Reuse Analysis via Affine Factorization

Ryan Job
ryan.job@colostate.edu
Colorado State University
Fort Collins, Colorado, USA

Sanjay Rajopadhye
sanjay.rajopadhye@colostate.edu
Colorado State University
Fort Collins, Colorado, USA

Abstract

Affine maps are a crucial structure within the polyhedral model of computation, essential for expressing programs in the model. Detecting values which are reused by a collection of affine maps within the same program expression can enable a number of optimization techniques: reducing the amount of storage needed, the time required for computing values, and even *value based* common sub-expression elimination, as well as more sophisticated optimizations like reduction simplification that reduce the asymptotic complexity. We present an algorithm to factorize a dependence map so that the maximum potential reuse is exposed.

CCS Concepts: **• Software and its engineering → Incremental compilers**; **Software performance**; **• Computing methodologies → Representation of mathematical functions**; **• Theory of computation → Program analysis**; **Abstraction**; **• General and reference → Performance**.

Keywords: polyhedral compilation, algorithmic complexity, program transformation

1 Introduction

The polyhedral model provides a mathematical framework for analyzing and transforming a domain-specific class of computations. *Affine control loops* are a proper subset of the model [2]. These programs are often written as a set of nested `for` loops where the upper and lower bounds are affine expressions using only compile-time constants, run-time constants (often referred to as *size parameters*), and the index variables of any enclosing affine control loops, but the model also include polyhedral equational programs.

Consider equation 1, which specifies a 3-dimensional array (or variable).

$$Y[i,j,k] = A[i+k] + B[i+j+k] \tag{1}$$

A straightforward way to compute `Y` is wrap this statement in a triply-nested `for` loop, one loop for each of the three dimensions of `Y`. Notice that there is a common `i+k` sub-expression used to index both `A` and `B`, and that neither `i` nor `k` are used in any other way. This allows us to rewrite equation 1 as shown in equation 2, where we introduced a 2-dimensional temporary variable `Z`.

$$\begin{aligned} Y[i,j,k] &= Z[i+k,j] \\ Z[x,y] &= A[x] + B[x+y] \end{aligned} \tag{2}$$

If the *right-hand side expression* of equation 1 appears within a triply-nested loop, one for each of the dimensions of `Y`, this factorization identifies common sub-expressions which could be optimized. Depending on how `Y` is used, this may reduce the time or space complexity of the loop program from $O(N^3)$ down to $O(N^2)$. We have identified one use case for this: reduction simplification via the Gautam and Rajopadhye algorithm [3]. This use case is especially interesting, as it can be combined with the existing implementation available in the AlphaZ system [17] and recent improvements [9] to fully automate the maximal simplification of reductions in the polyhedral model.

Our primary goal with this paper is to demonstrate an algorithm for factorizing affine maps such that the right factor has the smallest rank. Our secondary goal is to demonstrate its use for the identification of common sub-expressions, including its benefit to reduction simplification [3]. As part of this, we describe our plans to implement this as a pass within the AlphaZ system. Finally, we hope to promote discussion about other use cases for this analysis. Mathematically, our ideas are simple enough, and we originally expected that they would be well known in the literature. However, we have not found reference to such analyses in polyhedral compilation.

The remainder of this paper is organized as follows. In section 2, we review the Hermite normal form and its properties, as it forms the basis of the affine map factorization algorithm, which we next present in section 3, and describe how it can be implemented using the Integer Set Library (`isl`) (section 4). In section 5, we discuss how to use the algorithm to automate the Gautam and Rajopadhye reduction simplification algorithm. Section 6 describes our ongoing work to create a compiler pass in AlphaZ to leverage factorization and expose maximal reuse within an expression. Finally, we discuss related work in section 7 and conclude this paper with section 8.

