ &\quad (\lambda i\ j.\ A'\ i\ j * x'\ j)\ A \end{aligned}$$

Further simplification is not possible at this point, nor can we modify the *vlist* relation inside *ilist*. Luckily, ILIST-VLIST matches our goal, lifting the inner *vlist* to the outermost level and permitting to further operate on the concrete parameters of *vlist*. Note that ILIST-VLIST has two assumptions, resulting in new subgoals

$$\mathit{ilist}\ m\ ?Q\ ?B'\ A \tag{3.10}$$

and

$$\begin{aligned} &\forall i < m.\ \mathit{vlist}\ n\ (\lambda j.\ \mathit{op} =)\ \{0\}\ (\lambda j.\ A'\ i\ j * x'\ j) \\ &\qquad \Big(\mathit{map}\ (\lambda v.\ \mathit{snd}\ v * x\ !\ \mathit{fst}\ v) \\ &\qquad\quad (\mathit{filter}\ (\lambda v.\ \mathit{snd}\ v \neq 0)\ (\mathit{enum}\ (A\ !\ i)))\Big) \end{aligned} \tag{3.11}$$

In Eq. (3.10), $?Q$ and $?B'$ are existentially quantified variables. They do not get instantiated when we apply ILIST-VLIST, and the subgoal in Eq. (3.10) merely certifies that A has length n. Therefore, the prover is allowed to instantiate them arbitrarily and Eq. (3.10) is discharged by the assumption Eq. (3.5).

The rules VLIST-MAP, ALIST-FILTER and ALIST-ENUM can now be applied to Eq. (3.11). Note that applying them amounts to the effect of simplification using the rule in Eq. (3.9). However, they can be applied regardless of the way in which the three operations—*map*, *filter* and *enum*—are intertwined. Therefore, they are applicable in numerous cases where the context imposed by Eq. (3.9) is too restrictive.

[4]The predicate P and the vector z' are arbitrary, they just help to state that x is a list of length m.

$$\frac{\mathit{ilist}\ n\ (\lambda i\ a\ b.\ P\ i\ a\ (f\ b))\ x'\ x}{\mathit{ilist}\ n\ P\ x'\ (\mathit{map}\ f\ x)}\ \textsc{ilist-map}$$

$$\frac{\mathit{ilist}\ m\ (\lambda i\ r'\ r.\ \mathit{vlist}\ n\ (\lambda j.\mathit{op} =)\ \{0\}\ r'\ (f\ r))\ A'\ A}{\mathit{ilist}\ m\ (\lambda i\ r'\ r.\ r' = \mathit{listsum}\ (f\ r))\ (\lambda\ i.\ \Sigma\ j < n.\ A'\ i\ j)\ A}\ \textsc{ilist-listsum}$$

$$\frac{\begin{array}{c}\mathit{ilist}\ m\ Q\ B'\ A\\ \forall i < m.\ \mathit{vlist}\ n\ (P\ i)\ (D\ i)\ (f\ (A'\ i)\ i)\ (g\ (A\ !\ i)\ i)\end{array}}{\mathit{ilist}\ m\ (\lambda i\ r'\ r.\ \mathit{vlist}\ n\ (P\ i)\ (D\ i)\ (f\ r'\ i)\ (g\ r\ i))\ A'\ A}\ \textsc{ilist-vlist}$$

$$\frac{\mathit{alist}\ m\ (\lambda i\ r'\ r.\ P\ i\ r'\ (f\ (i,\ r)))\ D\ x'\ x}{\mathit{vlist}\ m\ P\ D\ x'\ (\mathit{map}\ f\ x)}\ \textsc{vlist-map}$$

$$\frac{\begin{array}{c}\mathit{alist}\ n\ P\ D\ x'\ x\\ (\forall i < n.\ \forall v\ v'.\ \neg\ Q\ (i,\ v) \wedge P\ i\ v'\ v \longrightarrow v' \in D)\end{array}}{\mathit{alist}\ n\ P\ D\ x'\ (\mathit{filter}\ Q\ x)}\ \textsc{alist-filter}$$

$$\frac{\mathit{ilist}\ m\ P\ x'\ x}{\mathit{alist}\ m\ P\ D\ x'\ (\mathit{enum}\ x)}\ \textsc{alist-enum}$$

$$\frac{\begin{array}{c}\mathit{ilist}\ n\ (\lambda i\ v'\ v.\ v' = f\ i\ v)\ x'\ z\\ \mathit{ilist}\ n\ (\lambda i\ v'\ v.\ v' = g\ i\ v)\ y'\ z\end{array}}{\mathit{ilist}\ n\ (\lambda i\ v'\ v.\ v' = f\ i\ v * g\ i\ v)\ (\lambda i.\ x'\ i * y'\ i)\ z}\ \textsc{ilist-mult}$$

$$\frac{\mathit{ilist}_v\ m\ x'\ y \qquad \mathit{ilist}\ m\ P\ z'\ x}{\mathit{ilist}\ m\ (\lambda i\ v'\ v.\ v' = y\ !\ i)\ x'\ x}\ \textsc{ilist-nth}^4$$

$$\frac{\mathit{ilist}_M\ m\ n\ A'\ A \qquad i < m}{\mathit{ilist}_v\ n\ (A'\ i)\ (A\ !\ i)}\ \textsc{ilist}_v \rightarrow \textsc{ilist}_M$$

Figure 3.1: Introduction rules for verifying CSR SpMV

The ALIST-FILTER rule forces us to prove that *filter* only removes default values, in the form of the following new subgoals:

$$\begin{aligned}
&\forall i < m.\ \forall j < n.\ \forall v\ v'.\\
&\quad \neg\ snd\ (j,\ v) \neq 0 \wedge v' = v * x\ !\ j \ \longrightarrow\ v' \in \{0\} \qquad (3.12)\\
&\forall i < m.\\
&\quad ilist\ n\ (\lambda j\ v'\ v.\ v' = v * x\ !\ j)\ (\lambda j.\ A'\ i\ j * x'\ j)\ (A\ !\ i)
\end{aligned}$$

Fortunately, the subgoal in Eq. (3.12) is completely discharged by the simplifier. The remaining goal is solved using the ILIST-MULT, ILIST-NTH, and $\text{ILIST}_v \rightarrow \text{ILIST}_M$, as well as the assumptions Eq. (3.5) and Eq. (3.6).

3.6 Automating the proof

The above proof outline already dictates a simple proof method. Isabelle's tactical language [62] provides us with ample methods and combinators that can be used to implement custom proof tactics. Our proof method is implemented as follows:

1. The *simplifier* attempts to rewrite the goal until no further rewrites are applicable, returning the new goal. If no rewrite rule could be applied, it returns an empty goal.

2. The *resolution* tactic attempts to apply each of the introduction rules and returns a new goal state for each of the matches. It is possible that more than one rule matches a given goal, e.g. ILIST-MAP and ILIST-NTH both match $ilist\ n\ (\lambda i\ v'\ v.\ v' = y\ !\ i)\ x'\ (map\ f\ x)$, resulting in a sequence of alternative goal states to be proved.

Invoking the proof method leads to a depth-first search on the combination of the two sub-methods. It maintains a sequence of goal states, initially containing only the main goal. After each successful application of either sub-method, the result is prepended to the head of the sequence. A failure at any level causes the search to backtrack and continue with the next available goal state. When the top element of the goal state sequence is empty, the main goal has been discharged and the proof is complete.

3.7 Verifying additional sparse formats

We examine to what extent our prover design allows us to verify additional formats without adding excessively many rules. By this we validate the basic principles that guide our domain theory development: minimizing reliance on format-specific rules, avoiding duplication of logic, and keeping representation relations general, for example by keeping the type of

the value stored in the matrix parametric. In this section, we extend our prover to verify additional formats that are strictly more complex than CSR.

For each of these formats, we consider only a single implementation variant from those presented in Section 2.4 and Section 2.5, as a representative for a larger class of similar implementations. Our experience indicates that our prover can overcome variations in implementations—both syntactic and operational—thanks to Isabelle's simplifier successfully canonicalizing these differences, requiring only minor tweaks to the prover's rule base.

3.7.1 Jagged diagonals (JAD)

A prominent feature of JAD's proof goal is the double use of transpose, once during compression (`jad`) and once during multiplication (`jadmv`). In modeling transposition in Isabelle we use the *takeWhile* list operator, which extracts the longest contiguous prefix of list elements (in this case, compressed rows) that satisfy some predicate (the row contains at least k elements, for a varying value of k). The proof relies on the premise that compressed rows are sorted by decreasing length prior to being transposed, in which case the *takeWhile* operation is the same as filtering. More introduction rules were added for handling other operators used in these functions, including *infl*, *rev* and *sort_key*. This was sufficient for the prover to complete the proof.

The ability to prove full functional correctness of JAD SpMV documents the strength of our prover. No other verification framework that we know of can (i) handle the complex data transformations in JAD compression, and (ii) prove correctness of arithmetic operations on the resulting sparse representation.

3.7.2 Coordinate (COO)

The COO format is challenging because it associates matrix values with both row and column coordinates, and also because it requires concatenation and aggregation operations. It turns out that the COO pair coordinates do not call for a new representation relation. In fact, thanks to how the functions `coo` and `coomv` are composed, we need to handle the pair coordinates only between concatenation (in `coo`) and aggregation (in `coomv`). The simplifier moves these two functions together; therefore, we introduce a rule to relate the representation of the input and output of *aggr* (*concat xs*), allowing the prover to automatically complete the proof.

$$\frac{\begin{array}{l}\mathit{vlist}\ n\ (\lambda i.\ \mathit{vlist}\ m\ (\lambda j\ a\ b.\ a = \mathit{snd}\ b\ \wedge i = \mathit{fst}\ b)\ \{0\}) \\ \qquad \{x.\forall j < m.x\ j = 0\}\ M\ xs\end{array}}{\begin{array}{l}\mathit{alist}\ n\ (\lambda i.\ \mathit{vlist}\ m\ (\lambda i.op =)\ \{0\}) \\ \qquad \{x.\forall j < m.x\ j = 0\}\ M\ (\mathit{aggr}\ (\mathit{concat}\ xs))\end{array}}\ \text{ALIST-AGGR-CONCAT}$$

3.7.3 Compressed Sparse Columns (CSC)

As CSC exhibits a peculiar use of concatenation and aggregation operations, it is handled similarly to COO. In contrast to COO, the input list to `concat` represents a transposed matrix, hence we use a rule similar to ALIST-AGGR-CONCAT above, but with a transposed matrix M.

3.7.4 Sparse CSR (SCSR)

The SCSR format applies another layer of compression on top of ordinary CSR. SCSR demonstrates the ability of our prover to peel off the additional compression layer and prove correctness of the overall result, while requiring only two rules in addition to those needed by CSR (see Table 3.2).

3.7.5 Blocked formats

It is easy to extend our prover to blocked formats because our matrices are of parametric type—the prover can work with matrices of numbers as well as with matrices whose elements are matrices. Parameterization of matrices was expressed with Isabelle/HOL type classes, which are used to restrict types in introduction rules.

We use a theory of finite matrices [49]. Here, too, the size of a matrix is not encoded in the matrix type (denoted α matrix) but it is required that matrix dimensions are bounded. To represent matrices as abstract values, we introduce the *matrix* conversion function:

$$matrix :: \text{nat} \to \text{nat} \to (\text{nat} \to \text{nat} \to \alpha) \to \alpha \text{ matrix}$$

The first two parameters specify the row and column dimensions, respectively. The third parameter is the abstract value encoded into the matrix.

We also extend the theory with the following conversion functions, which allow us to model operations on blocked compressed formats:

$$\begin{aligned} block\_vector &:: \text{nat} \to \text{nat} \to [\alpha] \to [\alpha \text{ matrix}] \\ block\_matrix &:: \text{nat} \to \text{nat} \to \text{nat} \to \text{nat} \to [[\alpha]] \to [[\alpha \text{ matrix}]] \\ concat\_vectors &:: \text{nat} \to [\alpha \text{ matrix}] \to [\alpha] \end{aligned}$$

The operation *block_vector* $m\ k\ x$ transforms the list x of length nk into a list of n k-vectors; *block_matrix* $m\ n\ k\ l\ A$ transforms the object A, representing an $mk \times nl$-matrix, into an object representing an $m \times n$-matrix of $k \times l$-blocks; *concat_vectors* $k\ x$ is the inverse operation, unpacking k-vectors into a single long vector. These are used to model, respectively, the LL functions `block`, `vblock` and `concat` in the implementation of blocked formats, respectively. Note that these functions use a richer set of parameters compared to

$$\frac{k \neq 0 \qquad ilist_v\ n\ (\lambda i.\ block\ k\ 1\ (\lambda i'\ j.\ A'\ (i*k+i')))\ A}{ilist_v\ (n*k)\ A'\ (concat\_vectors\ k\ A)}\ \text{ILIST-CONCAT\_VECTORS}$$

$$\frac{l \neq 0 \qquad ilist_v\ (m*l)\ x'\ x}{ilist_v\ m\ (\lambda i.\ block\ l\ 1\ (\lambda i'\ j'.\ x\ (i*l+i')))\ (block\_vector\ m\ l\ x)}\ \text{ILIST-BLOCK\_VECTOR}$$

$$\frac{k \neq 0 \qquad l \neq 0 \qquad ilist_M\ (m*k)\ (n*l)\ A'\ A}{ilist_M\ m\ n\ (\lambda i\ j.\ block\ k\ l\ (\lambda i'\ j'.A'(i*k+i')(j*l+j')))\ (block\_matrix\ m\ n\ k\ l\ A)}\ \text{ILIST-BLOCK\_MATRIX}$$

Figure 3.2: Introduction rules for verifying blocked sparse formats

their LL origins (e.g., the number of blocks). We argue that this is a reasonable price to pay for being able to verify blocked formats.

This code shows register-blocked CSR implementation in Isabelle:

$$rbcsr\ m\ n\ k\ l\ A = map\ (filter\ (\lambda(i,v).\ v \neq 0) \circ enum)\ (block\_matrix\ m\ n\ k\ l\ A)$$

$$\begin{aligned} &rbcsrmv\ m\ n\ k\ l\ (A,\ x) = \\ &\quad concat\_vectors\ k \\ &\quad\quad (map\ (listsum \circ \\ &\quad\quad\quad (map\ (\lambda v.snd\ v * block\_vector\ n\ l\ x\ !\ fst\ v)))\ A) \end{aligned}$$

We require that block dimensions are greater than zero and properly divide the respective matrix and vector dimensions. The correctness theorem follows:

$$\begin{aligned} &k \neq 0 \wedge l \neq 0 \wedge ilist_M\ (m*k)\ (n*l)\ A'\ A \wedge ilist_v\ (n*l)\ x'\ x \\ &\longrightarrow ilist_v\ (m*k)\ (\lambda i.\ \Sigma j < n*l.\ A'\ i\ j * x'\ j)\ (rbcsrmv\ m\ n\ k\ l\ (rbcsr\ m\ n\ k\ l\ A,\ x)) \end{aligned} \tag{3.13}$$

After adding the introduction rules in Figure 3.2 and a few rewrite rules for *matrix*, the prover automatically proves Eq. (3.13).

3.8 Evaluation

The verifier described in this chapter can prove full functional correctness of a variety of sparse formats, quickly and automatically. In the following, we describe our experience with other approaches for verifying sparse matrix programs. We then evaluate the success of our current approach in terms of applicability and extensibility.

Prior to focusing on functional programming models for implementing sparse codes, we experimented with different approaches to verifying imperative programs. However, even the simple CSR format proved rather complex to handle in any of these approaches, which we describe here:

Hoare logic / VCGen. We began experimenting with Hoarse-style pen-and-paper proofs, using either first-order judgements or inductive predicates in describing representation invariants of the sparse format. This convinced us that invariants in first-order logic of even relatively small programs are not amenable to manipulation by hand. While the use of inductive predicates showed more potential to scale, it was also more restrictive to work with. One particular problem has to do with the direction in which inductive definitions are folded and unfolded. This is a known problem in logic domains that deal with inductive constructions, and requires the use of inversion techniques [23, 19].

We next turned to mechanized verification of our CSR sparse matrix routines using ESC/Java [28], a verification condition generator (VCGen) and checker for Java programs. We were able to prove that a sparse matrix-vector multiplication returns the desired result with respect to a dense matrix multiplication. However, this approach proved impractical for two reasons:

- The first-order invariants and the reasoning provided by the framework were insufficient for capturing the full correctness property. For example, we could not reason directly about a sum of a sequence of values or account for arbitrary permutations of values. Consequently, our proofs were limited to cases were the nonzero values of the matrix are compressed in the original column order, whereas in general they need not be.
- Developing proofs amounted to writing very large loop invariants, preconditions and postconditions. These largely outnumbered the actual lines of code by a factor of 3–7 depending on the procedure being verified.

The aforementioned limitations rendered this approach inappropriate and impractical. We were motivated to look for more automated techniques and better abstractions for handling the complexity of sparse matrix manipulation.

Abstract interpretation. We used TVLA [50], a parametric framework based on three-valued logic, in an attempt to find a more automated technique for reasoning about

sparse matrices via powerful abstractions. We modeled dense and sparse matrix data structures using nested lists, with additional structural constraints and predicates for tracking dense-to-sparse mapping. We succeeded in analyzing a simple sparse format construction and inferred the desired invariant concerning the mapping of values between the dense matrix and the sparse representation. However, the verification of matrix-vector multiplication led to a combinatorial state explosion due to array random access. Moreover, the TVLA formulation was suited for capturing relational invariants and had no built-in support for asserting arithmetic properties such as equivalence of terms. It, too, thus proved inappropriate for the verification problem at hand.

Satisfiability. We experimented with verifying sparse codes using SAT-based bounded model checking. This technique was successfully used in verifying full functional correctness of small programs involving arithmetic, arrays and loops in the context of inductive synthesis [55, 54]. It uses a straightforward translation from imperative semantics to circuits, posing the correctness problem as unsatisfiability of the inverse property. Unfortunately, our prover scaled poorly for operations on even the simplest CSR sparse matrix format.

The aforementioned experiments convinced us that verifying the correctness of sparse matrix routines calls for a higher level of program abstraction and, accordingly, a logical framework capable of tracking intricate properties of such programs compactly and efficiently. Turning our attention to functional programs allowed us to replace explicit loops over arrays with maps and a fixed set of reductions over lists, which in turn simplified the formulation and encapsulation of inductive invariants.

We now analyze the coverage of proof rules in our domain theory and project on its extensibility and applicability to new formats. We hypothesize that, thanks to the use of parametric representation relations, our rule base exhibits a high degree of reusability, and is therefore likely to extend to additional formats with diminishing implementation efforts.

In total, 24 rules were needed to prove our sparse formats, including both introduction and simplification rules. Introduction rules were needed to (i) reason about some language construct such as **`map`**, `sum` and **`filter`**, in the context of a certain representation (e.g., rules ILIST-MAP, ILIST-LISTSUM, ALIST-FILTER in Figure 3.1); and (ii) formalize algebraic operations on vector and matrix representations, such as extracting an inner representation relation (ILIST-VLIST) and substituting a vector representation with a matrix representation ($\text{ILIST}_v \rightarrow \text{ILIST}_M$). Most operators were handled by a single introduction rule; a few (e.g., *map*) required one rule per representation relation.

Table 3.2 summarizes the reuse of the rules that were needed for proving five sparse formats. On average, fewer than 19% of rules used by a particular format are specific to this format, while over 66% of these rules are used by at least three additional formats, a significant level of reuse. This implies that the large majority of rules needed to verify

Reuse degree	# Rules	Avg. LOC	CSR	SCSR	COO	CSC	JAD	Total	%
1	11	28.3	1		2	5	3	11	18.3
2	3	6.3			3	3		6	10.0
3	1	6.0	1	1			1	3	5.0
4	5	6.4	3	5	4	3	5	20	33.3
5	4	2.3	4	4	4	4	4	20	33.3

Table 3.2: Rule reuse in sparse matrix format proofs

new formats is already in use by other formats. Even of the rules needed for more complex formats (CSC and JAD), only up to a third are format-specific.

On the other hand, format-specific rules tend to be harder to prove, as indicated by the average number of lines of Isar [62] code required to prove the rules. A detailed examination reveals that two rules for handling an *aggr-concat* sequence (used in CSC and COO) account for over a hundred lines each. We believe that these rules can be refactored for better reuse of simpler lemmas and greater automation. Note that most of the effort in proving JAD was invested in stating and proving the simplification of *transpose-transpose* composition. Table 3.2 does not account for these rules, as they are quite general and were implemented as an extension to Isabelle's theory of lists.

3.9 Discussion

This chapter shows how sparse format implementations written in LL can be verified for full functional correctness using a higher-order logic theorem prover. We are not aware of previous work on verifying full functional correctness of sparse matrix codes. We are not even aware of work that verified their memory safety without explicitly provided loop invariants.

Our own attempts at verification included ESC/Java, TVLA and SAT-based bounded model checking, neither of which was satisfactory (see Section 1.2, p. 1). Furthermore, neither of these tools was capable of proving higher-order properties like the ones we currently prove. This led us to raising the level of abstraction and deferring to purely functional programs where loops are replaced with comprehensions and specialized reduction operators.

The use of parametric representation relations allows us to verify a variety of different formats using a relatively small number of proof rules. By keeping proof rules general and preferring resolution of specific terms (via introduction rules) over simplification of compound expressions, we are able to achieve a high degree of reuse of rules. This suggests that extension of our proof theory to new formats is likely to require only incremental theory development efforts.

The development of effective verification frameworks for any domain in higher-order logic

requires deep proficiency in interactive theorem proving theory and practice. This often hinders its attractiveness as a viable method for developing verification frameworks. Most notably, higher-order theorem proving is characterized by a dichotomy between the program operational specification (as expressed in LL functions) and reasoning on deeper semantic invariants, such as those encapsulated by higher-order representation relations. Alternatives to HOL, such as dependently typed languages and their associated proof logic [22], mark a step forward in integrating deeper reasoning with programming and may represent a more scalable approach to provably correct domain-specific programming.

Our Isabelle/HOL theory of sparse matrices, along with the proofs of formats mentioned in this chapter, are publicly available as a source code repository. We encourage the interested reader to download it, here: `https://bitbucket.org/garnold/isabelle-sparse`

Chapter 4

Compiling LL Programs

4.1 Overview

In this chapter we consider the problem of translating programs written in LL into efficient data-parallel C code. Our base compilation approach is straightforward: we deploy a syntax-directed translation from high-level language constructs to a target C program, modeling the monadic dataflow in the LL program via value assignment to temporaries. We use a NESL-like flattening scheme for representing arbitrarily nested datatypes and insist on properly abstracting them using hierarchical C types and macros. We use a simple analysis to infer the sizes of output buffers and deploy an allocation strategy for temporary buffers that reduces memory management overhead. The result is a simple code generator that produces clean and well structured C programs, which are well-suited for further tweaking and reuse by programmers. However, it leaves a lot to be desired in terms of low-level performance, and does not utilize parallel hardware. The rest of this chapter describes a set of sequential optimizations and a parallelization scheme, which together yield parallel code whose dataflow, communication and synchronization resemble those of handwritten kernels. We conclude by evaluating the performance of two example kernels generated from LL programs on a multicore machine and show that they are competitive with handwritten C implementations.

4.2 Introduction to compilation of LL code

4.2.1 Compiling a high-level vector language

There have been several languages that raised the level of abstraction in which vector programs are written. Early examples include set-based languages like SETL [51], the APL family of languages for processing massive array-based data [35], and the influential FP/FL function-level languages [2, 3]. Recent dynamic languages (e.g., Python) incorporate high-

level constructs such as list comprehension, which are a convenient alternative form for dependence-free loops. These language all share the goal of promoting productivity and ease of programming. However, their implementations are based on dynamic interpretation and deemphasize the low-level performance of programs.

The goal of the LL compiler is to generate efficient C code that can be deployed in sparse matrix programs and is amenable to further reuse. This chapter focuses on four aspects of the compilation process:

Parsing and semantic analysis. The compiler front-end performs the mundane tasks necessary to obtain a typed abstract syntax tree (AST) representation of the LL program. In addition to inferring the type of each functional program node, it also infers symbolic size boundaries of output values that are generated at each node. This information is necessary for proper allocation of memory buffers during program execution. The front-end is described in Section 4.3.

Datatype representation. We implement a data abstraction layer that hides the intricacies of compound object handling. As a result, the translation process is fairly straightforward, and the code generated clean and relatively easy to maintain and reuse. A data abstraction layer maps high-level LL types that were found to be used in the program—including primitives, pairs, lists and nested types—to their corresponding C representation. In doing so, we insist on (i) keeping the coherent view of compound data object intact, providing a consistent API for manipulating them; as well as (ii) obtaining a "flat" layout of nested list types, which ensures a minimal memory footprint, efficient handling of lists through various functions, and good runtime characteristics. The data abstraction layer is described in Section 4.4.

Syntax-directed translation. We deploy a syntax-directed translation of LL functions into C. Dataflow edges—which are implicit in LL in the form of function composition and pair constructors—translate into value assignment to temporaries. Map and filter constructs translate to loops that traverse an input list object. Built-in functions are implemented as library function calls. This process is described in Section 4.5.

Sequential optimization. The sparse kernels generated by naïve translation from LL can be seven times slower than a handwritten version. In order to mitigate this problem, we identify a small set of optimizations that (i) allow an underlying C compiler to optimize loops; (ii) replace dynamic buffer allocation with static array declarations when possible; (iii) eliminate the use of temporary buffers between adjacent loops; and (iv) streamline the traversal of nested data structures in nested loops. These optimizations are described in Section 4.6.

4.2.2 Nested data-parallelism

High-level languages have been used for specifying implicitly parallel computations. In LL, we deploy a simple heuristic loop parallelization scheme that results in programs with long sequential execution threads, which are well-suited for execution on multicore platforms. This is different from the approach taken in previous work, most notably NESL [12], which pioneered the field of nested data-parallel languages.

