Tile Size Selection for Optimized Memory Reuse in High-Level Synthesis

Junyi Liu, John Wickerson and George A. Constantinides
Department of Electrical and Electronic Engineering, Imperial College London, SW7 2AZ, United Kingdom
{junyi.liu13, j.wickerson, g.constantinides}@imperial.ac.uk

***Abstract*—High-level synthesis (HLS) is well capable of generating control and computation circuits for FPGA accelerators, but still requires sufficient human effort to tackle the challenge of memory and communication bottlenecks. One important approach for improving data locality is to apply loop tiling on memory-intensive loops. Loop tiling is a well-known compiler technique that partitions the iteration space of a loop nest into chunks (or 'tiles') whose associated data can fit into size-constrained fast memory. The size of the tiles, which can significantly affect the memory requirement, is usually determined by partial enumeration. In this paper, we propose an analytical methodology to select a tile size for optimized memory reuse in HLS. A parametric polyhedral model is introduced to capture memory usage analytically for arbitrary tile sizes. To determine the tile size for data reuse in constrained on-chip memory, an algorithm is then developed to optimize over this model, using non-linear solvers to minimize communication overhead. Experimental results on three representative loops show that, compared to random enumeration with the same time budget, our proposed method can produce tile sizes that lead to a 75% average reduction in communication overhead. A case study with real hardware prototyping also demonstrates the benefits of using the proposed tile size selection.**

I. Introduction

High-level synthesis (HLS) has been successful at improving the design productivity for many FPGA applications. It is well known that a computation-intensive loop can be significantly accelerated in FPGAs by using loop unrolling, pipelining [1] and memory partitioning [2] in modern HLS such as Xilinx Vivado HLS [3], Intel FPGA SDK for OpenCL [4] and LegUp [5]. However, when loops process a large amount of data, the communication overhead can become a performance bottleneck, limiting possible computational parallelism.

Current HLS has limited support for managing on-chip memory efficiently. Unlike CPUs, FPGA accelerators lack a comprehensive cache system integrated into the logic fabric. However, given a loop nest shown in Listing 1, loop tiling can be applied to execute partial loop iterations with limited on-chip memory. Generally, the hardware implementation of loop tiling can be abstracted as in Fig. 1, where all three loop levels are tiled, and the loop tile consists of the inner three loop dimensions. The interface is in charge of loading and storing the data set through various interconnects with different bandwidth specifications. Some buffers need to be instantiated in on-chip memory to keep the relevant data locally for the loop tile. When $\texttt{Si} = \texttt{Ni}$, $\texttt{Sj} = \texttt{Nj}$, and $\texttt{Sk} = \texttt{Nk}$, there is enough on-chip memory to store all the data for computation, so that the interface only needs to transfer the entire dataset

```
for(i=0; i<Ni: i++){
  for(j=0; j<Nj: j++){
    for(k=0; k<Nk: k++){
      B[i][j] += A[i][10*j+k]; }}}
```

Listing 1: A three-dimensional loop nest.

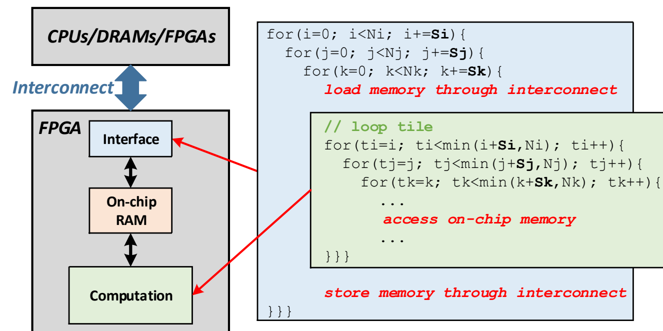


Fig. 1: Loop tiling and its mapping to a general FPGA accelerator system.

in and out once. When the original loop nest requires much more memory than is available on-chip, the tile size need to be reduced significantly. For example, an extreme case is that $\texttt{Si} = \texttt{Sj} = \texttt{Sk} = 1$, where the associated data needs to be transferred for every iteration of the loop body. When there are data reuse patterns in the memory accesses as shown in Listing 1, smaller on-chip memory can lead to more frequent data transfer and especially keeping less reused data on chip. Such communication overhead can consume a lot of time in the execution of the FPGA accelerator. Therefore, using the on-chip memory efficiently in hardware design involves the difficult problem of optimizing data reuse.

To exploit data locality, recent research [6], [7], [8] has suggested applying loop transformations to gather data-related iterations into a loop tile. The data elements accessed by these iterations are close to each other in terms of addressing distance, and therefore they can be packed into the on-chip memory. However, how to determine the size of loop tiles in order to maximize data reuse in HLS has not been well investigated. Different memory accesses in loop statements are related to different loop dimensions, which also leads to different data reuse patterns. For example, in Listing 1, the write access to the array `B` is not related to iterator `k`, and some read accesses to the array `A` may visit the same data elements among the loop iterations of `j`. Additionally, the

memory access functions can be even more complex after loop transformations. Given limited on-chip memory, it can be hard in practice to decide which loop dimensions should be assigned with more iterations for less communication overhead. Therefore, in this paper, we formally reason about how to determine the tile size instead of finding one by exhaustive or random enumeration.

The objective of this work is to provide a mathematical approach that quickly selects a tile size for optimized data reuse. We start from the symbolic analysis of data reuse patterns and the associated data layouts in on-chip memory. Our approach generates a parametric polyhedral model of the memory mapping from the iterations of a loop tile, where the tile sizes, *e.g.* `Si`, `Sj`, and `Sk` in Fig. 1, are treated as a set of parameters. The model can provide parametric expressions to calculate memory usage, which are then optimized over, to determine the tile size with optimization solvers. In particular, we make the following contributions:

- A symbolic approach to generate a parametric polyhedral model for calculating the memory usage of a tiled loop from its sizing parameters (Section III).
- An optimization algorithm that uses non-linear solvers to find, for a given amount of on-chip memory, the tile size for minimizing communication overhead (Section IV).
- An evaluation with three representative loop kernels, which compares our selection approach against time-constrained random enumeration (Section V-B).
- A case study of a loop nest from a convolutional neural network (CNN) with large memory usage, which evaluates our approach in real HLS-based FPGA design (Section V-C).