2 Hermite Normal Form

The Hermite normal form (HNF) of integer matrices forms the core of our algorithm for automatically factorizing affine maps. HNF can be though of as an analog to reduced row echelon form, but for matrices of integers as opposed to real

IMPACT 2024, January 17, 2024, Munich, Germany
2024.

numbers. We will begin with a definition of the form and several of its useful properties [11, 14].

Definition 2.1. An $m \times n$ integer matrix H is in Hermite Normal Form if it satisfies the following conditions:

1. The leading coefficient of a row (the first non-zero element) occurs strictly to the right of the leading coefficient of the row above.
2. All leading coefficients are strictly positive.
3. All entries above a leading coefficient are non-negative and strictly less than the leading coefficient they are above.
4. Any rows of zeros are below all other rows.

Lemma 2.2. *Let M be an $m \times n$ integer matrix. There exists a unique $m \times n$ integer matrix H in Hermite Normal Form such that $H = U \cdot M$ for some unimodular matrix U [14].*

Definition 2.3. A unimodular matrix is a square, invertible, integer matrix with determinant of ±1.

From this information, we can derive an alternative formulation of the Hermite Normal Form. We will use this alternative as the basis for the factorization algorithm.

Theorem 2.4. *For every integer matrix M, there exists a matrix H of the same dimension in Hermite Normal Form and a unimodular matrix U such that $M = U^{-1} \cdot H$.*

Proof. By lemma 2.2, there exists H and U such that $H = U \cdot M$. By definition 2.3, U is invertible, meaning U^{-1} exists. Multiplying U^{-1} on the left of H produces the following:

$$U^{-1} \cdot H = U^{-1} \cdot (U \cdot M) = M$$

□

When we view these matrices as affine maps, H transforms the domain of the maps to an intermediate space, which U^{-1} then transforms into the range of M. If H has rows of zeros, all vectors in the intermediate space will end in the same number of zeros. Calculating and retaining these values is unnecessary, so we will eliminate them.

Theorem 2.5. *Let H and U be the matrices produced when calculating the HNF of some matrix M. If H contains one or more rows of zeros, these rows can be eliminated from the bottom of H, and the same number of columns can be eliminated from the right of U^{-1} while maintaining $M = U^{-1} \cdot H$.*

Proof. Hermite normal form restricts any rows of all 0's to be the bottom rows of H. Let m be the number of such rows. When U^{-1} multiplies H on the left, the elements in the rightmost m columns of U^{-1} will only ever be multiplied with the elements in the bottom m rows of H. Multiplication by 0 always results in 0. All multiplications are accumulated using addition, so these products of 0 do not contribute to the result. Thus, the bottom m rows of H and the rightmost m columns of U^{-1} can be dropped without affecting the result. □

3 Factorization of Affine Transformations

We now describe the algorithm used to factorize affine maps which all have the same domain. First, we discuss the augmented matrix representation used to represent affine maps, which handles both size parameters and translations. Second, we cover the intution behind the algorithm. Finally, we present the algorithm itself.

3.1 Augmented Matrix Representation

Affine maps consist of a linear transformation plus a translation. This is shown in equation 3.

$$y = Ax + b \tag{3}$$

Here, x and y are, respectively, vectors in the domain and range of the transformation, A represents the linear transformation, and b is the translation. We use the widely used augmented matrix representation, where the input vector x is augmented to include a constant 1, and the translation vector b is concatenated as a new column of the linear transformation matrix A. This is shown in equation 4

$$y = \left[\begin{array}{c|c} A & b \end{array}\right] \cdot \left[\begin{array}{c} x \\ 1 \end{array}\right] \tag{4}$$

As in most polyhedral tools, we define spaces using two kinds of variables: *index variables* and *size parameters*. Our algorithm treats them homogeneously. As a convention, we simply write the size parameters first and separately track how many there are.