In NESL, a programmer can write an expression for multiplying pairs of numbers in a two-level nested list (i.e., a list of lists of pairs) using nested comprehension expressions, as follows:

```
{ {x * y : x, y in Ai} : Ai in A }
```

The idea behind the NESL data parallel compilation process—known as the *vectorization transform*—is to translate nested loops that operate over nested lists into a chain of vector primitives that operate over flat and segmented vectors containing atomic value types. The example NESL expression shown above is compiled into the following single vector operation, where `Ax` and `Ay` are vector operands that contain all the first and second values from pairs in `A`:

```
vec_mul (Ax, Ay)
```

It is now up to a vector abstraction layer [10] and the underlying hardware to parallelize this computation step and execute it efficiently. On the parallel, lockstep-style SIMD machine of the era (such as the Thinking Machines CM-2) this could be done very efficiently with perfect load balancing. Note that parallel workload distribution is oblivious of the lengths of nested lists, which may be arbitrary. Once the pair-wise multiplication is computed, the result vector is broken down into lists whose lengths are the same as those in the nested list `A`. Furthermore, the NESL compiler can vectorize arbitrarily nested loop constructs by systematically lifting the semantics of loop bodies and flattening nested list objects [13].

We argue that the parallelism induced by NESL's vectorization transform is critically flawed from the viewpoint of modern architectures, where computational cores are fewer and vastly more powerful, and synchronization and communication are expensive. We demonstrate this claim using a simple SpMV example.

Let `A` be a sparse matrix in CSR format (see Section 2.4.1). Multiplying `A` by a vector `x` can be expressed using the following NESL comprehension syntax:

```
{ sum ({ Aij * x[j] : j, Aij in Ai }) : Ai in A }
```

The NESL compiler stores the nested lists of `A` in three separate vectors: one vector contains all the column indexes, a second contains all nonzero cell contents, and a third encodes the boundaries of each of the compressed rows (called *segments*). Here, we name them `Ajs`, `Aijs` and `Aseg`, respectively. It then transforms the function into a sequence of three vector operations: the first fetches the cell values in `x` that correspond to *all* the column indexes in `Ajs`; the second multiplies those values element-wise with *all* their corresponding nonzeros in `Aijs`; the third sums the resulting products *within the segments* defined by `Aseg`.

```
t1 = vec_idx (x, Ajs)
t2 = vec_mul (Aijs, t1)
out = seg_sum (Aseg, t2)
```

The parallel execution dataflow of this code is shown in Figure 4.1. The red frames indicate units of parallel execution. It is well-suited for massively parallel SIMD architectures, where fine-grained parallelism improves the overall throughput. However, several attributes make it unsuitable for execution on prevailing parallel architectures, such as shared memory multicore machines: (i) multiple workload distribution points (red arrows) induce load balancing and scheduling overhead; (ii) single primitive operations (`idx`, `mul`) in parallel blocks prohibit instruction-level optimization (pipelines, superscalar); (iii) unnecessary communication through memory buffers (black arrows) induces latencies and saturates memory bandwidth; and (iv) multiple synchronization points (green arrows) induce signaling overhead and lead to frequent idling.

A more desirable execution scheme is shown in Figure 4.2. It trades parallelism granularity for exposing longer instruction sequences, thus improving on all of the aforementioned issues.[1] This style of parallelism is widespread in SPMD programming, where a single block of code executes in parallel on different chunks of data. Fewer synchronization points allow each of the threads to run more efficiently. Optimizations are then applied to improve the sequential performance of threads yet further, by eliminating various communication overheads.

It is generally possible to reconstruct synchronization-free serial execution flow from vectorized programs, as was described by Chatterjee [21]. However, it requires multiple analysis steps for recovering dependencies between inputs and outputs of adjacent functional blocks and reestablishing nested data structures that were lost in the course of vectorization. These analyses are done conservatively and may fail to discover some dependencies. Moreover, certain NESL transformations—e.g., the distribution of free variables to all list elements when compiling comprehensions—cannot be reversed without knowledge of the original high-level program.

[1] Note that even the communication through temporary buffer `t2` is avoidable if the summation is fused into the preceding parallel loop, accumulating the sum into a scalar register within each segment (see Section 4.6).

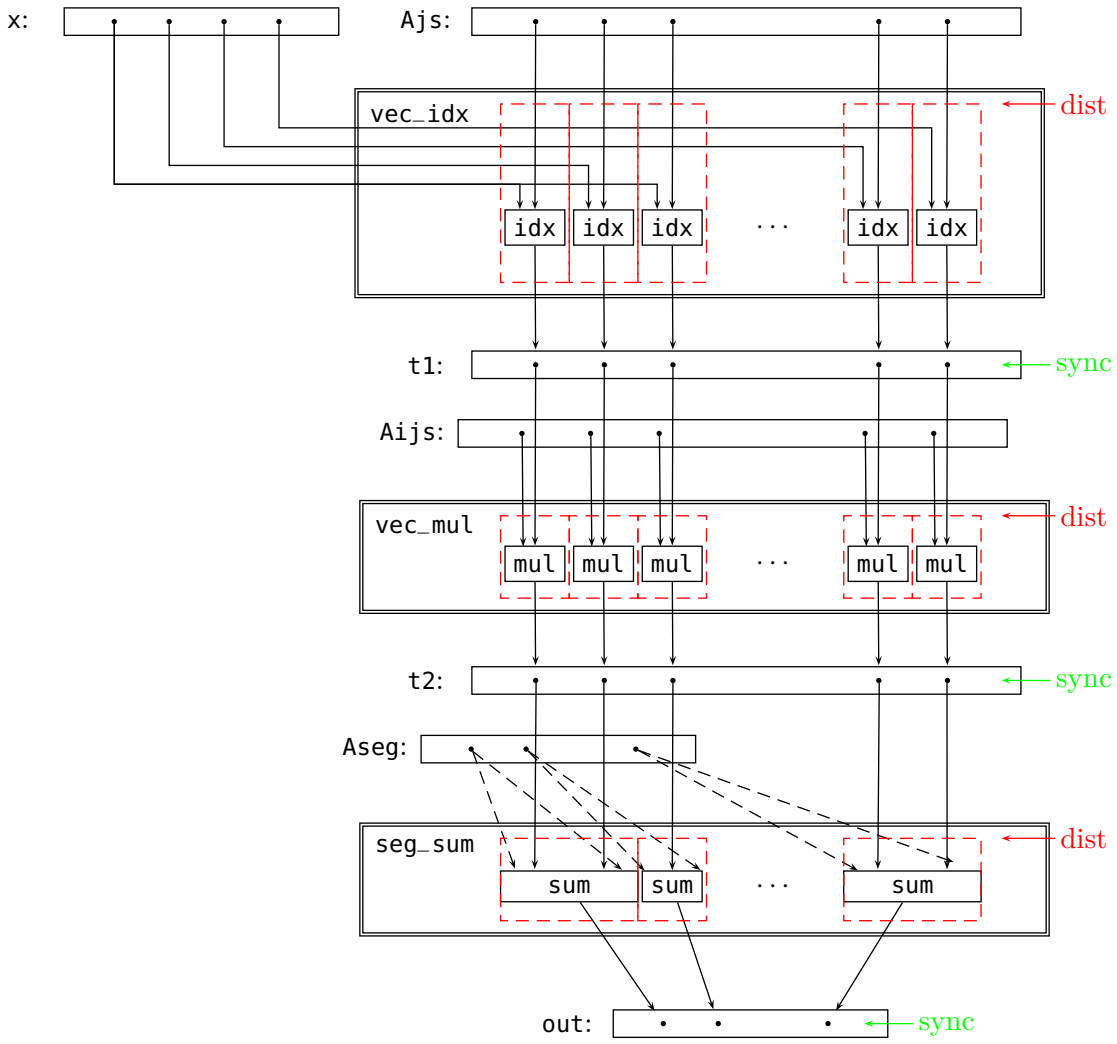


Figure 4.1: Fully vectorized parallel CSR SpMV

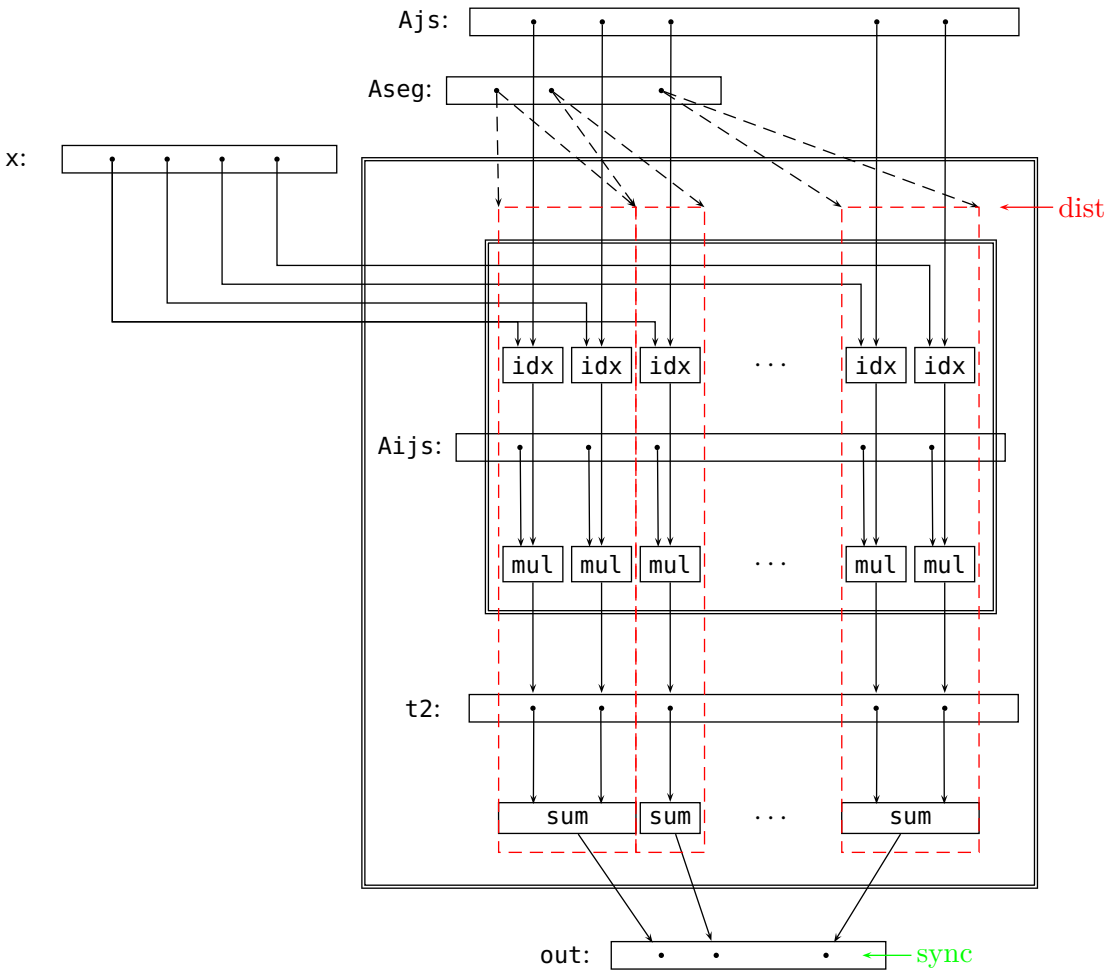


Figure 4.2: CSR SpMV with coarse-grained parallelism

We generate SPMD-style data parallel code directly from LL programs. We do so by applying a simple parallelization transformation, which converts maps with certain characteristics into parallel loops. In generating low-level parallel code we use OpenMP [20], a widespread and portable framework that provides a convenient abstraction for dispatching and synchronization of parallel threads. Our layout of nested list data structures allows us to quickly partition data for processing by parallel threads in a load balanced fashion. This parallelization scheme results in synchronization free serial execution flows within parallel threads, as exemplified in Figure 4.2. It is described in Section 4.7. As shown in Section 4.8.3, LL generated parallel code achieves substantial speedups on multicore architectures.

In this chapter we do not address SIMD parallelism and consider it a topic for future research. Modern day SIMD, such as SSE instructions on Intel processors, can be applied to loops *within* parallel threads, and is therefore orthogonal to the kind of parallelism that we generate. Furthermore, we experimented with replacing ordinary double precision reads, multiplications and additions with SSE intrinsics in the reference SpMV kernel shown in Figure 4.18. Our measurements indicated that the throughput of the SIMD variant is within 0.87–1.08 (median 0.97) of the non-SIMD one, depending on the matrix at hand. We conclude that the two-fold double precision floating point SIMD available with current SSE technology is not substantially beneficial for SpMV performance.

4.3 The LL compiler front-end

The LL front-end consists of a parser and a semantic analyzer. The latter includes a program normalization phase, type inference, inlining, and size inference.

4.3.1 Parsing, normalization, type inference and inlining

The parser implements the grammar that expands the constructs shown in Table 2.1 (p. 15) through Table 2.3 (p. 16). We use a GLR parser generator (Bison) to avoid tedious disambiguation between nested name binding constructs and tuple functions.[2] The output of the parser is an abstract syntax tree (AST), which provides a hierarchical, unambiguous view of the program structure.

Parsing is followed by a program normalization phase, where syntactic sugar—including infix/prefix operators, application, multi-arity tuples and list comprehension—is expanded into canonical forms.

The LL compiler deploys a standard Hindley-Milner [44] style global type inference algorithm. The type system accommodates type parameters, which can instantiate to arbitrary

[2] Since name binding in LL can contain nested parentheses, in parsing an expression such as `(a, (b, c)):...` it is only the colon that disambiguates a name binding from a tuple function. In LR, this causes a reduce/reduce conflict exceeding any fixed lookahead, which necessitates a delayed evaluation.

types. For example, the type of an integer constant is $\tau \to \text{int}$, where τ is a type parameter.[3] The inference of types in compound constructs is based on a prescribed set of rules corresponding to the semantics of each construct. The type of such constructs often depends on the input/output types of its components. For example, the type of a pair `(`f_1`, `f_2`)` depends on the types of f_1 and f_2. Therefore, type inference of LL programs is a recursive, bottom-up process.

Hindley-Milner type inference uses *unification* to constrain type parameters into more concrete forms. In LL, unification occurs in the following cases: (i) a pipe construct f `->` g, where the output type of f is unified with the input type of g; (ii) a pair constructor `(`f`, `g`)`, where the input types of f and g are unified; (iii) a filter construct **`filter`** f, where the output type of f is unified with bool; and (iv) a name binding construct `(x ::` τ_1`, y ::` τ_2`):` f, where the input type is unified with an optionally provided type annotation (τ_1, τ_2). A type error occurs when unification is attempted with two incompatible types: a list with a non-list, a pair with a non-pair, or two different primitive types (e.g., int and bool).

Unification is known to infer the most general types for a program. This means that the inferred types of some constructs may still contain type parameters. For example, the type of the following function for swapping the components of a pair is $(\tau_1, \tau_2) \to (\tau_2, \tau_1)$:

```
def swap = (snd, fst)
```

As with built-in functions, a parametric type of a user defined function is further constrained and instantiated when the function is used in the context of another function. For example, if the aforementioned `swap` function is to be used is the following context, its type parameters τ_1 and τ_2 will both instantiate to int due to being unified with the input type of `sub`:

```
def swap_and_sub = swap -> sub
```

That said, once an entire program is typed, only those functions whose types are fully instantiated will be considered for code generation.

Finally, the LL front-end inlines user-defined functions into their invocations, instantiating type parameters in inlined definitions accordingly. Full inlining in LL is possible thanks to the absence of recursion. While inlining may cause bloating, we find that it is necessary for improving the performance of generated code.

To demonstrate the output of the LL front-end, consider the dense matrix-vector multiplication introduced in Section 2.2.2 (p. 10). The AST representation of this function definition after parsing, normalization and type inference is shown in Figure 4.3. The double-lined box represents a function definition action (see Table 2.3); single-lined boxes represent functional blocks (see Table 2.1 and Table 2.2) and are each assigned input and output types (see Section 2.2.1, p. 9).

[3] In LL, we treat constants as functions that ignore their inputs, and hence have an arbitrary input type. This eliminates the distinction between "functions" and "values", and allows to treat both uniformly.

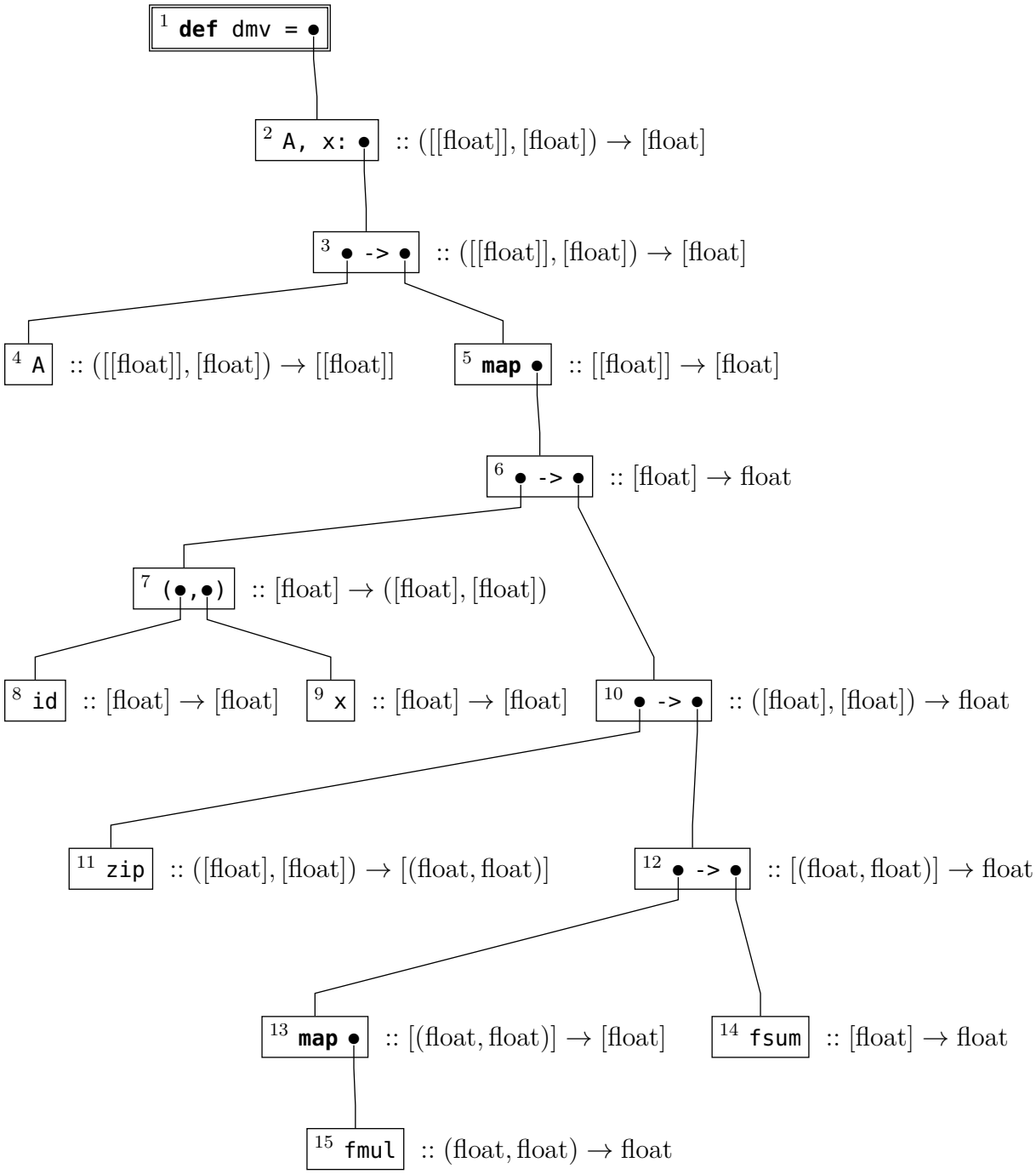


Figure 4.3: Typed AST representation of dense matrix-vector multiplication

4.3.2 Size inference

The purpose of size inference is to associate with each node in a function's AST the size of the output of that node. This information is used by the back-end to generate memory allocation directives. Size inference uses symbolic descriptors for capturing dependencies between lengths of lists at different locations in the program. The analysis propagates size descriptors via symbolic execution. The result is an over-approximation of the actual sizes of intermediate values in the computation, relative to the size of the input.

A size descriptor of a data value in LL is a symbolic construct whose hierarchical structure mirrors the type of that value. The size of an atomic value—e.g., a number—is denoted by the terminal symbol '$\cdot$'. The size of a pair is denoted by (s_1, s_2), where s_1 and s_2 denote the sizes of the first and second components, respectively. The size of a list is denoted by $[l : s]$, where l is the length of the list and s the size of elements in the list. A length is either an identifier that represents an actual runtime value, or is unknown and denoted by '?'. For nested lists, we use cascaded length descriptors of the form $l_1/l_2/\ldots/l_k$, where k is the depth of the list nesting and, for each $1 \leq i < k$, l_{i+1} is an upper bound on the sum of lengths of all the lists whose length is represented by l_i. Cascaded length descriptors are important when reasoning about the lengths of nested list objects. In particular, the fact that $l_1 \leq l_i$ for $i > 1$ later allows the code generator more freedom in allocating buffers for temporary computation values: it may extract the runtime value for l_1 and use it for allocating an output buffer; or it can use the runtime value for l_i, which may be available prior to entering the map, to make a single buffer allocation that is large enough to contain any single list output. As shown in Section 4.4, the low-level layout of nested lists is designed to capture the total length of lists at a given nesting depth, making it easy to extract at runtime.

For example, consider the AST representation of a dense matrix-vector multiplication shown in Figure 4.3. The input to this function is a pair consisting of a dense matrix of type [[float]], which is later bound to `A`, and a vector of type [float], which is later bound to `x`. The analysis associates with the input a type descriptor of the form $([l_1 : [?/l_2 : \cdot]], [l_3 : \cdot])$. This means that `A` has l_1 rows, and while the length of each row is not known a priori, and might even differ from row to row, the sum of lengths of all rows is l_2:

$$l_2 = \sum_{i=0}^{l_1-1} \texttt{len (A[}i\texttt{])}$$

The vector `x` has l_3 elements.

Figure 4.4 is analogous to Figure 4.3 and shows the result of size inference applied to the same function. Each node is associated with an input and output size descriptor. We trace the process of size inference for this simple function:

- The analysis of the binding construct (node 2) decouples the size descriptors for the dense matrix and vector and associates them with the bound names `A` and `x`, respectively. Consequently, the analysis of the named function `A` (node 4) returns the

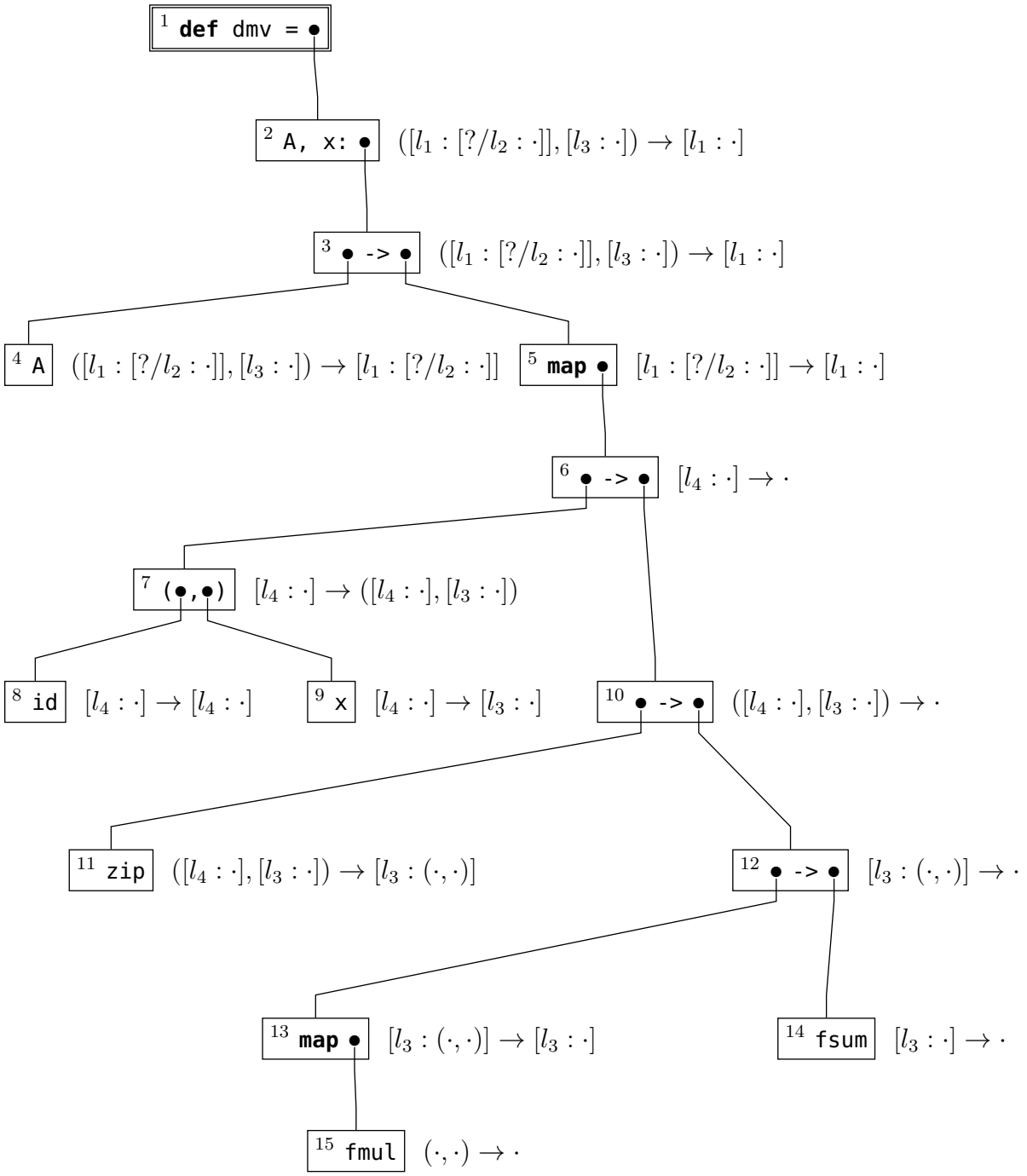


Figure 4.4: Size inference of dense matrix-vector multiplication

associated size descriptor as its output size, $[l_1 : [?/l_2 : \cdot]]$, which is forwarded via the pipe (node 3) as the input size to the outer map (node 5).