II. Related Work

Loop tiling is a widely used loop transformation that improves the data locality, and the loop performance can also be affected by the tile size selection. Bindhugula *et al.* [6] developed an automatic tool using polyhedral model to optimize the data locality of loop tiling on multi-core processors. In software compilation, tile size selection is focused on optimizing data reuse for the cache systems in CPUs [9], [10]. Following [6], Pouchet *et al.* [7] developed the polyhedral analysis of data reuse in loop tiling for HLS. To determine a tile size, they only enumerated 100 tile sizes with different power-of-two values on each loop dimensions. Unlike such limited enumeration, Peemen *et al.* [8] propose to construct a specific cost model considering both data reuse and loop transformation for a given loop. Then, they use a bounded enumeration with this model, but its search time can still grow quickly when the search space expands with the increase of available on-chip memory. In many recent works on designing FPGA accelerators, large-scale enumeration is widely used to select a tile size. Especially for CNN accelerators, exhaustive enumeration (as used in [11]) and random enumeration (as used in [12]) are used for design space exploration. However, all these enumeration approaches are time consuming when there is a large space of tile sizes to explore.

III. Symbolic Analysis of Memory Mapping

As shown in Fig. 1, the FPGA accelerator could spend a significant amount of time to transferring data due to limited on-chip memory. It is worthwhile evaluating the size of on-chip memory required from the symbolic loop tile size so that we can further understand how data reuse is affected by tile size selection. In this section, we start by analysing memory mapping used for the local buffers in an FPGA accelerator. A symbolic approach is introduced to construct a modular memory mapping [13] from the array accesses in a given loop. This analysis enables the calculation of the memory usage of a loop tile with the tile size as parameters.

A. Modular Memory Mapping

For a given array access in a loop nest, after loop tiling, the on-chip memory contains – at any given point in time – only some of the data elements from the original array. An intuitive memory mapping is to use the original array index function to link loop iterations to on-chip memory elements. Intuitively, these data elements can be stored with the same layout in the on-chip memory as in the off-chip memory. However, using the same data addressing scheme for all tile sizes can cost much more on-chip memory than necessary in some cases. For example, if we use the original form `A[ti][10*tj+tk]` as the read access in the loop tile of Fig. 1, the number of on-chip data elements $\#M = \texttt{Si} \times (10 \times (\texttt{Sj} - 1) + (\texttt{Sk} - 1) + 1)$. If, say, tile sizes $\texttt{Si} = 4$, $\texttt{Sj} = 2$, and $\texttt{Sk} = 2$ are selected, then 48 on-chip data elements need to be used. However, the alternative form `A[ti][tj][tk]` can be used instead for on-chip memory mapping, because $\texttt{Sk} < 10$. In this case, $\#M = \texttt{Si} \times \texttt{Sj} \times \texttt{Sk}$, which just requires 16 on-chip data elements. Such significant memory reduction is already demonstrated in a particular form of loop tiling as shown in [14], where a modular memory mapping is used to generate similar transformations. Therefore, our analysis is based on the construction of the following modular mapping from the original array accesses in the loop.

Definition 1 (Modular Mapping)**.** *Given a n-dimensional loop nest, a modular mapping is defined as a function of the form*

$$g(v) = Gv \bmod s, \tag{1}$$

where $G \in \mathbb{Z}^{n\times n}$, $s \in \mathbb{N}^n_{>0}$ and "mod" represents element-wise modulo. This function maps the n-dimensional iteration vector v to the index vector of an n-dimensional array. The size of each dimension of this array is determined by the corresponding value in s, and the memory size required by this mapping is at most $\prod_{i=1}^{n} s_i$.

Unfortunately, the technique developed in [14] required numerical constants for the loop bounds. This is inapplicable when we are trying to *determine* loop tile sizes, as these sizes are undetermined at compile time and therefore remain as symbolic constants. Therefore, in this paper, we extend the approach of [14] to the parametric case. The result is – rather than a number indicating the required memory size – an explicit *function* from tile size vector to memory requirement,

opening up the possibility to optimize this function directly. The main innovation of our methodology is the symbolic approach of generating the modular memory mapping.

B. Abstracting Parametric Sets from the Loop Tile

Given a n-dimensional loop tile (*e.g.*, the inner three loop dimensions shown in Fig. 1), the analysis starts with the iteration domain of this tile.

Definition 2 (Tile Iteration Domain)**.** *The tile iteration domain is a parametric set of iteration vectors from a rectangular tile, of the form*

$$\mathcal{D}^{p,b} = \left\{ v \in \mathbb{Z}^n \;\middle|\; p \leq v \leq p + b \right\},$$

where $p \in \mathbb{Z}^n$ represents the parametric start points of the loop tile in all loop dimensions, and $b \in \mathbb{N}^n$ represents the parametric size of the loop tile.

Also, we need to capture the array access patterns from the original loop before tiling.

Definition 3 (Array Indexing Function)**.** *Given a w-dimensional array access in the original loop, its indexing function is an affine expression of the form:*

$$f(v) = Av + c,$$

where $v \in \mathcal{D}^{p,b}$, $A \in \mathbb{Z}^{w \times n}$ is a coefficient matrix, and c is a integer constant vector.

For the example loop shown in Listing 1, we have $v = [\mathtt{ti}, \mathtt{tj}, \mathtt{tk}]^T$, and the array indexing function of the original read access `A[ti][10*tj+tk]` is $f(v) = \left[\begin{smallmatrix} 1 & 0 & 0 \\ 0 & 10 & 1 \end{smallmatrix}\right] v$.

Data reuse happens when a memory element is accessed by multiple loop iterations. For example, when $f(v) = f(v')$, iterations v and v' access the same memory element. We can further derive $A(v - v') = 0$, where $v - v'$ represents the iteration difference vector.

Definition 4 (Tile Iteration Difference Set)**.** *The tile iteration difference set is the parametric set of difference vectors $\delta = v - v'$ for $v, v' \in \mathcal{D}^{p,b}$, i.e.*

$$\mathcal{K}^b = \left\{ \delta \;\middle|\; -b \leq \delta \leq b \right\}.$$

We will later make use of the property that this parametric set is always a 0-symmetric polytope [13] independent of parameters p.