3.2 Intuition

Recall the 3-dimensional array from equation 1, repeated below for convenience.

$$Y[i, j, k] = A[i + k] + B[i + j + k]$$

On the right-hand side of the equation, we have two affine maps: one to index the input A and the other to index the input B. Both have the same domain: the (i, j, k) space defined by Y. Using our augmented matrix representation, we can represent these maps M_A and M_B as shown in equation 5.

$$\begin{aligned} M_A &= \left[\begin{array}{ccc} 1 & 0 & 1 \end{array}\right] \\ M_B &= \left[\begin{array}{ccc} 1 & 1 & 1 \end{array}\right] \end{aligned} \tag{5}$$

We can factorize these maps such that they have a common right factor (which we call H) as shown in equation 6.

$$\begin{aligned} M_A &= \left[\begin{array}{cc} 1 & 0 \end{array}\right] \cdot H \\ M_B &= \left[\begin{array}{cc} 1 & 1 \end{array}\right] \cdot H \\ \text{where } H &= \left[\begin{array}{ccc} 1 & 0 & 1 \\ 0 & 1 & 0 \end{array}\right] \end{aligned} \tag{6}$$

This common right factor allows us to introduce a new 2-dimensional variable in the same manner as shown in equation 2, repeated below for convenience.

$$Y[i,j,k] = Z[i+k,j]$$
$$Z[x,y] = A[x] + B[x+y]$$

This can be done for any common factor of the maps, but our goal is to expose the maximum reuse. This means that the null-space or *kernel* of the factor must be the intersection of the null-spaces of the original maps. This holds if and only if equation 7 holds.

$$\ker(H) = \ker\left(\begin{bmatrix} M_A \\ M_B \end{bmatrix}\right) \tag{7}$$

Now, consider the row-style HNF. Per Theorem 2.4, we know that $M = U^{-1}H$. If we simply define M as being the matrix formed by concatenating M_A and M_B, as shown on the right-hand side of equation 7, we now have a way to compute the desired right-hand factor. The remaining left-hand factors can then be found as rows of U^{-1}, completing the factorization of the original affine maps.

One way to think of this factorization is that we find the smallest subspace containing all the ranges of the maps. The common right-hand factor, H, defines an affine map from the original domain of the original maps to this smallest subspace, referred to as the *intermediate space*. Then, we construct maps from this intermediate space into the ranges of each of the original maps using U^{-1}. Pulling out the common factor gives us the desired factorization.

3.3 Core Algorithm

We now describe how to factorize a set of affine maps (see Algorithm 1).

Let $T = [T_1, T_2, \cdots]$ be a list of affine maps, and M_i be the augmented matrix representing the map T_i. We first construct a single combined matrix M as shown in equation 8.

$$M = \begin{bmatrix} M_1 \\ M_2 \\ \vdots \end{bmatrix} \tag{8}$$

Next, we calculate the row-oriented HNF of M and call it H. This matrix represents the basis vectors of the intermediate space containing the ranges of all the original affine maps. We assume that this calculation also produces the matrix U^{-1} (represented by Q in Algorithm 1), described previously in theorem 2.4. We may use any existing algorithm for calculating these matrices, as there are many versions [4, 13, 14].

From lemma 2.2, H will have the same number of rows as M. However, if there is reuse among the original affine maps, M will be rank deficient, meaning H will have at least one row of zeros. Per theorem 2.4, we can simply drop these rows from H along with the associated columns from Q, removing any unnecessary dimensions from the intermediate space.

The final step in the factorization is to break Q into the individual maps from the intermediate space to each of the ranges of the original maps. Since the range of Q is the same as the range of M, we can simply break up Q into sub-matrices Q_i in the same way we constructed M from the matrices M_i, but in reverse. That is, if M_i is stored as rows j through k of M, we can form Q_i from rows j through k of Q. We then represent each of the maps T_i as the composition of H with Q_i. As H is common to all of the maps, it can be factored out, completing the factorization of the original affine maps.