- The analysis of the outer map (node 5) peels off the outermost length of the input size descriptor—in this case, l_1—and substitutes a fresh length identifier l_4 for the unknown inner length '?'. As a result, the size descriptor at the entry to the map body (node 6) is $[l_4 : \cdot]$, where l_4 is only defined in the context of the map. This step also emits a constraint on the new length symbol, namely $l_4 \leq l_2$; we know that l_2 represents the sum of lengths of all lists whose length is represented by l_4. This constraint is important: it allows the code generator to use either the runtime value of l_4 to allocate an output buffer inside the map, or it can use the runtime value of l_2 for allocation prior to entering the map, knowing that it is over-approximating l_4.

- The output size descriptor of the pair construct (node 7) is the pair of output descriptors of its components: `id` (node 8) merely propagates its input size, $[l_4 : \cdot]$; `x` (node 9) returns the size descriptor associated with the bound name, $[l_3 : \cdot]$. Hence, the output size of node 7 is $([l_4 : \cdot], [l_3 : \cdot])$.

- The rule for inferring the output size of `zip` (node 11) does the following:

  1. Returns as output a list size descriptor whose length is one of the lengths of its two inputs, and whose element size is the pair of its inputs' element sizes. In this example, it is $[l_3 : (\cdot, \cdot)]$.
  2. Emits a constraint on the length of the two inputs, namely $l_4 = l_3$.

  Note that every built-in LL function has a similar rule for inferring its output size given its input size. These rules may introduce new length symbols and constraints. The detailed list of inference rules is omitted.

- The analysis of the inner map (node 13) peels off the outermost length—in this case, l_3—and propagates $(\cdot, \cdot)$ to the inner function, `fmul` (node 15). The rule for the latter returns $\cdot$. Leaving the map, the analysis re-wraps this size descriptor with the previously removed length, resulting in $[l_3 : \cdot]$.

- The analysis of `fsum` (node 14) returns $\cdot$. This is propagated back as the output size of nodes 12, 10 and 6.

- Finally, the analysis of the outer map (node 5) re-wraps the size resulting from the body function (node 6) with the previously removed length, resulting in $[l_1 : \cdot]$. This size is propagted back as the output size of nodes 3 and 2.

As shown in this example, the size inference analysis revealed the necessary information for allocating sufficiently large buffers for values emitted during the computation of the function. Specifically, it found that:

1. The buffer necessary to store the output of the inner map (node 13) should be of length l_3, which is the length of the input right-hand side vector.

2. The buffer necessary to store the output of the outer map (node 5)—which is also the output of the whole function—should be of length l_1, the same as the length (i.e., number of rows) of the input matrix.

Constraints emitted during the process can be used for validating runtime invariants by adding assertions to the generated code. In this example, upon entering the body of the outer map, the code generator can emit a statement of the form `assert (`l_4 `==` l_3`)`, where both lengths are replaced with expressions that extract the appropriate list lengths.

The size inference analysis applies nicely to the benchmarks evaluated in Section 4.8, allowing precise and efficient allocation of buffers (see Section 4.5):

- For CSR SpMV, it precisely infers that (i) the size of the intermediate list of products generated for each compressed row is equal to the length of that row, and does not exceed the sum of lengths of all the rows; and (ii) the size of the output vector is the same as the number of rows in the compressed input matrix.

  These map, respectively, to the outputs of nodes 7 and 5 (as well as 3 and 2) in Figure 4.10. In the example generated code in Figure 4.11, they determine the sizes of the temporary `v4` (declared in line 13) and the output value `*out`.

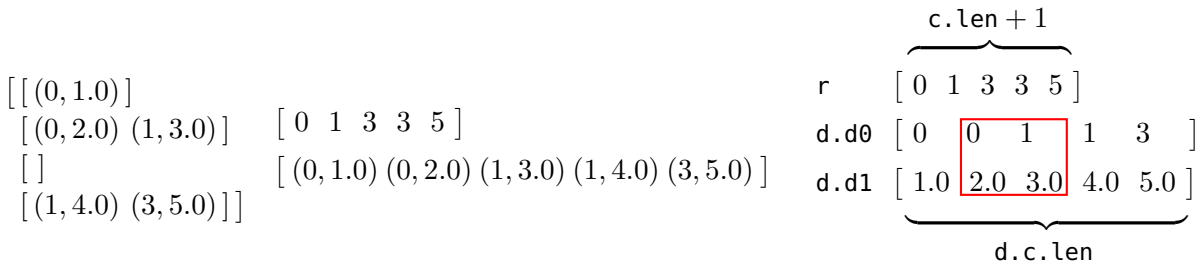
- In the case of RBCSR the analysis infers that (i) the size of the list of products computed as part of the dense block multiplication equals the column dimension of the block; (ii) the computation of dense products along a row results in a list whose length is bounded by the total number of blocks, and its elements are vectors whose size is the row dimension of blocks; (iii) the outermost map over the rows of the blocked matrix results in a list whose length is equal to the length of the input matrix (i.e., the number of rows of blocks), and its elements are vectors whose length is the row dimension of blocks.[4]

These facts allow for a sound and efficient allocation scheme that can be generated at compile-time.

4.4 Representation of high-level datatype

The translation of LL programs relies on a data representation scheme that maps operations on high-level data values to C objects. The goal of this representation is two-fold. On the one hand, it implements an efficient representation of nested lists and pairs that minimizes memory footprint and improves the runtime characteristics of memory usage. This is

[4]Precise size inference for RBCSR is made possible thanks to incorporating fixed-length list inference (see Section 4.6.1).



(a) Nested (b) Flattened list of lists (c) Segregated list of pairs

Figure 4.5: Flattening of a CSR representation

achieved via datatype flattening, which maps arbitrarily nested data values onto an aggregate of flat vectors. On the other hand, it hides the complexities of a low-level representation and facilitates a straightforward translation process of LL programs into C. It does so by providing a data abstraction layer that consists of a coherent type hierarchy and an API for manipulating objects that represent nested data values.

4.4.1 Datatype flattening

This technique maps a high-level nested datatype onto a small number of flat arrays containing primitive data values. Flattened data structures are necessary for efficient sequential processing as well as data parallelism. In the sequential case, they minimize memory fragmentation and dramatically improve the temporal locality of looping constructs, maximizing cache utilization and improving the effectiveness of prefetching. Flattening also turns high-level operations like zip/unzip and concatenation into trivial, constant time ones. In the parallel context, datatype flattening facilitates *vectorization*, which is the basis for NESL's data parallelization, described in Section 4.2.2. The idea was first introduced in the early work of Blelloch and Sabot [13].

Flattening is based on two principles: (i) a list of lists of elements of type τ maps to a long, consecutive list of τ's and a buffer of segment descriptors (integers) determining the start/end indexes of each of the inner lists; and (ii) a list of pairs of types τ_1 and τ_2 maps to two lists of elements of types τ_1 and τ_2, respectively.

Figure 4.5 demonstrates flattening on a small CSR matrix data object: (a) the high-level, nested data structure; (b) a segmented representation obtained by flattening of the nested list; (c) a pair of segmented lists obtained by segregating the values of the pairs in the list. Note that the two resulting segmented lists (denoted by `d.d0` and `d.d1`) share the same segment buffer (denoted by `r`). The end result is equivalent to a standard array-based representation of CSR used by a majority of scientific applications.[5] Finally, flattening can be

[5]Notice that the flattened representation in Figure 4.5(c) is identical to the low-level data structure shown

Type specification $\mathcal{T}[\![\tau]\!]$:

$$
\begin{aligned}
\mathcal{T}[\![\text{int}]\!] &\rightarrow \texttt{int} \\
\mathcal{T}[\![\text{float}]\!] &\rightarrow \texttt{double} \\
\mathcal{T}[\![\text{bool}]\!] &\rightarrow \texttt{bool} \\
\mathcal{T}[\![(\tau_0,\tau_1)]\!] &\rightarrow \texttt{struct \{}\mathcal{L}[\![\tau_0]\!]\texttt{ d0; }\mathcal{L}[\![\tau_1]\!]\texttt{ d1;\}} \\
\mathcal{T}[\![[\tau]]\!] &\rightarrow \texttt{struct \{l\_t c; }\mathcal{T}'[\![[\tau],\varepsilon]\!]\texttt{\}} \\
\\
\mathcal{T}'[\![[(\tau_0,\tau_1)],i]\!] &\rightarrow \mathcal{T}'[\![[\tau_0],i\texttt{0}]\!]\ \mathcal{T}'[\![[\tau_1],i\texttt{1}]\!] \\
\mathcal{T}'[\![[[\tau]],i]\!] &\rightarrow \texttt{size\_t *r}i\texttt{; }\mathcal{L}[\![[\tau]]\!]\texttt{ d}i\texttt{;} \\
\mathcal{T}'[\![[\tau],i]\!] &\rightarrow \mathcal{L}[\![\tau]\!]\texttt{ *d}i\texttt{;}
\end{aligned}
$$

Canonical type name $\mathcal{L}[\![\tau]\!]$:

$$
\begin{aligned}
\mathcal{L}[\![\text{int}]\!] &\rightarrow \texttt{int} \\
\mathcal{L}[\![\text{float}]\!] &\rightarrow \texttt{double} \\
\mathcal{L}[\![\text{bool}]\!] &\rightarrow \texttt{bool} \\
\mathcal{L}[\![\tau]\!] &\rightarrow \mathcal{L}'[\![\tau]\!]\texttt{\_t} \\
\\
\mathcal{L}'[\![\text{int}]\!] &\rightarrow \texttt{i} \\
\mathcal{L}'[\![\text{float}]\!] &\rightarrow \texttt{f} \\
\mathcal{L}'[\![\text{bool}]\!] &\rightarrow \texttt{b} \\
\mathcal{L}'[\![(\tau_0,\tau_1)]\!] &\rightarrow \texttt{p}\mathcal{L}'[\![\tau_0]\!]\mathcal{L}'[\![\tau_1]\!] \\
\mathcal{L}'[\![[\tau]]\!] &\rightarrow \texttt{l}\mathcal{L}'[\![\tau]\!]
\end{aligned}
$$

Figure 4.6: Mapping of LL types to C

applied to any nested type of arbitrary depth by recursively applying the two aforementioned principles.

4.4.2 Data abstraction layer

The data abstraction layer does two things: (i) it systematically packs flattened data representations into hierarchical C type definitions; and (ii) it generates a set of API methods for manipulating objects of these types. This allows us to separate the handling of data values from the generation of code that implements the flow of data values through the program. At the same time, data abstraction does not prohibit optimization of data representation. As we show in Section 4.6, optimizing the data representation is done by enhancing the type declarations together with the methods that are used for manipulating them.

The first role of the data abstraction layer is to map all LL types that are being used by the program to encapsulated and modular type definitions in C. The result of mapping an LL type τ is a C type declaration:

$$\texttt{typedef }\mathcal{T}[\![\tau]\!]\ \mathcal{L}[\![\tau]\!]\texttt{;}$$

Here, $\mathcal{T}[\![\tau]\!]$ generates a C type specification and $\mathcal{L}[\![\tau]\!]$ a canonical type name for τ. These two functions are defined in Figure 4.6. $\mathcal{T}[\![\tau]\!]$ implements the recursive datatype flattening described in Section 4.4.1. However, it only applies one of the two flattening principles, which applies to the type at hand, τ. It then uses $\mathcal{L}[\![\tau']\!]$ to declare a field that corresponds to a nested type τ', and delegates further flattening to the generation of $\mathcal{T}[\![\tau']\!]$. The names generated by $\mathcal{L}[\![\tau]\!]$ take the form $\texttt{p}t_1t_2$ when τ is a pair type and $\texttt{l}t$ when it is a list type.

in Figure 2.1(b) (p. 17).

```c
/* Type: [[float]] */
typedef struct {
  l_t c;      /* Bookkeeping data */
  size_t *r;  /* Row segments */
  lf_t d;     /* Flattened rows */
} llf_t;

/* Type: [float] */
typedef struct {
  l_t c;      /* Bookkeeping data */
  double *d;  /* Row content */
} lf_t;
```

(a) Dense matrix [[float]]

```c
/* Type: [[(int, float)]] */
typedef struct {
  l_t c;       /* Bookkeeping data */
  size_t *r;   /* Row segments */
  lpif_t d;    /* Flattened rows */
} llpif_t;

/* Type: [(int, float)] */
typedef struct {
  l_t c;       /* Bookkeeping data */
  int *d0;     /* Column indexes */
  double *d1;  /* Nonzero values */
} lpif_t;
```

(b) CSR matrix [[(int, float)]]

Figure 4.7: Implementation of LL matrix types in C

For example, `llf_t` stands for a "list of lists of floats" (used for representing a dense matrix) and `llpif_t` is a "list of lists of pairs of integers and floats" (representing a CSR matrix).

A simple example of a hierarchical, flattened datatype is that of a dense matrix whose high-level type is [[float]]. The result of datatype generation is a C type `llf_t`, which in turn uses another C type `lf_t` for storing the concatenated rows of the matrix. The definition of the two types in shown in Figure 4.7(a). Here, the buffer `r` in the encapsulating type, `llf_t`, contains the segment descriptors that split the contiguous list of numerical values `d.d` into rows. Fields of type `l_t` are used for internal bookkeeping of their respective objects, such as storing the lengths of lists: `c.len` is length of the segment list and `d.c.len` is the total length of the flattened list of matrix rows; `c.size` and `d.c.size` store the allocated size of these buffers, respectively. Note that `lf_t` is a valid representation for [float] and can be used elsewhere. This leads to a consistent use of types throughout the generated code.

As a slightly more complicated example, consider the type [[(int, float)]] that is used for CSR matrices such as the one in Figure 4.5(a). The result of datatype generation is a C type `llpif_t`, which uses another C type `lpif_t` for storing the actual contiguous data elements of the nested lists of pairs. The definition of the two types is shown in Figure 4.7(b). The mapping of fields of the encapsulating type, `llpif_t`, to actual data content is exemplified in Figure 4.5(c). Notice how the list of pairs is stored using two separate buffers, `d.d0` (for column indexes) and `d.d1` (for nonzeros). The lengths stored in `c.len` and `d.c.len` in this case are 4 and 5, respectively.

$$
\begin{aligned}
P_{\text{len}}(v) &= P(v, P'_{\text{len}}) \\
P_{\text{term}}(v) &= P(v, P'_{\text{term}}) \\
P_{\text{mono}}(v) &= P(v, P'_{\text{mono}})
\end{aligned}
$$

$$
P(v,p) = \begin{cases}
P(v.\texttt{d0}, p) \wedge P(v.\texttt{d1}, p) & \text{if } \textbf{\texttt{typeof(}}v\texttt{)} = \mathcal{L}[\![(t_0, t_1)]\!] \\
P'(v, p, [t], \varepsilon) & \text{if } \textbf{\texttt{typeof(}}v\texttt{)} = \mathcal{L}[\![[t]]\!] \\
\mathit{true} & \text{otherwise}
\end{cases}
$$

$$
P'(v,p,[t],i) = \begin{cases}
P'(v, p, [t_0], i\texttt{0}) \wedge P'(v, p, [t_1], i\texttt{1}) & \text{if } t = (t_0, t_1) \\
p(v, [t], i) \wedge P(v.\texttt{d}i, p) & \text{if } t = [t'] \\
p(v, [t], i) & \text{otherwise}
\end{cases}
$$

$$
P'_{\text{len}}(v,[t],i) = \begin{cases}
v.\texttt{c.len} < v.\texttt{c.size} & \text{if } t = [t'] \\
v.\texttt{c.len} \leq v.\texttt{c.size} & \text{otherwise}
\end{cases}
$$

$$
P'_{\text{term}}(v,[t],i) = \begin{cases}
v.\texttt{r}i\texttt{[}v.\texttt{c.len]} = v.\texttt{d}i\texttt{.c.len} & \text{if } t = [t'] \\
\mathit{true} & \text{otherwise}
\end{cases}
$$

$$
P'_{\text{mono}}(v,[t],i) = \begin{cases}
\forall k \;.\; 0 \leq k < v.\texttt{c.len} \implies v.\texttt{r}i\texttt{[}k\texttt{]} \leq v.\texttt{r}i\texttt{[}k+1\texttt{]} & \text{if } t = [t'] \\
\mathit{true} & \text{otherwise}
\end{cases}
$$

Figure 4.8: Data structure integrity constraints

`init`$_\tau$ (`t:` τ, l_1`: int, ...,` l_k`: int`)	Initialize t with nested lengths l_1 through l_k
`len (`s`: [`τ`])`	Return the number of elements in list s
`get`$_{[\tau]}$ (`t:` τ, s`: [`τ`],` j`: int`)	Get the j-th element of s into t
`set`$_{[\tau]}$ (`t: [`τ`],` s`:` τ, j`: int`)	Set the (fixed size) j-th element of t to s
`iter`$_{[\tau]}$ (i`: [`τ`]′,` s`: [`τ`])`	Initialize i for iterating over s
`read`$_{[\tau]}$ (`t:` τ, i`: [`τ`]′)`	Read the next element of iterator i to t
`append`$_{[\tau]}$ (`t:` τ, s`: [`τ`])`	Setup t for appending an element to s
`write`$_{[\tau]}$ (`t: [`τ`],` s`:` τ`)`	Commit appended value in s to t
`pair (t:` (τ_1, τ_2)`,` s_1`:` τ_1, s_2`:` τ_2`)`	Form a pair t of the values of s_1 and s_2

Table 4.1: Data layer API used by generated code

Several invariants govern the integrity of C data objects whose types are generated by the process described in Figure 4.6. They ensure that the content of segment buffers are sane and that the lengths of different buffers are properly correlated. These invariants are specified in Figure 4.8. The higher-order predicates P and P' follow the inductive structure of the nested type, whereas the specialized predicates P_{len}, P_{term} and P_{mono} enforce particular properties:

Sanity of buffer lengths. For each bookkeeping field c that governs a data buffer, c`.len` $\leq$ c`.size`; if c governs a segments buffer, then c`.len` $<$ c`.size`.[6] Note that a single bookkeeping field may govern both data and segment descriptor buffers, in which case the stricter requirement applies. This predicate is defined by P_{len}.

Consistency of terminal segment value. For each segments buffer r that is controlled by a field c and governs a list field d, the index that terminates the final segment of the list must equal the length of the list, namely r`[`c`.len]` $=$ d`.c.len`. This predicate is defined by P_{term}.

Monotonicity of segment values. For each segments buffer r that is controlled by field c, r`[`k`]` $\leq$ r`[`$k+1$`]` for all $0 \leq k <$ c`.len`. This predicate is defined by P_{mono}.

The second role of the data abstraction layer is to provide an API for compound datatypes, which hides the complexities of handling segment offsets, multiple assignments and memory management, and allows simple code generation. The API is shown in Table 4.1. Note that some of these methods have type-specific implementation that is generated by the type abstraction layer (e.g., `get`$_{[\tau]}$), while others are type agnostic (e.g., `pair`). All of these methods induce a constant time overhead: read/write iterators are implemented via aliasing to the

[6]Segment values encode both the start and end indexes of each segment, hence the length of a segment buffer must be strictly greater than the number of sublists it specifies.

data buffers; the `set` operation, which performs actual copying, is only provided for lists of elements of a known fixed size and hence induces a constant time overhead as well. Note that the actual methods are implemented as macros, and hence expand to operations at the caller's context with no invocation overhead. Following are example `get` methods for types [[float]] (`llf_t`) and [[(int, float)]] (`llpif_t`), used for extracting the `j`-th row of a dense and a CSR matrix, respectively.

```
#define get_llf(t, s, j) \
  lf_t t; \                              /* Declare target object */
  t.c.len = (s).r[j + 1] - (s).r[j]; \   /* Compute sublist length */
  t.d = (s).d.d + (s).r[j];              /* Alias value array */

#define get_llpif(t, s, j) \
  lpif_t t; \                            /* Declare target object */
  t.c.len = (s).r[j + 1] - (s).r[j]; \   /* Compute sublist length */
  t.d0 = (s).d.d0 + (s).r[j]; \          /* Alias index array */
  t.d1 = (s).d.d1 + (s).r[j];            /* Alias nonzero value array */
```

For example, applying `get_llpif` with `j` equals 1 to the CSR representation in Figure 4.5(c) will initialize a target object of type `lpif_t` (see definition above). Its `d0` and `d1` fields will alias the subparts of the index and nonzero value buffers, respectively, that correspond to the second compressed row. The length that will be stored in the `c` field of the target object is 2. The actual data that is referenced by this object is highlighted in red in Figure 4.5(c).

Data layer API method implementations are generated automatically for each type that is being used in the program. In general, their generation follows the hierarchical structure of LL types and their corresponding C types. This process is guided by the same rules that were used for generating type definitions, shown in Figure 4.6. The full specification of the API method generation process is omitted.

4.5 Syntax-directed translation

The LL compiler back-end applies a set of rules that entails a recursive translation of the program. It is applied to the root of each function definition, generating a C implementation for each. A subset of these translation rules is shown in Figure 4.9. One kind of rules, denoted by $\mathcal{F}[\![a]\!]$, governs the translation of actions, such as a function definition. The second kind of rules, denoted by $\mathcal{F}'[\![e, \Gamma, v_{\text{in}}, v_{\text{out}}]\!]$, translates functional building blocks and takes four arguments: e is the functional construct to be translated, Γ is the environment

[7] Note that the implementation of some built-in functions is specialized for different types. The code generator annotates the invocation of such functions accordingly.

$$\mathcal{F}[\![\texttt{def}\ f\texttt{=}\ e]\!] \rightarrow$$
```
void def_f (L[[κ'(e)]] *out, L[[κ(e)]] *in)
{
  F'[[e, ∅, *in, *out]]
}
```

$$\mathcal{F}'[\![v_1\texttt{,}\ v_2\texttt{:}\ e, \Gamma, v_{\text{in}}, v_{\text{out}}]\!] \rightarrow$$
```
L[[τ1]] v1;
F'[[fst, Γ, v_in, v1]]
L[[τ2]] v2;
F'[[snd, Γ, v_in, v2]]
F'[[e, Γ ∪ {v1, v2}, v_in, v_out]]
```
where $\kappa(e) = (\tau_1, \tau_2)$

$$\mathcal{F}'[\![e_1\ \texttt{->}\ e_2, \Gamma, v_{\text{in}}, v_{\text{out}}]\!] \rightarrow$$
```
L[[κ'(e1)]] v;
F'[[e1, Γ, v_in, v]]
F'[[e2, Γ, v, v_out]]
```
where v is fresh

$$\mathcal{F}'[\![f, \Gamma, v_{\text{in}}, v_{\text{out}}]\!] \rightarrow$$
```
{ v_out = f;        if f ∈ Γ
{ f (v_out, v_in);  otherwise[7]
```

$$\mathcal{F}'[\![\texttt{map}\ e, \Gamma, v_{\text{in}}, v_{\text{out}}]\!] \rightarrow$$
```
append_[κ'(e)] (v', v_out);
int v_j;
for (v_j = 0; v_j < len (v_in);
     v_j++) {
  get_[κ(e)] (v, v_in, v_j);
  F'[[e, Γ, v, v']]
  write_[κ'(e)] (v_out, v');
}
```
where v_{j}, v, v' are fresh

$$\mathcal{F}'[\![(e_1\texttt{,}\ e_2), \Gamma, v_{\text{in}}, v_{\text{out}}]\!] \rightarrow$$
```
L[[κ'(e1)]] v1;
F'[[e1, Γ, v_in, v1]]
L[[κ'(e2)]] v2;
F'[[e2, Γ, v_in, v2]]
pair (v_out, v1, v2);
```
where v_1, v_2 are fresh

Figure 4.9: Syntax-directed translation rules

of bound names, v_{in} is an input identifier and v_{out} is an output identifier. Translation rules makes use of the input and output types of the functional unit to which they are applied, extracted using $\kappa(e)$ and $\kappa'(e)$, respectively. The function $\mathcal{L}[\![\tau]\!]$ obtains the name of the C type that implements the LL type τ (see Figure 4.6). The translation also uses the datatype API methods shown in Table 4.1.

The translation rules constitute a mapping from the LL high-level dataflow model to the more traditional control flow and dataflow used in imperative programs:

- The translation of name binding declares new variables and assigns to them the respective components of the input. It then translates the inner function in an environment that contains these newly declared variables.
- A pipeline (left-to-right functional composition) can be viewed as an explicit dataflow edge. Its translation propagates the output of the first function as input to the second function using a temporary.
- Named functions translate to either (i) copying from a bound name variable, if the name corresponds to a variable that is present in the current environment; or (ii) a function invocation.
- A map translates to a loop that iterates over the elements of the list, applies the inner function to each, and appends the results to the output list.
- Pair construction merely embeds its two inputs in a single output variable.

Note that the rules in Figure 4.9 are simplified for presentation purposes. For example, they do not handle name scoping, buffer management, runtime checks and error conditions.

Consider the CSR SpMV kernel that was introduced in Section 2.4.1 (p. 17). Its typed AST representation is shown in Figure 4.10. Figure 4.11 shows the C code that is generated from this AST using the rules in Figure 4.9. We annotated certain lines with comments that specify the LL construct for which code is being generated, along with the identifiers that specify the input/output dataflow.

Notice how buffer allocation is handled in this example. Thanks to size inference, we can determine that the length of the output (`*out`) equals the number of rows in the input matrix (`v1`) and initialize it accordingly (line 12). Also, the length of the temporary buffer `v4` is known to be equal to the length of each row processed by the inner map (loop in lines 26–48), or `len (v3)`. However, size inference also tells us that this buffer is no larger than the sum of lengths of all rows, or `len (v1.d)`. It is therefore beneficial to initialize it *once* outside the loop nest (line 14) and reset it to "empty" right before the inner loop (line 21).