Remark. *This analysis is focused on rectangular tiling, which is widely used in compiler optimizations for both software and HLS. It is believed that the extension to other tiling shapes can also be supported by using the general Minkowski sum $\oplus$ to calculate $\mathcal{K}^b = \mathcal{D}^{p,b} \oplus (-\mathcal{D}^{p,b})$, but we leave this extension for future work.*

C. Analyzing the Basis Vectors of Data Reuse

With $f(v)$ from Definition 3 and $\delta \in \mathcal{K}^b$ from Definition 4, we can abstract the data reuse pattern from the set of equations: $A\delta = 0$. In linear algebraic terms, we are interested in that part of the kernel (null space) of the coefficient matrix A also within $\mathcal{K}^b$, *i.e.* $\ker(A) \cap \mathcal{K}^b$. It is convenient to represent this kernel as a linear combination of a set of $q < n$ basis vectors such that $A\delta = A\Lambda_{\text{reuse}} t = 0$, where $\Lambda_{\text{reuse}} = [\lambda_1, \lambda_2, \ldots, \lambda_q]$ is the reuse basis, and q is the nullity of A.

Since t is an arbitrary integer vector, we use the *Hermite normal form* [15] of A to determine Λ_{reuse} so that $A\Lambda_{\text{reuse}}$ is a zero matrix. Calculation of the Hermite normal form involves determining a unimodular matrix U_{reuse} such that

$$\begin{aligned} &[H_{\text{reuse}}\ 0] = AU_{\text{reuse}} \\ &\Rightarrow 0 = [H_{\text{reuse}}\ 0]\,[0\ I]^T = AU_{\text{reuse}}\,[0\ I]^T \\ &\Rightarrow \Lambda_{\text{reuse}} = U_{\text{reuse}}\,[0\ I]^T \end{aligned} \tag{2}$$

where H_{reuse} is a lower triangular matrix, and I is a $q \times q$ identity matrix. For example, the read access to array `A` in Listing 1 has its $\Lambda_{\text{reuse}} = [0, 1, -10]^T$ with $q = 1$, because

$$\underbrace{\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}}_{[H_{\text{reuse}}\ 0]} \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & 0 & 0 \\ 0 & 10 & 1 \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & -10 \end{bmatrix}}_{U_{\text{reuse}}} \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}.$$

To characterize the data reuse patterns by the vectors in $ker(A) \cap \mathcal{K}^b$, we use the following parametric constraints to capture potentially redundant basis vectors, generalising Liu *et al.*'s observations to the parametric case [14]:

$$\Lambda_{\text{reuse}} t \in \mathcal{K}^b \;\Rightarrow\; -b \leq \Lambda_{\text{reuse}} t \leq b.$$

With these inequalities, symbolic *Fourier-Motzkin elimination* [15] is applied to derive the projected constraints on each dimension of t, which has the affine form:

$$-\alpha_i^T b - \gamma_i \leq t_i \leq \alpha_i^T b + \gamma_i,$$

where $1 \leq i \leq q$, t_i corresponds to the i-th dimension of the subspace of Λ_{reuse}, and α_i, γ_i are constant vectors.

In the context of loop tiling, b has a upper bound corresponding to the maximal tile size. If $|t_i| < 1$ is proved to be always true, the ith basis vector can be removed because its integer coefficient t_i can only be zero. When $|t_i| < 1$ is not always true, we keep this basis vector but record the condition $d_i(b)$ under which it can be removed:

$$d_i(b) = \alpha_i^T b + \gamma_i < 1. \tag{3}$$

For example, the condition $d_1(b)$ of the Λ_{reuse} of the array `A` in Listing 1 is $b_3/10 < 1$. These various conditions will be applied to adapt the final memory mapping during the tile size selection described in Section IV.

D. Constructing the Kernel of the Modular Mapping

From the reuse basis of the given array access, we can generate a new basis Λ^b of the form $\Lambda^b = [\Lambda_{\text{reuse}}\ \Lambda^b_{\text{uniq}}] \in \mathbb{Z}^{n \times n}$. This basis represents the parametric kernel of the final modular mapping, where the original reuse patterns are preserved by the existence of the reuse basis Λ_{reuse}. The parametric basis vectors in Λ^b_{uniq} are required to be linearly independent of Λ_{reuse}. Moreover, they should not introduce additional reuse patterns into the modular mapping, *i.e.*

$$G(\Lambda^b_{\text{uniq}} t) \bmod s = 0 \wedge A(\Lambda^b_{\text{uniq}} t) \neq 0 \Rightarrow \Lambda^b_{\text{uniq}} t \notin \mathcal{K}^b. \tag{4}$$

These constraints ensure that no unwanted reuse patterns from Λ_{uniq} are in $\mathcal{K}^b$. Therefore, the basis vectors of Λ^b_{uniq}

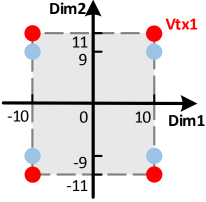


Fig. 2: Vertex projection for the array `A` in Listing 1 when $b = [10, 101, 101]^T$: the eight vertices in the original 3D space are projected into the 2D space of Λ_{orth}, and the symbolic coordinates of Vtx1 are $(b_1, 10 * b_2/101 + b_3/101)$.

correspond to the unique memory accesses in the loop tile, which also affect the memory size of the modular mapping.

Based on the method discussed in [13] and [14], we derive Λ^b_{uniq} as a symbolic lattice in the orthogonal space of Λ_{reuse}. In linear algebra, the null space of Λ^T_{reuse} is orthogonal to the column space of Λ_{reuse}; in other words, the kernel of Λ^T_{reuse} is orthogonal to the basis vectors of Λ_{reuse}. Thus, this kernel, denoted by Λ_{orth} with $n-q$ basis vectors, is derived by the Hermite normal form of Λ^T_{reuse} as in (2). For example, the array `A` in Listing 1 has its $\Lambda_{\text{orth}} = \left[\begin{smallmatrix} 1 & 0 & 0 \\ 0 & 10 & 1 \end{smallmatrix}\right]^T$.