Algorithm 1 Algorithm for factorizing affine maps

```
Input: A list matrices $M_i$ representing affine maps, all with
    the same $D$-dimensional domain.
Output: A common right factor $H$ and left factors $Q_i$.
 1: procedure FactorizeMaps($M_0 \dots M_n$)
 2:     $M \leftarrow$ Concatenate($M_0 \dots M_n$)
 3:     $H, U \leftarrow$ HermiteNormalForm($M$)
 4:     $Q \leftarrow$ MatrixInverse($U$)
 5:     for $r$ = Rows(H) $-1 \dots 0$ do
 6:         if IsRowOfZeros($H, r$) then
 7:             $H \leftarrow$ DropRow($H, r$)
 8:             $Q \leftarrow$ DropCol($Q, r$)
 9:         end if
10:     end for
11:     $start \leftarrow 0$
12:     for $i = 0 \dots n$ do
13:         $end \leftarrow start$ + Rows($M_i$)
14:         $Q_i \leftarrow$ GetRows($Q, start, end$)
15:         $start \leftarrow end$
16:     end for
17:     return $H, Q_0 \dots Q_n$
18: end procedure
```

4 Implementation with `isl`

The Integer Set Library (`isl`) is a commonly used tool for describing and manipulating integer sets in the polyhedral model, which includes support for affine expressions and matrix operations [15]. We have created a GitHub repository[1] which implements the affine map factorization algorithm using the `islpy` library [5]. This is presented as a Jupyter notebook with a running example, making it readily available to a wider community.

Our implementation only uses two of the structures available for representing affine expressions: the `Aff` class represents an affine expression with a single output, and the `MultiAff` class is an ordered collection of affine expressions which each represent one dimension of the multidimensional output. Each is used either where necessary or most convenient based on the API calls available.

The `Mat` class represents a matrix in `isl`. The API already contains a function to compute the Hermite normal form,

[1]https://github.com/ryanjob42/FactorizingAffineMaps

Table 1. Alpha Expressions syntax. Binary operators may be written in infix notation.

Expression	**Syntax**
Constants	Constant name or symbol
Operators	op $(\text{Expr}_1, \ldots, \text{Expr}_N)$
Case	case $\text{Expr}_1; \ldots; \text{Expr}_N$esac
Restriction	$\mathcal{D}'$: Expr
Dependence	Expr.$(z \rightarrow f(z))$
Reductions	reduce($\oplus$, $(z \rightarrow f_p(z))$, Expr)

including both the U matrix and its inverse (referred to as `Q` in the API). `isl` implements the column-oriented form, so we transposed all inputs and outputs to match the augmented matrix representation shown in Section 3.1.

For record keeping to ensure correctness, we used a few other classes. The `Space` and `LocalSpace` classes define the space in which an expression resides. We opted to use one versus the other based solely on the requirements of the API. The `Id` class represents an identifier for a dimension in a space. We used it to give each dimension a name so it can be tracked through the creation and modification of different objects.

5 Simplifying Reductions

AlphaZ is a tool for exploring program transformations and optimizations within the polyhedral model [17]. Programs are specified as equations in the Alpha language as defined by Mauras [7] and later extended by Le Verge to include reductions [6], modeled as polyhedral collections of values combined with an associative and commutative operator to produce collections of values. Table 1 summarizes the syntax of Alpha expressions.

Consider the simplest form of a reduction, expressed mathematically in equation 9, and in the Alpha code below it.

$$Y[f_p(z)] = \bigoplus_{z \in \mathcal{D}} X[f_r(z)] \tag{9}$$

```
Y = reduce(op, f_p, X[f_r(z)])
```

It uses two affine functions, a *projection* function f_p and a *read* function f_r. The reduction itself is performed over a polyhedral domain, $\mathcal{D}$, called its *body*. The read function maps points in $\mathcal{D}$ to points in the domain of X, and the projection function maps these values to points in the output variable Y.

