Fast Linear Programming through Transprecision Computing on Small and Sparse Data

TOBIAS GROSSER[*], University of Edinburgh, United Kingdom
THEODOROS THEODORIDIS, ETH Zurich, Switzerland
MAXIMILIAN FALKENSTEIN, ETH Zurich, Switzerland
ARJUN PITCHANATHAN, IIIT Hyderabad, India
MICHAEL KRUSE, Argonne National Labs, United States of America
MANUEL RIGGER, ETH Zurich, Switzerland
ZHENDONG SU, ETH Zurich, Switzerland
TORSTEN HOEFLER, ETH Zurich, Switzerland

A plethora of program analysis and optimization techniques rely on linear programming at their heart. However, such techniques are often considered too slow for production use. While today's best solvers are optimized for complex problems with thousands of dimensions, linear programming, as used in compilers, is typically applied to small and **seemingly trivial problems, but to many instances** in a single compilation run. As a result, compilers do not benefit from decades of research on optimizing large-scale linear programming. We design a simplex solver targeted at compilers. A novel theory of **transprecision computation** applied from individual elements to full data-structures provides the computational foundation. By carefully combining it with optimized representations for **small and sparse matrices** and **specialized small-coefficient algorithms**, we (1) reduce memory traffic, (2) exploit wide vectors, and (3) use low-precision arithmetic units effectively. We evaluate our work by embedding our solver into a state-of-the-art integer set library and implement one essential operation, coalescing, on top of our transprecision solver. Our evaluation shows more than an **order-of-magnitude speedup** on the core simplex pivot operation and a **mean speedup of 3.2x (vs. GMP) and 4.6x (vs. IMath)** for the optimized coalescing operation. Our results demonstrate that our optimizations exploit the wide SIMD instructions of modern microarchitectures effectively. We expect our work to provide foundations for a future integer set library that uses transprecision arithmetic to accelerate compiler analyses.

CCS Concepts: • **Mathematics of computing** → **Mathematical software performance**; **Solvers**; • **Computing methodologies** → *Vector / streaming algorithms*; *Optimization algorithms*; • **General and reference** → *Performance*; *Experimentation.*

Additional Key Words and Phrases: Simplex, Linear Programming, Transprecision, Presburger Arithmetic

[*]Part of this work was performed at ETH Zurich.

Authors' addresses: Tobias Grosser, School of Informatics, University of Edinburgh, United Kingdom, tobias.grosser@ed.ac.uk; Theodoros Theodoridis, Department of Computer Science, ETH Zurich, Switzerland, theodoros.theodoridis@inf.ethz.ch; Maximilian Falkenstein, Department of Computer Science, ETH Zurich, Switzerland, falkensm@student.ethz.ch; Arjun Pitchanathan, IIIT Hyderabad, India, arjun.p@research.iiit.ac.in; Michael Kruse, Argonne National Labs, United States of America, michael.kruse@anl.gov; Manuel Rigger, Department of Computer Science, ETH Zurich, Switzerland, manuel.rigger@inf.ethz.ch; Zhendong Su, Department of Computer Science, ETH Zurich, Switzerland, zhendong.su@inf.ethz.ch; Torsten Hoefler, Department of Computer Science, ETH Zurich, Switzerland, torsten.hoefler@inf.ethz.ch.

2475-1421/2020/11-ART195
https://doi.org/10.1145/3428263

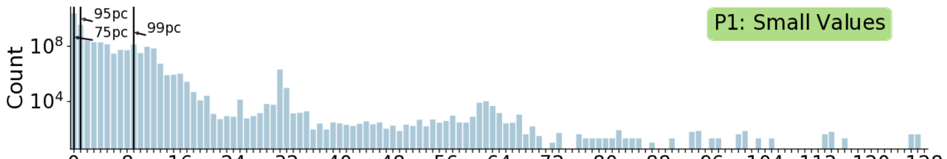
ACM Reference Format:
Tobias Grosser, Theodoros Theodoridis, Maximilian Falkenstein, Arjun Pitchanathan, Michael Kruse, Manuel Rigger, Zhendong Su, and Torsten Hoefler. 2020. Fast Linear Programming through Transprecision Computing on Small and Sparse Data. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 195 (November 2020), 28 pages. https://doi.org/10.1145/3428263

1 INTRODUCTION

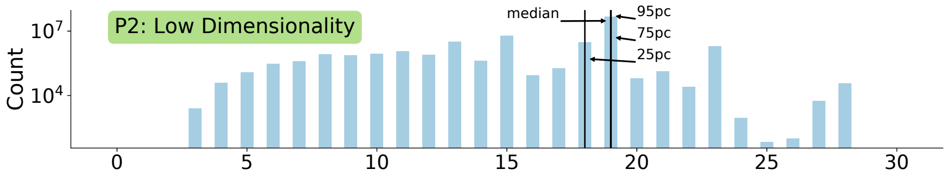
Linear programming (LP) with rational constraints [Detlefs et al. 2005; Nelson 1981] and arithmetic over integer sets [Verdoolaege 2010], which uses rational linear programming at its core, have become established building blocks for program analysis and optimization. Bao et al. [2017] and Gysi et al. [2019] model, for example, the exact behavior of a cache hierarchy as a sequence of operations on integer sets. Like many other program analyses, their cache models derive integer sets directly from source code elements, such that the structural properties of the input programs have an immediate influence on the structure of the resulting constraints and cost functions. Linear programming solvers nowadays can scale to enormous problems by using intricate pivoting strategies and sparse matrix formats. However, the properties of linear programs as they arise in program analysis are very atypical. As a result, they do not have a constraint structure on which today's solver implementations excel. Modern linear programming solvers focus on solving large individual problems with thousands of variables and constraints. In program analysis, we solve small problems with only a handful of variables and constraints (corresponding to source-level variables, loop nests, and array dimensions), over and over again. In addition, solvers for most of the other scientific and technical fields use (inexact) floating-point arithmetic while rational arbitrary precision arithmetic (e.g., GMP [Granlund et al. 2015]) provides the exactness guarantees that are desirable in the programming languages world. This observation led us to ask: how can one design a fast linear programming solver tailored towards program analysis?

We characterize the linear programming problems that arise during program analysis to understand which properties make them special. For our analysis, we use the state-of-the-art integer set library (isl) [Verdoolaege 2010] that implements integer set arithmetic as typically used by polyhedral compiler analyses. It provides the mathematical foundations for a wide range of program analyses and optimizations [Baghdadi et al. 2015; Bao et al. 2017; Grosser et al. 2012; Gysi et al. 2019; Vasilache et al. 2018; Verdoolaege et al. 2013]. While isl implements many essential algorithms, this paper focuses on optimizing the simplex solver at its core and its use in one key operation, integer set coalescing. Coalescing uses linear programming to determine relationships between pairs of convex integer sets to combine them into a single convex set [Verdoolaege 2015]. Merging convex sets is important to obtain and maintain a concise representation, which enables isl to efficiently reason about integer sets. Simplifying integer sets that model program properties such as cache hits and misses, memory footprints, data volumes, optimized loop structures, and similar properties through coalescing is a typical use case for linear programming in the context of program analysis and optimization. While the desirable goal of optimizing a full integer set library requires a long-term effort (Section 2.1), we believe that innovations aimed at the rational simplex and its use in coalescing provide important foundations for further innovation in this space.

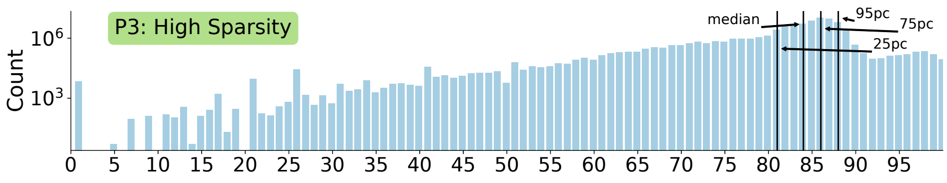
We obtain detailed statistics by tracing and analyzing linear programs in three key areas: analytical modeling, polyhedral loop optimization, and accelerator code generation. We run the Haystack analytical cache model [Gysi et al. 2019] on 30 computational kernels in Polybench [Pouchet 2012], run the Polly polyhedral loop optimizer on Polybench and the Polly regression tests, and use PPCG [Verdoolaege et al. 2013] to map stencil kernels to GPUs through hyrid-hexagonal tiling [Grosser et al. 2014]. Using isl, we extract a representative set of 465,460 linear problems



(a) Value Size - Bitwidth of coefficients in binary representation



(b) Dimensionality - Number of matrix columns



(c) Sparsity - Percent of zero coefficients

Fig. 1. Linear programming for program analysis has unique characteristics visible in 465,460 simplex tableaux: more than 99% of all coefficients fit 8-bit integers, the typical column count fits modern vector registers (up to 32 16-bit elements), and the matrices are sparse (median sparsity 84%).

that arise during integer set coalescing. Our experience leads us to conjecture that the typical problem instances are small, which we now validate by measuring three key properties:

(1) the size of the constraint coefficients (number of bits required to represent them),
(2) the number of dimensions in each constraint, and
(3) the sparsity (number of zeros) in the constraints.

Figure 1 provides the results of our analysis. Our bit-width analysis (Figure 1a) shows that more than 99% of all coefficients require less than 10 bits, and only a tiny number of inputs require more than 64 bits. While larger coefficients are rare, we observe up to 127 bits. Such coefficients typically represent important corner-cases for which no practical upper bound is known. To ensure correctness, the use of arbitrary precision arithmetic (e.g., GMP) is standard in constraint libraries such as `isl`. Our dimensionality analysis (Figure 1b) shows that all input matrices have a small number of columns, at most 28 and usually less than 20 columns. Matrices typically also have less than 64 constraints. Our sparsity analysis (Figure 1c) shows a median sparsity of over 80%. Linear programming solvers have been exploiting sparsity [Houstis et al. 1990; Kreutzer et al. 2014; Monakov et al. 2010] for a long time, but the sparsity we observe is unusual. While classical algorithms commonly process a single large matrix, we process thousands of small matrices one after another. In summary, we identify three properties: (P1) small values, (P2) low dimensionality, and (P3) high sparsity – three opportunities to optimize linear solvers for program analysis.

We introduce a high-performance simplex solver tailored for program analysis that takes advantage of native integers, SIMDization, as well as custom-designed algorithms and data structures.

Most integers that we observed (Figure 1a) fit into 16 bits; performing vectorized arithmetic on small native integers is vastly cheaper than using an arbitrary precision library. To take advantage of the small-scale sparsity, we introduce a novel micro-sparsity format suitable for small matrices. This representation supports operations such as querying which rows have non-zero values in a particular column. Such functionality is not available in traditional sparse formats. We combine all of the above ideas in *transprecision integer matrices*, that is, matrices that operate on the smallest native integer type that can represent the stored numbers and switch to wider ones on demand. Transprecision matrices enable straightforward vectorization of row-wise operations by specializing for a fixed integer type and column count, while at the same time guaranteeing correctness. Only when large values arise in a computation do transprecision matrices transition to high precision types. Therefore, they limit the associated high costs to cases where this is unavoidable. We use transprecision arithmetic combined with new algorithms specialized for small integers to accelerate both the core simplex solver and one key operation (coalesce) of a state-of-the-art integer set library. With transprecision integers, as a less intrusive variant of transprecision computing, we also provide benefits to libraries that have not been designed with transprecision in mind. Our combined work provides important foundations for the future development of a full integer set library (see Section 2.1) that we envision to be designed from the ground up for high-performance matrix-level transprecision.