To satisfy the conditions in (4), we need to find a lattice L^b in the subspace corresponding to Λ_{orth}, so that $\Lambda^b_{\text{uniq}} = \Lambda_{\text{orth}} L^b$, and $\Lambda_{\text{orth}} L^b t \notin \mathcal{K}^b$ when $t \neq 0$. In geometric terms, this lattice should have no intersection with the polytope of $\mathcal{K}^b$ except the zero point. To characterize the shape of $\mathcal{K}^b$, we project the parametric vertices of $\mathcal{K}^b$ onto the subspace corresponding to Λ_{orth}. The vertices of $\mathcal{K}^b$ are represented by a matrix V^b where each column vector contains the coordinates of a vertex. Generally, they are obtained by the vertex enumeration of $\mathcal{K}^b$. In this analysis, since $\mathcal{K}^b$ can be geometrically represented as a parametric box, its vertices can be enumerated by the combination of upper or lower bound from each dimension. Then, we apply the following linear transformation to generate the projected parametric vertices:

$$V^b_{\text{orth}} = (\Lambda^T_{\text{orth}} \Lambda_{\text{orth}})^{-1} \Lambda^T_{\text{orth}} V^b, \tag{5}$$

where $V^b_{\text{orth}} \in \mathbb{Z}^{(n-q)\times N_{vtx}}, V^b \in \mathbb{Z}^{n \times N_{\text{vtx}}}$ and $N_{\text{vtx}} = 2^n$. For example, as illustrated in Fig. 2, we have $V^b_{orth} = \left[\begin{smallmatrix} 1 & 0 & 0 \\ 0 & 10/101 & 1/101 \end{smallmatrix}\right] V^b$ for the array `A` in Listing 1.

With the projected vertices, we need to generate a proper dense lattice, which can minimize the memory requirement affected by $\Lambda_{\text{orth}} L^b$. The reader is referred to [13] for more details about the theory and methods of constructing such a lattice. However, it is noteworthy that we restrict our symbolic analysis to select a rectangular lattice in this step, which makes L^b become an $(n-q) \times (n-q)$ parametric diagonal matrix. The diagonal property of L^b is exploited in generating the modular mapping in Section III-E. To make the lattice dense, L^b is determined by the symbolically maximal element in each row vector of $V^b_{orth} = [r^b_1, r^b_2, \ldots, r^b_{n-q}]^T$ in (5) such that

$$L^b = \text{diag}\left([l^b_1, l^b_2, \ldots, l^b_{n-q}]\right), \tag{6}$$

where $l^b_i = \lfloor \max(r^b_i) \rfloor + 1$ is the i-th diagonal element and $1 \leq i \leq n-q$. On the i-th basis vector of Λ_{orth}, l^b_i is the smallest integer value greater than the maximal projected vertex coordinate of $\mathcal{K}^b$. For example, we have $l^b_1 = b_1 + 1$ and $l^b_2 = \lfloor 10 * b_2/101 + b_3/101 \rfloor + 1$ in Fig. 2. Therefore, due to the 0-symmetry of the projected vertices, it is guaranteed that the selected dense lattice only intersects the parametric polytope of $\mathcal{K}^b$ at the zero point.

Finally, we can construct $\Lambda^b = [\Lambda_{\text{reuse}} \ \Lambda_{\text{orth}} L^b]$ as the kernel of the modular mapping of the given array access.

E. Calculating the Memory Size

According to Proposition 2 in [13], the modular mapping shown in (1) can be derived from the *Smith normal form* [15] of Λ^b such that

$$S = G \Lambda^b U, \tag{7}$$

where G and U are unimodular matrices, and $S = \text{diag}(s)$ is a diagonal matrix whose diagonal entries are the elements of s. However, in our symbolic analysis, we propose an alternative approach that avoids checking divisibility of multivariate polynomials from l^b when calculating the Smith normal form.

The first change is to use matrix diagonalization instead of the Smith normal form. In general, matrix diagonalization is the first step of computing the Smith normal form, as used in [16]. The second step is to make the diagonal elements of S satisfy that s_i divides s_{i+1} for $1 \leq i \leq n-1$. We note that these conditions of divisibility are not necessary for calculating the memory size of the modular mapping in (1). Therefore, it is safe to skip the second step of the Smith normal form. Since the proof of using (7) to construct modular mapping is also valid for matrix diagonalization, we can still use (7) to represent the form of matrix diagonalization.

The second change is to avoid the symbolic calculation of the greatest common divisor when diagonalizing Λ^b. The parameters of Λ^b are introduced by L^b, which can be separated out as $\Lambda^b = \Lambda_{\text{const}} \left[\begin{smallmatrix} I & 0 \\ 0 & L^b \end{smallmatrix}\right]$ where $\Lambda_{\text{const}} = [\Lambda_{\text{reuse}} \ \Lambda_{\text{orth}}]$ is constant. Then, we transform diagonalizing Λ_{const} into (7):

$$\begin{aligned} & X = G \Lambda_{\text{const}} Y \\ & \Rightarrow X \left[\begin{smallmatrix} I & 0 \\ 0 & L^b \end{smallmatrix}\right] = G \Lambda^b \left[\begin{smallmatrix} I & 0 \\ 0 & L^b \end{smallmatrix}\right]^{-1} Y \left[\begin{smallmatrix} I & 0 \\ 0 & L^b \end{smallmatrix}\right]. \end{aligned} \tag{8}$$

where Y is a unimodular matrix, and $X = diag(x)$. For example, the Λ_{const} of the read access to array `A` in Listing 1 has the following diagonalization:

$$\underbrace{\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 101 \end{bmatrix}}_{X} = \underbrace{\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 10 \\ -10 & 0 & 1 \end{bmatrix}}_{G} \underbrace{\begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & -10 \end{bmatrix}}_{\Lambda_{\text{const}}} \underbrace{\begin{bmatrix} 1 & 0 & -10 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}}_{Y}.$$

We can make (8) equivalent to (7) when $\left[\begin{smallmatrix} I & 0 \\ 0 & L^b \end{smallmatrix}\right]^{-1} Y \left[\begin{smallmatrix} I & 0 \\ 0 & L^b \end{smallmatrix}\right]$ is a unimodular integer matrix. Since the determinant of this matrix is already ± 1, we just need to ensure the entries are also integers. This leads to the following constraints of our symbolic approach:

$$\begin{cases} y_{i,j} \bmod l^b_{i-q} = 0, & \text{for all } 1 \leq j \leq q, \\ (y_{i,j} l^b_j) \bmod l^b_{i-q} = 0, & \text{for all } q < j \leq n, \end{cases} \tag{9}$$

for all $q+1 \leq i \leq n$. In practice, we note that all of these constraints can be easily satisfied due to simple column manipulation in the matrix diagonalization of Λ_{const}. For all of our benchmark loops in Section V, the $(q+1)$-th to n-th row vectors of Y are found to be equal to $[0 \ I^{(n-q)\times(n-q)}]$.