AlphaZ includes an implementation of the reduction simplification transformation. Currently, it requires as input a *reuse vector* ρ, along which the *value of the reduction body expression* is invariant. Given this, the transformation rewrites the reduction to exploit the reuse, and such that the domains of the resulting equations asymptotically fewer points. For equation 9, ρ must belong to the null-space (or kernel) of $f_r(z)$.

When the reduction body is not as simple as in equation 9, but an arbitrary expression as specified by the rules of Table 1, there remains the issue of automatically determining the space of legal values of ρ. However, the affine factorization algorithm can be used on the affine dependence functions, producing a single function for $f_r(z)$. It is then straightforward to calculate its null-space and select any vector ρ in this space. Thus, this manual process can be automated, eliminating the need for human analysis of reuse in the reduction body.

5.1 Example Reduction

Consider the reduction in equation 10, which contains the same expression as shown previously in equation 1.

$$R[i,j] = \min_k \big(A[i+k] + B[i+j+k]\big) \tag{10}$$

Computing the entirety of R as written would be an $O(N^3)$ operation. However, we can apply affine factorization to easily find a reuse vector, then apply reduction simplification to compute R in $O(N^2)$ time.

Rewriting the indexing expressions in our augmented matrix representation (see Section 3.1) and concatenating them (see line 2 of Algorithm 1) produces matrix M as shown in equation 11.

$$M = \begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix} \tag{11}$$

Computing the HNF and finding the inverse of the unimodular matrix (see lines 3 and 4 of Algorithm 1) produces H and Q as shown in equation 12.

$$\begin{gathered} H = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \\ Q = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \\ M = Q \cdot H \end{gathered} \tag{12}$$

Since H does not contain any rows of zeros, no rows of H or columns of Q need to be dropped (see lines 5-10 of Algorithm 1). Since each of the original expressions only had one dimension, we can rewrite them using the appropriate row of Q (see lines 11-16 of Algorithm 1). This allows the reduction to be rewritten as shown in equation 13.

$$\begin{gathered} R[i,j] = \min_k \big(Z[i+k, j]\big) \\ \text{where } Z[x,y] = A[x] + B[x+y] \end{gathered} \tag{13}$$

This reduction is now in the simplified form as shown in equation 9. To perform reduction simplification, we must select a reuse vector in the null-space of the *read* function. In this case, the read function is the one represented by the matrix H, or equivalently, $(i,j,k \rightarrow i+k, j)$. The null-space contains any vector where $-i = k$ and $j = 0$, so we will semi-arbitrarily select $(1, 0, -1)$.

Table 2. A Subset of the AlphaZ Normalization Rules

Source Expression	**Replacement**	**Condition**
$e.f$	e	if $f(z) = z$
$(e.f_1).f_2$	$e.f$	where $f = f_1 \circ f_2$
$D_1 : (D_2 : e)$	$D : e$	where $D = D_1 \cap D_2$
$(D : e).f$	$D' : (e.f)$	where $D' = f^{-1}(D)$
$(e_1 \oplus e_2).f$	$(e_1.f) \oplus (e_2.f)$	
$(D : e_1) \oplus e_2$	$D : (e_1 \oplus e_2)$	
$e_1 \oplus (D : e_2)$	$D : (e_1 \oplus e_2)$	
$e \oplus (\text{case } e_1; \ldots; \text{ esac})$	$\text{case } (e \oplus e_1); \ldots; \text{ esac}$	

Notice that a column j of R can be implemented as a scan over the same column of Z. In this case, each point $R[i, j]$ can be computed as the maximum of $R[i+1, j]$ and $Z[i, j]$. The reuse vector selected indicates this reuse to AlphaZ. Thus, we can achieve a final $O(N^2)$ rewrite of the reduction as shown in equation 14.

$$R[i, j] = \begin{cases} Z[N, j] & \text{if } i = N \\ \min\big(R[i+1, j],\ Z[i, j]\big) & \text{otherwise} \end{cases} \tag{14}$$

6 Factorization Visitor in AlphaZ

AlphaZ implements a normalization transformation originally described by Mauras as a pass which rewrites expressions according to a set of normalization rules. Table 2 summarizes a subset of these rules [16]. This is implemented as a visitor over the Alpha abstract syntax tree.