Our contributions are:

- Transprecision computing tools
  - *An overflow-aware SIMD library* that extends C++ std::simd with overflow detection
  - *Transprecision integers* combine fast native arithmetic with a high-precision fall-back
  - *Transprecision matrices* combine native SIMD arithmetic with a high-precision fall-back
- Transprecision constraint solving
  - A *transprecision simplex solver* that exploits transprecision matrices
  - A *transprecision coalesce operation* that exploits the fast simplex solver
- Algorithm and data-structure optimizations for small and sparse matrices
  - A *vectorized small GCD* algorithm
  - A *micro-sparsity format* (MSF) for small and sparse matrices
- A performance evaluation comparing to the state-of-the-art integer set library `isl`
  - More than an *order-of-magnitude speedup* for the core pivot operation in our simplex solver
  - A mean *speedup of* 3.2× (vs. GMP) and (4.6x vs. IMath) for integer set coalescing

2 BACKGROUND

We first provide background of how simplex-based linear programming is used in a state-of-the-art integer set library (Section 2.1). We then describe the core pieces of the simplex algorithm variant that we later optimize with transprecision matrices, our micro-sparsity format, and algorithm specialization (Section 2.2). We then summarize the differences between typical matrices and the small, sparse ones we are interested in (Section 2.3). Finally, we discuss the importance of high-performance program analyses (Section 2.4).

2.1 Simplex at the Core of Integer Set Arithmetic

While the simplex algorithm is at the heart of many program analyses, various algorithms typically extend the rational simplex solver to enable reasoning about integer polyhedra or full Presburger arithmetic [Haase 2018; Presburger 1929]. With the integer set library `isl`, the use of full Presburger arithmetic has become standard in the field of polyhedral loop analysis and optimization. To better

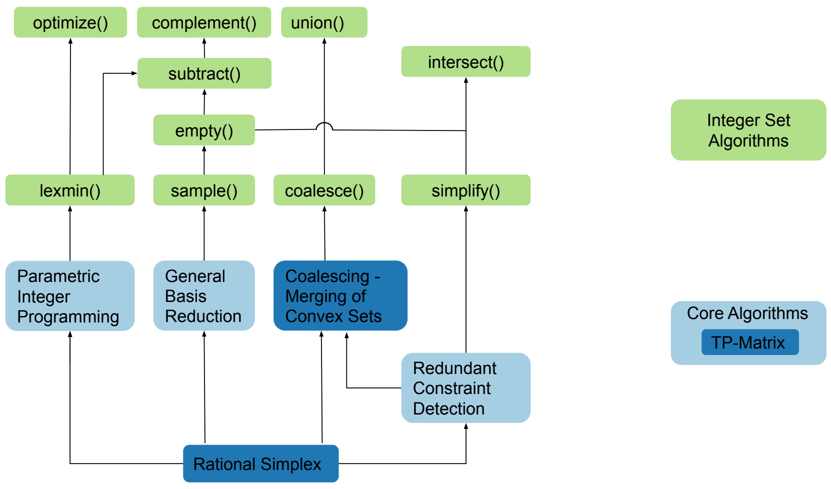


Fig. 2. A rational simplex solver is at the core of solving integer set problems in compilers.

understand the simplex algorithm's importance, we overview the main algorithms that drive a modern integer set library. We describe their interaction and how they use a rational simplex solver.

Figure 2 illustrates the different components of an integer set library, such as `isl`. The foundation for all the algorithms is a rational simplex solver. Parametric integer programming [Feautrier 1988] in combination with integer cuts [Schriver 1986] implemented on top of the rational simplex provides support for the lexicographic optimization of problems with multible parametric symbols. Integer emptiness of a constraint set is computed via generalized basis reduction [Lovász and Scarf 1992], which uses the rational simplex to compute rational extreme values along basis directions. Finally, the rational simplex is used to detect rationally redundant constraints and to verify if two constraint sets can be coalesced into a single convex object.

On top of these core algorithms, a theory of integer sets is implemented. While some operations are simple (e.g., the complement of a set A is a subtraction of A from a universe set), others are complex. For example, the subtraction itself uses integer emptiness checks to reduce the result's size and relies on lexicographic optimization to reason about existential quantification. The connections in Figure 2 visualize essential dependences (e.g., the use of lexmin for subtraction) as well as dependences to optimize performance (e.g., emptiness checks during subtract or coalescing after union). Verdoolaege [2020] provides further information on isl.

Due to the size and complexity of an integer set library (around 100,000 LoC for isl), we propose a multi-step approach to designing a novel transprecision integer set library. The first step in this work is the optimization of the rational simplex solver, which provides the foundation for all other components. It is unique in that it does not depend on any other complex algorithms. In theory, we can optimize it down to the assembly code and even introduce non-trivial changes such as the use of data-structure-level transprecision computations. To exercise the resulting simplex solver and evaluate its performance, we port one higher-level algorithm to the new simplex solver. Our choice is the coalesce algorithm, which combines pairs of convex objects and only requires a fast simplex solver and the detection of redundant constraints. In a second step (not part of this work), the integer simplex as well as parametric integer programming, two algorithms that typically share many low-level data-structures (e.g., the tableau) with the rational simplex, must be re-designed on top of the transprecision data structures of our rational simplex. Finally, in step three (also not part of this work), the high-level integer set algorithms must be re-designed with transprecision in mind. As step two and three are long term objectives that are not immediately attainable, this work introduces transprecision integers as an incremental and minimally invasive approach that provides some transprecision benefits for existing integer set implementations (e.g., isl). In summary, our proposed approach provides immediate benefits and a long-term path towards the future implementation of a full high-performance integer set library.

2.2 Linear Programming and the Simplex Algorithm

Linear programming is an optimization method widely used for modeling various problems such as planning, program analysis, routing, and scheduling. A linear program in standard form has a linear objective function subject to linear equality and non-negativity constraints:

$$\begin{array}{ll} \text{minimize} & \mathbf{c}^{\mathsf{T}}\mathbf{x} \\ \text{subject to} & \mathbf{A}\ \mathbf{x} = \mathbf{b} \\ & \quad \mathbf{x} \geq \mathbf{0} \end{array}$$

where $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ are the variables of the problem, $\mathbf{c} = (c_1, c_2, \ldots, c_n)$ are the coefficients of the objective function, and $\mathbf{A}$ a $p \times n$ is the matrix which, together with $\mathbf{b} = \left(b_1, b_2, \ldots, b_p\right)$, $b_i \in \mathbb{N}$, encodes the constraints of the problem. The Simplex algorithm [Bertsimas and Tsitsiklis 1997] is commonly used to solve linear programs. It also allows us to reason about inequality constraints, either by modeling them explicitly or by translating them into a combination of equality and non-negativity constraints. For program analysis, we also use it to check the feasibility of sets of constraints and detect redundancies in them. For example, is the inequality system with $\alpha > \beta$, $\beta > \gamma$, and $\gamma > \alpha$ feasible? Or, given $\alpha > \beta$, $\beta > \gamma$, and $\alpha > \gamma$, is the third inequality redundant?

The main data structure used by the simplex algorithm is called the tableau. The tableau encodes the equalities in a matrix (and a few auxiliary arrays). Each row of the tableau represents a linear equation $y[i] = \alpha[i, 0] + \sum_j \alpha[i, j]x[j]$. To answer queries like the examples in the previous paragraph, the tableau is transformed through a series of steps until the system is obviously feasible or not feasible. These transformations are called pivots, and they are the main performance bottleneck. A pivot operation solves a linear equation (that of the pivot row) for one of its variables (the pivot column) and updates the rest of the tableau through a series of row-wise operations:

$$\begin{array}{c|cc} & \text{pivot column} & \text{any other column} \\ \hline \text{pivot row} & \alpha & \beta \\ \text{any other row} & \gamma & \delta \end{array} \quad \Rightarrow \quad \begin{array}{c|cc} & \text{pivot column} & \text{any other column} \\ \hline \text{pivot row} & \frac{1}{\alpha} & -\frac{\beta}{\alpha} \\ \text{any other row} & \frac{\gamma}{\alpha} & \delta - \frac{\beta\gamma}{\alpha} \end{array}$$

Pivoting can produce rational entries, which cannot be represented as floating-point numbers due to potential inaccuracies. Such inaccuracies may be acceptable in other applications, but in our use case they could lead to miscompilations or wrong analysis results, so we require perfect accuracy. One option is to use rational arithmetic (e.g., GMP's `mpq_t`). However, an alternative is more amenable to optimizations: using a single common denominator for each row. This alternative representation simplifies operations, e.g., the division by α in the pivot row and the inversion of the pivot element can be performed by exchanging α's numerator with the row's denominator. Also, the row-wise operations can be vectorized easily. One issue with maintaining an explicit denominator is that the integers in the tableau can quickly grow large. Large integers arise because operations such as addition increase the denominators (e.g., by multiplying them together). This numeric explosion can be prevented by normalizing each row (simplifying the fractions). Normalization computes the row-wise greatest common divisor (gcd) and divides the whole row by it.

	Dense	Sparse
Big	HPC ML	Large Scale Linear Optimization
Small	Computer Graphics	Program Analysis Constraint Libraries

Fig. 3. Problem domains have different typical matrix sizes and densities.

A potential implementation of the pivot would perform the following steps:

(1) Process the pivot row. The pivot element, α, is inverted, and the other pivot row entries, β, are negated and divided by α. Let d_p be the common denominator of the pivot row. Then $\alpha = \alpha_n/d_p$ and $\beta = \beta_n/d_p$, so we have $1/\alpha = d_p/\alpha_n$ and $-\beta/\alpha = -\frac{\beta_n/d_p}{\alpha_n/d_p} = -\beta_n/\alpha_n$. Therefore, we can perform the pivot row transform by swapping the numerator α_n with the pivot row's common denominator d_p and adjusting the signs of the non-pivot column entries, $\beta'_n = -\beta_n$. The new pivot row is divided throughout by its gcd so that its entries remain as small as possible.[1]

(2) Process non-pivot rows. If the pivot column entry, γ, is equal to zero, then this row does not require processing, since $\gamma/\alpha = \gamma$ and $\delta - (\beta\gamma)/\alpha = \delta$, if $\gamma = 0$. Otherwise, the pivot column entry $\gamma = \gamma_n/d_o$ is transformed to $\gamma/\alpha = \frac{\gamma_n/d_o}{\alpha_n/d_p} = \frac{\gamma_n d_p}{\alpha_n d_o}$. The rest of the entries $\delta = \delta_n/d_o$ are transformed to $\delta - (\beta\gamma)/\alpha = \delta_n/d_o - \frac{(\beta_n/d_p)(\gamma_n/d_o)}{\alpha_n/d_p} = \frac{\delta_n\alpha_n - \beta_n\gamma_n}{\alpha_n d_o}$. These transformations can be implemented as follows. First, we update the common denominator $d'_o = d_o a_n$. Then we multiply the pivot column entry by d_p (which is now the value of the pivot column entry of the pivot row after transformation). We then multiply all non-pivot column entries by a_n. Finally, we add to the result the product $\beta'_n\gamma_n$ (pivot column entries of the current row multiplied by the negated pivot row entries of the corresponding column). These row-wise operations can be fully vectorized if the common denominator representation is used. The common denominators do not have to be updated separately; they can be updated as part of the vectorized arithmetic.

2.3 Matrices: Big vs. Small and Dense vs. Sparse

Operations on matrices are prevalent across various computation domains such as machine learning, signal processing, linear programming, scientific computing, computer graphics, and graph processing. Matrix types differ among domains (Figure 3). For example, large sparse matrices can be used to solve differential equations [Zlatev 1991] or large-scale linear programs [Gill et al. 1984]; an example for large dense matrices is deep learning [Chetlur et al. 2014]. Small dense matrices are standard in domains such as computer graphics, for which specialized hardware exists. One mostly unexplored area are matrices in constraint libraries for programming languages such as PolyLib [Loechner 1999], PPL [Bagnara et al. 2008], and `isl`. Our initial analysis suggests that such libraries operate on integer or rational matrices that are often small, sparse, and with a small value range. An additional constraint imposed by these libraries is the use of arbitrary precision arithmetic (e.g., GMP [Granlund et al. 2015]). Even though numbers are usually small, they can sometimes grow large, and using unchecked fixed-width integers would lead to incorrect results. Arbitrary precision arithmetic prevents vectorization (operations are hidden behind library calls that act as an optimization barrier for compilers) and as a result, most of the hardware resources (very wide vector units) are underutilized.