It should be noted that the code generated by the LL compiler does not currently implement proper memory deallocation. We anticipate that, due to the cycle-free nature of

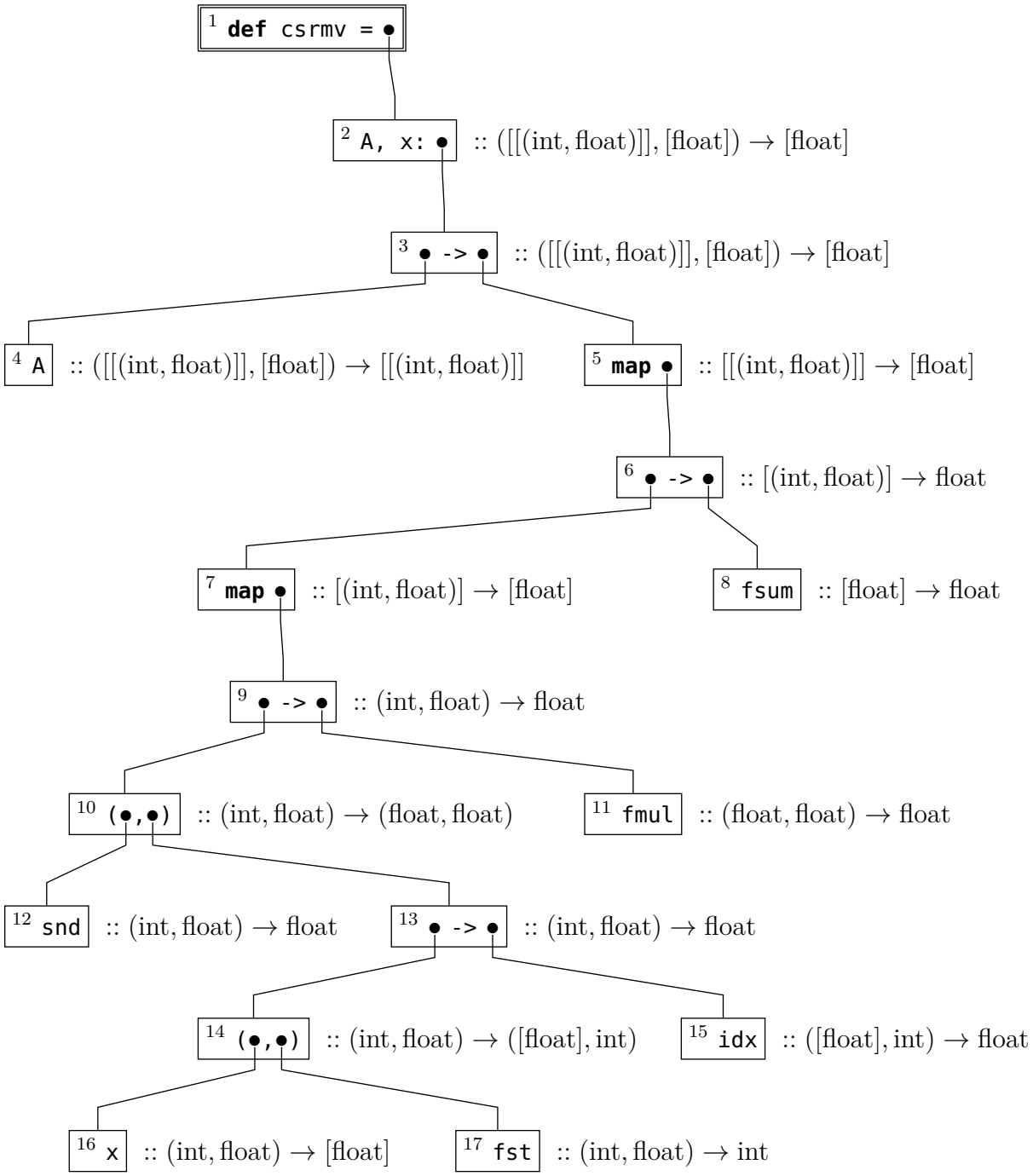


Figure 4.10: Typed AST representation of CSR SpMV

```
1  void def_csrmv (lf_t *out, pllpiflf_t *in)
2  {
3    llpif_t A;                              /* A, x: A -> map (...) / *out <- *in */
4    fst (A, *in);
5    lf_t x;
6    snd (x, *in);
7
8    llpif_t v1;                                  /* A -> map (...) / *out <- *in */
9
10   v1 = A;
11
12   init_lf (*out, len (v1));                       /* Initialize result vector. */
13   lf_t v4;
14   init_lf (v4, len (v1.d));        /* Initialize vector of products (advanced). */
15
16   append_lf (v2, *out);                /* map (map (...) -> fsum) / *out <- v1  */
17   int j1;
18   for (j1 = 0; j1 < len (v1); j1++) {
19     get_llpif (v3, v1, j1);
20
21     empty_lf (v4);                           /* map (...) -> fsum /   v2 <- v3  */
22
23     append_lf (v5, v4);               /* map ((..., ...) -> fmul) /   v4 <- v3  */
24     int j2;
25     for (j2 = 0; j2 < len (v3); j2++) {
26       get_lpif (v6, v3, j2);
27
28       pff_t v7;                 /* (snd, (x, fst) -> idx) -> fmul /   v5 <- v6  */
29
30       double v8;                         /* (snd, (x, fst) -> idx) /   v7 <- v6  */
31       snd (v8, v6);
32
33       double v9;
34       plfi_t v10;                               /* (x, fst) -> idx /   v9 <- v6  */
35
36       lf_t v11;                                       /* (x, fst) /  v10 <- v6  */
37       v11 = x;
38       int v12;
39       fst (v12, v6);
40       pair (v10, v11, v12);
41
42       idx_plfi (v9, v10);                                 /* idx /   v9 <- v10 */
43
44       pair (v7, v8, v9);
45
46       fmul (v5, v7);                                    /* fmul /   v5 <- v7  */
47       write_lf (v4, v5);
48     }
49
50     fsum (v2, v4);                                      /* fsum /   v2 <- v4  */
51     write_lf (*out, v2);
52   }
53 }
```

Figure 4.11: Generated code for CSR SpMV

aliasing in LL programs, a reference counting based scheme will constitute a precise and efficient solution.[8]

Finally, the performance characteristics of this naïvely generated C code is suboptimal compared to what is reasonable to expect from a handwritten variant. Most notably, the use of memory buffers for storing intermediate results in loops (i.e., products of compressed rows) is highly detrimental to the overall performance of the code: not only does it induce unnecessary memory write and read operations, it also pollutes the cache and tampers with locality of reference. In the following section we show techniques for eliminating such inefficiencies.

4.6 Sequential optimization

Loop-based code that is generated using the naïve translation scheme shown in Section 4.5 suffers from several shortcomings and needs to be further optimized in order to perform on par with handwritten C code. Inefficiencies are attributed to (i) redundant reads/writes to memory, which can be eliminated through loop fusion, data structure optimization, and streamlined indexing of buffers; and (ii) unoptimized loops, which can be optimized via a more rigorous type analysis. This section presents three separate techniques that our compiler deploys in order to eliminate the above redundancies. Our goal is not to perform low-level optimizations such as register mapping and loop unrolling; rather, it is to make the generated code amenable to such optimization by an underlying C compiler.

As an overarching example, consider the register-blocked CSR (RBCSR) SpMV shown in Section 2.5.2 (p. 28). It reprises here, with the inner dense matrix-vector function (`dmv`) replaced with its definition.[9] We assume that the dimensions of blocks are fixed at r rows and c columns and that they are computation constants.

```
def rbcsrmv (A, x) =
 xb = block (c, x):
  A ->
  [[(snd, xb[fst]) ->
    Aij, xj:
     Aij -> [(id, xj) -> zip -> [fmul] -> fsum]] -> fvsum] ->
  concat
```

To avoid the full complexity of this relatively subtle kernel we break it down into smaller parts: (i) we use the inner loops that compute the dense matrix-vector multiplication of

[8] A similar scheme is used by the interpreter of the K language—a descendant of APL that is used as the basis for high-throughput database systems—and is known to be efficient and robust.

[9] Inlining of user-defined functions is part of the processing done by the LL compiler front-end.

blocks—highlighted by a red frame—in demonstrating the effect of fixed list length inference (Section 4.6.1); (ii) we focus on the composition of maps with summation operations—highlighted by a green frame—in demonstrating the effect of reduction fusion (Section 4.6.2); finally, (iii) we focus on the looping structure of the whole function in demonstrating the effect of nested index flattening (Section 4.6.3).

4.6.1 List fixed-length inference

The goal of this technique is to exploit information about lengths of lists that is known at compile-time in facilitating several code improvements. We extend our high-level types for capturing constant list lengths and deploy a propagation-based inference for finding such lengths. We then extend the back-end to take advantage of this knowledge in generating enhanced datatype layouts and emitting loop code that is more amenable to further optimization.

Consider the multiplication of dense blocks inside the RBCSR SpMV function shown above (highlighted by a red frame). Without additional information, the inferred types for `Aij` and `xj` are, respectively, [[float]] and [float]. This can be seen in the typed AST of the dense multiplication function in Figure 4.3 (where they are named `A` and `x`, respectively). This leads to three separate yet related outcomes:

1. The data abstraction layer will generate an ordinary low-level nested list representation for `Aij`, as shown in Figure 4.7(a). The outer aggregate, `llf_t`, will include a segment buffer `r`. However, as the length of rows in blocks is fixed at c, we can predict that the sequence of values in the segment buffer must be in increments of c; and since the number of rows is r, we can predict that its length will be $r + 1$. An example sequence of segment values is $0, c, 2c, \ldots, rc$.[10]

   In accordance with the data structure layout, the methods generated for accessing rows inside a dense block such as `get_llf` are obliged to refer to the segment buffer in determining the offset and length of a given row. The result is a suboptimal data representation and a redundant computation that uses it: using segment buffers for representing fixed length sublists results in excess memory usage and unnecessary read-/write operations.

   It is also of note that the relaxed view on the dimensions of dense blocks has further effects outside the dense multiplication sub-function:

   - The input matrix `A`—of type [[(int, [[float]])]], or `llpillf_t`—uses a redundant segment buffer for determining the boundaries of whole blocks, although they are known to be increments of the number of lines in a block, r.

[10]Note that segment offsets are not necessarily zero-based so there are many possible such sequences.

- The blocked vector `xb`—of type [[float]], or `llf_t`—uses a redundant segment buffer for determining the boundaries of blocks, although they are known to be increments of c.
- The output of the second map—of type [[float]], or `llf_t`—uses a redundant segment buffer for determining the boundaries of the dense matrix-vector products, although they are known to be increments of r.

By extension, the data abstraction methods that govern the handling of these low-level type representations are suboptimal.

2. As shown in Figure 4.9 and exemplified in Figure 4.11, the boundaries of the loops that correspond to the two maps over dense blocks will be determined at runtime by the length of the list that is being iterated upon. This is suboptimal because we can predict that these two loops will iterate exactly r and c times, respectively. Given this knowledge, and given that r and c are often small constants, we could take advantage of unconditional loop unrolling and eliminate looping overhead entirely.

3. The output of the inner loop—the one that implements `[fmul]`—will write to a buffer whose size depends on some runtime length (see Section 4.3.2). This is suboptimal because we can predict that it will be of size c. Had we retained this knowledge, we could declare this buffer as an array on the stack. This would have saved the dynamic allocation overhead in this case. But more importantly, in conjunction with unconditional loop unrolling it would allow the C compiler to map array cells to registers and to substitute array dereferences in loops with operations on registers.[11]

We solve these inefficiencies by extending LL types to capture constant list lengths. The user can annotate certain values in the program with nested list types that are known to have a fixed length. The compiler propagates this knowledge—along with knowledge gathered from other numerical constants in the program—and enriches the previously inferred generic list types with constant qualifiers. The compiler back-end, in turn, uses this knowledge for (i) generating efficient data representation and API methods for nested types with constant list lengths; (ii) setting constant loop boundaries; and (iii) allocating fixed length buffers on the stack. As is evident from the evaluation in Section 4.8.2, exploiting these properties at the level of C code is paramount to realizing the benefits of register blocking.

Inferring length-enriched list types. Fixed-length extended list types are of the form $[\tau]_n$, where $n \in \mathbb{N}$. For example, a user can specialize a dense matrix-vector multiplication kernel for a specific block size, say 4×3, as follows:

[11] Taking full advantage of such optimizations in the RBCSR case also depends on proper fusion of the `fsum` operation into the preceding loop, see Section 4.6.2.

```
def dmv (A :: [[float]{3}]{4}, x :: [float]{3}) = ...
```

The compiler front-end enriches list types with fixed-length qualifiers via an abstraction-based propagation algorithm. Abstract descriptors for this analysis are analogous to length-extended LL types, with the added feature that integer types can optionally be marked by constant qualifiers as well. This allows the analysis to track confluences between integer constants and constant lengths of lists.

The analysis itself is similar to the size inference described in Section 4.3.2. Starting from the root, it extracts length-enriched user type annotations and merges them with previously inferred generic types to create an input abstract descriptor; it then uses node-specific rules to propagate constant information through the AST. The result is a function whose typed nodes are enriched with constant qualifiers.

As an example, consider a length-enriched dense matrix-vector multiplication. Its AST version is identical to the one in Figure 4.3. But now we have the additional knowledge that the input type of the function body (node 2) is $([[\text{float}]_3]_4, [\text{float}]_3)$. Following are the highlights of the analysis process:

- The inferred length-enriched descriptor at the input to the first map (node 5) is $[[\text{float}]_3]_4$. The outer list dimension is peeled off, propagating $[\text{float}]_3$ as the input to the inner function (node 6).
- It is easy to see how the inferred length-enriched abstraction descriptor at the input to the `zip` function (node 11) is $([\text{float}]_3, [\text{float}]_3)$. Propagation of length-enriched types through this node results in $[(\text{float}, \text{float})]_3$.
- The latter is propagated as the input descriptor to the second map (node 13). The analysis of this node infers that the output descriptor is $[\text{float}]_3$.
- Finally, the analysis of the outer map (node 5) yields a length-enriched output descriptor of $[\text{float}]_4$.

Evidently, the analysis discovered the necessary facts about fixed-length qualifiers for the back-end to optimize data object representation and allocation, as well as loop boundaries.

The process exemplified above can be applied to the fully fledged RBCSR SpMV that is annotated as follows:

```
def rbcsrmv (A :: [[(int, [[float]{3}]{4})]], x :: [float]) =
 xb = block (3, x): ...
```

In this case, it also uses the fact that the descriptor at the input to `block` is $(\text{int}_3, [\text{float}])$, yielding an output type $[[\text{float}]_3]$. This is propagated through the outermost two maps, causing `xj` to have a length-enriched type of $[\text{float}]_3$. From here, the analysis is the same as that of the dense multiplication kernel.

Enhanced low-level datatype representation. The type abstraction layer is extended to handle fixed-length list types accordingly. We map LL list types of the form $[\tau]_n$, with any value of n, to a single parametric low-level type. For example, both $[\text{float}]_4$ and $[\text{float}]_3$ map to the same type `nlf_t`. Here, the `n` qualifier indicates that the following list takes a length parameter. The back-end is responsible for substituting constant values for length parameters during code generation, based on the specific constant lengths inferred for each program location.

A data structure that implements a nested list with a fixed inner list length does not contain a segment buffer. Instead, it contains a scalar offset value to the first element, stored in the `b` field. The length of each segment (a.k.a., stride) comes from the provided length parameter. For example, the nested type $[[\text{float}]_3]_4$ translates to the following type declarations, which are the fixed-length equivalents of the types shown in Figure 4.7(a):

```
/* Type: [[float]{n0}]{n1} */
typedef struct {
  nl_t c;
  int b;  /* Offset to beginning of first segment */
  nlf_t d;
} nlnlf_t;

/* Type: [float]{n} */
typedef struct {
  nl_t c;
  double *d;
} nlf_t;
```

Notice that the type `nlf_t` contains the same data as the plain `lf_t` in Figure 4.7(a).[12] As hinted above, the difference is in the associated length parameter, which will be attached to every use of this type in the generated code.

The data abstraction layer generates specialized methods for handling length-enriched lists. For example, getting the `j`-th sublist of an object of type $[[\text{float}]_3]_4$ is done by the following macro. Here, the type parameters are marked `n0` (for the inner lists) and `n1` (for the outer list). Contrast this implementation with that of `get_llf` shown in Section 4.4.2.

```
#define get_nlnlf(t, s, j, n0, n1) \
  nlf_t t; \
  t.c.len = n0; \
  t.d = (s).d.d + (s).b + (n0) * (j);
```

[12]The bookkeeping field for fixed-length lists of type `nl_t` is identical in content to the plain `l_t` used with ordinary lists. We differentiate the type name to distinguish its use in different contexts.

It should be noted that the C compiler disseminates type parameters during macro expansion, then treats them as constant literals through subsequent optimization phases. For example, the multiplication expression `(n0) * (j)` in the above macro is replaced with an increment by a constant (in this case 3) via an ordinary strength reduction optimization.

Another enhancement is concerned with initialization macros, where dynamic buffer allocations are replaced with array declarations. Buffer pointers—such as `.d.d` in the above example—then point to the statically allocated buffers. Interestingly, a standard optimizing compiler (GCC) can optimize dereferences to such buffers: array cells are mapped to registers, and array dereferences with fixed offsets translate to operations on those registers, instead of memory.

Constant loop boundaries. A map that is applied to a fixed-length list entails a loop whose boundary conditions are constant expressions. An optimizing compiler can unroll these loops entirely, eliminating conditional branches and substituting constants for uses of loop indexes.

The overall result is code that largely resembles what an expert programmer achieves through laborious coding: manually unrolled loops, careful optimization of register use, and elimination of unnecessary segment buffers. Fixed-length inference is embedded in the LL compiler front-end, as part of the semantic analysis phase. Enhancements to datatype handling and code generation are implemented in the back-end. The extension to code generation is embedded in the existing syntax-directed translation and does not necessitate an additional AST traversal.

4.6.2 Reduction fusion

The goal of fusion is to eliminate writing of loop output to temporary memory buffers. We perform local AST rewriting and annotate map nodes with a new notion of *embedded reducers*, which results in a significantly faster code with less communication through memory.

Figure 4.12 shows a partial AST representation of the RBCSR SpMV kernel that includes the sequence of three innermost nested maps. The first map (node 2) iterates over dense blocks along a compressed row; the second map (node 5) iterates over rows of a dense block; the third map (node 8) computes products along a dense row. Here we assume that list types of dense blocks are annotated with fixed dimensions of 4×3 (see Section 4.6.1).

Embedded reducers. Our first observation is that the outputs of the maps at nodes 2 and 8 are used as direct inputs for summation operations (nodes 3 and 9, respectively), via composition operators (nodes 1 and 7). These are highlighted by red frames in Figure 4.12. The translation rules in Figure 4.9 dictate that a memory buffer is allocated for storing the intermediate result (a list). The use of an intermediate buffer induces significant performance penalty: every iteration of the preceding loop writes to this buffer, and every iteration of

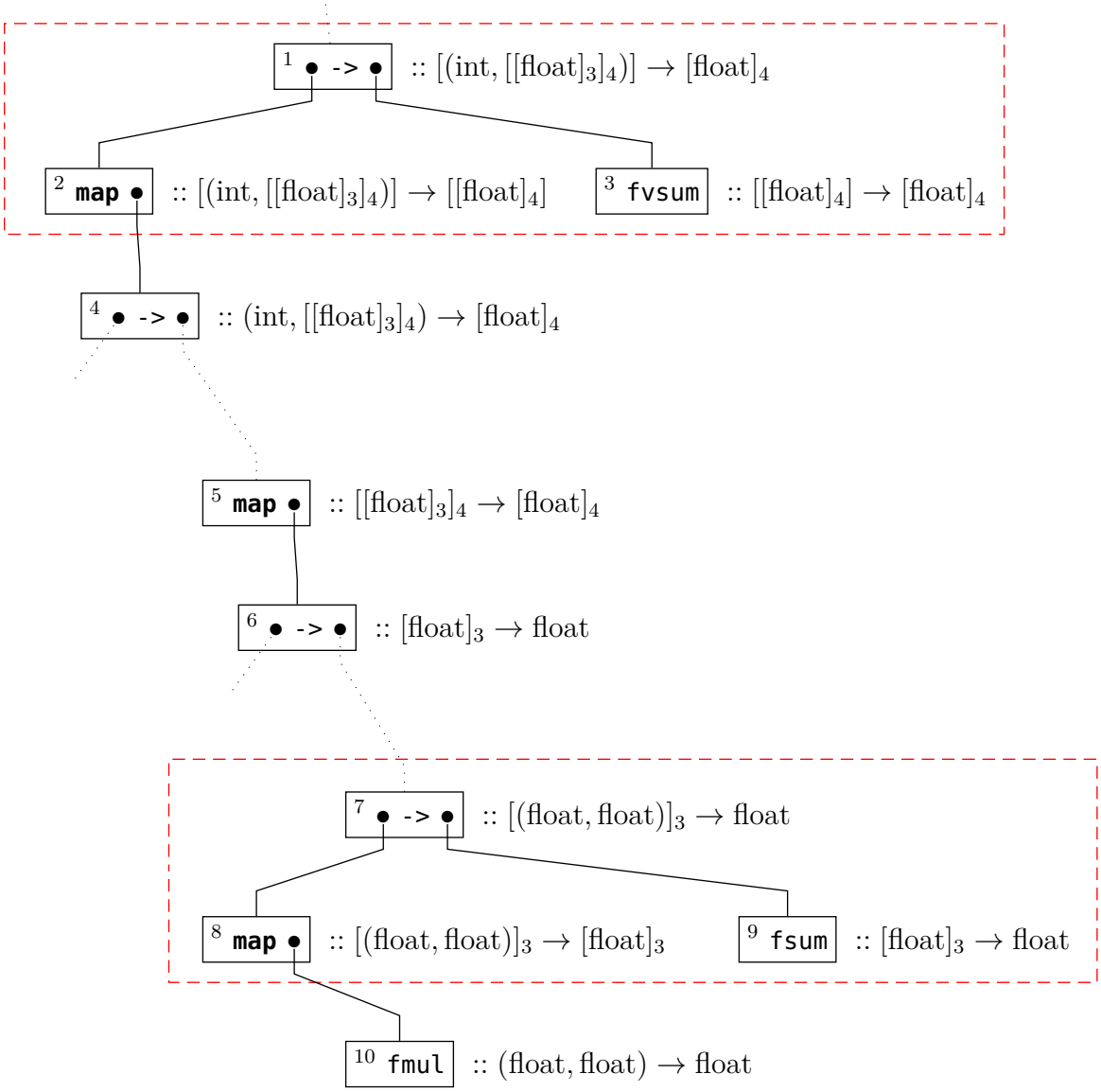


Figure 4.12: Typed AST representation of RBCSR SpMV inner maps

the succeeding reduction (summation in this case) reads from it. Things get much worse in the presence of cache misses (e.g., when the buffer gets big enough). Finally, the allocation overhead is also considered, although it can be amortized by moving it outside the loop nest.

We eliminate the communication penalty by embedding summation operations into the maps that precede them. Essentially, we rewrite a function of the form **map** ($f_{\texttt{map}}$) -> $f_{\texttt{reduce}}$ into **map**/$f_{\texttt{reduce}}$ ($f_{\texttt{map}}$). This technique is an instance of *loop fusion*, a well-known optimization for improving performance of adjacent loops. In LL, where adjacent functional blocks communicate through intermediate variables, fusion is particularly beneficial. We focus on fusing maps that are followed by reductions, as seen in the RBCSR example. We call the specialization of a map to incorporate a reduction functionality an *embedded reducer*.

Figure 4.13 shows the effect of incorporating embedded reducers into the highlighted maps in Figure 4.12. The reducer-equipped maps (nodes 1 and 7) are annotated appropriately, instructing the code generator to emit summation code directly inside the loops. In this case, the former map (node 1) accumulates outputs of type $[\text{float}]_4$; the latter map (node 7) accumulates outputs of type float.

Reducer derivation. We now observe that the second map (node 5)—highlighted by a red frame in Figure 4.13—is the last function to be executed in the body of the first map (node 1). Its output is a list of floats, which is then summed by the embedded reducer at node 1. This computation is redundant, because the value computed on the i-th iteration of node 6 (a single float) can be directly added to the i-th position of the reducer at node 1 (a list of floats). Hence, the second map (node 5) can avoid writing to a temporary output buffer in this case.

We remedy this inefficiency by propagating a *derivative* of an embedded reducer of a map (in this case, node 1) as we process its body. When the last operation in the body is itself a map (in this case, node 5), it "consumes" the derived reducer: the inner map can then use its own iteration index i to accumulate its i-th output directly into the i-th position of the outer map's reducer.

Figure 4.14 shows the effect of deriving a reducer for the second map (node 5). A derived reducer is annotated with the number of the outer node, where the original reducer was defined (node 1). The reducer at the outer node is marked "consumed" (denoted by parentheses), indicating that the summation operation takes place at an inner node.