Based on (8), we can obtain the symbolic form of the modulus vector s in (1) from X and L^b in (6), so that

$$\det(S) = \det(X)\det(L^b) = \prod_{i=1}^{n} x_i \prod_{i=1}^{n-q} (\lfloor \max(r^b_i) \rfloor + 1) \tag{10}$$

Therefore, we can use $\det(X)\det(L^b)$ to calculate the symbolic memory size of the modular mapping for loop tiling, which is only related to the parametric tile size vector b.

For the generated modular mapping, the actual size of the i-th dimension of the memory access is $g_i b$, where g_i represents the i-th row vector of G. As mentioned in [14], some modulus s_i may be larger than $g_i b$, so $g_i b$ can be used to calculate a tighter memory size. In our symbolic analysis, we will use $g_i b$ instead of s_i for the memory size calculation only when $g_i b$ is proved to be always smaller than s_i for all tile sizes.

IV. Tile Size Selection Algorithm

In this section, we use the memory model generated in Section III to establish a communication cost model of the entire tiled loop. Then, an optimization algorithm is introduced to find the tile size leading to the minimum communication cost.

A. Model of Communication Cost

A given loop may have several read/write memory accesses to different arrays. Each of these array accesses can be related to a different subset of the loop iterators. For a given array access in the un-tiled loop, the position of the innermost loop dimension related to this access is denoted by pos. In this paper, the loop dimension is counted from outermost to innermost in a variable vector. For instance, b_1 represents the tile size on the outermost loop dimension, and b_n corresponds to the innermost. In the example loop shown in Listing 1, the pos of the access to the array `B` is equal to 2. The original loop size is denoted by a vector $b^{\max}$, which also represents the maximal tile size. In the same example loop, $b_1^{\max} = \texttt{Ni}$ is the maximal value of `Si` for the iterator `ti` in the loop tile. Then, we can use $\prod_{i=1}^{pos}((b_i^{\max}+1)/(b_i+1))$ to calculate the communication frequency, which represents number of loads from or stores to off-chip memory occurs. This calculation is accurate for the nested loops with constant and/or parametric trip counts. When a trip count varies with the iterators of its outer loops, the communication frequency will be over-approximated. For example, given a triangular iteration space, we count the number of tiles in its rectangular hull, which roughly doubles the exact number. Since the tile shape is already fixed, our later optimization of minimizing data transfer is still valid with such over approximation.

To calculate the volume of transferred data for a given array access k, we can multiply its on-chip memory size derived from (10) by its communication frequency. The total communication cost can be formulated as

$$Cost(b) = \sum_{k=1}^{N}\left(c_k \det(X_k)\det(L_k^b)\prod_{i=1}^{pos_k}\frac{b_i^{\max}+1}{b_i+1}\right), \quad (11)$$

where N is the number of distinct array accesses, and the coefficient $c_k \in \{1,2\}$ is determined by whether the k-th array access requires only a single load/store per iteration or both a load and a store. The data reuse between each pair of consecutive data transfers considered in [7], [8] can also be incorporated into our approach to further reduce the communication overhead, but we leave this for the future work.

Algorithm 1 `optSize()`: solve the optimization problem

Input: memory budget M^{bgt}, maximal tile sizes $b^{\max}$, additional constraints $f(b)$.

1: $b^{\text{GP}} \leftarrow$ `solveGP(`$\widetilde{Cost}(b), M^{\text{bgt}}, b^{\max}, f(b)$`)`
2: $b^{\text{pre}} \leftarrow$ `round(`b^{GP}`)`
3: **if** $\exists i.\, b_i^{GP} \le 1.5$ **then**
4: $\quad b^{\text{res}} \leftarrow$ `solveBNB(`$Tile(b), M^{\text{bgt}}, b^{\text{pre}}, b^{\max}, f(b)$`)`
5: **else**
6: $\quad b^{\text{res}} \leftarrow$ `solveBNB(`$\widetilde{Cost}(b), M^{\text{bgt}}, b^{\text{pre}}, b^{\max}, f(b)$`)`
7: **end if**
8: **return** b^{res}

Because of the non-trivial non-linearity introduced by $(b_i+1)^{-1}$ and $\det(L_k^b) = \prod_{i=1}^{n-q}(\lfloor\max(r_{k,i}^b)\rfloor+1)$ derived from (6), we propose to optimize over the following approximated total communication cost modified from (11):

$$\widetilde{Cost}(b) = \sum_{k=1}^{N}\left(c_k \det(X_k)\det({L'}_k^b)\prod_{i=1}^{pos_k}\frac{b_i^{\max}}{b_i}\right), \quad (12)$$

where $\det({L'}_k^b) = \prod_{i=1}^{n-q}(\max(r_{k,i}^b)+1)$. In this cost model, the floor operators in $\det(L_k^b)$ are removed, and the two "+1" in the calculation of the communication frequency are dropped. Thus, (12) is a posynomial and can be optimized as a geometric programming problem [17]. Moreover, the $\max()$ operation can be handled by the toolbox used for solving the optimization problem in Section IV-C.

B. Optimization Algorithm

The goal of our tile size selection is to reduce the overhead of communication as much as possible with a given on-chip memory budget M^{bgt} for all array accesses. This optimization can be realized by minimizing the cost model in (12) subject to a constrained on-chip memory size. Nevertheless, the limitation of using this model is that the value of some posynomial item may be dominated by $1/b_i$ when b_i is close to zeros, *i.e.* when only one iteration from the i-th loop dimension is executed in the loop tile.

As shown in Algorithm 1, we propose a two-stage optimization algorithm that uses geometric programming (GP) and branch-and-bound (BNB) solvers. In Line 1, the continuous GP problem below is first solved:

$$\begin{aligned}\min_{b\in\mathbb{R}^n}\quad & \widetilde{Cost}(b)\\ \text{s.t.}\quad & 0 \le b \le b^{\max},\ f(b) \le 0\ ,\\ & \textstyle\sum_{k=1}^{N}\det(X_k)\det({L'}_k^b) \le M^{\text{bgt}}\end{aligned} \quad (13)$$

The tile size b is constrained in a box so that each dimension of the loop tile can have at least one iteration and up to the number of iterations in the original loop. $f(b)$ represents the additional constraints that may be introduced for fixing or limiting the size of some loop tile dimensions. The last constraint ensures that on-chip memory usage is within the budget M^{bgt}. Then, the elements of the generated b^{GP} are rounded to their nearest integer values in Line 2 as b^{pre}, which

Algorithm 2 `OptSelect`: Select an optimized tile size

Input: memory budget M^{bgt}, maximal tile sizes $b^{\max}$, basis changing conditions $d(b)$, additional constraints $f(b)$.