Our goal with the affine factorization algorithm is to develop a new visitor similar to the normalization one. This would apply the factorization to pull a dependence function up through the abstract syntax tree. This dependence function will expose the maximum reuse available. With this, reduction simplification can be extended to automatically find vectors in the reuse space.

AlphaZ uses `isl` to represent and manipulate affine functions, similar to our existing Python implementation of the affine factorization algorithm (see Section 4). Although AlphaZ is written in Java and accesses `isl` via a foreign function interface, re-implementing the factorization algorithm will be relatively straightforward.

7 Related Work

Common subexpression elimination (CSE) is a compiler optimization which detects repeated expressions whose value does not change between executions [12]. At a high level, it identifies two instances of the same sub-expression where the input values do not change between the two instances. The value computed from the first instance is saved and reused for the second instance, eliminating the re-computation. The affine map factorization can be used as a new method for identifying common sub-expressions.

The global CSE implementation by John Cocke performs CSE at a global level by constructing and analyzing a graph representation of a program [1]. This can identify common sub-expressions and indicate whether they can be eliminated, or at least moved so it is computed a fewer number of times.

Identifying unnecessary computations can also be performed by analyzing the generated assembly. Monniaux and Six show that, by unrolling the first iteration of a loop, instructions which produce the same result can be identified [8]. Their optimization strategy identifies operations which only need to be done once in the first, unrolled iteration of the loop, then removed for all later iterations.

Pasko et al. demonstrated how multiplication by a constant scalar or matrix can also be optimized with CSE techniques [10]. Repeated bit patterns are identified and their partial results saved. The saved results are combined via a series of shift and accumulate operations. Applying this optimization can either allow faster hardware blocks to be synthesized, or fewer operations needed overall in the case of CPU-based matrix multiplication.

8 Conclusion

We have demonstrated how affine maps can be factorized to pull out a single common map with the maximum dimensionality kernel. We are working on using this to develop a *fully automatic* reduction simplifier. We expect that, other than this, there are other use cases for dependence factorization, such as *value based* common sub-expression identification and analysis. We hope to promote discussion about what these use cases may be.

References

[1] John Cocke. 1970. Global common subexpression elimination. *ACM SIGPLAN Notices*, 5, 7, (July 1, 1970), 20–24. DOI: 10.1145/390013.808480.

[2] Paul Feautrier. 1991. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20, 1, (Feb. 1, 1991), 23–53. DOI: 10.1007/BF01407931.

[3] Gautam and S. Rajopadhye. 2006. Simplifying reductions. In *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (POPL '06). Association for Computing Machinery, New York, NY, USA, (Jan. 11, 2006), 30–41. ISBN: 978-1-59593-027-9. DOI: 10.1145/1111037.1111041.

[4] Ravindran Kannan and Achim Bachem. 1979. Polynomial algorithms for computing the smith and hermite normal forms of an integer matrix. *SIAM Journal on Computing*, 8, 4, (Nov. 1979), 499–507. Publisher: Society for Industrial and Applied Mathematics. DOI: 10.1137/0208040.

[5] Andreas Kloeckner. [n. d.] Islpy 2023.2.5 documentation. Retrieved Oct. 30, 2023 from https://documen.tician.de/islpy/index.html.

[6] H. Le Verge. 1992. Reduction operators in alpha. In *PARLE '92 Parallel Architectures and Languages Europe* (Lecture Notes in Computer Science). Daniel Etiemble and Jean-Claude Syre, (Eds.) Springer, Berlin, Heidelberg, 397–411. ISBN: 978-3-540-47250-6. DOI: 10.1007/3-540-55599-4_101.