Reducer shadowing. We now observe that the embedded reducer at the first map (node 1)—highlighted by a green frame in Figure 4.13—may not be as efficient as it should. In particular, updating each element in this object of type $[\text{float}]_4$ requires that we read its current value (i.e., compute the correct offset and fetch the value from memory), perform the addition, and store the new value. It may be necessary to use a memory allocated buffer even when the output has a known fixed length, as in this case. The reason is that it may be nested in yet another map and be part of a larger buffer of unknown size. This is actually

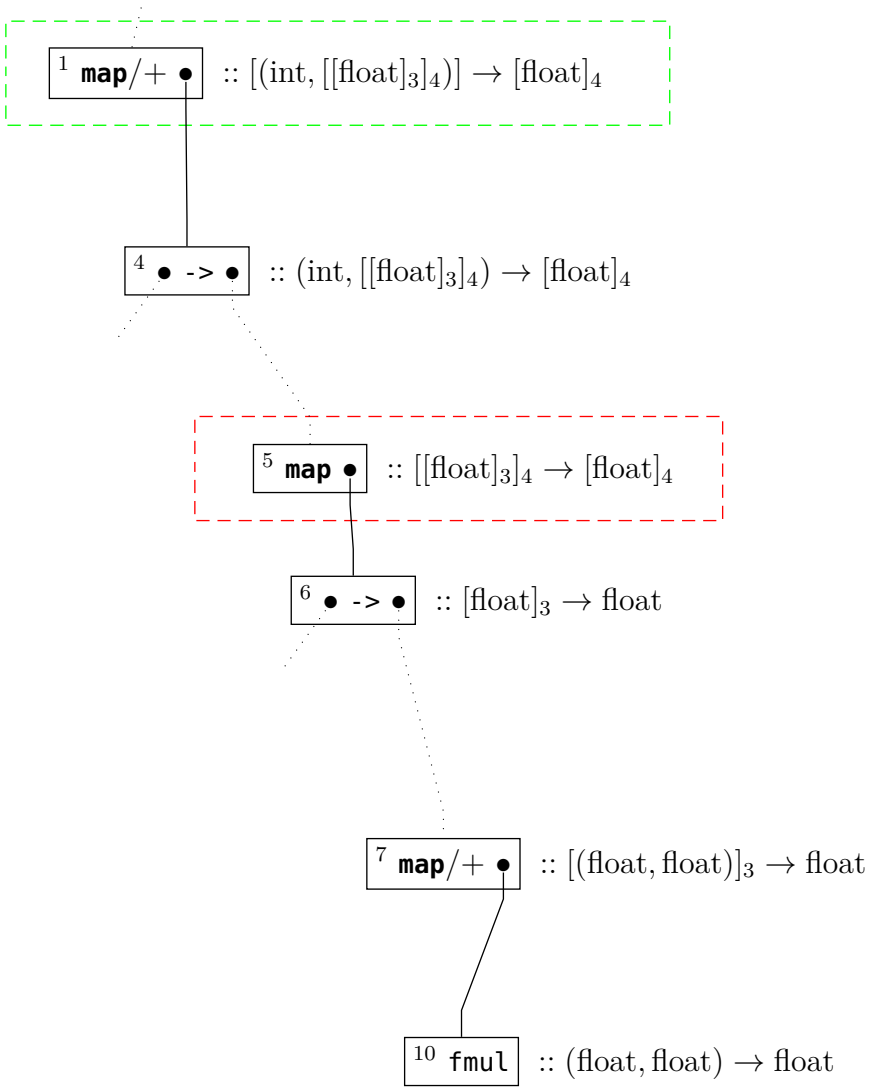


Figure 4.13: Embedded reducers in RBCSR SpMV

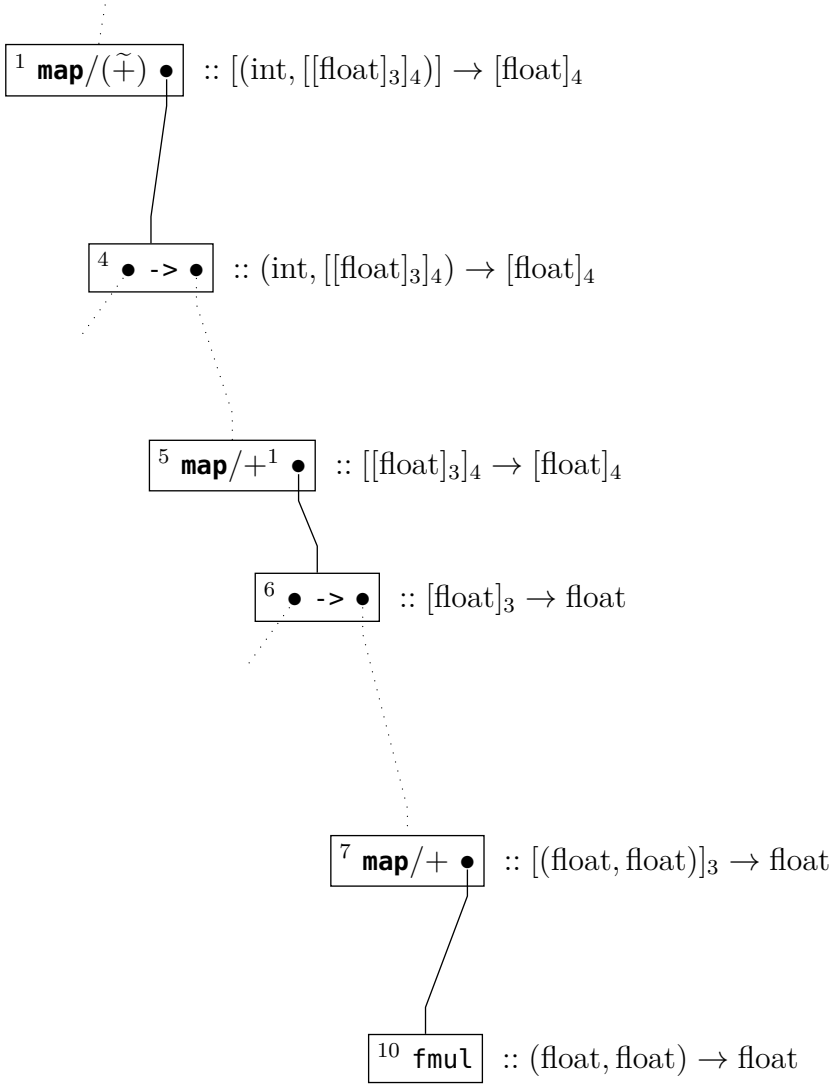


Figure 4.14: Complete reducer optimization in RBCSR SpMV

the case with the RBCSR example, where the vectors resulting from the map at node 1 are concatenated to form the final result.

We mitigate this problem by *shadowing* embedded reducers whose type is a list of statically known size. The idea is to allocate a temporary "shadow" buffer on the stack, which has the same length as the reducer, and accumulate values to this buffer. Once the accumulation is completed, the shadow reducer is copied ("deshadowed") once onto the actual output buffer. Shadowing can boost performance of nested maps significantly. This is especially true when inner loops are unrolled, and a shadow reducer can be mapped to registers.

Figure 4.14 shows the final AST after shadowing the reducer at the first map (node 1). A shadowed reducer is denoted by a tilde.

This completes the reducer optimization for the RBCSR kernel, and results in the following dataflow:

- The third map (node 7) accumulates the results of multiplication into a scalar reducer variable directly. This variable can be optimized into a register by the C compiler.
- The second map (node 5) accumulates the values in the latter variable into the i-th position of the shadow reducer of the outer map (node 1). Thanks to the optimizations described in Section 4.6.1, accumulation to this shadow reducer amounts to operations on registers, rather than memory.
- At the end of each iteration of the outer map (node 1) the content of the shadow reducer is copied to the permanent output buffer.

The overall result is code whose dataflow largely resembles a carefully handcrafted low-level implementation, with minimal communication through memory. The embedded reducer logic does not require a separate transformation phase and is implemented as an extension to the syntax-directed translation: (i) embedding is triggered by matching patterns in the AST and performing local tree rewrites; (ii) reducer derivation is done by propagation through the recursive translation process; and (iii) reducer shadowing is a node-local optimization.

4.6.3 Nested index flattening

As shown in Section 4.5, a map induces successive reads of values from an input list, using the list's `get` method. Nested maps over deeply nested data structures require, at each level, that offsets of sublists be computed and pointers initialized accordingly. Often times this involves reading from segment buffers. When done repeatedly throughout the loop nest, these reads amount to a substantial portion of the execution time.

Nonetheless, we observe that nested maps often encode a linear sweep over one or more buffers containing primitive values. This is due to the data structures that are generated by datatype flattening (see Section 4.4.1). For example, consider the complete map nest in

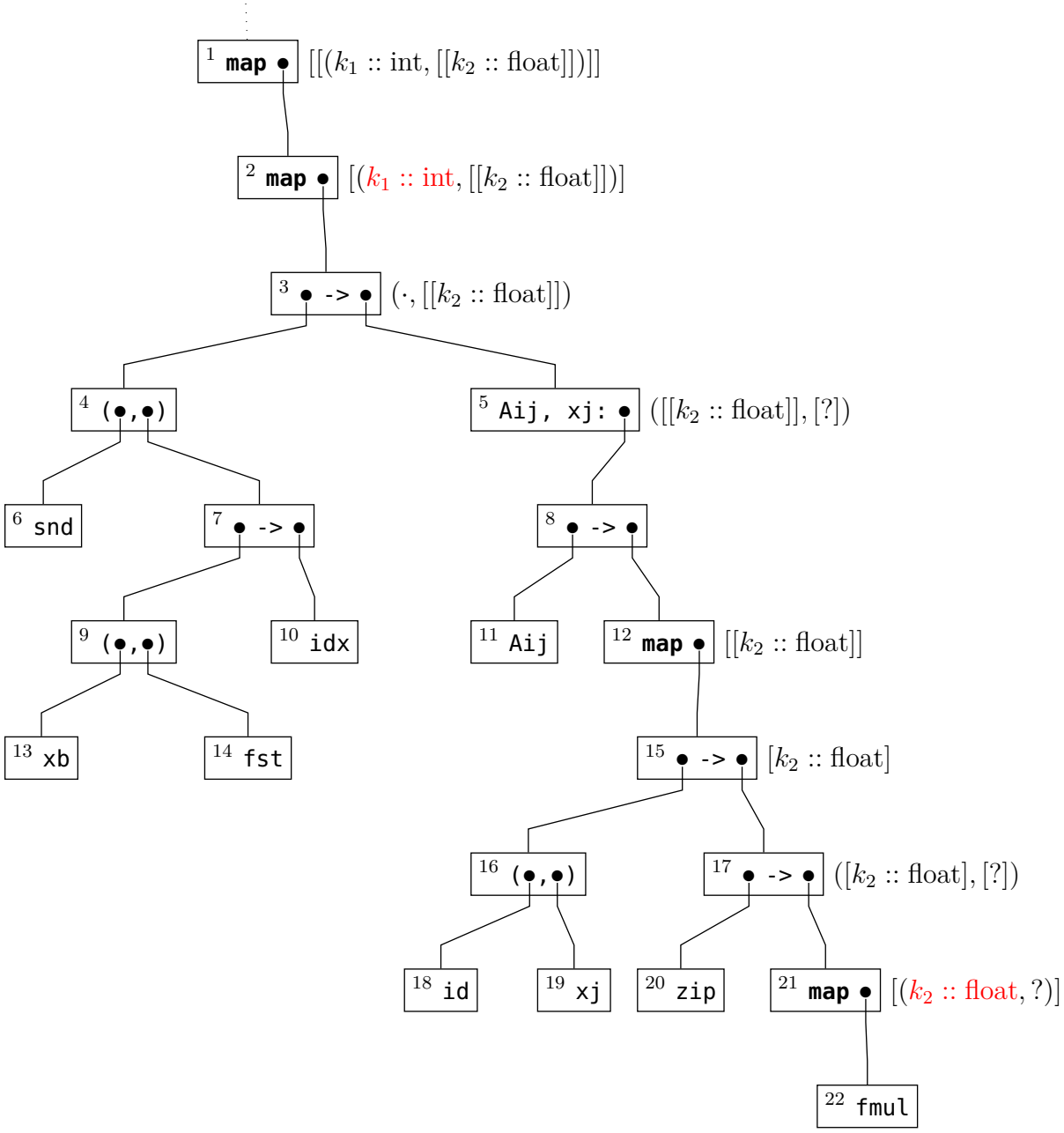


Figure 4.15: Flat index propagation in RBCSR SpMV

RBCSR multiplication, shown in Figure 4.15. This figure omits type annotations; for now we ignore the annotations to the right of some of the nodes. This computation induces two separate linear traversals: (i) the block index integer values are read through subsequent iterations of the second map (node 2); (ii) the floating point values inside blocks are read through subsequent iterations of the fourth map (node 21).

Our goal is to identify linear traversal patterns of buffers containing primitive values, and to replace naïve sublist extraction with a more economical, linearly advancing pointer.

Identifying linear traversal. Candidates for linear traversal are maps whose input type is a list that contains a primitive value type that is not nested within another list. In this example, the input type of node 2 is $[(\text{int}, [[\text{float}]])]$, suggesting that the integer component may be linearly read, which is indeed the case.

However, it is also necessary to ensure that the order of reads from each candidate buffer is not interrupted by any of the nodes between the function entry point and the node at which they occur. For example, the input for node 21 is of type $[(\text{float}, \text{float})]$, suggesting that both float components may be linearly read. In this case, however, only the first component—containing values in dense blocks—induces linear reads from a buffer. The second float component represents values in the right-hand side vector, whose access pattern is non-linear: it is governed by block index values and is therefore random.

Furthermore, it is also essential to track how a pointer that is initialized to some buffer outside the loop nest is to be used by the loop in which the same buffer is read. For example, the buffer containing dense block floating point values, and that is nested in four levels of lists (i.e., $[[(\text{int}, [[\text{float}]])]]$) at the entry to the outermost loop, is actually accessed as the left-hand side components of the list of pairs (i.e., $[(\text{float}, \text{float})]$) that is processed by the innermost map (node 21).

We implement a dataflow analysis that infers, for each functional node in the program, the set of buffers whose values are read in a linear fashion. It also associates with each such buffer a symbol representing the pointer that is set to point to this buffer at the entry to the function. Here, too, we use an abstraction of buffer pointers—called an *index tree*—that mirrors the structure of LL types: a primitive value that is nested in one or more loops can be assigned a pointer symbol, k. For presentation purposes, we also annotate pointers with their value type. In our example, the abstract descriptor at the entry (node 1) is $[[(k_1 :: \text{int}, [[k_2 :: \text{float}]])]]$, indicating that a two-level nested buffer of integer values is pointed to by k_1, and a four-level nested buffer of floats is pointed to by k_2.

Similarly to size inference (Section 4.3.2) and fixed list length inference (Section 4.6.1), the inference of "flat indexes" in LL functions is performed by propagating abstract descriptors using node-specific semantic rules. In the following, we trace the analysis of the RBCSR SpMV function shown in Figure 4.15. The descriptors to the right of some of the nodes show the inferred index trees at their input.

- The descriptor at the input to the outermost map (node 1) indicates two- and four-level nested pointers to an integer and float buffer, respectively. The map rule peels off the outermost list layer and propagates the inner descriptor to the body function.

- At the second map (node 2) we find that the left-hand side integer input component can be read using a linear pointer, k_1. This fact is used by the code generator when emitting `get` method calls at the beginning of the loop body that implements the map (see below).

  The analysis then peels off another list layer. The descriptor obtained is a pair whose left component is a primitive (denoted by '$\cdot$') and whose right component contains a pointer to a float buffer (k_2) inside a two-level nested list.

- The analysis of a pair constructor (nodes 4 and 9) merely aggregates the output descriptors of its two descendants. The `snd` and `fst` functions (nodes 6 and 14) return the respective components of their input, $[[k_2 :: \text{float}]]$ and '$\cdot$'.

  The inference of flat indexes through `idx` (node 10) is interesting to note. The descriptor at the input to this function is a pair whose left component is a doubly-nested list of floats. However, even if the input descriptor for this input contains a definite index symbol—e.g., $[[k' :: \text{float}]]$—the output cannot contain such a pointer: the `idx` function uses its right input component to select an arbitrary element from its left input component. Therefore, the output descriptor in this case must be [?].

  Consequently, the descriptor at the input to node 5 is $([[k_2 :: \text{float}]], [?])$.

- The analysis of a name binding is more tricky. In the general case, binding may introduce implicit dataflow edges into the program. Therefore, it is not safe to assume that an input descriptor to a binding construct that contains a valid linear pointer (e.g., $[[k_2 :: \text{float}]]$ which is bound to `Aij` at node 5) can be forwarded to the output of a corresponding named function (e.g., `Aij` at node 11). Extra care must be given to multiple uses of the named function, which necessitate replication of the original input pointers. In this case, however, it is safe to propagate the descriptor as is at node 11.

- The analysis of the third map (node 12) is analogous to that of the previous maps. The pair constructor (node 16) and its descendants `id` and `xj` (nodes 18 and 19) result in $([k_2 :: \text{float}], [?])$ at the input to node 17.

- Interestingly, the `zip` function preserves pointers in its argument, although it rearranges their structure into a list of pairs. In this case, it results in $[(k_2 :: \text{float}, ?)]$ at the input to node 21.

- Arriving at the fourth map (node 21) we find that the left-hand side float input component can be read using a linear pointer, k_2. The analysis then removes the list layer, propagating $(\cdot, \cdot)$ to `fmul` (node 22).

Notice that the analysis identified the two linear buffer reads that are present in the RBCSR SpMV kernel, highlighted in red in Figure 4.15.

Flat indexed datatype methods. Having identified the linear pointers available at each map, the LL compiler can specialize code generation to take advantage of this knowledge.

The first part of this specialization is the initialization of pointers to suitable buffers. The data abstraction layer extends the datatype API with `index` methods for each type that is used in the program. These methods initialize a set of pointers to point to the beginning of particular buffers in the object they are given as argument. For example, `index_llpxillxf` initializes two pointers to an object of type [[(int, [[float]])]], as marked by the `x` qualifier: one pointer points to the integer buffer containing block indexes (`xi`); the second pointer points to the floating point buffer containing dense block content (`xf`). These pointers model the two linear traversals that were discovered by our analysis. The `index` methods are hierarchical and modular, similar to other methods generated by the data abstraction layer. They take a sequence of indexes to be initialized, a source object `s`, and an offset `off`. For the RBCSR example, they look as follows:

```
#define index_llpxillxf(k1, k2, s, off) \
  index_lpxillxf (k1, k2, (s).d, (s).r[off]);

#define index_lpxillxf(k1, k2, s, off) \
  int *k1 = (s).d0 + (off); \
  index_llxf (k2, (s).d1, (off));

#define index_llxf(k1, s, off) \
  index_lxf (k1, (s).d, (s).r[off]);

#define index_lxf(k1, s, off) \
  double *k1 = (s).d + (off);
```

Note that the use of an `index` method is analogous to invoking a sequence of `get` operations to extract a particular primitive value out of a nested data object. The differences are that (i) no intermediate values such as sublists are being constructed; and (ii) the value that is being extracted is a reference to a location in a buffer, rather than the content stored at that location.

In order to take advantage of indexes in loops, the data abstraction layer also generates enhanced `get` methods that make use of indexed buffers. For example, `get_lpxillxf` implements an efficient extraction of the integer component. This method will be used by the code generated for the second map in Figure 4.15 (node 2), instead of the ordinary `get_lpillf`. Here, the value assigned to `t.d0` is read using a direct dereference of pointer `k1`:

```
#define get_lpxillxf(t, s, j, k1, k2) \
  pillf_t t; \
  t.d0 = k1[j]; \
  t.d1.c.len = (s).r1[j + 1] - (s).r1[j]; \
  t.d1.r = (s).d1.r + (s).r1[j]; \
  t.d1.d = (s).d1.d;
```

The overall effect of index flattening is crucial for deeply nested loops, such as those exhibited in RBCSR SpMV. Both inference and emission of index-aware API calls are implemented as part of the syntax-directed translation process done by the compiler back-end. The data abstraction layer is extended accordingly.

4.7 Implicit parallelism

The LL compiler deploys a simple, syntax-based parallelization scheme, where maps are converted into data-parallel loops and pair constructors into parallel tasks. A simple heuristic guides the selection of constructs to parallelize, which prohibits nested parallelism and favors data parallelism over task parallelism. Partitioning of data accounts for proper load balancing. This results in coarse-grained parallelism, exhibiting long synchronization-free threads, as shown in Section 4.2.2. It goes hand-in-hand with the sequential optimizations described in Section 4.6 and achieves good utilization of resources on multicore machines with real-world datasets.

4.7.1 Parallelization transformation

We identify two forms of parallelism, corresponding to syntactic constructs in LL programs: (i) maps represent an implicit data-parallel computation on a list; (ii) a pair constructor represents an implicit task-parallel computation on any object. In both cases, the parallel threads are free from data dependencies, thanks to LL's dataflow model and its side-effect-free semantics.

The above constructs also entail a basic distribution/synchronization scheme that corresponds to the program's syntactic structure: (i) parallel loops entail data distribution prior to entering the loop and a synchronization right past the loop's scope; (ii) pair constructors entail a synchronization point right past the constructor's scope. While this may result in an unnecessarily fined-grained synchronization, it proved sufficient in our benchmarks (see Section 4.8). The scheme can be coarsened via further analysis (see discussion in Section 4.9).

Parallelization of LL functions is implemented as an AST transformation and performed prior to translation. We follow a simple heuristic in deciding which constructs should be parallelized:

- A map is parallelized only if it is not nested in a parallel map and if the output type of its inner function has a fixed size. The latter requirement means that a partitioning of the input already gives us a definite partition of the output buffers. This allows the loop body to write directly to the memory section that corresponds to its output chunk within a single, flattened data object (see Section 4.4.1).

  The parallelization of maps replaces `[`f`]` with `[|`f`|]`, which is an explicitly parallel map whose semantics is otherwise equivalent to that of the sequential map.

- A pair is parallelized only if it is not nested in a parallel map and does not contain a parallel map.

  The parallelization of pairs replaces `(`f`, `g`)` with `(|`f`, `g`|)`, an explicitly parallel pair constructor.

The resulting scheme does not contain nested data parallelism, and strictly favors data parallelism over task parallelism. While this conceptually limits the amount of parallelism that can be extracted from a given program, we found it to be sufficient for the typical uses of LL programs, namely applying a uniform transformation to a large (possibly nested) list object. Similar observations were made by Catanzaro et al. in the context of generating parallel GPU code [15]. In the rest of this chapter we focus solely on data parallel loops, which is the only form of parallelism present in our benchmarks.

The result of this transformation is an explicitly parallel LL function. It is then handed to the code generator for emitting parallel low-level code. The parallelized versions of the example kernels used earlier is similar, both having their outermost map turned into a parallel map: for the CSR SpMV in Figure 4.10 it is node 5; for the (partial) RBCSR SpMV shown in Figure 4.15 it is node 1.

4.7.2 Generating parallel code with OpenMP

The LL back-end generates parallel C code by emitting OpenMP directives. Figure 4.16 shows the skeleton of a parallel CSR SpMV kernel generated by our compiler. Contrast this code with the sequential version shown in Figure 4.11. We highlight the differences between the parallel map (lines 9–37) and an ordinary sequential map:

- Declaring a special partition variable (`p1` on line 8) of type [int]. The values in this list will mark the start and ending indexes of the input list that will be processed by each of the parallel threads. Note that the computation of the actual partition happens inside the parallel block, as it depends on the number of threads spawned at runtime.

- The whole loop code block is marked with `#pragma omp parallel` (line 9). This directive instructs OpenMP to spawn an execution of the same code block on multiple threads.

```
1  void def_csrmv_par (lf_t *out, pllpiflf_t *in)
2  {
3    llpif_t A;
4    ll_fst (A, *in);
5    lf_t x;
6    ll_snd (x, *in);
7
8    li_t p1;
9  #pragma omp parallel default(none) shared(p1)
10   {
11 #pragma omp single
12     {
13       int p2;      /* Partition workload. */
14       p2 = omp_get_num_threads ();
15       init_li (p1, p2);
16       aux_partition (p1, A, p2);
17     }
18
19     int p3;    /* Get current partition. */
20     int v2;
21     int v3;
22     p3 = omp_get_thread_num ();
23     aux_getpart (v2, v3, p1, p3);
24
25     int j1;
26     index (k1, k2, A, v2);
27     for (j1 = v2; j1 < v3; j1++) {
28       get (v4, A, j1, k1, k2);
29       double v5;
30
31       /* Loop body... */
32
33       set_lf (*out, v5, j1);
34     }
35   }
36   (*out).c.len = (A).c.len;
37 }
```

Figure 4.16: Skeleton of generated code for parallel CSR SpMV

- Inside the parallel block, we use a `#pragma omp single` (line 11) to specify a code block to be executed only by the leader thread, with a barrier right after it. In this block we (i) obtain the dynamic number of threads in the current group using `omp_get_num_threads` (line 14); and (ii) invoke a library function `aux_partition` (line 16) to compute a partition of the input data, to be stored in `p1`.

- Next, each thread (i) obtains its number using `omp_get_thread_num` (line 22); then (ii) uses this number to obtain the start and ending indexes marking its share of work by calling the `aux_getpart` library function (line 23).

- The loop itself is slightly different from an ordinary sequential loop, as its boundaries are determined by the partition (`v2` and `v3`) and are not necessarily zero-based. Consequently, it must use `get` and `set` methods with an explicit index (`j1`) upon reading the input value and writing the output, respectively. Note that `set` is well defined because the output type has a fixed size that is statically known.

- Past the loop body, we set the length of the output object to its overall aggregate length (line 36).

The advantages of using OpenMP are evident from this example. It gives us the right amount of control over parallel invocation, while abstracting away low-level thread management and synchronization. Our benchmarks indicate that the overhead due to OpenMP directives is negligible. OpenMP is a natural choice for implementing data parallel computations on a shared memory, multicore platform. However, it is likely not a good fit for hybrid architectures such as GPUs, which require more elaborate abstractions such as CUDA [48, 32].

Our partitioning function, `aux_partition`, is designed to achieve two goals:

1. Load balancing with respect to nested list lengths. In the case of a list of lists—e.g., $[[(\text{int}, \text{float})]]$ in CSR SpMV—it will consider the *total length* of nested sublists in each partition chunk as a proxy for the actual workload, rather than the *number* of sublists in each chunk. This is important because the lengths of nested lists may be not uniformly distributed.

2. Prefer utilization of all computational cores over assigning an average chunk to each participating core. In effect, we may end up with several underloaded partitions, but we are less likely to have idling ones.