[1]This step does not change the actual numbers but only their representations, e.g., $2/4 = 0.5 \longrightarrow 1/2 = 0.5$.

Matrix formats used in sparse linear algebra can take advantage of the sparsity and enable the use of vector units. However, they are unsuitable for the previously mentioned constraint libraries as they are meant for large and structured matrices. Their data structures introduce memory indirections through their sparsity encoding, and result in more expensive memory loads and stores. For very small matrices such as those we are interested in, the space savings would be minimal and would not justify the overhead.

2.4 The Importance of High-Performance Program Analysis

The analysis and optimization of programs are often time-critical. A classical compilation flow, where a single program is compiled into a binary before it is deployed for production use, is often assumed not to be time-critical. However, this assumption is not always correct. While for research, a slightly higher compile time cost paid once is rarely problematic, for production compilers such as LLVM, compilation time increases are carefully observed. As a result, even small increases often prevent the adoption and further evolution of research results through the LLVM community. Another area where compile times are critical is iterative compilation [Pouchet et al. 2008, 2007] and auto-tuning [Pfaffe et al. 2019]. In this setting a compiler (or just the Presburger library) is run many times with different configurations to find (close to) optimal configurations. Here, the run time cost directly affects the size of the space that can be explored and, consequently, directly impacts the kind of questions that can be answered.

Advanced analyses are also increasingly used to provide interactive user feedback. One key motivation of the HayStack cache model [Gysi et al. 2019] was the idea of memory-aware programming where code editors provide immediate feedback on memory behavior to programmers. Like code-completion, this feedback is time-critical, as already reaction times above 100ms are perceived as disruption. While cache models that take seconds to complete are useful, these models need to be at least an order of magnitude faster to enable smooth memory-aware programming. As polyhedral reasoning can be expensive (Section 4), efforts towards improving the performance of the underlying math libraries are critical.

3 SMALL AND SPARSE TRANSPRECISION COMPUTING

We observed that linear programming is a key algorithm for program analysis that is typically applied to problems with particular properties: the integer coefficients have small values and the matrices that arise are typically sparse and of small size (Figure 1). We introduce three techniques to take advantage of these properties:

- **Transprecision Computing Tools - Section 3.1**
  We transparently switch between integer types, at a granularity from individual elements to full matrices, and enable high-performance processing through code specialization and vectorization.

- **The Micro-Sparsity Format MSF - Section 3.2**
  We use a micro-sparsity format that takes advantage of the sparse structure of small matrices.

- **Small Value Algorithms - Section 3.3**
  We introduce new algorithms that take advantage of the small value range of integers when computing greatest common divisors as needed for constraint normalization.

We combine these techniques into a new transprecision simplex solver (Section 3.4) that provides high-performance for program analysis problems.

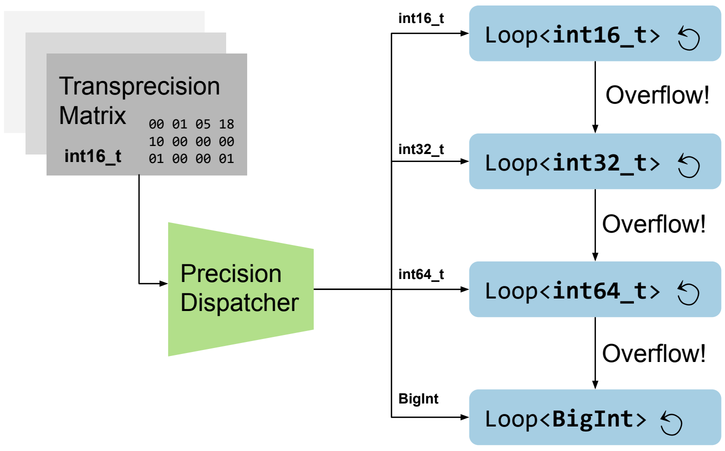


Fig. 4. Transprecision loop multi-versioning: multiple high-performance versions of hot loops in transprecision code are generated. At runtime, the appropriate version based on the actual integer type is dispatched. In the event of an overflow, execution is diverted to a higher precision version.

3.1 Transprecision Computing

To benefit from our observations that matrices in constraint libraries are typically small and store primarily small integers, we propose *Transprecision Matrices*. The core idea is to use the smallest possible integer type provided by the target architecture for the entire matrix, and fall back to a larger integer type when needed. Using the smallest possible integers allows us to utilize the underlying hardware efficiently. Furthermore, by switching to higher precision when integer overflows occur, correct behavior is guaranteed.

More formally, assume an architecture that natively supports a set of integer *types*; such as $types = \{int8, int16, int32, int64\}$. Let $rank(type)$ be the ordering of types according to their bitwidth; rank 0 corresponds to the smallest type, e.g., $rank(int8) = 0$ because $int8$ is the smallest type in $types$. Let $min(type)$ and $max(type)$ respectively denote the smallest and largest integer value that can be represented by that type. Initially, a transprecision matrix is initialized to the smallest integer type (i.e., $rank(type) = 0$). When an operation result stores an integer val into the matrix it must be checked that $val \geq min(type)$ and $val \leq max(type)$. If the condition does not hold, then a new type is chosen according to $\arg\min_{type} rank(type)$ under the constraints $val \geq min(type)$ and $val \leq max(type)$, and the matrix is transparently (to the user) widened to that type. If none of the native integer types can represent the integer, we must fall back to higher-precision types (e.g., `__int128` or GMP).

Transprecision matrices enable high performance for all integer types through code specialization, referred to as *multi-versioning*. Initially, we introduce transprecision matrices without any code specialization. Using performance profiling, we then analyze the code and identify often executed loops that are slow due to the use of high-precision computations and costly per-access dispatches into our transprecision matrices. Each of these hot loops is multi-versioned for each concrete integer matrix type and column count[2] to enable vectorization, and the appropriate versions are dispatched at runtime (Figure 4) and at loop granularity. Depending on the underlying integer

[2]Column counts are rounded up to multiples of the available vector lanes for this type.

width and the number of columns of the transprecision matrix, the appropriate version of the loop is executed at runtime.

The critical dispatch is implemented using metaprogramming with C++ 14 variadic templates. We synthesize a large switch statement where each case specializes a type-generic (Section 3.1.3) user-provided function parameter for a given element type and column count. The following restrictions apply: there must not be loop-carried dependencies, and each loop iteration must process one row of the matrix (handling overflows would require additional bookkeeping if loop carried dependencies exist or if multiple rows were to be processed). By using the tightest possible integer width, fewer memory transactions are needed, and more elements can be processed without increasing the number of executed instructions; this is possible through vectorization. For example, in AVX-512, a single register can hold one of the following: (1) 64×`int8_t`, (2) 32×`int16_t`, (3) 16×`int32_t`, or (4) 8×`int64_t`. Arithmetic instructions that operate on such registers process all elements in parallel at the same time.

```
void genericFunction(TransprecisionMatrix &a, TransprecisionMatrix &b) {
    TransprecisionMatrix c(a.nRows(), a.nCols());
    for (int i = 0; i < a.nRows(); ++i)
        for (int j = 0; j < a.nCols(); ++j)
            // Automatic integer type change on overflow
            c[i,j] = a[i,j] + b[i,j];
    c.dispatchLoop (
      [] (auto row, const auto &vector_of_twos) {
        // Body of the Loop
        //
        // The body is provided as type-generic code. Parameters are declared
        // as 'auto' such that they can be specialized to the requested
        // combinations of element type and vector width. The appropriate
        // version of the specialized loop body is dispatched at runtime,
        // depending on the underlying type of matrix c. Overflows are detected
        // and handled transparently by switching to a wider integer version of
        // the loop.
        return row * vector_of_twos;
      },
      [] (auto &matrix) {
        // Loop invariant state
        //
        // The loop body is provided as type-generic code that is specialized to
        // each integer type and column width. It is executed once for each
        // version of the loop that is executed.
        decltype(matrix)::VectorType vector_of_twos{2};
        return vector_of_twos;
      }
    );
}
```

Listing 1. Example transprecision code: the multiversioned loop dispatcher expects a loop body and loop invariant state generator as arguments.

Arbitrary precision arithmetic guarantees that the results of operations are always representable, with the only limitation being the amount of memory required to store them. Arithmetic operations on native (fixed-size) integers can overflow when an operation's result exceeds the range of a scalar type [Dietz et al. 2015]. While CPUs have flags set whenever an overflow occurs, it is impossible to directly access them in C/C++. However, compiler builtins such as `__builtin_add_overflow` are provided by mature compilers [Rigger et al. 2019], which, apart from performing the arithmetic, also return the overflow flag status. The multi-versioned hot loops are implemented in terms of such intrinsics: arithmetic operations are checked, and the overflow status is reported to the dispatcher.

The transprecision dispatcher handles overflows. If an overflow occurs, then the results cannot be stored in the original matrix (e.g., if the matrix is currently using 16-bit integers and the result requires 32-bits). The current loop iteration is aborted, the matrix is widened (copied to a matrix of identical dimensions but using a wider integer type), and processing of the aborted iteration is restarted. If additional loop-invariant state exists, then it is also widened.

We implement transprecision matrices as a C++ library. The example in Listing 1 illustrates their use. Lines 1-30 define a function which includes one transprecision loop and a set of compile-time generated specialized loops. Its input matrices are transprecision and, thus, their integer type can be any native or arbitrary-precision type. Lines 3-6 create a new transprecision matrix and store in it the sum of the two inputs. The arithmetic is scalar. By default, transprecision matrices are initialized with the narrowest integer type, `int16_t` (while narrower types do exist, e.g., `int8_t`, they are impractical due to the lack of basic vectorized instructions such as multiplication Table 1). If the results of all the additions that are executed at line 6 fit in 16-bits, then the actual integer width used will not change. If one or more of the results are values that require wider integers, then `c` will be automatically widened. Such scalar loops are meant for non-performance critical code; hot code should use multi-version loops, such as the one shown in Lines 7-29.

The dispatcher accepts two lambdas:

(1) The loop body: its first argument is the row to be processed and its second the loop-invariant state, a vector of twos. Line 18 multiplies the current row with the vector of twos and returns the result, which also includes an overflow flag (the multiplication and overflow check are vectorized; see Section 3.1.3 for more details).
(2) The loop invariant state generator: it generates a vector with all values being equal to two (a minimal example). The invariant state is computed before the first iteration. It is recomputed whenever an overflow occurs to ensure that the types of the newly generated state match the type of the widened matrix.

The dispatcher chooses the appropriate version of the loop depending on the underlying type of matrix `c` and switches to a larger bitwidth type if overflows occur.

3.1.1 Transprecision at Different Granularities. Precision switching can occur at granularities more fine-granular than matrix-level transprecision (Figure 5). Each granularity comes with its drawbacks and advantages in terms of performance and flexibility for user code:

(1) **Element-wise transprecision** is the most flexible and least performant variant. It maintains for each scalar value a separate type, promotes individual integers independently, and dispatches at each scalar access. Element-wise transprecision is the least intrusive variant since all precision switching logic is encapsulated in a type that can be used as a drop-in replacement for any scalar type. The drawback is that all operations are complex: they have to perform type-checking and promotion, and as a result, they cannot be vectorized. As a result, the dispatch cost for processing a full matrix is in $O(\#rows * \#cols)$.
(2) **Row-wise transprecision** retains flexibility from a user's perspective while allowing for first performance optimizations. Each matrix row can switch between integer widths independently. With this approach, operations can be vectorized, but type-checking and promotion

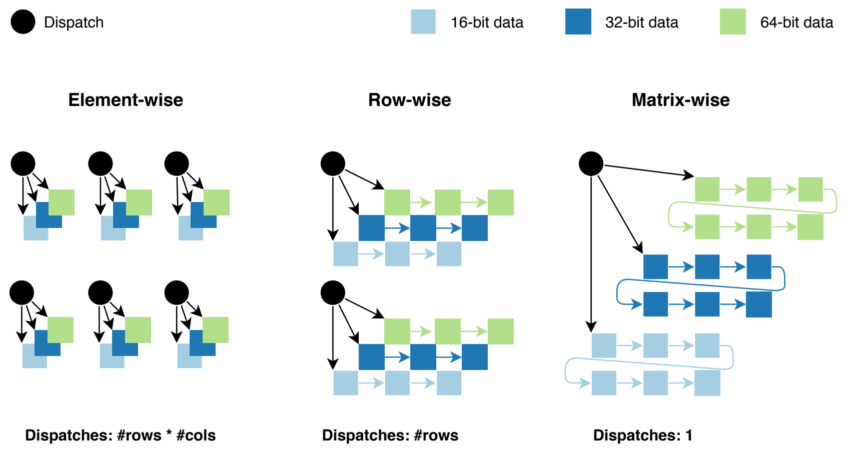


Fig. 5. The dispatch cost decreases from element-wise to matrix-wise transprecision.

are still required. Type-promotion is further complicated as operations must now be specialized for any pair of types that two rows may use. The dispatch cost for processing a full matrix is in $O(\#rows)$.