1: $b^{\text{cur}} \leftarrow \texttt{optSize}(M^{budget}, b^{\max}, f(b))$
2: $cm^{\text{cur}} \leftarrow Cost(b^{\text{cur}})$
3: $d'(b) \leftarrow \texttt{checkCond}(d(b^{\text{cur}}) < 1))$
4: **while** $d'(b) \neq \emptyset$ **do**
5: $\quad d(b) \leftarrow d(b) \setminus d'(b)$
6: $\quad f(b) \leftarrow f(b) \cup d'(b)$
7: $\quad \texttt{modifyMemMap}(d'(b))$
8: $\quad b^{\text{new}} \leftarrow \texttt{optSize}(M^{\text{bgt}}, b^{\max}, f(b))$
9: $\quad cm^{\text{new}} \leftarrow Cost(b^{\text{new}})$
10: $\quad$ **if** $cm^{\text{new}} < cm^{\text{cur}}$ **then**
11: $\quad\quad b^{\text{cur}} \leftarrow b^{\text{new}}$
12: $\quad\quad cm^{\text{cur}} \leftarrow cm^{\text{new}}$
13: $\quad$ **end if**
14: $\quad d'(b) \leftarrow \texttt{checkCond}(d(b^{\text{cur}}) < 1))$
15: **end while**
16: **return** b^{cur}

is used as the start point for the second optimization stage. In this stage, we solve a mixed-integer optimization problem:

$$\begin{aligned} \min_{b\in\mathbb{Z}^n} \quad & Obj(b) \\ \text{s.t.} \quad & b^{\text{init}} = b^{\text{pre}}\ , 0 \le b \le b^{\max},\ f(b) \le 0\ , \\ & \textstyle\sum_{k=1}^{N} \det(X_k)\det(L'^{b}_{k}) \le M^{\text{bgt}} \end{aligned} \tag{14}$$

where $Obj(b)$ represents the objective function, and b^{init} represents the start point of the BNB solver. When any element of b^{GP} is found to be close to zero in Line 3, we propose to use the alternative objective function $Tile(b)$ (in Line 4) instead of $\widetilde{Cost}(b)$ (in Line 6) to address the limitation of (12). $Tile(b)$ is formulated to reflect the volume of tile iterations such that

$$Tile(b) = -\textstyle\sum_{i=1}^{n}(b_i^{m_i} + 1),$$

where m_i is the number of significant array accesses with $pos \ge i$ (*e.g.* $m_1 = m_2 = 2$ and $m_3 = 1$ in Listing 1). The power of m_i helps to differentiate the contribution of each loop dimension to the size of accessed memory. Starting from the initial point b^{pre}, this alternative optimization is to maximize the number of tile iterations with a given memory size.

The process of selecting an optimized tile size is abstracted in Algorithm 2. In Line 1, `optSize()` is used to produce a tile size based on the initial modular mapping generated by (8). Then, $Cost(b)$ in (11) is calculated as the communication cost to indicate the quality of the input tile size. In Line 4–15, we iteratively modify the modular mapping and regenerate a new tile size. The conditions from (3) are evaluated in $\texttt{checkCond}(d(b^{\text{cur}}) < 1)$ to find which basis vectors in Λ_{reuse} can be removed. The satisfied conditions stored in $d'(b)$ are removed from $d(b)$ and added into the additional constraints $f(b)$. In Line 7, according to $d'(b)$, a new modular mapping is constructed with the same symbolic analysis as in Sections III-D and III-E. Based on the new cost model and the associated constraints, a new optimization problem is solved to produce a new tile size in Line 8. After this, the communication costs of the current tile size b^{cur} and the new tile size b^{new} are compared, so that the size with lower cost will be kept for the next iteration. In the end, when no more basis vectors can be removed from Λ_{reuse} according to (3), we take b^{cur} as the final tile size selection.

C. Implementation

Both the symbolic analysis and the tile size selection algorithm have been implemented in MATLAB as a single toolbox, which is open-source in a public GitHub repository (https://github.com/Junyi-Liu/PolyTSS). The symbolic analysis of memory mapping uses the MATLAB Symbolic Math Toolbox. The selection algorithm uses YALMIP [18] to model and solve the optimization problems, which utilizes `fmincon` and `bnb` solvers from the MATLAB Optimization Toolbox.

V. Experiments

We have also implemented a time-constrained random enumeration for comparison. Two different approaches are run in MATLAB R2016a on a desktop PC with Intel i7-3770 CPU.

A. Benchmarks

MMM (matrix-matrix multiplication). We consider the loop calculating $C^{500\times300} = A^{500\times400}B^{400\times300}$. It is a 3-dimensional loop with three array accesses, and transfers at least 4.7×10^5 elements (include loading and storing).

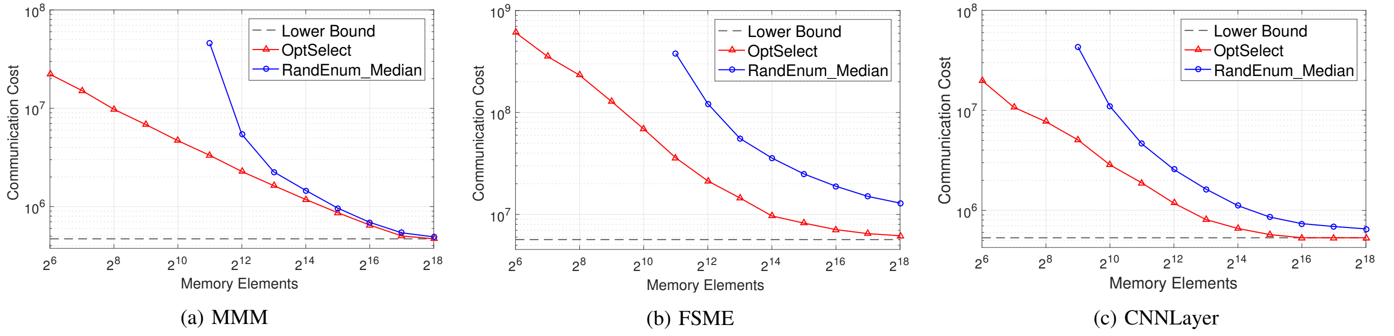
FSME (full search motion estimation). This is a critical loop of image processing in video coding. We use an 8-dimensional loop processing 1280x720 images from [8]. It contains three main array accesses and transfers at least 5.67×10^6 elements.