Partitioning implements the algorithm shown in Figure 4.17. It takes as input a segment buffer S describing n segments and a number of partition chunks k. The start/end positions of segment $1 \leq j \leq n$ are represented by S_{i-1} and S_i, respectively.[13] Note that all the arithmetic operations in this algorithm—including division—have integer semantics. It returns a buffer P with the following guarantees:

[13]This is the standard representation of a segment buffer shown in Section 4.4.1.

```
function PARTITION(S,n,k)
    P_0 ← 0
    for i ← 1...k do
        t ← ((S_n − S_0)·i)/k + S_0
        j ← (n − P_{i−1})/(k − i + 1) + P_{i−1}
        while S_j < t do j ← j + 1
        while j > P_{i−1} ∧ S_{j−1} > t do j ← j − 1
        P_i ← j
    end
    return P
end
```

Figure 4.17: Nested list partitioning algorithm

- P is a valid segmentation of S, namely $P_0 = 0$, $P_k = n$, and $\forall 1 \leq i \leq k \;.\; P_{i-1} \leq P_i$.

  This stems from the fact the that partition chunks are formed by a non-decreasing sequence of $k + 1$ indexes, the first of which is zero. The computation of the target coverage t guarantees that $P_k = S_n$.

- For each $1 \leq i \leq k$, the end index of the i-th partition chunk is the smallest that covers $\frac{i}{k}$ of the total lengths of all segments. Formally, $S_{P_i} - S_0 \geq \frac{(S_n - S_0) \cdot i}{k}$ and $S_{\max\{P_i - 1, P_{i-1}\}} - S_0 \leq \frac{(S_n - S_0) \cdot i}{k}$.

  This is due to the two-step process by which the end index of the next partition is determined: (i) guessing a good candidate, assuming a uniform length distribution of the remaining segments; (ii) adjusting the initial guess until it satisfies the minimal coverage requirement.

This algorithm converges quickly in practice, especially when the lengths of the input list segments are more-or-less even. In this case, it will be linear by k (the number of threads). This is much preferred over a naïve implementation that is linear by n (the number of segments).

4.8 Evaluation

We experimented with both sequential performance and parallel scaling of compiled LL code. Our benchmarks include two kernels—SpMV of CSR and RBCSR formats—run on a variety of real-world matrix inputs. Our results indicate that optimized sequential LL code can perform as fast as handcrafted C code, and that LL's parallelization scheme achieves

Name	Rows × Columns	Nonzeros	Nonozeros/row	Total density
bcsstk35	30237 × 30237	1450163	47.96	0.00159
crystk03	24696 × 24696	1751178	70.91	0.00287
dense2	2000 × 2000	4000000	2000.00	1.00000
mac_econ_fwd500	206500 × 206500	1273389	6.17	0.00003
nasasrb	54870 × 54870	2677324	48.79	0.00089
olafu	16146 × 16146	1015156	62.87	0.00389
qcd5_4	49152 × 49152	1916928	39.00	0.00079
raefsky3	21200 × 21200	1488768	70.22	0.00331
rail4284	4284 × 1092610	11279748	2632.99	0.00241
rma10	46835 × 46835	2374001	50.69	0.00108
scircuit	170998 × 170998	958936	5.61	0.00003
venkat01	62424 × 62424	1717792	27.52	0.00044

Table 4.2: Benchmark matrix suite

substantial speedups on shared memory architectures.[14] They also support the viability of our approach of optimizing LL programs only to the extent necessary for a good underlying C compiler to perform low-level optimization.

4.8.1 Preliminary

In our experiments we have used a set of 12 matrices whose characteristics are shown in Table 4.2. Five of them—`mac_econ_fwd500`, `qcd5_4`, `rail4284`, `rma10` and `scircuit`—are unstructured, unsymmetric matrices that were studied in the context of optimization for GPU by Williams et al. [64]. Of the remaining matrices, four are symmetric (`bcsstk35`, `crystk03`, `nasasrb`, `olafu`) and two are unsymmetric (`raefsky3` and `venkat01`). The `dense2` matrix is a dense matrix in sparse format. All of the matrices, with the exception of `dense2` and `qcd5_4`, can be found on the UFL Sparse Matrix Collection [24].

We ran our benchmarks on an Intel Core i7-2600 3.4 GHz machine with 4 GB of DDR3 memory. This is a single-socket quad-core processor capable of running up to 8 virtual threads via Intel's Hyper-Threading Technology (HTT).[15] The CPU frequency can be scaled down to 1.6 GHz, which we use in evaluating scalability of parallel execution (see Section 4.8.3). We used GCC version 4.4.3. In evaluating GCC's ability to optimize C code that is generated from LL programs, we used the `-O3`, `-m64` and `-funroll-all-loops` flags. These flags trigger the highest degree of performance optimizations, instruct the compiler to generate 64-bit instructions and to unroll all loops to the fullest extent possible, including

[14] We did not directly compare against a handwritten parallel SpMV. Intel's Math Kernel Library (MKL), the state-of-the-art in parallel sparse linear algebra, is closed source.

[15] In our benchmarks we only use up to four threads utilizing the four physical cores.

```
int i, next_i, k = R[0];
for (i = 0, next_i = 1; i < m; i = next_i, next_i++) {
  double y = 0.0;          /* Accumulate row output. */
  int next_R = R[next_i];  /* Sparse row boundary. */
  while (k < next_R) {
    int j = J[k];
    double v = V[k];
    y += v * X[j];         /* Compute inner product. */
    k++;
  }
  Y[i] = y;                /* Store row output. */
}
```

Figure 4.18: Reference C implementation of CSR SpMV

those whose termination condition is not known at compile-time; we found that this entails a slight performance improvement over the ordinary `-funroll-loops` flag, which only unrolls loops whose bounds are statically known.

In our runs we insisted on sampling at least five different runs of each configuration of matrix, kernel, optimizations, and block size. In reporting performance we use the median result for each such configuration and indicate the range of lowest and highest results with an error bar. The standard unit we use for performance is "million floating point operations per second" (or MFLOP/s for short). This value represents the overall throughput of a numerical algebra kernel with respect to the minimal number of operations necessary to compute the result. In the case of SpMV, it is twice the number of nonzero values in the matrix, as each nonzero induces one multiplication and one addition. Note that all floating point values and operations in our code are double precision (64-bit).

4.8.2 Sequential performance

We evaluate the sequential performance of LL generated code for the CSR and RBCSR SpMV kernels. Our hypotheses are: (i) for the non-blocked kernel, embedded reducers and flattened indexing lead to C code that is as fast as handwritten sequential code; (ii) register blocking requires fixed-length list inference as well in order to yield similarly competitive C code.

For comparison, we use a standard reference C implementation. CSR SpMV is implemented as a straightforward doubly-nested loop. It is shown in Figure 4.18. We considered three slightly different variants of the same kernel and chose the one for which GCC produced the best performing machine code. Our RBCSR implementation was derived from an optimized RBCSR kernel used in the OSKI sparse linear algebra library. The original OSKI

```
int k = *Rb++;
while (mb--) {
  double y[RB_ROWS] = { 0 };   /* Accumulate block row output. */
  int i, next_k = *Rb++;       /* Block sparse row boundary. */

  while (k++ < next_k) {
    int jb = *Jb++;            /* Get block index. */
    for (i = 0; i < RB_ROWS; i++) {
      double *xb = X + jb * RB_COLS;
      int blkncol = RB_COLS;
      while (blkncol--)
        y[i] += *V++ * *xb++;  /* Dense row inner product. */
    }
  }
  k--;

  for (i = 0; i < RB_ROWS; i++)
    *Y++ = y[i];               /* Store block row output. */
}
```

Figure 4.19: Reference C implementation of RBCSR SpMV

version reads particular values from the `x` vector into individual scalar variables (declared with the **`register`** qualifier). It also accumulates the multiplication results along a block row into individually declared variables. Accordingly, the two loops that compute the dense block multiplication are fully unrolled by hand. This ensures that the benefit of register blocking is being exploited, but results in a large number of kernel variants tailored for different block sizes. Unlike in OSKI, we refrain from unrolling the two innermost loops that perform the dense block multiplication. Consequently, we do not use scalar variables for reading values from the `x` vector nor to accumulate products along a row. We found that GCC is fully capable of doing this for us, resulting in code that has the same performance characteristics as the manually unrolled version, yet is parametrizable by block size. Our C implementation of RBCSR is shown Figure 4.19. Here, too, we considered two slight variants and chose the one with the better performance characteristics.

We measured the performance of the LL generated code with different sequential optimizations applied. For CSR, we used the following combinations:

- an unoptimized variant;
- a variant using index flattening (see Section 4.6.3);
- a variant using embedded reducers (see Section 4.6.2);
- and a variant using both.

For RBCSR, we used the following combinations:

- an unoptimized variant;
- a variant using fixed-length list inference (see Section 4.6.1);
- a variant adding index flattening to fixed-length lists;
- a variant adding embedded reducers to fixed-length lists;
- and a variant that combines them all.

We found it useless to try further combinations without fixed-length list inference; without it, the presumed advantage of RBCSR is nonexistent.

In measuring RBCSR performance, we also tested each kernel with a variety of block sizes. Due to the complexity of modern day architectures, and the uncertainty regarding intricate compiler optimizations and their effect on performance, memory optimizations such as register blocking are considered highly unpredictable; finding the best performing block size is often left for an automatic performance tuning framework (a.k.a., autotuner). Therefore, we report the best performing block sizes among 25 different ones ranging from 2×2 to 8×8 cells.

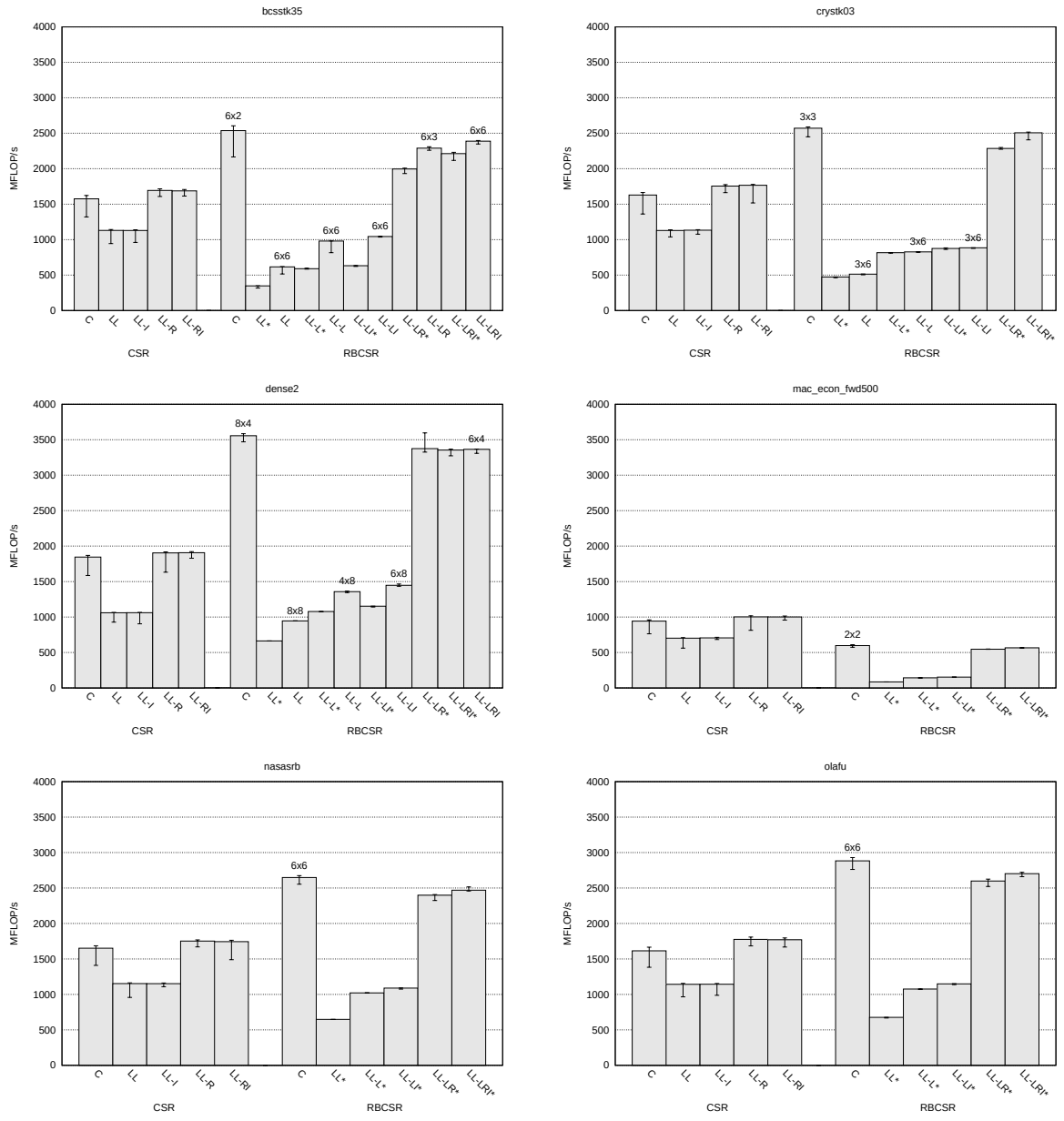


Figure 4.20: Performance of LL generated sequential SpMV (part 1)

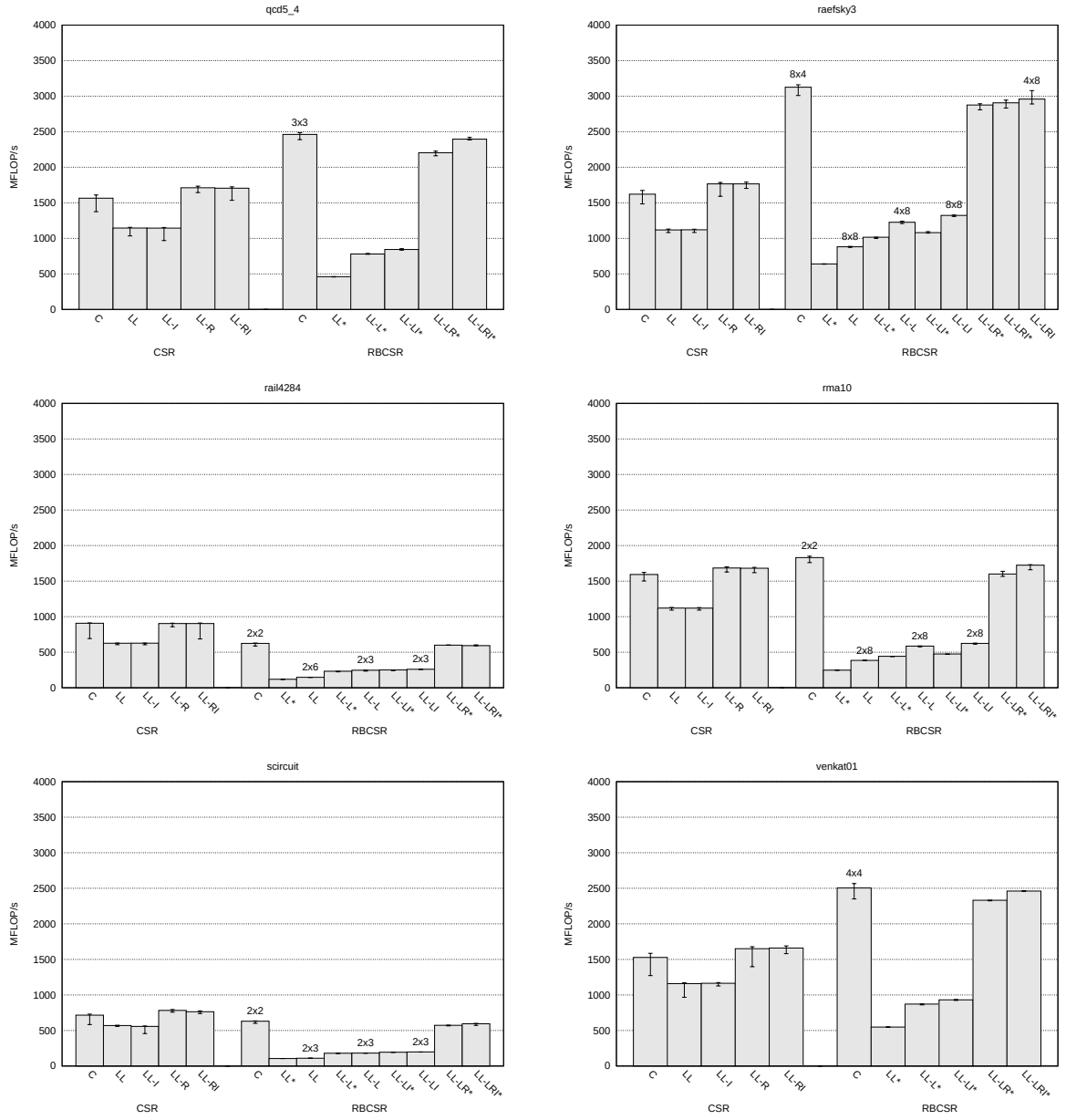


Figure 4.21: Performance of LL generated sequential SpMV (part 2)

The benchmark results for the sequential SpMV kernels are shown in Figure 4.20 and Figure 4.21. Each of the graphs shows the performance of C and LL implementations of two kernels for a given matrix out of the set shown in Table 4.2. Each column in a graph depicts the median MFLOP/s achieved by the corresponding implementation. The columns are split into two groups, the left-hand side showing the performance for CSR and the right-hand side showing RBCSR. In each of the groups, the 'C' column indicates the reference implementation (see Figure 4.18 and Figure 4.19). The 'LL' columns indicate code that was generated from an LL program: the leftmost column uses none of the sequential optimizations described in Section 4.6; further columns add optimizations, where 'L' indicates fixed-length lists, 'I' indicates index flattening and 'R' indicates embedded reducers. In the RBCSR group, the 'C' column shows the performance using an optimal block size, which is shown in a label above the column. The performance of each of the LL generated implementations is measured using the same block size, and these columns are marked by a star. However, in cases were better performance was exhibited with an LL implementation using a different block size, an additional column is present with the alternative block size showing above it.

CSR. Without the reducer optimization, the performance of LL code is within a factor of 0.57–0.80 of the reference C implementation. With the reducer optimization it is within 1–1.09, essentially surpassing the throughput of the reference implementation on most benchmarks. The index flattening optimization has little to no effect on performance. This validates our hypothesis that an LL non-blocked SpMV kernel is competitive with handwritten C code.

RBCSR. First, it is interesting to note that register blocking can improves performance compared to ordinary CSR by a factor of up to 1.93 on some matrices, but it can worsen it for others. Indeed, register blocking is known to only be beneficial with matrices whose nonzeros tend to appear in small clusters. It is also interesting to see how different variants of RBCSR SpMV entail different optimal block sizes and how significant the effect of block size can be. While some matrices entail a uniform optimal block size across all implementations—e.g., `nasasrb`, `olafu`, `qcd5_4` and `venkat01`—others perform best at varying block sizes—e.g., `bcsstk35`, `dense2` and `rma10`.

The unoptimized LL implementation of RBCSR performs within a factor of 0.14–0.28 of the reference implementation, depending on the choice of block size. Enabling fixed-length list inference, and consequently allowing the C compiler to unroll the innermost loops, speeds it up to 0.23–0.39. However, it is not until the reducer optimization is enabled that we see substantial improvement in performance, at a factor of 0.78–0.96 of the reference implementation, depending on block size. Here we can also see the benefit of the index flattening optimization: as the loop nest is deeper, avoiding repeated reading of segment buffers becomes more beneficial. With all the optimizations combined and with an optimal block size selection, the LL code performs within 0.94–0.98 of the reference implementation.

This validates our hypothesis regarding the competitiveness of LL code for the blocked SpMV kernel.

4.8.3 Parallel scaling

We evaluate the scaling of parallel LL generated code for the CSR and RBCSR SpMV kernels. Our hypotheses are: (i) our parallelization scheme is well-suited for a shared memory architecture such as multicore, increasing performance accordingly with the number of cores used; (ii) our parallelization scheme induces minimal overhead and a exhibits a good utilization of resources.

The comparison methodology for parallel kernel performance is different from the one used in the sequential case. Here we measure the performance of a single parallel LL implementation running on one, two and four cores. The implementation chosen in this case is one with all the sequential optimizations mentioned enabled; we also enable the parallelization transformation (see Section 4.7), designated by 'P'.

The benchmark results for the parallel SpMV kernels are shown in Figure 4.22 and Figure 4.23. Each of the graphs shows the performance of CSR and RBCSR kernels with a specific matrix out of the set shown in Table 4.2. Each column in a graph depicts the median MFLOP/s achieved by the corresponding kernel executed on a given number of cores. Error bars indicate the lowest and highest results. Here, too, the columns are split into two groups, the left-hand side showing the performance of CSR and the right-hand side showing RBCSR. In each group we present the performance of the corresponding kernel on one, two and four cores. As in the sequential case, for each parallel execution of RBCSR we use the same block size as the single-core optimal block size (marked by a star). However, in cases were a different block size yields better performance, a second column shows the optimal performance with the corresponding block size shown above it. Finally, Figure 4.24 and Figure 4.25 present the relative speedup of parallel runs compared to their respective single-core run.

CSR. Running CSR SpMV on two cores yields a speedup of 1.58–1.96 with a median speedup of 1.62; running it on four cores yields 1.71–3.17 with a median of 1.85. The two matrices achieving the most substantial speedups—`mac_econ_fwd500` and `scircuit`—both exhibit low single-core throughput at 949 and 682 MFLOP/s, respectively. Running on four cores increases their throughput to about 2200 MFLOP/s. This can be attributed to the matrices' special structure, which leads to a computational bottleneck in the sequential case: the former has a largely diagonal structure; in the latter case it is due to the proximity of column indexes of nonzeros across neighboring rows. In both cases, parallel execution on multiple cores leads to better utilization of the memory bandwidth and cache, and proves highly beneficial.

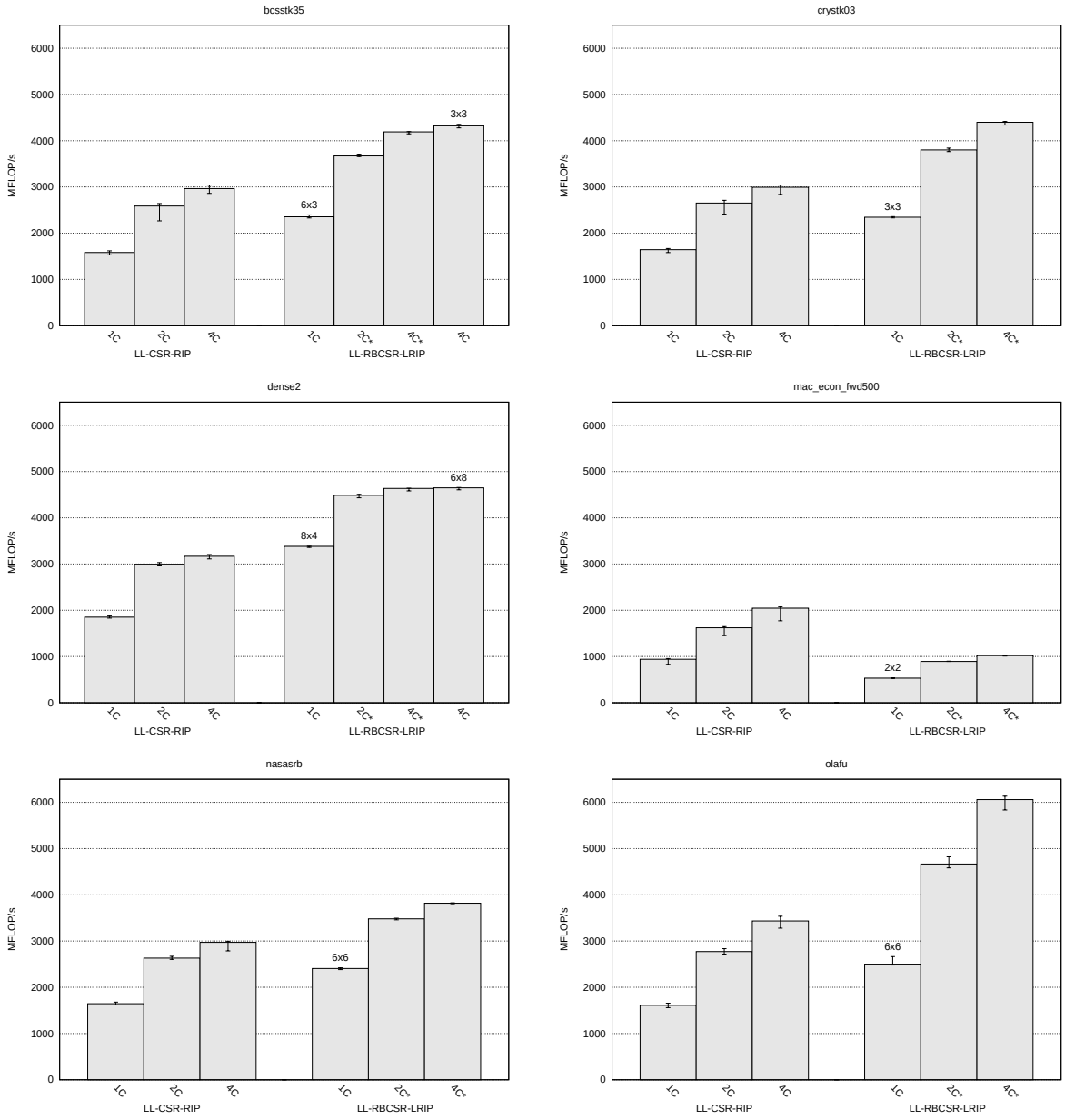


Figure 4.22: Performance of LL generated parallel SpMV (part 1)