(3) **Matrix-wise transprecision** maintains a uniform integer type across a full matrix and always switches all integers to the minimal type that can be used across the full matrix. Type-checking and promotions happen at a coarser granularity. As a result, the dispatch cost for processing a full matrix is in $O(1)$.

In addition to matrix-wise transprecision (Section 3.1), we also implemented element-wise transprecision (Section 3.1.2). We also introduce row-wise transprecision as a concept but noted that the large majority of matrices we observe (Figure 1) are uniformly small, such that matrix-wise transprecision is preferable due to its lower dispatch costs and its less complicated implementation.

3.1.2 Element-Wise Transprecision Arithmetic. Element-wise transprecision integers transparently and independently switch between implementations, even if stored in matrices. Our element-wise transprecision integers are built on top of existing arbitrary-precision libraries: a transprecision integer is either a pointer to an arbitrary-precision integer or a fixed-width integer embedded into that pointer. To implement this, we take advantage of pointer alignment, which ensures that the value of the least significant bit (LSB) of a pointer is always zero. It can, consequently, be used to discriminate the integer type. For example, if the LSB is zero, then the pointer points to an arbitrary precision integer, else a 32-bit integer is embedded in the upper bits of the pointer (and the pointer is not actually a pointer); this integer can be extracted via bit manipulation. Operations on pairs of transprecision integers promote their arguments if necessary. In Section 4.4, we evaluate the performance impact of transprecision integers on polyhedral compilation.

3.1.3 Type-Generic Vector Code Generation. To enable vectorized transprecision programming, we need integer-type independent code that is lowered efficiently to target-specific instructions. The simplest solution is to write scalar code and rely on auto-vectorization. However, there are no guarantees that the auto-vectorizer generates efficient code or even vectorized code at all; the programmer would have to inspect the generated assembly and modify the source code such that, if possible, the auto-vectorization is successful. An alternative is to manually write vector intrinsics or inline assembly and hand-tune the code, but in modern instruction set extensions, such

Table 1. The AVX-512 instruction set is not complete, but the available instructions depend on the bitwidth of the elements they process. Only for 16-bit, AVX-512 has native instructions for computing the high-bits of a multiplication as well as element-wise saturation – both important components for overflow checks.

Instruction	**8-bit**	**16-bit**	**32-bit**	**64-bit**
SIMD multiplication	✗	✓	✓	✓
SIMD multiplication - high-bits	✗	✓	✓	✗
SIMD addition	✓	✓	✓	✓
SIMD addition - with saturation	✓	✓	✗	✗

as AVX-512, the available combinations of instructions and types are not uniform (Table 1), e.g., when operating on 16-bit or 32-bit integers, it is possible to extract the high-bits of multiplications which are necessary for overflow checks, but this is not possible in the case of 64-bit integers. Also, depending on the number of active elements in a vector register, an instruction that operates on half or a quarter register can reduce the execution cost (higher throughput). Transprecision code written with intrinsics would have to take all of these details into account and be manually written many times. Instead of these approaches, we only use the experimental C++ interface `std::simd` [Hoberock 2019] and rely on LLVM [Lattner and Adve 2004] to perform instruction selection and generate high-performance vectorized code.

```
template<typename INT, unsigned N>
bool multiply(VEC<INT, N> a, VEC<INT, N> b, VEC<INT, N>& result) {
  return std::any_of(multiply_with_overflow(a, b, &result));
}
```

Listing 2. Vectorized multiplication with overflow check.

```
vpmulhw %zmm0,%zmm1,%zmm2
vpmullw %zmm0,%zmm1,%zmm0
vpsraw $0xf,%zmm0,%zmm3
vpcmpneqw %zmm3,%zmm2,%k0
xor %eax,%eax
kortestd %k0,%k0
setne %al
```

Listing 3. Assembly for 32 element 16-bit vectorized multiplication with overflow checking.

Three components are necessary to make transprecision code work: a vector programming interface, overflow detection for vectors, and efficient code generation. Our interface is based on `std::simd` that already provides vector types templated on the element type and the vector width. Overflow checks are essential for transprecision code since they allow us to switch between integer widths on-demand. Overflow checks for vector types are already implemented in LLVM. However, they are not exposed in LLVM's C/C++ front end, Clang; and `std::simd` does not support overflow checking either. Builtins for scalar overflow checking exist (e.g., `__builtin_mul_overflow`). However, they do not work for vector types. We extend Clang such that we can use these builtins to accept vector types as well, thus enabling Clang/LLVM to generate end-to-end high-performance vector code with overflow checks. While our particular implementation based on LLVM's libcxx and the clang/LLVM vector lowering is toolchain specific, we believe that the design we propose is also suited for GCC and other C/C++ compilers. The combination of these three components enables us to write code like Listing 2.

The code (Listing 2) is element type and integer width agnostic; the compiler can automatically generate all necessary versions. We show as an example the generated code for 16-bit integers and 32 elements per vector (Listing 3). The first instruction, `vpmulhw`, multiplies two pairs of 32 16-bit integers in registers `zmm0` and `zmm1`, and stores the high 16 bits of the products in `zmm2`. The second instruction, `vpmullw`, multiplies the same inputs and stores the products' low 16-bits back in `zmm0`. The third instruction, `vpsraw`, performs a right arithmetic shift to the previous product and stores the results in `zmm3`. The fourth instruction, `vpcmpneqw`, compares the high bits of the multiplication, stored in `zmm2`, with the right shifted lower bits, stored in `zmm3`. If they differ, an overflow has occurred and the remaining instructions write this result in `al` (`eax` and `al` are different views of the same register). The right arithmetic shift by 15 ensures that if the product's low bits represent a non-negative number then the result will be `0x0000`; otherwise, it will be `0xFFFF`. These are exactly the values the high bits should contain in the corresponding cases if no overflow occurred.

Overflow checking, `std::simd`, and LLVM's efficient code generation together allows us to:

- Write type and vector width generic code once and instantiate it statically for all necessary combinations of types and width, e.g., (`int16_t`, `int32_t`, ...) × (8, 16, 32, ...).
- Take advantage of LLVM's ability to lower such code to the target architecture and exploit instructions that are available for a subset of the integer types.
- Check for overflows efficiently and enable transprecision computations.

These components are sufficient to write the type-generic high-performance vector code that we need to implement the body of our loop dispatcher (Listing 1). To further increase readability of these loop bodies and hide any explicit overflow tracking, we can optionally introduce an overflow-aware vector type that encapsulates a `std::simd` data vector and a `std::simd` overflow vector and automatically keeps track of overflows. As a result, we derive automatically the desired high-performance variants of our code.

3.2 Micro-Sparsity

Our goal is to benefit from matrix sparsity in order to improve performance. There are various sparse matrix formats used in linear algebra, e.g., compressed sparse row/columns, ELLPACK [Houstis et al. 1990], Sliced ELLPACK [Monakov et al. 2010], SELL-C-sigma [Kreutzer et al. 2014], which take advantage of the underlying structure of the matrices and can enable vectorization on such matrices. Such representations can significantly reduce the memory footprint of matrices and enable processing of inputs, which would be impracticably large if stored densely. Our use case has requirements which are not supported by these formats; for example, we require the ability to efficiently determine which rows of a matrix are non-zero at a specific column (this is used in the common pivoting operation); current sparsity formats do not allow efficiently querying row information. With current dense or sparse formats a whole or a large part of it is loaded from main memory (since CPUs load at the granularity of entire cache lines, e.g., 64 bytes on x86) even if a single byte is requested. This results in an avoidable (for the rows that are zero at the specified column) latency penalty. With our proposed sparsity information, only the necessary rows are actually loaded.

We propose a new format for small and sparse matrices: the matrices are still stored densely, but additional sparsity information is maintained. Using this information, queries such as "all rows which are non-zero at column *j*" are implemented with a low number of bitwise instructions. These queries enable us to accelerate both the pivot operation itself and the selection of the pivot elements. While there are algorithms that would benefit from accelerated row-wise queries (e.g, finding a pivot column for a fixed row), our analysis has shown that these queries are not common enough to require acceleration. Consequently, our matrix format focuses on column-wise queries.

To accelerate column-wise queries, we store the sparsity information itself column-wise: we store in each 64-bit integer the zero/non-zero state of a single column with up to 64 rows. Since we are targeting small matrices, we can store all the sparsity information in a handful of integers. As an example, for the following matrix:

$$\begin{matrix} 0 & 1 & 0 & 0 & 0 & 5 & 0 & 5 & 0 \\ 0 & 0 & 7 & 0 & 0 & 0 & 2 & 1 & 0 \\ 0 & 0 & 0 & 9 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 8 & 3 & 3 & 2 & 3 \end{matrix}$$

the sparsity information would be stored in nine integers (shown in binary):

0000 0001 0010 0100 1000 1001 1010 1011 1100

each entry represents the sparsity of a single column, e.g., the last entry is 1100, this means that the first two entries of the last column are zero and the other two are non-zero. With this representation, querying which rows are non zero at a particular column is very efficient: instead of iterating over all rows, we check the sparsity entry in the column of interest:

```
1 SparsityInfo rows_to_process = tab.get_sparsity(pivot_col);
2 rows_to_process.remove_row(pivot_row);
3 while (rows_to_process) {
4     int row_id = rows_to_process.next_row();
5     // Process row
6 }
```

Listing 4. Example use of micro-sparsity.

We first get the sparsity vector for column `pivot_col` (Line 1). Then we drop row `pivot_row` because we do not want to process it (Line 2), regardless of its entries (this is implemented by setting the corresponding bit to 0). The loop iterates while there are still unprocessed rows (Lines 3-6). The next row is obtained by finding the next set bit in `rows_to_process` (Line 4); at the same time, this bit is set to 0 so that this row is only processed once.

3.3 Algorithm Specialization for Small Integers

Algorithms are typically optimized for asymptotic complexity. In contrast, we want to optimize for known value ranges; we take advantage of the fact that inputs are generally small to optimize one of the common operations during pivoting: normalization. A row is normalized by dividing it by the gcd of its elements. A typical implementation would sequentially compute pairwise gcds using Euclid's algorithm. Euclid's algorithm requires multiple modulo operations, which are very expensive even on native integers (10s of cycles in modern CPUs). An alternative way to compute the gcd of two or more numbers is through their prime factorization. For example, $gcd(198, 3036, 51612) = gcd(\mathbf{2} \times \mathbf{3} \times 3 \times \mathbf{11}, \mathbf{2} \times \mathbf{3} \times \mathbf{11} \times 17, \mathbf{2} \times 2 \times \mathbf{3} \times \mathbf{11} \times 23) = \mathbf{2} \times \mathbf{3} \times \mathbf{11} = 66$. The gcd of a set of numbers is the product of their common prime factors (including multiples). Prime factorization is very expensive in general [Crandall and Pomerance 2006], but it can be precomputed for a set of common small values.