[7] Christophe Mauras. 1989. *Alpha : un langage equationnel pour la conception et la programmation d'architectures paralleles synchrones*. These de doctorat. Rennes 1, (Jan. 1, 1989). Retrieved Nov. 9, 2023 from https://www.theses.fr/1989REN10116.

[8] David Monniaux and Cyril Six. 2021. Simple, light, yet formally verified, global common subexpression elimination and loop-invariant code motion. In *Proceedings of the 22nd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems* (LCTES 2021). Association for Computing Machinery, New York, NY, USA, (June 22, 2021), 85–96. ISBN: 978-1-4503-8472-8. DOI: 10.1145/3461648.3463850.

[9] Louis Narmour, Tomofumi Yuki, and Sanjay Rajopadhye. 2023. Maximal simplification of polyhedral reductions. (Sept. 21, 2023). Retrieved Oct. 27, 2023 from http://arxiv.org/abs/2309.11826 arXiv: 2309.11826[cs].

[10] R. Pasko, P. Schaumont, V. Derudder, S. Vernalde, and D. Durackova. 1999. A new algorithm for elimination of common subexpressions. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18, 1, (Jan. 1999), 58–68. Conference Name: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. DOI: 10.1109/43.739059.

[11] Alexander Schrijver. 1998. *Theory of Linear and Integer Programming*. John Wiley & Sons, (June 11, 1998). 488 pp. ISBN: 978-0-471-98232-6.

[12] Y. N. Srikant Shankar Priti, (Ed.) The Compiler Design Handbook: Optimizations and Machine Code Generation. CRC Press, Boca Raton, (Sept. 24, 2002). 928 pp. ISBN: 978-0-429-18670-7. DOI: 10.1201/9781420040579.

[13] Arne Storjohann and George Labahn. 1996. Asymptotically fast computation of hermite normal forms of integer matrices. In *Proceedings of the 1996 international symposium on Symbolic and algebraic computation - ISSAC '96*. the 1996 international symposium. ACM Press, Zurich, Switzerland, 259–266. ISBN: 978-0-89791-796-4. DOI: 10.1145/236869.237083.

[14] Vasilios Tourloupis. 2013. Hermite normal forms and its cryptographic applications. *University of Wollongong Thesis Collection 1954-2016*, (Jan. 1, 2013). https://ro.uow.edu.au/theses/3788.

[15] Sven Verdoolaege. 2010. Isl: an integer set library for the polyhedral model. In *Mathematical Software – ICMS 2010* (Lecture Notes in Computer Science). Komei Fukuda, Joris van der Hoeven, Michael Joswig, and Nobuki Takayama, (Eds.) Springer, Berlin, Heidelberg, 299–302. ISBN: 978-3-642-15582-6. DOI: 10.1007/978-3-642-15582-6_49.

[16] Tomofumi Yuki, Vamshi Basupalli, Gautam Gupta, Guillaume Iooss, DaeGon Kim, Tanveer Pathan, Pradeep Srinivasa, Yun Zou, and Sanjay Rajopadhye. 2012. AlphaZ: a system for analysis, transformation, and code generation in the polyhedral equational model. https://www.cs.colostate.edu/TechReports/Reports/2012/tr12-101.pdf.

[17] Tomofumi Yuki, Gautam Gupta, DaeGon Kim, Tanveer Pathan, and Sanjay Rajopadhye. 2013. AlphaZ: a system for design space exploration in the polyhedral model. In *Languages and Compilers for Parallel Computing* (Lecture Notes in Computer Science). Hironori Kasahara and Keiji Kimura, (Eds.) Springer, Berlin, Heidelberg, 17–31. ISBN: 978-3-642-37658-0. DOI: 10.1007/978-3-642-37658-0_2.