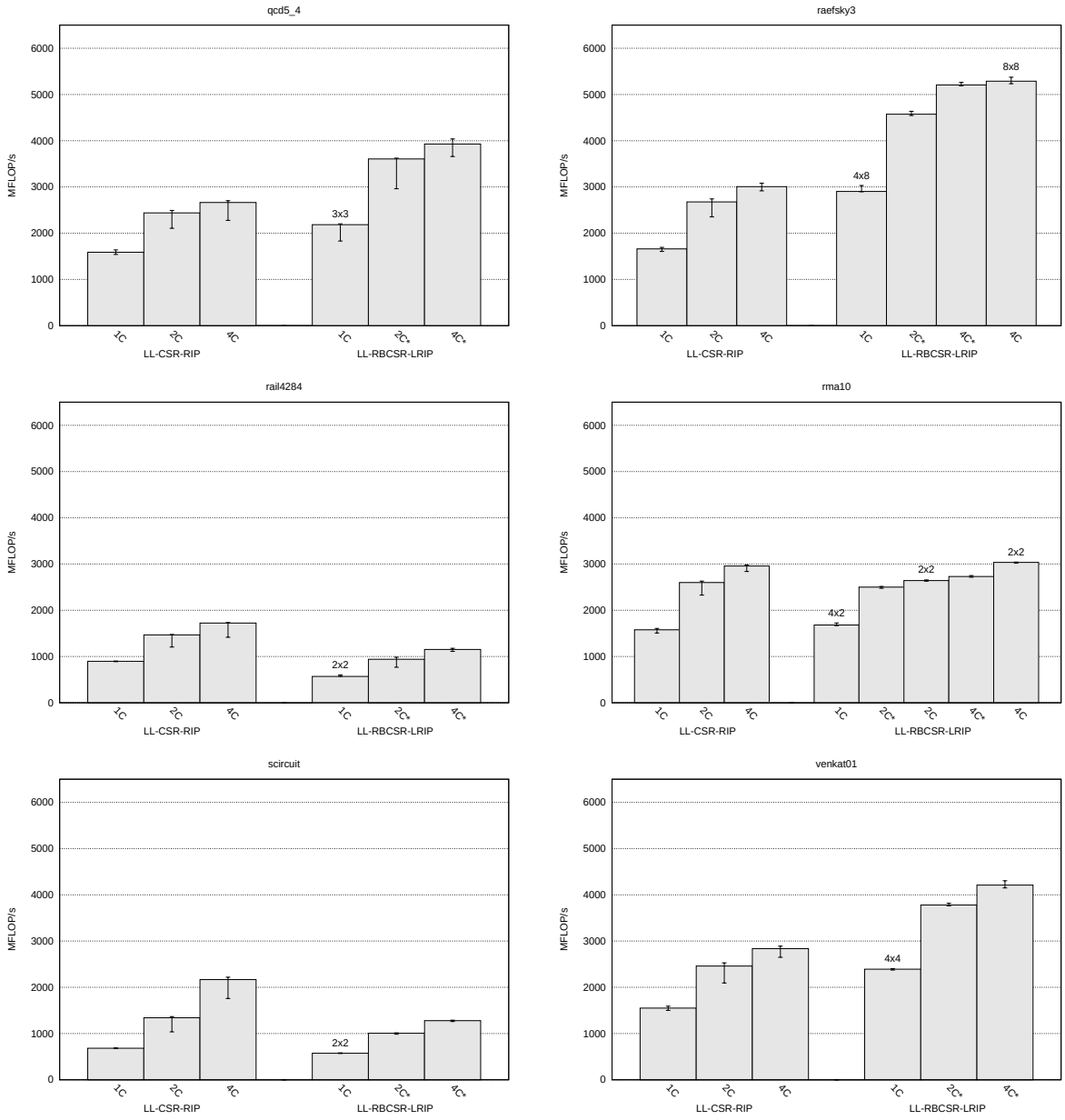


Figure 4.23: Performance of LL generated parallel SpMV (part 2)

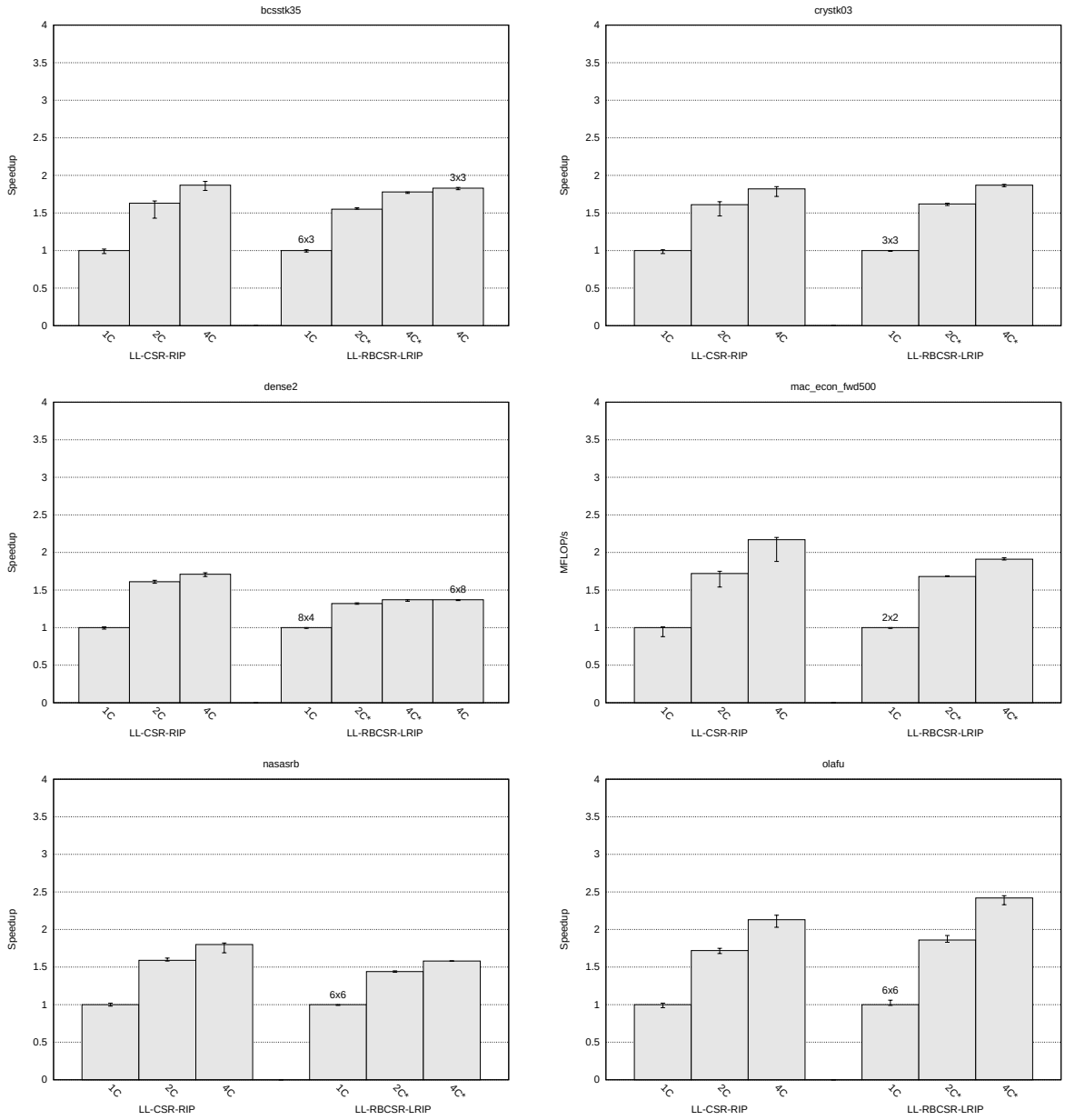


Figure 4.24: Relative speedup in LL generated parallel SpMV (part 1)

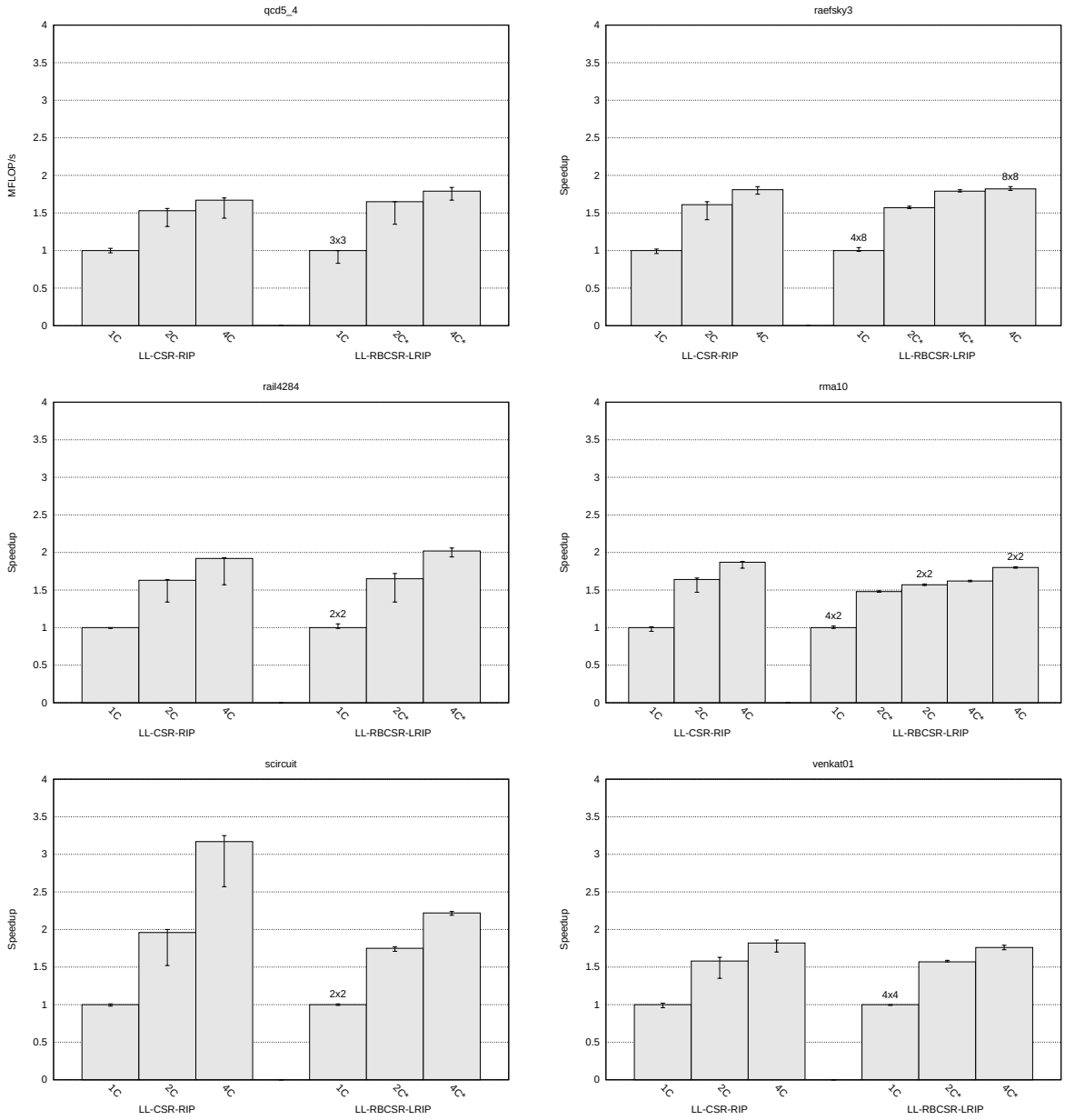


Figure 4.25: Relative speedup of LL generated parallel SpMV (part 2)

RBCSR. The speedup of RBCSR due to parallel execution on two cores lies with 1.32–1.86 of the single-core performance with a median speedup of 1.57; running on four cores yields between 1.37–2.42 with a median of 1.81. A notable speedup is seen with `olafu`, a highly clustered matrix whose manipulation benefits largely from register blocking. It is the only matrix for which a quad-core RBCSR tops the 5000 MFLOP/s mark by a large gap, achieving 6060 MFLOP/s. Here, too, we see how different block sizes induce different performance characteristics, although the difference between them is not as significant as with varying optimization levels in sequential kernels.

The observed speedups with both CSR and RBCSR appear to be quiet modest, suggesting a sub-linear scaling of parallel execution of LL kernels. However, it may be that speedups are limited due to the memory bound nature of SpMV. To validate this possibility, we executed the same set of benchmarks with the CPU frequency scaled down to a low of 1.6 GHz. This decreases the computational throughput of the processor and thus reduces the gap between the CPU speed and memory latency.

The results of this benchmark are shown in Figure 4.26 and Figure 4.27. The relative speedups of these parallel runs compared to their respective single-core run are shown in Figure 4.28 and Figure 4.29. These graphs are structured analogously to the ones in Figure 4.22 through Figure 4.24.

We first notice that single-core performance of both CSR and RBCSR has decreased to 0.52–0.61 (median 0.55) of the corresponding execution at the native processor speed. This is to be expected as the single-core execution did not saturate the memory bandwidth in either of the cases.

On the other hand, we notice that quad-core performance of both kernels is within 0.6–0.95 (median 0.88) of the corresponding execution at the native processor speed. In effect, the overall throughput of a quad-core execution with a slowed down CPU frequency is close to the maximal throughput achieved with native CPU frequency. With the scaled down CPU frequency, the relative speedups due to dual-core execution are 1.82–1.94 (median 1.86) for CSR and 1.87–1.98 (median 1.91) for RBCSR. The speedups running on four cores are 2.79–3.59 (median 2.93) for CSR and 2.44–3.31 (median 2.86) for RBCSR.

All of this evidence suggests that the humble speedups observed with the native CPU frequency can indeed be attributed to memory bandwidth saturation and are inherent to SpMV. Together with the previous results, it validates our hypothesis that LL's parallelization scheme is well-suited for multicores and that performance increases with the number of cores used.

To validate the second hypothesis regarding resource utilization on parallel executions, we measured the distribution of floating point operations across the different threads in two- and four-core runs. For RBCSR, we used a single block size of 2×2 as representative of other block sizes. Table 4.3 shows the difference in FLOP count between the most loaded thread and the least loaded one for each test, both as an absolute number and as a percentage

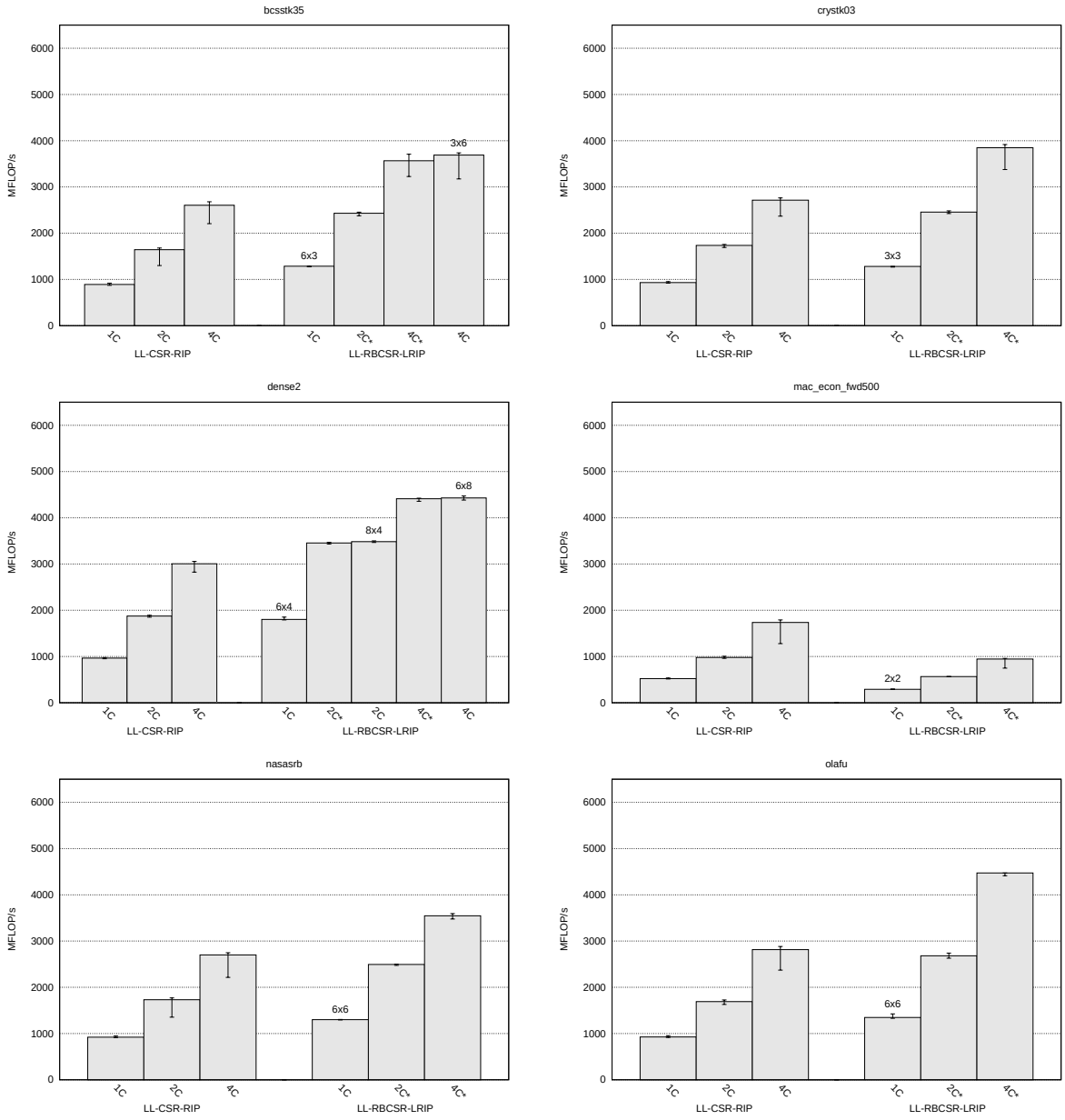


Figure 4.26: Parallel performance with scaled down CPU frequency (part 1)

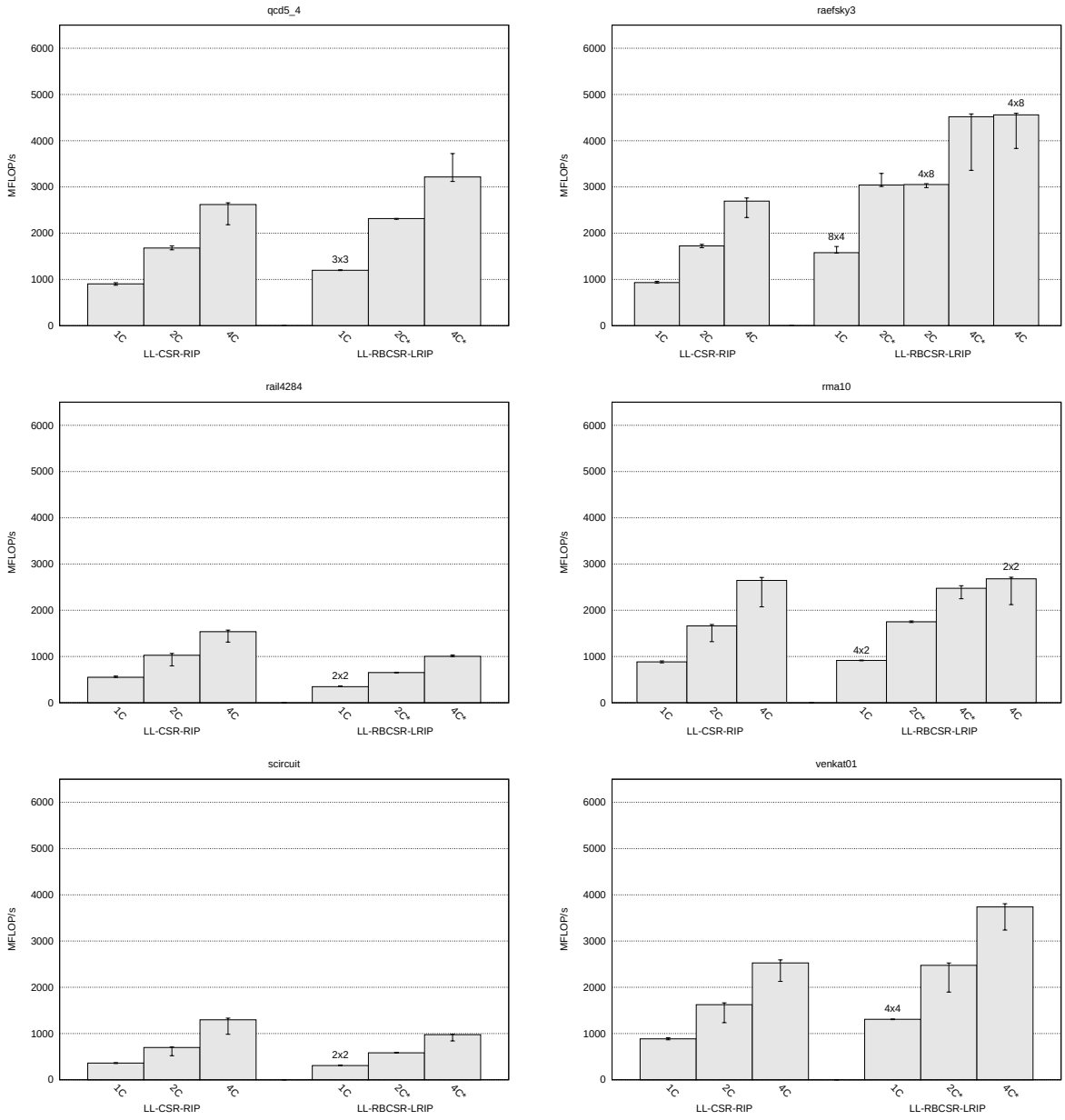


Figure 4.27: Parallel performance with scaled down CPU frequency (part 2)

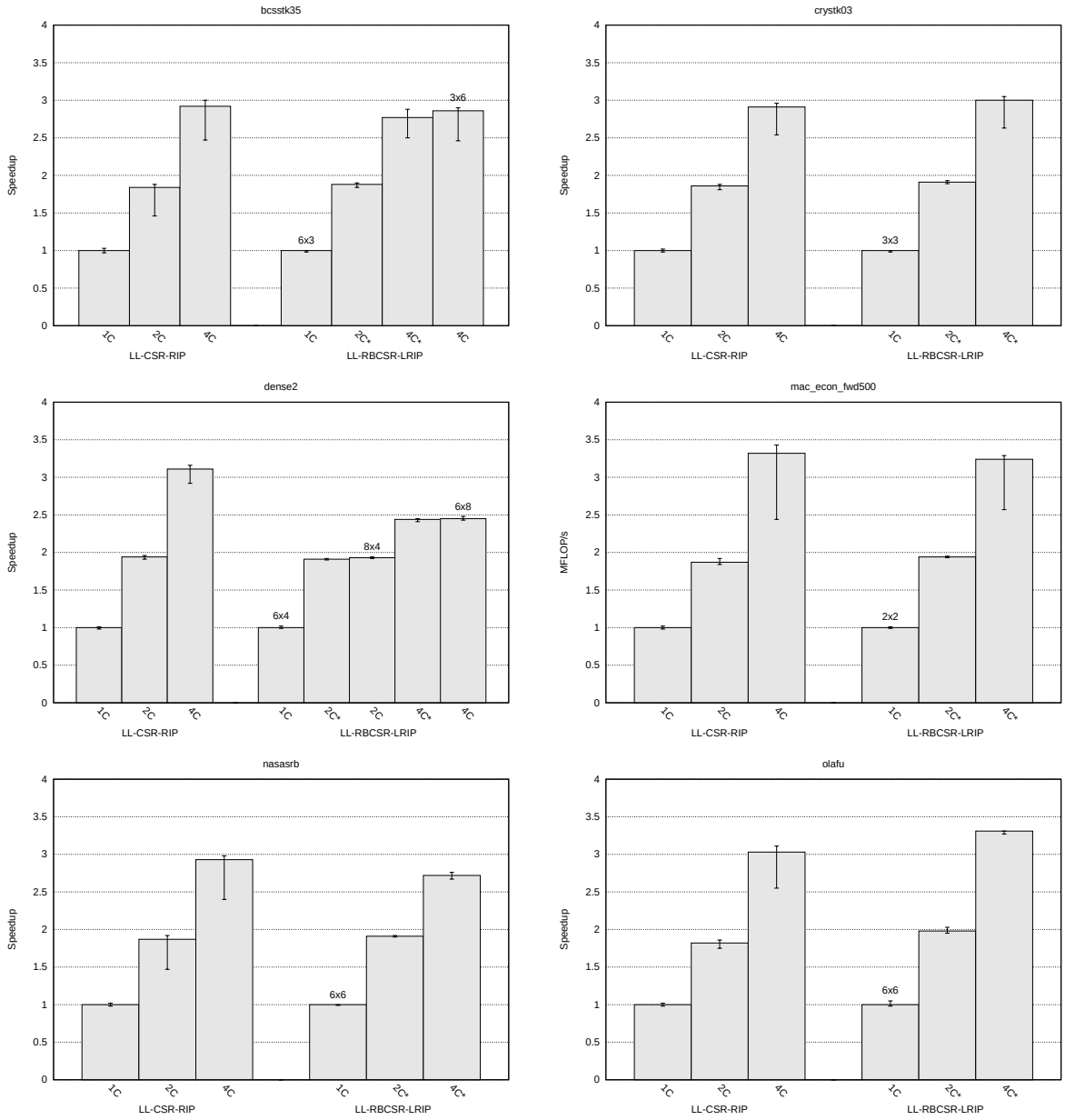


Figure 4.28: Parallel speedup with scaled down CPU frequency (part 1)