We propose a compact prime factorization representation to enable high-performance normalization: we use bits to represent prime factors. In binary, each bit represents a power of two, e.g., $101_b = 1 \times 2^0 + 0 \times 2^1 + 1 \times 2^2 = 5$. In the proposed prime factor base, each bit represents a prime factor. A prime factorization can include multiples of one or more factors, e.g., $27 = 3 \times 3 \times 3$. Thus, more than one bit might be necessary per prime number. With 32-bits all numbers up to 82 (starting at 2) can be represented contiguously. While some larger numbers can be represented as well, there are gaps in between, e.g., we cannot represent 83 because it would require an additional bit, but we can represent $84 = 2 \times 2 \times 3 \times 7$. We only need multiple factors for 2, 3, 5, and 7, e.g., $64 = 2^6$. The following table shows which prime number is represented by each bit:

Bit-Position	**Prime Factor**							
$0 \ldots 7$	2	2	2	2	2	2	3	3
$8 \ldots 15$	3	3	5	5	7	7	11	13
$16 \ldots 23$	17	19	23	29	31	37	41	43
$24 \ldots 31$	47	53	59	61	67	71	73	79

Example: the prime factorization of 60 is $\{2, 2, 3, 5\}$. In the 32-bit prime factor base, it is 10001000011.

With integers up to 82 pre-factorized and stored in prime factor base 32, computing the gcd requires only a logical AND reduction. Using AVX-512, for up to 16 integers, this requires a few vector shuffles and logical AND operations that perform a vertical reduction on the integer vector. The computed gcd is then used as the divisor. This division in prime factor base is an XOR operation: if a factor is present in both the divisor and the dividend, then it should be removed, e.g., $\frac{76}{38} = \frac{2 \times 2 \times 19}{2 \times 19} = 2$. The conversion back to normal integers can be performed as a lookup.

3.4 Transprecision Simplex Algorithm

We implemented the variant of the simplex algorithm described in Section 2.2 by combining transprecision matrices, small-scale sparsity information, and small integer algorithm specialization. We use a transprecision matrix to represent the tableau (Section 3.1). The parts of the algorithm which are not performance-critical directly use transprecision matrices as a drop-in replacement for regular matrices. The tableau's underlying integer type is automatically and transparently widened whenever necessary. We currently do not automatically revert back to smaller types in case coefficients shrink, as the solvers in Presburger libraries are typically short-lived and re-created when needed. The performance-critical parts, such as the pivot operations, are implemented with our high-performance vectorized interface (Section 3.1.3); multiple versions are generated at compile-time, and the appropriate version is dispatched at runtime. We use sparsity information to further accelerate the pivoting step by reducing memory traffic (Section 3.2). Finally, we use our small-integer specialized gcd computation to accelerate the row normalization step (Section 3.3). All matrices are allocated on the heap as dense one-dimensional array similar to how isl uses a single one-dimensional stretch of memory to store its data. While stack allocations are likely to reduce memory management costs and yield additional performance improvements, our preliminary analyses suggest only a limited speedup potential. We left this optimization for future work.

4 EVALUATION

We evaluate the performance of transprecision arithmetic, small-scale sparsity, and algorithm specialization for small integers by integrating our transprecision simplex solver in `isl`, which is used by many applications such as polyhedral compilation [Grosser et al. 2012], cache modeling [Gysi et al. 2019], and accelerator mapping [Grosser et al. 2014]. Operations on integer sets can incur large runtime overheads. For example, `isl`-based operations account for the majority of the compilation time when using LLVM and Polly on Polybench (Figure 6): 71% on average and up to 93%.

Integer set coalescing [Verdoolaege 2015] is one such operation. It simplifies composite maps (polyhedra) by merging (coalesceing) their convex components and relies on a simplex solver to determine redundant equalities and inequalities. Polly (Figure 7a), cache modeling (Figure 7b), as

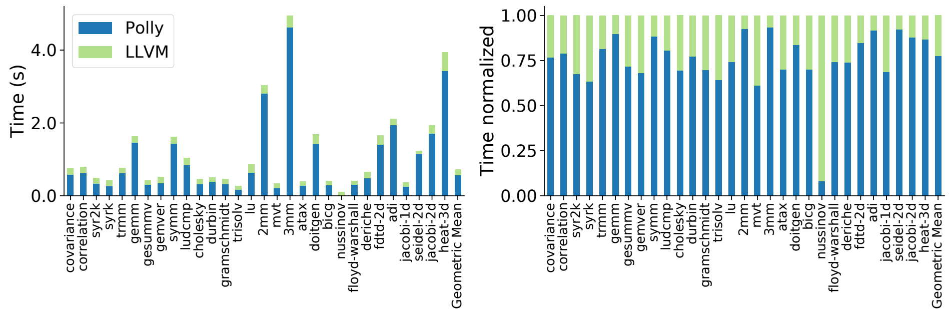
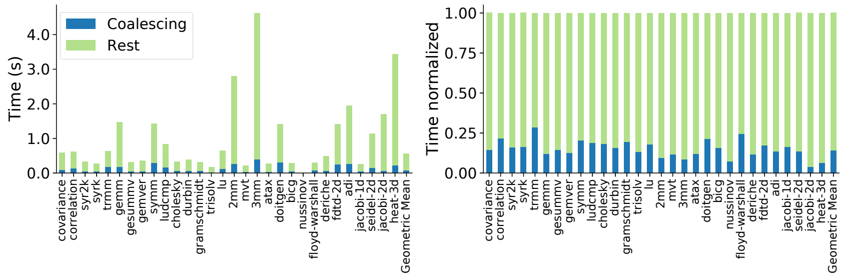
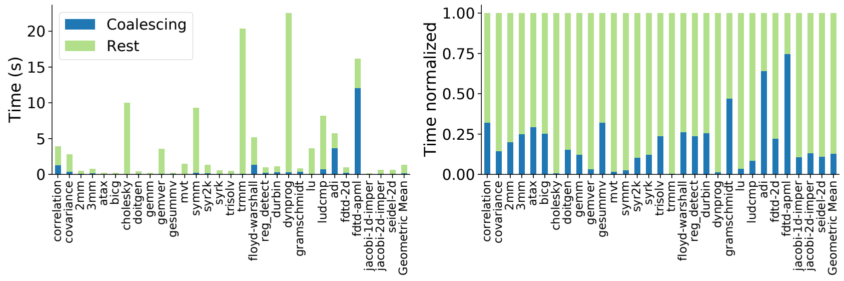


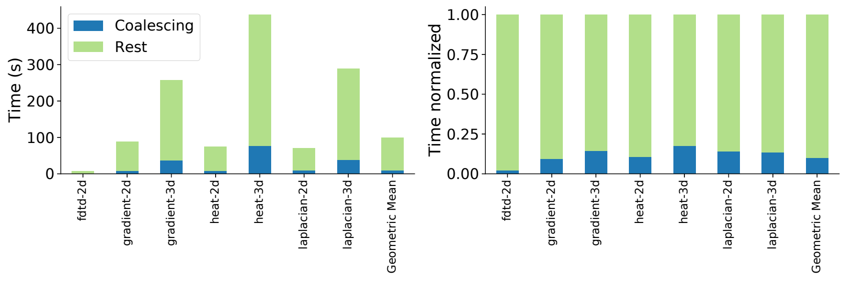
Fig. 6. Compilation Times of Polybench (LLVM 9 with Polly): the polyhedral part (Polly and `isl` dominate the compilation runtime, 71% on average and up to 93%.



(a) Polyhedral Compilation on Polybench: 14% on average and up to 28% in coalescing.



(b) Cache Modeling on Polybench: 11% on average and up to 74% in coalescing.



(c) Accelerator Mapping with Hexagonal Tiling: 10% on average and up to 17% in coalescing.

Fig. 7. Polyhedral compiler spend 10–14% on average and up to 74% in a single operation, coalescing.

well as accelerator mapping (Figure 7c) depend on coalescing. We use the coalesce operation as part of our evaluation because a) a non-trivial amount of time is spent on it in a variety of applications, b) it only relies on the simplex solver. Therefore, it is feasible to independently optimize it without re-engineering large parts of `isl` (Section 2.1).

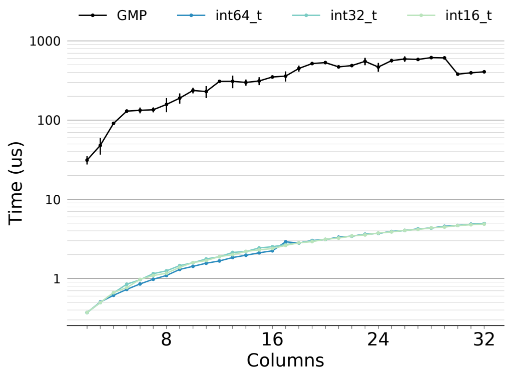
We also analyzed the longest-running Polly compilation on the 3mm polybench benchmark to understand if we can easily point out other hot functions. Unfortunately, the design of `isl` makes such an analysis hard. Linux perf gets seemingly confused by `isl`'s widespread use of function pointers, callbacks, and higher-order functions (e.g., `isl_union_set_foreach`, `isl_ast_node_foreach_descendant_top_down`) and the call-graph cycles their use introduce. Tracing with Valgrind's callgrind [Weidendorfer 2008] worked reliably and showed that 75% of the runtime is spent in a big cycle involving isl and Polly. In this cycle, 40% of the runtime is spent in AST generation (isl_ast_from_schedule_node or one of its children) [Grosser et al. 2015]. Due to the overall recursive design, this does not allow us to conclude that the actual schedule tree construction itself is so expensive, but a non-trivial amount of time is likely spent on modifying isl objects (e.g., adjusting memory access functions on-the-fly). The hottest functions (without considering time spent in children) are the pivot function itself, followed by `isl_seq_inner_product`, `isl_seq_abs_min_non_zero`, `isl_seq_combine`, `isl_mat_product`, various other `isl_seq*` functions, as well as malloc and free. While these functions cannot be easily attributed to higher-level operations (e.g., subtract, intersect), they are typically used to modify a sequence (or matrix) of constraints when implementing higher-level algorithms. This consequently provides a weak indication that faster constraint processing through SIMDization and the reduction of memory allocation (e.g., through transprecision computing) could accelerate other components of `isl`.

To evaluate our transprecision simplex solver, we use the cache model analyzer [Gysi et al. 2019] on Polybench to generate coalescing inputs (Section 4.5). The cache model analyzer's runtime varies from 0.1 to more than 22.5 seconds, depending on the input benchmark (Figure 7b). The total coalescing time per benchmark varies from 0.01 to around 12 seconds (up to 75% of the total runtime). The number of coalescing operations per benchmark vary from 31 up to 672.

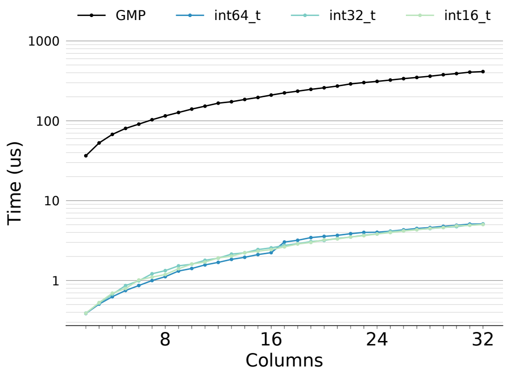
In the following subsections, we first microbenchmark individual optimizations to understand their impact on the core pivot loop. We demonstrate:

- Overhead of arbitrary-precision arithmetic (Section 4.1.1)
- Benefits of vectorization (Section 4.1.2)
- Overhead of transprecision arithmetic (Section 4.1.3)
- Benefits of micro-sparsity (Section 4.2)
- Benefits of specialization of small integers (Section 4.3)

All microbenchmarks are aimed to explore the performance impact on the full range of potential inputs. We explore this space by exploiting the fact that the dynamic behavior of the core pivot loop does not depend on particular coefficient values but only on the size of the matrices and their sparsity. As a result, we can synthesize microbenchmarks that cover the full space of potential inputs. All microbenchmarks are run under two different regimes: hot and cold cache. The hot regime repeatedly operates on the same inputs. The cold regime operates on a set of data whose memory size is large enough to exceed the size of the last level cache. Both cases are executed in a loop for at least one second, after which the mean runtime is calculated. We then show the impact of transprecision integers on polyhedral compilation (Section 4.4). Finally, we benchmark `isl`'s coalescing operation. That is, we use our transprecision simplex solver to accelerate it and measure the total runtime improvement (Section 4.5). Both our code and the isl benchmarks use an isl 0.21 development build from Jan 2019 (git hash 1d10ba4cb5) as foundation. All benchmarks



(a) Hot cache



(b) Cold cache

Fig. 8. For the simplex row operations, arbitrary precision integers have a significant cost compared to (scalar) native integer arithmetic.

are run single-threaded on a Google Cloud N2 instance using an Intel Xeon 3.1 Ghz Cascade Lake processor (family 6, model 5, stepping 7, extended family 0, extended model 5) with 32KB L1 data and instruction cache, 1024 KB unified L2 cache, 24.8 MB unified L3 and AVX-512 vector extensions (foundation, double & quad word, confict detection, byte and word instructions, vector length).