CNNLayer (convolutional neural network layer). We use the 6-dimensional loop of the third convolutional layer in AlexNet [19], which processes largest amount of data among all five convolutional layers. It has three array accesses and transfers at least 5.32×10^5 elements.

B. Comparison to Random Enumeration

For our benchmarks, it is infeasible to finish the exhaustive enumeration in a reasonable amount of time. Therefore, we choose to use a time-constrained random enumeration (`RandEnum`) for comparing tile size selection. It use the same model from the analysis in Section III to calculate memory size and communication cost without the feature of modifying the memory mapping. In our experiments, we run `RandEnum` with a time limit and select the best tile size from all assessed sizes. For each run of `RandEnum`, we shuffle the seed of the random number generator and use uniform distribution for enumerating tile sizes. For a given time limit, `RandEnum` is run for 100 times so that the statistics of all selected communication costs can be derived for evaluation.

1) Selection quality: In our experiments for comparing selection quality, we firstly run our approach (`OptSelect`) with a series of power-of-two memory budgets for all benchmarks. The summary of the execution time of `OptSelect` and `RandEnum` is shown in Table I. `OptSelect` can have varying execution time due to the iterative behaviour in Algorithm 2. To have a fair comparison, we give two approaches the



(a) MMM (b) FSME (c) CNNLayer

Fig. 3: Comparing the quality of tile size selection.

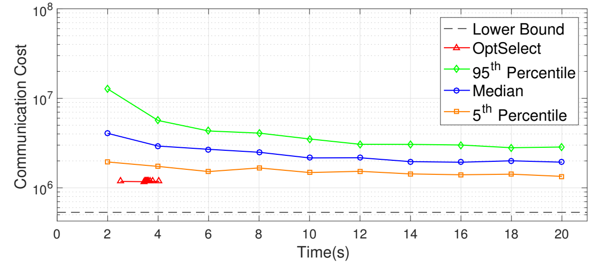
TABLE I. Experimental details of comparing selection quality.

Benchmark	`OptSelect`			`RandEnum`		
	Minimal Time (s)	Average Time (s)	Maximal Time (s)	Search Time (s)	Average Samples	Search Space
MMM	0.25	1.51	3.34	4	334	6.0×10^7
FSME	1.71	10.13	20.39	21	956	7.6×10^9
CNNLayer	0.55	1.94	5.69	6	302	7.5×10^7

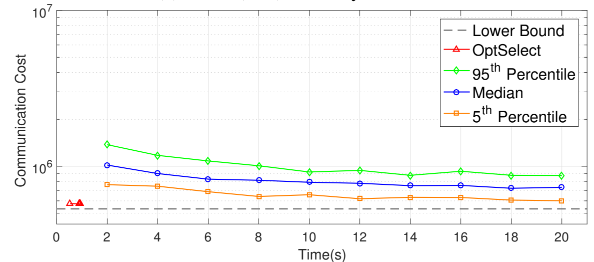
same amount of time to find a solution. For each benchmark, the time allotted to `RandEnum` is set to the ceiling of the maximal execution time of `OptSelect`.

In Fig. 3, the communication costs of the tile size selected by two approaches are plotted, where the cost values are calculated by (11). The lower bound represents the minimal amount of array elements that needs to be transferred. For each memory budget, the median cost from 100 different runs of `RandEnum` is compared to the solution selected by `OptSelect`. As shown in Table I, `RandEnum` only has time to assess a very small number of samples from the large search space. When there is a small amount of memory elements available, `RandEnum` can frequently fail to find any valid tile size in the time limit. It is shown in Fig. 3 that `RandEnum` can produce the selections satisfying memory constraints in over 50% runs only when the given memory budget is larger than 1024 elements. Moreover, `OptSelect` performs the selection much better than `RandEnum` in terms of minimizing the communication overhead. Here, we consider the cost gap as the decrease between the communication costs of `RandEnum` and `OptSelect`. As shown in Fig. 3, this cost gap becomes smaller with the increase of available memory elements, because large on-chip memories are more likely to allow more data reuse even with randomly selected tile sizes. Compared to `RandEnum`, our approach can reduce the communication overheads of **MMM**, **FSME** and **CNNLayer** by 51.65%, 88.45% and 84.27% on average.

2) Selection speed: We further evaluate the selection speed with the benchmark loop of **CNNLayer**. In these experiments, we run `RandEnum` for a wide range of time limits and adjust the execution time of `OptSelect` by changing the internal iteration limit of `solveBNB` in Algorithm 1. Specifically, the time limit of `RandEnum` is incremented from 2 to 20 seconds with a interval of 2 seconds, and the iteration limit of `solveBNB` is incremented from 0 to 300 with a interval



(a) 4096 (2^{12}) memory elements



(b) 32768 (2^{15}) memory elements

Fig. 4: Comparing the selection speed with **CNNLayer**.

of 30. In Fig. 4, for each time limit, the median, 5th and 95th percentile of the communication costs selected by `RandEnum` are plotted. In addition, `OptSelect` did not modify the memory mapping for the two chosen memory budgets. Thus, we can observe that the execution time of `OptSelect` only spans within a small range of time.

Given a small amount of memory as shown in Fig. 4a, `RandEnum` cannot provide steady tile size selection with respect to the communication cost. Its variation and median of the selected communication costs can become slightly smaller when time limit increases, but the reduction of communication overhead still remains to be as large as 53.85% at the time limit of 20 seconds. Then, we increase the memory budget to 32768 memory elements, which is to compare the selection speed when on-chip memory is relatively large. As shown In Fig. 4b, our approach still performs much better, with over 79% reduction of communication overhead, and its communication cost is close to the lower bound. It can also be observed that `RandEnum` requires a lot of time to produce a relatively good tile size with little variation.