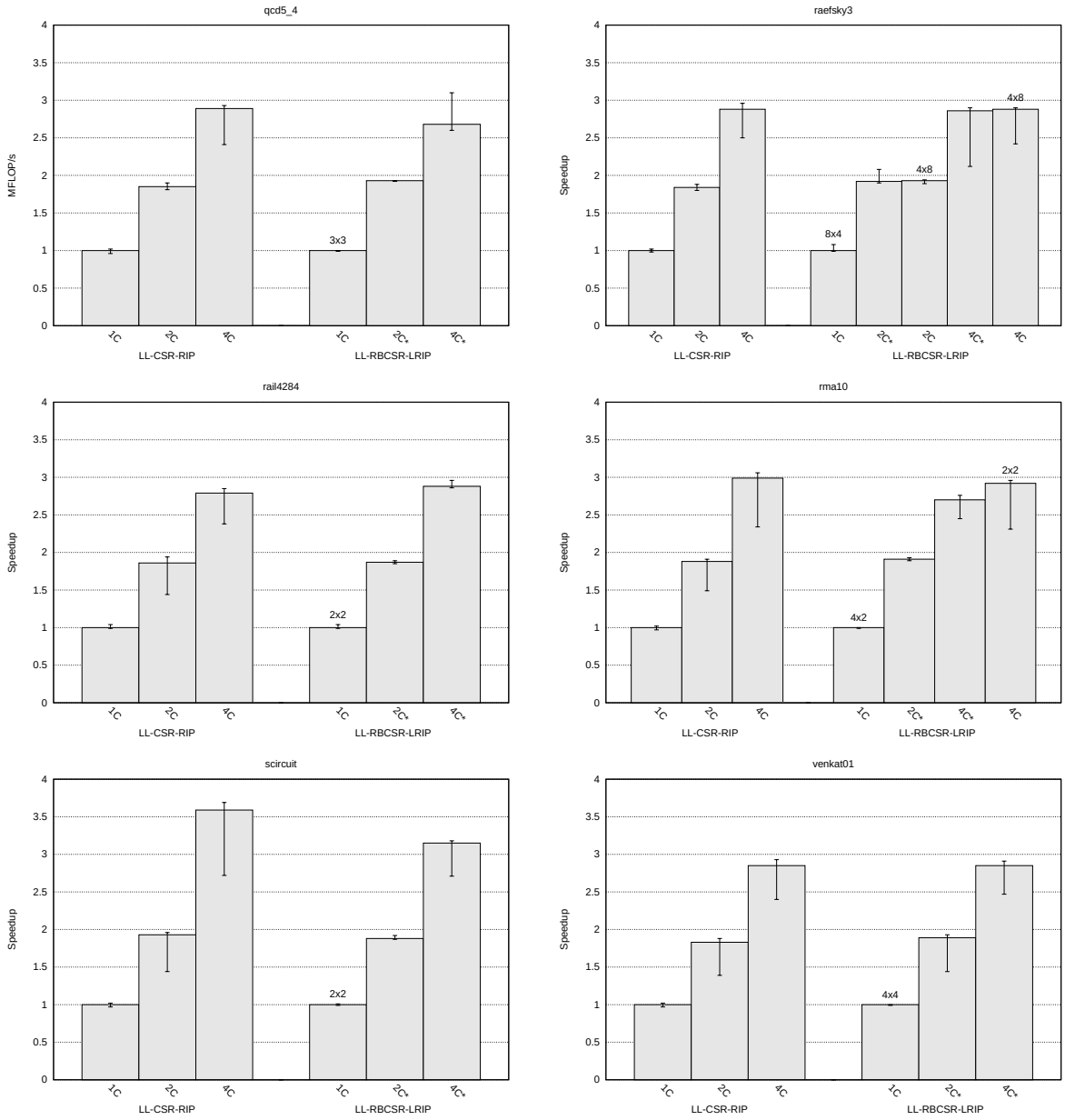


Figure 4.29: Parallel speedup with scaled down CPU frequency (part 2)

Test	Two threads				Four threads			
	CSR		RBCSR		CSR		RBCSR	
bcsstk35	82	0.01%	344	0.02%	78	0.01%	336	0.04%
crystk03	16	0.00%	16	0.00%	214	0.02%	608	0.06%
dense2	-		-		-		-	
mac_econ_fwd500	2	0.00%	80	0.00%	22	0.00%	56	0.00%
nasasrb	24	0.00%	24	0.00%	112	0.01%	312	0.02%
olafu	192	0.02%	168	0.01%	224	0.04%	168	0.03%
qcd5_4	-		-		-		-	
raefsky3	160	0.01%	448	0.03%	208	0.03%	352	0.05%
rail4284	12260	0.11%	18376	0.08%	12684	0.23%	50560	0.43%
rma10	194	0.01%	928	0.03%	346	0.03%	712	0.05%
scircuit	4	0.00%	16	0.00%	20	0.00%	32	0.00%
venkat01	16	0.00%	128	0.01%	80	0.01%	192	0.02%

Table 4.3: Workload imbalance in multi-threaded execution of LL kernels

of the workload of the least loaded thread. For all but one matrix, the observed maximal load imbalance is as high as 0.06% and as many as 928 FLOPs. In the case of `rail4284`, the load imbalance is as high as 0.23% (12684 FLOPs) for CSR and 0.43% (50560 FLOPs) for RBCSR running with four threads. This is due to the large variation in the number of nonzeros per row in this matrix. In all of these cases, workload distribution across multiple threads is very good.

We also measured the actual execution time of individual threads in two- and four-way parallel executions of both CSR and RBCSR. The results indicate a 40–50 microsecond difference between the fastest and slowest threads, in most of the test cases. In some cases larger gaps of up to 300 microseconds were observed, and in one case we observed a 2.5 millisecond gap. However, there was no correlation between these results and the difference in FLOPs shown in Table 4.3. This leads us to categorize these differences as fluctuations due to the execution environment.

Finally, we measured the actual time spent on partitioning the workload by the master thread during multithreaded execution. Distributing the workload for four threads took up to 11 microseconds for CSR and up to 7 microseconds for RBCSR. This is a negligible overhead compared to the time spent in actual computation. We conclude that the parallelization of LL kernels exhibits a good utilization of resources and induces minimal overhead.

4.9 Discussion

The results in this chapter are promising in two ways: (i) they demonstrate that, for the particular domain of sparse matrix kernels, it is possible for domain experts to directly

experiment with programing at a high-level and generate low-level code that is immediately usable; and (ii) they show that trading parallelism granularity for reduced communication can make NESL-like programs execute efficiently on modern day architectures such as shared memory multicore.

Other recent projects have had similar goals. Notable among them is Copperhead [15], a Python specializer that compiles high-level data parallel code to run on a GPU. Unlike LL, Copperhead compiles functions in a just-in-time fashion, which gives it the extra advantage of being able to tune the code to the actual input at hand. However, Copperhead does not place as much emphasis on optimizing sequential execution threads as the LL compiler does. Most notably, Copperhead nested lists can be either entirely flexible or entirely fixed (i.e., a multidimensional array), but not an mix of both. Unlike LL, optimizing a nested list structure to a cube representation is based on the actual input and decided at runtime. This makes Copperhead unsuitable for implementing tiling optimizations in sparse kernels. Additionally, Copperhead does not perform nested list access optimization such as reducer derivation and nested index flattening.

Data Parallel Haskell (DPH) is another recent effort that aims to embed the ideas introduced by NESL in a modern and actively developed functional language [18]. The Haskell type system is augmented to support explicitly parallel types, allowing to formally characterize data distribution and aggregation and obtain more control over the parallelization of different parts of the computation. It also augments the NESL vectorization transform to support additional types, and performs optimizations such as loop fusion. An example SpMV kernel is reported to perform within a factor of 0.4–0.5 of a reference C implementation. Although it scales nicely with the number of parallel execution cores, the gap in throughput between the reference and the generated C code is significant.

We address several shortcomings in the current implementation of the LL compiler and discuss opportunities for future research.

More rigorous type enrichment. The fixed list length enrichment described in Section 4.6.1 is only able to specialize lists whenever the propagation algorithm encounters a concrete constant length value. More broadly, we do not formally establish fixed-length lists as a strict subtype of regular lists in LL.

There are cases were a more powerful formalism could yield a stricter characterization of list types in LL programs. For example, we know that `block` always produces a list of lists of the same fixed length, even if we cannot infer at compile time an actual length constant. Accordingly, we could have used an efficient nested list representation for the output of this function, one that does not use a segment buffer at the outermost level.[16] Furthermore, having a proper subtyping relation for fixed-length lists would lead to more efficient type representations elsewhere in the program, due to unification.

[16] In fact, the representation of fixed-length sublists described in Section 4.6.1 is parametric by the actual length value.

Another possible enhancement is representing symbolic length expressions in the length analysis phase. Along with fixed-length list subtypes, this would allow to capture dependencies between different list objects in the program, even when the actual lengths are not compile-time constants.[17]

Better synchronization inference. Synchronization in parallel LL code is guided by its syntactic structure and may be unnecessarily fine grained. Similarly, elimination of synchronization points is done via AST rewriting, such as map fusion.

This scheme may benefit from a more informed notion of data distribution. For example, the decision as to whether or not two consecutive parallel maps should be fused should consider whether a single load distribution prior to the first map leads to properly balanced inputs for the second map's parallel threads as well. In this case, synchronization and distribution between the maps are not required, and they should be fused. Otherwise, it may be beneficial to impose a synchronization, and avoid fusing the loops.

Another shortcoming is concerned with unnecessarily fine-grained task parallelism. For example, if we can show that the two components resulting from a (parallel) pair constructor are used independently by different components of a subsequent pair constructor, then the body functions of these two constructs can be fused together and synchronization eliminated.

Synchronization analyses along similar lines were proposed by Chatterjee [21] and Catanzaro et al. [15]. We defer the implementation and evaluation of such methods to future work.

More parallelism. The parallelization scheme described in Section 4.7 can benefit from more informed and intricate program transformations.

One example is implementing vectorization of loops within parallel threads. Vectorization is key to utilizing computational resources on GPUs. It is, however, becoming gradually more effective in multicores with the advent of wider SIMD capabilities such as Intel AVX.

Another area for improvement is in the selection of maps for parallelization. The current scheme assumes (implicitly) that parallelizing and distributing the load at the outermost level is beneficial. However, this may not be the case in general: specifically, it may be that inner loops are better candidates for parallelization as they represent a more suitable workload for distribution (e.g., significantly more list elements).

Furthermore, the fact that LL does not parallelize nested maps means that some computations cannot be properly load balanced.[18] It may be beneficial to support nested

[17] A similar concept is used for inferring whole buffer lengths, described in Section 4.3.2.

[18] Similar concerns are discussed in the context of other nested data parallel languages, e.g. [15, 14].

parallel invocation, which can be guided by runtime information such as the total number of threads in encapsulating parallel loops and the relative payload assigned to a given nested map. All of these will make LL usable in situations where single-level parallelism is not enough.

Finally, the LL compiler should support parallelization of additional loop construct, specifically maps with non-fixed output types and filters. In both cases writing directly to a designated output buffers is not possible, since offsets are not known a priori. In the former case, an additional copying step is necessary for concatenating the results from parallel threads once their lengths are determined. Copying itself can be done in parallel. In the latter case, filtering can be implemented as a two-step process: first, a bitmap buffer is computed that indicates which elements satisfy the filter predicate, and the total length of these elements for each thread; second, write offsets are computed for each thread, and copying is performed to those offsets (in parallel). To compensate for possible workload imbalance in the copying phase, an intermediate load distribution step may be used as well.

Our implementation of the LL compiler, along with the benchmarking suite described in this chapter, are publicly available as a source code repository. We encourage the interested reader to download it, here: `https://bitbucket.org/garnold/llc`

Chapter 5

Conclusion

LL raises the level at which sparse formats are implemented: instead of low-level data structures and intricate computations of "one word at a time", programmers compose functional transformations on high-level data objects. LL makes dataflow explicit, allows natural composition and reuse of code, and makes it easy to track the invariants that govern the representation of matrix contents.

The limited expressiveness of LL proved to be central to our ability to automatically verify a variety of sparse formats. This eliminated the need to deal with provision and discovery of custom inductive invariants. The second important novelty is the identification of representation relations that govern the domain of sparse matrix formats, and their formalization as parametric predicates in HOL. This allowed us to keep our theory rule base small and reusable, and to separate the specification of representation invariants from the proof of transformations that manipulate them.

Finally, LL's functional semantics, fixed set of operators and simple dataflow model enables a straightforward compilation process that produces efficient, reusable and maintainable C code. Also central to our approach are the systematic mapping of high-level LL types to a hierarchical, compact C types; the identification of crucial optimizations for eliminating redundancies in sequential execution threads; and the implementation of a coarse-grained loop parallelization scheme that is well-suited for execution of data parallel sparse kernels on multicore platforms.

The three components of the LL project described in this work—language, verifier and compiler—constitute an ecosystem that improves the productivity of sparse matrix format developers. It is our hope that our results will serve as foundation for developing new methodologies and frameworks for programmers in this domain. This work entails a number of open questions and opportunities for further research, and in the following we mention several of them.

LL in the real world. It remains to show whether LL can successfully deal with a broader set of sparse formats and kernels other than SpMV. One aspect of this question pertains

to LL expressive limitations, and to possible future extensions (e.g., a general purpose reduction operator) to mitigate them.

Another part concerns expanding the compiler support for additional language primitives, better optimization, and more extensive heuristics. In general, it is unclear whether the approach we followed in implementing the compiler—namely, achieving superior sequential performance separate from (and prior to) program parallelization—will continue to be beneficial as it has been so far.

Notable in this context is the implementation of communication avoiding iterative methods, such as those described in [25, 45]. These techniques are based on the expansion of transitive data dependencies along an iterative application of a simpler sparse kernel such as SpMV. The goal is to infer a set of overlapping input chunks that will both improve the cache efficiency of the computation of particular chunks of the output, and eliminate the communication between adjacent chunks. Although we have had initial success with prototyping such a computation in LL, it remains to show whether it can be verified and compiled into efficient code.

Beyond compilation: autotuning and synthesis. Following the recent success of autotuning libraries for sparse matrix formats, it is an accepted fact that no single implementation is optimal across all machine and for all types of applications and workloads. One outcome of this observation is that no single deterministic compilation process will generate code that is universally optimal. Therefore, we consider the linking the LL compiler to an autotuning back-end an important research goal and a fertile ground for further collaboration.

On a separate note, we believe that our work on functional verification of sparse codes can be lifted to facilitate synthesis of new formats. Particularly interesting is the use of deductive proof rules in searching through a space of implementation variants. Generally speaking, this is allowing simplification to occur in both directions. Without proper means for bounding the implementation space exploration, such a process is bound to diverge and unlikely to yield useful results. However, we may be able to adapt techniques for efficiently pruning such exploration. One notable approach that is exemplified in the Denali superoptimizer [37] uses E-DAGs for representing potentially unbounded application of congruence rules in an efficient, bounded form [46]. Extending this technique for higher-order parametric congruence relations, as well as expanding application of introduction rules, can lead to interesting and useful results.

Sparse format conversion. In this work we focused on the relationship between a sparse representation and a dense mathematical object. Consequently, it was natural to use a sparse format constructor, which assumed as input a dense data structure, as a prerequisite for our verification method. However, dense matrix formats are of very little interest in reality: in most real-world situations, a portable and general purpose sparse format such as COO is used for encoding input payloads.

This observation changes the assumptions underlying some parts of our work. One interesting question is whether LL is applicable to conversion between different sparse representations. If so, it will be interesting to see whether our proof theory can handle input representations which are themselves (other) sparse formats.

Bibliography

[1] Nawaaz Ahmed, Nikolay Mateev, and Keshav Pingali. A framework for sparse matrix code synthesis from high-level specifications. In *Supercomputing (SC)*. IEEE, 2000.

[2] John Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM (CACM)*, 21(8):613–641, 1978.

[3] John Backus, John H. Williams, and Edward L. Wimmers. An introduction to the programming language FL. In *Research topics in functional programming*, pages 219–247. Addison-Wesley Longman, Boston, MA, USA, 1990.

[4] Aart J. C. Bik. *Compiler Support for Sparse Matrix Computations*. PhD thesis, Leiden Institute of Advanced Computer Science, 1996.

[5] Aart J. C. Bik, Peter Brinkhaus, Peter M. W. Knijnenburg, and Harry A. G. Wijshoff. The automatic generation of sparse primitives. *ACM Transactions on Mathematical Software (TOMS)*, 24(2):190–225, 1998.

[6] Aart J. C. Bik and Harry A. G. Wijshoff. Compilation techniques for sparse matrix computations. In *International Conference on Supercomputing (ICS)*, pages 416–424, 1993.

[7] Aart J. C. Bik and Harry A. G. Wijshoff. On automatic data structure selection and code generation for sparse computations. In *Workshops on Languages and Compilers for Parallel Computing (LCPC)*, pages 57–75, 1993.

[8] Richard S. Bird. *A calculus of functions for program derivation*, pages 287–307. Addison-Wesley Longman, Boston, MA, USA, 1990.

[9] Guy E. Blelloch and Siddhartha Chatterjee. VCODE: A data-parallel intermediate language. In *Frontiers of Massively Parallel Computation*, pages 471–480, 1990.

[10] Guy E. Blelloch, Siddhartha Chatterjee, Jonathan C. Hardwick, Margaret Reid-Miller, Jay Sipelstein, and Marco Zagha. CVL: A C vector library. Technical Report CMU-CS-93-114, School of Computer Science, Carnegie Mellon University, February 1993.

[11] Guy E. Blelloch, Siddhartha Chatterjee, Jonathan C. Hardwick, Jay Sipelstein, and Marco Zagha. Implementation of a portable nested data-parallel language. In *Principles and Practice of Parallel Programming (PPoPP)*, pages 102–111, 1993.

[12] Guy E. Blelloch, Siddhartha Chatterjee, Jonathan C. Hardwick, Jay Sipelstein, and Marco Zagha. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing (JPDC)*, 21:102–111, 1994.

[13] Guy E. Blelloch and Gary W. Sabot. Compiling collection-oriented languages onto massively parallel computers. *Journal of Parallel and Distributed Computing (JPDC)*, 8:119–134, 1990.

[14] Bryan Catanzaro. *Compilation Techniques for Embedded Data Parallel Languages*. PhD thesis, University of California, Berkeley, 2011.

[15] Bryan C. Catanzaro, Michael Garland, and Kurt Keutzer. Copperhead: compiling an embedded data parallel language. In *Principles and Practice of Parallel Programming (PPoPP)*, pages 47–56, 2011.

[16] Manuel M. T. Chakravarty, Gabriele Keller, and Simon Peyton Jones. Associated type synonyms. In *International Conference on Functional Programming (ICFP)*, pages 241–253. ACM, 2005.

[17] Manuel M. T. Chakravarty, Gabriele Keller, Simon Peyton Jones, and Simon Marlow. Associated types with class. In *Symposium on Principals of Programming Languages (POPL)*, pages 1–13. ACM, 2005.

[18] Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. Data parallel haskell: a status report. In *Workshop on Declarative Aspects of Multicore Programming (DAMP)*, pages 10–18. ACM, 2007.

[19] Bor-Yuh Evan Chang and Xavier Rival. Relational inductive shape analysis. In *Symposium on Principals of Programming Languages (POPL)*, pages 247–260, 2008.

[20] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming*. The MIT Press, October 2007.

[21] Siddhartha Chatterjee. Compiling nested data-parallel programs for shared-memory multiprocessors. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15:400–462, 1993.

[22] Adam Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *Programming Languages Design and Implementation (PLDI)*, pages 234–245, 2011.

[23] Cristina Cornes and Delphine Terrasse. Automating inversion of inductive predicates in Coq. In *TYPES*, pages 85–104, 1995.

[24] Tim Davis and Yifan Hu. The University of Florida sparse matrix collection. `http://www.cise.ufl.edu/research/sparse/matrices/`.

[25] James Demmel, Mark Hoemmen, Marghoob Mohiyuddin, and Katherine A. Yelick. Avoiding communication in sparse matrix computations. In *International Parallel and Distributed Processing Symposium*, pages 1–12, 2008.

[26] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, Hewlett-Packard Labs, July 2003.

[27] Jianjun Duan, Joe Hurd, Guodong Li, Scott Owens, Konrad Slind, and Junxing Zhang. Functional correctness proofs of encryption algorithms. In *Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, pages 519–533, 2005.

[28] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Programming Languages Design and Implementation (PLDI)*, pages 234–245, 2002.

[29] Anwar Ghuloum, Terry Smith, Gansha Wu, Xin Zhou, Jesse Fang, Peng Guo, Byoungro So, Mohan Rajagopalan, Yongjian Chen, and Biao Chen. Future-proof data parallel algorithms and software on intel multi-core architecture. *Intel Technology Journal*, 11(04), November 2007.

[30] Anwar Ghuloum, Eric Sprangle, Jesse Fang, Gansha Wu, and Xin Zhou. Ct: A flexible parallel programming model for tera-scale architectures, October 2007. Intel white paper.

[31] Jeremy Gibbons. A pointless derivation of radixsort. *Journal of Functional Programming*, 9(3):339–346, 1999.

[32] Wen-mei W. Hwu, editor. *GPU Computing Gems, Emerald Edition*. Morgan Kaufmann, February 2011.

[33] Eun-Jin Im. *Optimizing the performance of sparse matrix-vector multiplication*. PhD thesis, University of California, Berkeley, 2000.

[34] Eun-Jin Im and Katherine A. Yelick. Optimizing sparse matrix computations for register reuse in SPARSITY. In *International Conference on Computational Science*, pages 127–136, 2001.

[35] Kennth E. Iverson. *A Programming Language*. John Wiley and Sons, January 1962.

[36] Ankit Jain. pOSKI: An extensible autotuning framework to perform optimized SpMVs on multicore architectures. Master's thesis, University of California, Berkeley, 2008.

[37] Rajeev Joshi, Greg Nelson, and Keith Randall. Denali: a goal-directed superoptimizer. In *Programming Languages Design and Implementation (PLDI)*, pages 304–314, 2002.

[38] Vladimir Kotlyar and Keshav Pingali. Sparse code generation for imperfectly nested loops with dependences. In *International Conference on Supercomputing (ICS)*, pages 188–195, 1997.

[39] Vladimir Kotlyar, Keshav Pingali, and Paul Stodghill. A relational approach to the compilation of sparse matrix programs. In *Euro-Par*, pages 318–327, 1997.

[40] Alan LaMielle and Michelle Mills Strout. Enabling code generation within the sparse polyhedral framework. Technical Report CS-10-102, Colorado State University, March 2010.

[41] Lie-Quan Lee, Jeremy G. Siek, and Andrew Lumsdaine. Generic graph algorithms for sparse matrix ordering. In *International Symposium on Computing in Object-Oriented Parallel Environments (ISCOPE)*, pages 120–129, 1999.

[42] Lie-Quan Lee, Jeremy G. Siek, and Andrew Lumsdaine. The generic graph component library. pages 399–414, 1999.

[43] Nikolay Mateev, Keshav Pingali, Paul Stodghill, and Vladimir Kotlyar. Next-generation generic programming and its application to sparse matrix computations. In *International Conference on Supercomputing (ICS)*, pages 88–99, 2000.

[44] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences (JCSS)*, 17:348–375, 1978.

[45] Marghoob Mohiyuddin, Mark Hoemmen, James Demmel, and Katherine A. Yelick. Minimizing communication in sparse matrix solvers. In *Supercomputing (SC)*, 2009.

[46] Greg Nelson and Derek C. Oppen. Fast decision procedures based on congruence closure. *Journal of the ACM (JACM)*, 27(2):356–364, 1980.

[47] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[48] NVIDIA Corporation. *NVIDIA CUDA Reference Manual, Version 3.0*, February 2010.

[49] Steven Obua. *Flyspeck II: The Basic Linear Programs*. PhD thesis, Technische Universität München, 2008.

[50] Mooly Sagiv, Thomas W. Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(3):217–298, 2002.

[51] Jacob T. Schwartz. Set theory as a language for program specification and programming. Technical report, Courant Institute of Mathematical Sciences, New York University, 1970.

[52] Jeremy G. Siek and Andrew Lumsdaine. The Matrix Template Library: A generic programming approach to high performance numerical linear algebra. In *International Symposium on Computing in Object-Oriented Parallel Environments (ISCOPE)*, pages 59–70, 1998.

[53] Douglas R. Smith. KIDS: A semi-automatic program development system. *IEEE Transactions on Software Engineering*, 16:1024–1043, 1990.

[54] Armando Solar-Lezama, Gilad Arnold, Liviu Tancau, Rastislav Bodík, Vijay A. Saraswat, and Sanjit A. Seshia. Sketching stencils. In *Programming Languages Design and Implementation (PLDI)*, pages 167–178, 2007.

[55] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. Combinatorial sketching for finite programs. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 404–415, 2006.

[56] Paul Stodghill. *A Relational Approach to the Automatic Generation of Sequential Sparse Matrix Codes*. PhD thesis, Cornell University, 1997.

[57] Michelle Mills Strout, Larry Carter, and Jeanne Ferrante. Compile-time composition of run-time data and iteration reorderings. In *Programming Languages Design and Implementation (PLDI)*, pages 91–102, 2003.

[58] Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones, and Kevin Donnelly. System F with type equality coercions. In *Workshop on Types in Languages Design and Implementation (TLDI)*, pages 53–66. ACM, 2007.

[59] Richard Vuduc, James W. Demmel, and Katherine A. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. In *Scientific Discovery through Advanced Computing (SciDAC), Journal of Physics: Conference Series*, volume 16, pages 521–530, 2005.

[60] Richard W. Vuduc. *Automatic performance tuning of sparse matrix kernels*. PhD thesis, University of California, Berkeley, 2004.

[61] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Symposium on Principals of Programming Languages (POPL)*, pages 60–76. ACM, 1989.

[62] Makarius Wenzel. *The Isabelle/Isar Implementation*. Technische Universität München. `http://isabelle.in.tum.de/doc/implementation.pdf`.

[63] Martin Wildmoser and Tobias Nipkow. Certifying machine code safety: Shallow versus deep embedding. In *International Conference on Theorem Proving in Higher-Order Logics*, pages 305–320, 2004.

[64] Samuel Williams, Leonid Oliker, Richard W. Vuduc, John Shalf, Katherine A. Yelick, and James Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *Supercomputing (SC)*, pages 38:1–38:12, 2007.