4.1 Row-Wise Operations Performance

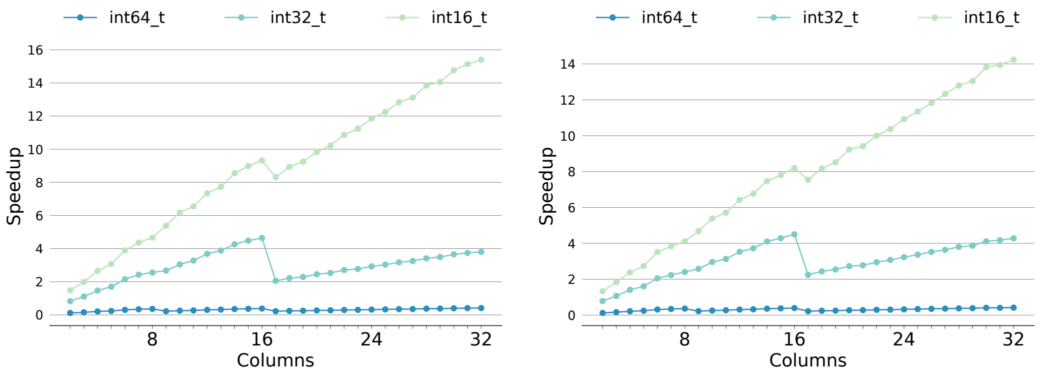
Simplex operations such as pivoting modify the tableau's rows. To understand the performance behavior of such row-wise operations in isolation, we benchmarked the core loop of the pivot operation (Section 2.2), excluding normalization. We subsequently benchmark the full pivot operation.

4.1.1 Overhead of Arbitrary Precision Arithmetic. Most of our data needs a small number of bits (Figure 1a). We microbenchmark the row-wise operations using both arbitrary precision integers (GMP) and native integers. Our results (Figure 8) confirm that scalar code with native integers is much faster than GMP, 25× for 16 columns and more than 80× for 32 columns using 16-bit integers.

Multiple factors cause this performance gap. Arbitrary precision arithmetic is complex and native integer arithmetic is directly implemented in hardware circuits. GMP potentially allocates additional memory to store the result of each operation. This can drastically reduce performance. Each operation is a library call; this incurs function call overhead and hinders compiler optimizations.

The transprecision interface does not suffer from these drawbacks. Typically, all arithmetic is performed with native integers. Thus, the data is stored contiguously, and no indirections or additional dynamic memory allocations are required. We only need to fall back to arbitrary precision arithmetic when the largest native integer is not wide enough.

4.1.2 Vectorization. We use C++'s experimental `std::simd` vector library to generate vectorized code (described in Section 3.1.3) and target AVX-512. The achieved speedup in the hot cache regime (Figure 9a) is close to 15× when using 16-bits and 32 columns. The cold cache speedup (Figure 9b) peaks at around 14× as we are limited by memory latency. The 32-bit version achieves speedups of up to 4×, which is 1/4 of what the 16-bit version does; one would reasonably expect 1/2 the performance. However, AVX-512 does not have instructions that compute the higher bits of 32-bit products (it does for 16-bit), and the compiler generates 64-bit code, which halves the effective vector register width. The 64-bit vectorized version is slower than the scalar one because LLVM is



(a) Hot cache

(b) Cold cache

Fig. 9. Speedup of vectorization with overflow checks over scalar code in row-wise operations of the pivot kernel for a varying number of columns. Vectorization speeds up `int16_t` and `int32_t` row-operations by taking advantage of data parallelism. There is a slowdown for `int64_t`, as the scalar loop benefits from scalar overflow flags, whereas the vector code uses vector instructions that cannot be lowered effectively due to missing 64-bit instructions (Table 1).

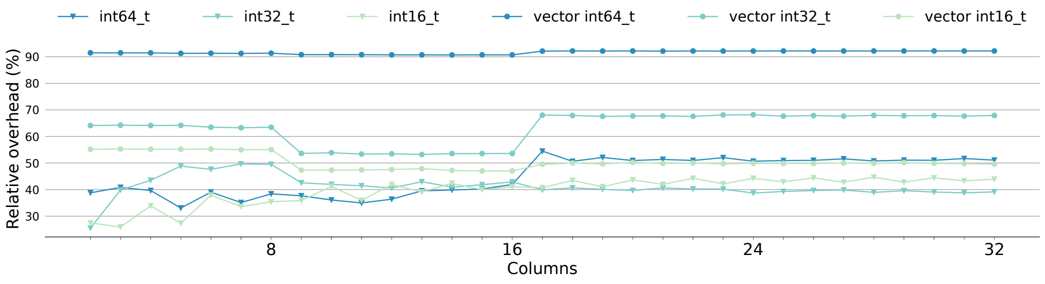


Fig. 10. The relative overhead of overflow checks in row-wise operations of the pivot kernel for a varying number of columns. The overhead is measured against a version that performs only the arithmetic and no overflow checks. Scalar overflow checks are cheaper than the vectorized ones since they are implemented in terms of the overflow status register.

generating inefficient code: the numbers are extracted one-by-one from vector registers, multiplied with scalar instructions, and then re-inserted in vector registers. There are no vector multiplication instructions that return the higher bits of a 64-bit product. Consequently, LLVM has to scalarize the code. Repeatedly extracting from and inserting in vector registers is expensive.

We compare the cost of overflow checks by benchmarking scalar and vectorized versions of a kernel, with and without overflow checks. Vectorized checks are more expensive (Figure 10). Scalar checks are implemented via reading the overflow flag (from the CPU status register), whereas the vectorized version checks for overflows by performing extra arithmetic. The former's performance overhead is between 40% and 60%, and the latter's is between 60% and 100%. Even though the vectorized checks induce more overhead, the speedup due to data parallelism is significant enough to justify their use.

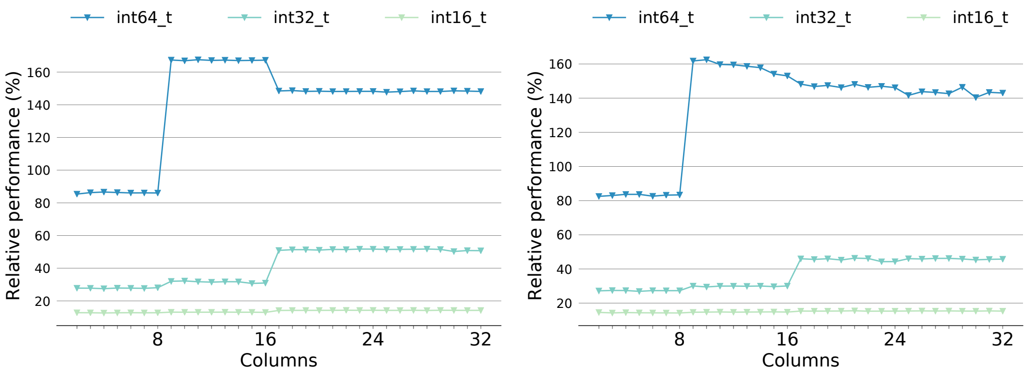


Fig. 11. Transprecision overhead in vectorized row-wise operations of pivot kernel hot loop (without the zero-check in line 16 and normalization) for a varying number of columns.

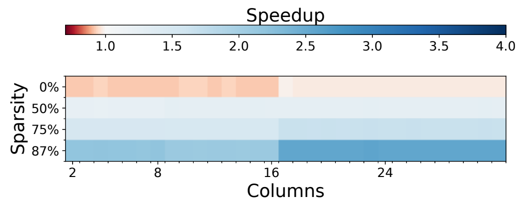
4.1.3 Transprecision Row-wise Operations. The transprecision version of the row-wise operations can fall back to higher integer widths, potentially to arbitrary precision, and this induces an additional overhead over the checked version. Switching integer widths requires copying the whole input matrix. However, we do not measure this cost; we are interested in the overhead of using the transprecision interface (this copying overhead is included in subsequent benchmarks, whenever applicable). The overhead is depicted in Figure 11. Our interface introduces overhead that differs between types, which is likely related to a non-optimal lowering in LLVM. While the overall overhead is non-trivial and could likely be reduced further, especially for lower-precision, the cost does not fundamentally reduce the large speedups we are already seeing. As a result, we can make these speedups available to our solver without having to pay the price of potentially correctness-reducing overflows.

4.2 Using Micro-Sparsity

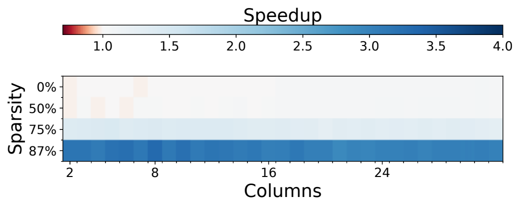
We now benchmark the full pivot iteration, which includes ignoring rows whose pivot column entries are equal to 0. The default strategy is to include a check in the hot loop that skips the appropriate rows. This is, however, inefficient because: a) A whole cache line (64 bytes in X86) is loaded even if only one element is needed. If the pivot entry is 0, all of the loaded data is useless. b) The additional branches in the hot loop complicate the code and hinder speculative execution.

We use micro-sparsity, described in Section 3.2, to decide which rows should be processed. This optimization has two effects: a) fewer instructions are executed and b) fewer cache lines are loaded. The microbenchmark results (Figure 12) show that the use of sparsity information in combination with the otherwise unchanged previous data structures is beneficial for sufficiently ($> 50\%$) sparse inputs. Our inputs are generally sparse (Figure 1c), and both in the hot and cold regimes, we observe speedups up to $3.5\times$ and $4\times$. Whenever the input sparsity is low, there is a $5 - 20\%$ performance hit. The speedups are higher with a cold cache because unnecessary cache lines are not loaded.

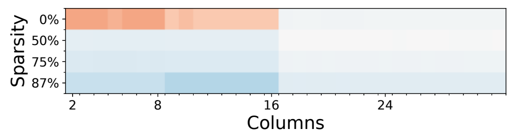
The combined performance gains of switching to native integers, vectorization, and micro-sparsity add up to $1,200\times$ in the hot cache regime and $750\times$ in the cold cache regime (Figure 13). While these performance numbers are impressive, we remind the reader that they describe the performance of a single loop measured in isolation and will not translate to the full simplex solver or even full applications.