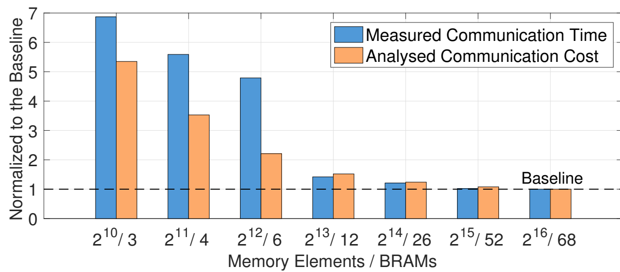


Fig. 5: Evaluating HLS-generated accelerators of **CNNLayer**.

C. Case Study

In this case study, Xilinx SDSoC 2016.1 is used as the HLS tool, and a ZedBoard with Xilinx Zynq SoC XC7Z020 is used for real hardware prototyping. We have implemented the FPGA accelerator of **CNNLayer** by HLS, which has the same hardware architecture as shown in Fig. 1. Both the accelerator and its communication network are clocked at 100 MHz. The loop tile is fully pipelined without unrolling and memory partitioning. Each datum is stored as a 16-bit unsigned integer number, and the data set is instantiated in the off-chip memory. To transfer the data in burst mode, the accelerator uses the master interfaces connected to the SoC bus through the high-performance AXI ports.

At first, we have synthesized a naive accelerator by HLS, which buffers all the data without loop tiling. This implementation requires 608 18Kb block RAMs (BRAMs), while there are only 280 BRAMs available on the chip. As shown in Fig. 3c, the accelerator with 2^{16} elements can already achieve the minimal communication cost, and therefore, we choose to use this memory budget as the baseline. In our experiments, we have prototyped the accelerators with the tile sizes selected for seven memory budgets ($\leq 2^{16}$ elements). Their execution times T_{exec} are measured with real hardware implementation. Then, we obtain the communication time with $T_{com} = T_{exec} - L_{tile} \times T_{clk}$, where L_{tile} represents the clock cycles of executing one loop tile (derived from HLS synthesis report) and $T_{clk} = 10\text{ns}$ is the clock period.

In Fig. 5, both the communication time T_{com} and the communication cost $Cost(b)$ from (11) are normalized to those of the accelerator with the baseline memory budget. When the cost overhead is under 50% (at 2^{13}), the overhead of the communication time are similar to those estimated by the cost model. When the memory budget is smaller than 2^{13} elements, a substantial increase of the communication time is observed. Since in these cases there are much frequent and small-amount accesses to the off-chip DDR memory, we believe that the gap of overhead is caused by the extra long latencies introduced in these access patterns. According to the BRAM usage shown in Fig. 5, it will be reasonable to use just 26 (instead of 608) BRAMs with the selected tile size to achieve fairly small communication overhead. Overall, this case study demonstrates that our tile size selection can be used at compile time to identify which memory budget is the minimum that covers most of data reuse and will not lead to significant communication overhead.

VI. Conclusion

In this paper, we introduce a mathematical method of selecting a tile size for optimized memory reuse. We introduce a symbolic analysis to calculate the on-chip memory size for loop tiling. And with the analyzed cost model, we also develop an optimization algorithm to select a tile size with the consideration of modifying memory mapping. The experimental results show that our approach can produce a tile size selection fast with high quality. Also, a case study demonstrates the advantages of using our selection approach in HLS for FPGA implementation.

Acknowledgments

The support of the EPSRC grants EP/I020357/1 and EP/K034448/1, the Royal Academy of Engineering, and Imagination Technologies is gratefully acknowledged. We thank the anonymous reviewers for their helpful suggestions.

References

[1] Z. Zhang and B. Liu, "SDC-based modulo scheduling for pipeline synthesis," in *ICCAD*, 2013.
[2] Y. Wang, P. Li, and J. Cong, "Theory and algorithm for generalized memory partitioning in high-level synthesis," in *FPGA*, 2014.
[3] Xilinx, *Vivado Design Suite User Guide: High-Level Synthesis*.
[4] Intel, *Intel FPGA SDK for OpenCL Programming Guide*.
[5] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "LegUp: High-level synthesis for FPGA-based processor/accelerator systems," in *FPGA*, 2011.
[6] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," *SIGPLAN Not.*, vol. 43, no. 6, pp. 101–113, Jun. 2008.
[7] L.-N. Pouchet, P. Zhang, P. Sadayappan, and J. Cong, "Polyhedral-based data reuse optimization for configurable computing," in *FPGA*, 2013.
[8] M. Peemen, B. Mesman, and H. Corporaal, "Inter-tile reuse optimization applied to bandwidth constrained embedded accelerators," in *DATE*, March 2015, pp. 169–174.
[9] S. Coleman and K. S. McKinley, "Tile size selection using cache organization and data layout," *SIGPLAN Not.*, vol. 30, no. 6, pp. 279–290, Jun. 1995.
[10] S. Mehta, G. Beeraka, and P.-C. Yew, "Tile size selection revisited," *ACM Trans. Archit. Code Optim.*, vol. 10, no. 4, pp. 35:1–35:27, Dec. 2013.
[11] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based accelerator design for deep convolutional neural networks," in *FPGA*, 2015.
[12] Y. Ma, Y. Cao, S. Vrudhula, and J.-s. Seo, "Optimizing loop operation and dataflow in FPGA acceleration of deep convolutional neural networks," in *FPGA*, 2017.
[13] A. Darte, R. Schreiber, and G. Villard, "Lattice-based memory allocation," *IEEE Transactions on Computers*, vol. 54, no. 10, pp. 1242–1257, Oct 2005.
[14] Q. Liu, G. A. Constantinides, K. Masselos, and P. Y. K. Cheung, "Automatic on-chip memory minimization for data reuse," in *FCCM*, 2007.
[15] A. Schrijver, *Theory of linear and integer programming*. John Wiley & Sons, 1998.
[16] T. Hu, *Integer programming and network flows*. Addison-Wesley Pub. Co., 1969.
[17] S. Boyd and L. Vandenberghe, *Convex optimization*. Cambridge university press, 2004.
[18] J. Lofberg, "YALMIP : a toolbox for modeling and optimization in MATLAB," in *2004 IEEE International Conference on Robotics and Automation*, Sept 2004, pp. 284–289.
[19] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25*. Curran Associates, Inc., 2012, pp. 1097–1105.