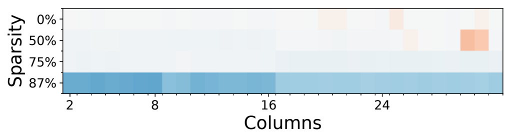
(a) 16-bit integers, hot cache



(b) 16-bit integers, cold cache

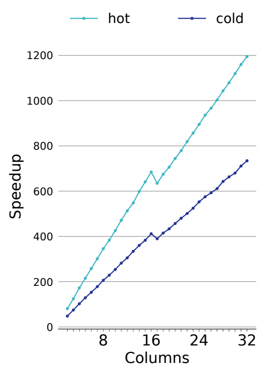


(c) 32-bit integers, hot cache

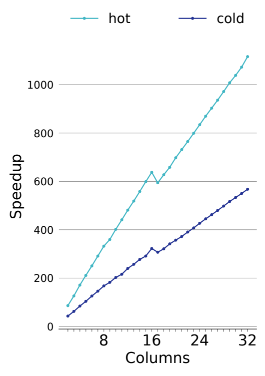


(d) 32-bit integers, cold cache

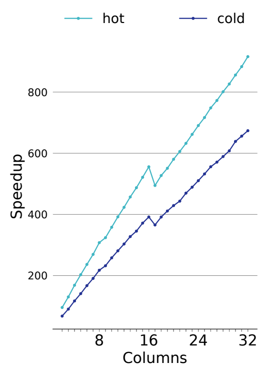
Fig. 12. Speedup of micro-sparsity over zero-check for a varying number of columns. The performance impact varies from no speedup to up to 15×. The cold cache regime benefits more because using micro-sparsity results in fewer memory transactions. For 64-bit numbers we see only minor differences as the cost of scalar arithmetic dominates.



(a) 50% Sparsity

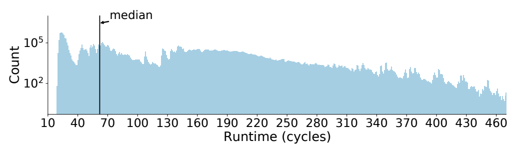


(b) 75% Sparsity

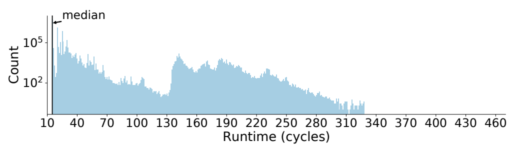


(c) 87.5% Sparsity

Fig. 13. Overall speedup of int16 types, vectorization, and sparsity vs. scalar (arithmetic precision).



(a) Original



(b) Optimized

Fig. 14. Runtime (in cycles) of GCD computation and normalization. The original version computes gcds using Euclid's algorithm while the optimized one uses a small integer specialized variant described in Section 3.3. The median runtime is 65 for the original and 18 for the optimized. The total runtime reduction is 3.4×.

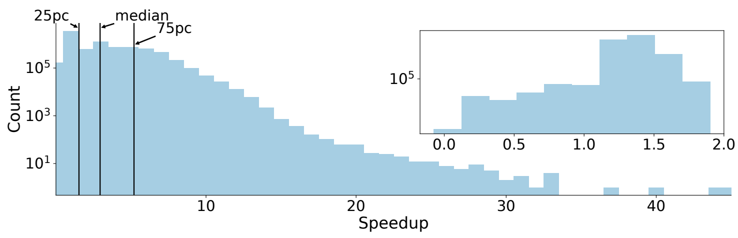


Fig. 15. Speedup distribution for specialized normalization. In more than 99.3% the specialized version is faster, the median speedup is 3.6×. The embedded (top-right) plot details the cases which are slower.

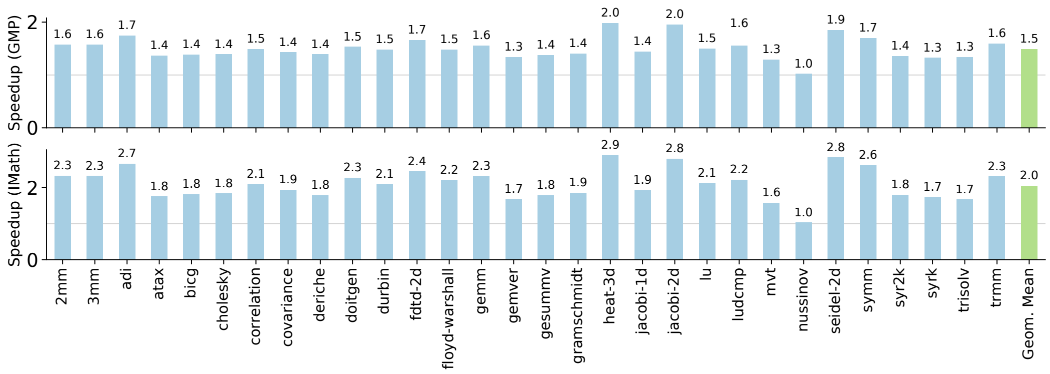


Fig. 16. Transprecision integers accelerate Polly on Pollybench by 1.5× (GMP) and 2.0× (IMath).

4.3 Specialization for Small Integers

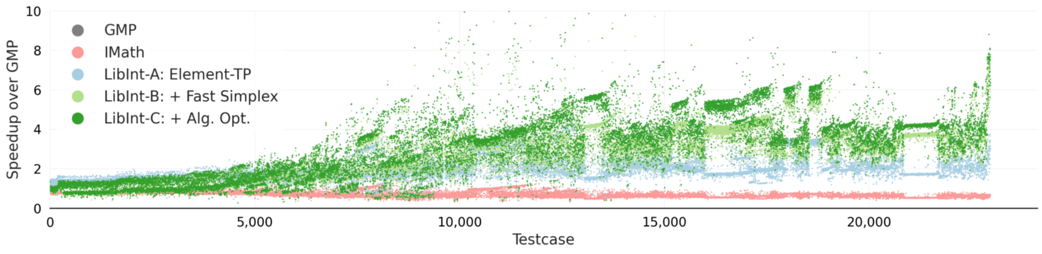
Normalization is used in `isl` to maintain small numbers in tableaux. It is done by dividing rows by their common gcd. For example, $\left[\frac{15}{3}, \frac{9}{3}, \frac{30}{3}\right]$ would be represented as $[3, 15, 9, 30]$, its gcd is 3 and the simplified row would be $\left[\frac{5}{1}, \frac{3}{1}, \frac{10}{1}\right]$ ($[1, 5, 3, 10]$). We optimized the normalization step with algorithm specialization for small integers (described in Section 3.3) while keeping all other data structures unchanged. We analyzed the optimized version using micro-benchmarks in combination with inputs extracted from our test cases (we traced all normalization inputs while executing coalescing test cases). The total runtime reduction between the original (Figure 14a) and optimized (Figure 14b) versions is 3.4×; the median speedup is 3.6× (Figure 15).

4.4 Performance Impact Transprecision Computing on Polyhedral Compilation

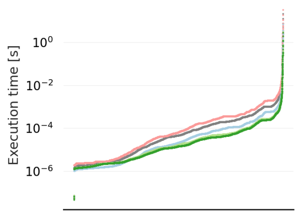
Polyhedral compilation (e.g., LLVM + Polly) relies heavily on arbitrary precision arithmetic. By default, `isl` uses GMP. However, due to license incompatibility LLVM and Polly rely on an alternative arbitrary precision arithmetic library, IMath [Fromberger 2019]. Element-wise transprecision integers implemented on top of IMath can significantly speed up polyhedral compilation (Figure 16): transprecision Polly, when used on Polybench, is 2.0× on average faster than Polly with IMath and 1.5× faster than Polly with GMP.

4.5 Performance Impact of Transprecision Simplex on Integer Set Coalescing

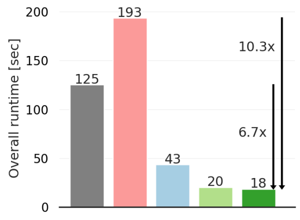
We assess the benefits of our work on linear programming for program analysis by measuring the performance benefits of optimizing one key operation on integer sets. Integer set coalescing is such a key operation (Section 2.1) that is important in Polly, HayStack, and PPCG (Section 4). For



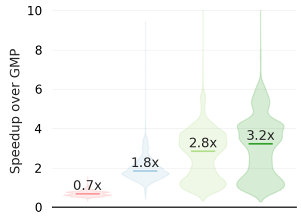
(a) Speedup distribution across individual test cases



(b) Execution time distribution



(c) Total execution time



(d) Average speedup

Fig. 17. Improved performance for coalescing when using LibInt instead of GMP (isl) or IMath (Polly). The accumulated execution time reduces with element-wise transprecision by 2.9× (GMP) and 4.5× (IMath). Our fast simplex and algorithmic improvements yield a further reduction to 6.7× (GMP) and 10.3× (IMath). The average speedup (median) of our optimizations is 3.2*x* (GMP) and 4.6*x* (IMath).

our analysis, we run the three tools on Polybench, the Polly test suite, and the hexagonal-tiling benchmarks. In these runs, we trace all integer sets that `isl` inspects when coalescing and obtain 22,961 test cases as a result. Coalesce is important across all components of a polyhedral compiler where it reduces, for example, code size during AST generation or simplifies the extracted SCoP models. Our tests cover the uses of coalescing in the following compilation tasks:

Tool	**SCoP Modeling**	**Dependences**	**Cache Modeling**	**Scheduling**	**Code Generation**
Polly	✓	✓	✗	✓	✓
PPCG	✓	✓	✗	✓	✓
HayStack	✓	✓	✓	✗	✗

Using our test cases, we compare different implementations. As a baseline we use `isl` with arbitrary precision arithmetic in its two main configurations. The first configuration uses GMP, which is the default arbitrary precision library in `isl`. The second configuration uses IMath, which is used in Polly due to its permissive LLVM-compatible license. We then compare this baseline with our transprecision-enabled variants of the coalesce algorithm. LibInt-A combines IMath with element-level transprecision arithmetic, LibInt-B also uses our fast simplex solver with matrix-level transprecision arithmetic, and LibInt-C adds additional algorithmic optimizations that reduce constraint canonicalization to favor effective SIMDization. All changes together make LibInt-C a coalesce implementation that has been optimized fully for transprecision computing.

Our results show that the speedups for the different test cases range from 0.25× to more than 10.0× (Figure 17a) compared to GMP and that execution times are short, ranging from microseconds to tens of seconds (Figure 17b). The total execution time of all benchmarks (Figure 17c) reduces from 2m06s (GMP) and 3m15s (IMath) to 43s (element-wise transprecision) and even 18s (all optimizations), which corresponds to a 6.7× (GMP) and 10.3× (IMath) speedup. The total execution time is a useful model for settings where many small problems are solved in sequence. Such settings are very typical in program analysis, where not individual queries, but the overall time of all queries – often across a full benchmark suite – determines the cost of an analysis. The average speedup (median) of element-wise transprecision (Figure 17d) is 1.8× (GMP) and 2.5× (IMath), which our fast simplex solver and the algorithmic optimizations increase to 3.2× (GMP) and 4.6× (IMath). While the large number of short-running tests impact the mean speedup, we continue to see a notable benefit of transprecision arithmetic.

Our results for one key operation demonstrate that we can effectively exploit modern SIMD hardware by leveraging low dimensionality and small integer values. However, it remains to be shown that similar speedups can be attained for more operators. The fact that most of the speedups we observe are not due to algorithmic changes makes us hopeful that our work provides a strong foundation for extending this work to a full Presburger library.

5 RELATED WORK

Mixed-precision iterative solvers Carson and Higham [2018] and Haidar et al. [2018a,b] take advantage of low-precision hardware, e.g., half-precision floats, to accelerate the solutions of linear systems. Such solvers can switch between low and high precisions multiple times, and they take advantage of the fact that different parts of the solver are less sensitive to rounding errors than others. Similarly to our approach, low precision arithmetic enables higher performance by reducing the number of necessary memory transactions and using wider vector units. A key difference between our work and mixed-precision iterative solvers is that we are operating on integers, and we have to handle overflows. When switching to low-precision floating-point types, the precision is reduced (fewer correct decimal digits), but this is not possible with integers: if the number cannot be represented in a given integer type, then the result is wrong and not less precise.

Mixed-precision machine learning. Training neural network models is very robust when it comes to floating-point errors [Gupta et al. 2015]. Mixed-precision hardware has been developed specifically for this use case [Markidis et al. 2018]. Similarly to iterative solvers, using lower precision does not result in abrupt catastrophic failures but only convergence can be affected.

Dynamic compilation approaches aim to increase the performance of object-oriented and dynamic languages. These approaches are typically implemented by an interpreter that collects run-time information, and a dynamic compiler that is subsequently invoked to produce efficient machine code using this information [Hölzle and Ungar 1994; Kotzmann et al. 2008]. Such runtimes implement speculative optimizations and, if a speculation fails, discard the compiled code using *deoptimization* [Hölzle et al. 1992] and continue execution in the interpreter. For example, in JavaScript, only a floating-point number type exists, but by using speculative optimizations, numbers can be represented as integers where possible [Würthinger et al. 2013]. As another example, an optimization that has been proposed for SmallTalk is *customization*, where methods are compiled for different receiver types, which are then dynamically selected [Chambers and Ungar 1989]. Similar to these approaches, our approach uses run-time feedback to improve performance. However, rather than improving language performance, we propose a new methodology for enabling the efficient support of linear programming in compilers.

Sub-polyhedral scheduling proposed by Upadrasta and Cohen [2013] exploits that many polyhedra in the context of polyhedral scheduling use two variables per inequality (TVPI). Our proposed optimization retains the full generality of the LP solver.

6 CONCLUSION

We present a new approach for optimizing linear programming that takes advantage of native integer types and vectorization to accelerate small-coefficient inputs as they commonly occur during program analysis. Thanks to both element and matrix-granularity transprecision, we can aggressively exploit small integer sizes while maintaining overflow-safety through automatic widening to larger types. We further exploit domain-specific information to develop a targeted micro-sparsity format and a SIMD-friendly small-value GCD algorithm to address key bottlenecks in our linear solver. By embedding our results in a state-of-the-art integer set library (isl), we show end-to-end performance improvements for the Polly loop optimizer by 2.0x (median) through minimal-invasive element-wise transprecision and a performance improvement of 3.2× (vs. GMP) and 4.6× (vs. IMath) for a single fully-specialized operation. We expect this work to serve as a first step towards a modern Presburger library based on transprecision arithmetic.

ACKNOWLEDGMENTS

This work is supported through the Swiss National Science Foundation under the Ambizione programme (grant PZ00P2168016) as well as ARM Ltd. and Xilinx Inc., in the context of Polly Labs.

REFERENCES

Riyadh Baghdadi, Ulysse Beaugnon, Albert Cohen, Tobias Grosser, Michael Kruse, Chandan Reddy, Sven Verdoolaege, Adam Betts, Alastair F Donaldson, Jeroen Ketema, et al. 2015. Pencil: A platform-neutral compute intermediate language for accelerator programming. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*. IEEE, 138–149. https://doi.org/10.1109/pact.2015.17

Roberto Bagnara, Patricia M Hill, and Enea Zaffanella. 2008. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Science of Computer Programming* 72, 1-2 (2008), 3–21. https://doi.org/10.1016/j.scico.2007.08.001

Wenlei Bao, Sriram Krishnamoorthy, Louis-Noel Pouchet, and P Sadayappan. 2017. Analytical modeling of cache behavior for affine programs. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 32. https://doi.org/10.1145/3158120

Dimitris Bertsimas and John N Tsitsiklis. 1997. *Introduction to linear optimization*. Vol. 6. Athena Scientific Belmont, MA.

Erin Carson and Nicholas J Higham. 2018. Accelerating the solution of linear systems by iterative refinement in three precisions. *SIAM Journal on Scientific Computing* 40, 2 (2018), A817–A847. https://doi.org/10.1137/17m1140819

C. Chambers and D. Ungar. 1989. Customization: Optimizing Compiler Technology for SELF, a Dynamically-typed Object-oriented Programming Language. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation (PLDI '89)*. 146–160. https://doi.org/10.1145/73141.74831

Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759* (2014).

Richard Crandall and Carl B Pomerance. 2006. *Prime numbers: a computational perspective*. Vol. 182. Springer Science & Business Media. https://doi.org/10.2307/3621190

David Detlefs, Greg Nelson, and James B Saxe. 2005. Simplify: a theorem prover for program checking. *Journal of the ACM (JACM)* 52, 3 (2005), 365–473. https://doi.org/10.1145/1066100.1066102

Will Dietz, Peng Li, John Regehr, and Vikram Adve. 2015. Understanding integer overflow in C/C++. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 25, 1 (2015), 2. https://doi.org/10.1109/icse.2012.6227142

Paul Feautrier. 1988. Parametric integer programming. *RAIRO-Operations Research* 22, 3 (1988), 243–268. https://doi.org/10.1051/ro/1988220302431

M. J. Fromberger. 2019. imath. https://github.com/creachadair/imath. Accessed: 2019-04-25.

Philip E Gill, Walter Murray, Michael A Saunders, and Margaret H Wright. 1984. Sparse matrix methods in optimization. *SIAM J. Sci. Statist. Comput.* 5, 3 (1984), 562–589. https://doi.org/10.21236/ada124397

Torbjrn Granlund et al. 2015. *GNU MP 6.0 Multiple precision arithmetic library*. Samurai Media Limited.

Tobias Grosser, Albert Cohen, Justin Holewinski, Ponuswamy Sadayappan, and Sven Verdoolaege. 2014. Hybrid hexagonal/classical tiling for GPUs. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*. 66–75. https://doi.org/10.1145/2544137.2544160

Tobias Grosser, Armin Groesslinger, and Christian Lengauer. 2012. Polly—performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters* 22, 04 (2012), 1250010. https://doi.org/10.1145/2925426.2926286

Tobias Grosser, Sven Verdoolaege, and Albert Cohen. 2015. Polyhedral AST generation is more than scanning polyhedra. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 37, 4 (2015), 1–50. https://doi.org/10.1145/2743016

Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. 2015. Deep learning with limited numerical precision. In *International Conference on Machine Learning*. 1737–1746.

Tobias Gysi, Tobias Grosser, Laurin Brandner, and Torsten Hoefler. 2019. A fast analytical model of fully associative caches. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 816–829. https://doi.org/10.1145/3314221.3314606

Christoph Haase. 2018. A survival guide to presburger arithmetic. *ACM SIGLOG News* 5, 3 (2018), 67–82. https://doi.org/10.1145/3242953.3242964

Azzam Haidar, Ahmad Abdelfattah, Mawussi Zounon, Panruo Wu, Srikara Pranesh, Stanimire Tomov, and Jack Dongarra. 2018a. The design of fast and energy-efficient linear solvers: On the potential of half-precision arithmetic and iterative refinement techniques. In *International Conference on Computational Science*. Springer, 586–600. https://doi.org/10.1007/978-3-319-93698-7_45

Azzam Haidar, Stanimire Tomov, Jack Dongarra, and Nicholas J Higham. 2018b. Harnessing GPU tensor cores for fast FP16 arithmetic to speed up mixed-precision iterative refinement solvers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. IEEE Press, 47. https://doi.org/10.1109/sc.2018.00050

Jared Hoberock. 2019. C++ Extensions for Parallelism Version 2 (Working Draft, N4808). Accessed: 2019-07-22.

Urs Hölzle, Craig Chambers, and David Ungar. 1992. Debugging Optimized Code with Dynamic Deoptimization. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation (PLDI '92)*. 32–43. https://doi.org/10.1145/143095.143114

Urs Hölzle and David Ungar. 1994. Optimizing Dynamically-dispatched Calls with Run-time Type Feedback. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation (PLDI '94)*. 326–336. https://doi.org/10.1145/178243.178478

Elias N Houstis, John R Rice, NP Chrisochoides, HC Karathanasis, PN Papochiou, EA Vavalis, and Ko Yang Wang. 1990. //ELLPACK: A Numerical Simulation Programming Environment for Parallel MIMD Machines. In *Proceedings of the 4th International Conference on Supercomputing (ICS '90)*. Association for Computing Machinery, New York, NY, USA, 96–107. https://doi.org/10.1145/77726.255144

Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. 2008. Design of the Java HotSpot Client Compiler for Java 6. *ACM Transactions on Architecture and Code Optimization (TACO)* 5, 1, Article 7 (May 2008), 32 pages. https://doi.org/10.1145/1369396.1370017

Moritz Kreutzer, Georg Hager, Gerhard Wellein, Holger Fehske, and Alan R Bishop. 2014. A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide SIMD units. *SIAM Journal on Scientific Computing* 36, 5 (2014), C401–C423. https://doi.org/10.1137/130930352

Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: feedback-directed and runtime optimization*. IEEE Computer Society, 75. https://doi.org/10.1109/cgo.2004.1281665

Vincent Loechner. 1999. PolyLib: A library for manipulating parameterized polyhedra.

László Lovász and Herbert E Scarf. 1992. The generalized basis reduction algorithm. *Mathematics of Operations Research* 17, 3 (1992), 751–764. https://doi.org/10.1287/moor.17.3.751

Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S Vetter. 2018. Nvidia tensor core programmability, performance & precision. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 522–531. https://doi.org/10.1109/ipdpsw.2018.00091

Alexander Monakov, Anton Lokhmotov, and Arutyun Avetisyan. 2010. Automatically tuning sparse matrix-vector multiplication for GPU architectures. In *International Conference on High-Performance Embedded Architectures and Compilers*. Springer, 111–125. https://doi.org/10.1007/978-3-642-11515-8_10

Charles Gregory Nelson. 1981. *Techniques for program verification*. Xerox. Palo Alto Research Center.

Philip Pfaffe, Tobias Grosser, and Martin Tillmann. 2019. Efficient hierarchical online-autotuning: a case study on polyhedral accelerator mapping. In *Proceedings of the ACM International Conference on Supercomputing*. 354–366. https://doi.org/10.1145/3330345.3330377

Louis-Noël Pouchet. 2012. Polybench: The polyhedral benchmark suite. (2012).

Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and John Cavazos. 2008. Iterative Optimization in the Polyhedral Model: Part II, Multidimensional Time. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*. Association for Computing Machinery, New York, NY, USA, 90–100. https://doi.org/10.1145/1375581.1375594

Louis-Noel Pouchet, Cedric Bastoul, Albert Cohen, and Nicolas Vasilache. 2007. Iterative optimization in the polyhedral model: Part I, one-dimensional time. In *International Symposium on Code Generation and Optimization (CGO'07)*. IEEE, 144–156. https://doi.org/10.1109/cgo.2007.21

Mojzesz Presburger. 1929. Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt in Comptes Rendus du I congres de Mathématiciens des Pays Slaves. *Slaves, Warsaw* (1929), 92–101.

Manuel Rigger, Stefan Marr, Bram Adams, and Hanspeter Mössenböck. 2019. Understanding GCC Builtins to Develop Better Tools. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019)*. 74–85. https://doi.org/10.1145/3338906.3338907

A Schriver. 1986. Theory of integer and linear programming.

Ramakrishna Upadrasta and Albert Cohen. 2013. Sub-Polyhedral Scheduling Using (Unit-)Two-Variable-per-Inequality Polyhedra. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '13)*. Association for Computing Machinery, New York, NY, USA, 483–496. https://doi.org/10.1145/2429069.2429127

Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730* (2018).

Sven Verdoolaege. 2010. isl: An integer set library for the polyhedral model. In *International Congress on Mathematical Software*. Springer, 299–302. https://doi.org/10.1007/978-3-642-15582-6_49

Sven Verdoolaege. 2015. Integer set coalescing. In *International Workshop on Polyhedral Compilation Techniques, Date: 2015/01/19-2015/01/19, Location: Amsterdam, The Netherlands*.

Sven Verdoolaege. 2020. Integer Set Library: Manual, Version 0.22.1. Retrieved from http://isl.gforge.inria.fr/manual.pdf on 31.08.2020.

Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, Jose Ignacio Gomez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral parallel code generation for CUDA. *ACM Transactions on Architecture and Code Optimization (TACO)* 9, 4 (2013), 54. https://doi.org/10.1145/2400682.2400713

Josef Weidendorfer. 2008. Sequential performance analysis with callgrind and kcachegrind. In *Tools for High Performance Computing*. Springer, 93–113. https://doi.org/10.1007/978-3-540-68564-7_7

Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to Rule Them All. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2013)*. 187–204. https://doi.org/10.1145/2509578.2509581

Zahari Zlatev. 1991. *Sparse Matrix Technique for Ordinary Differential Equations*. Springer Netherlands, 131–154. https://doi.org/10.1007/978-94-017-1116-6_8