Traditional Limitations of polyhedral

1. The main limitation of the polyhedral model is known to be its restriction to statically predictable, (loop)-based program parts.

2. performance over a variety of targets. So far, these successes have been limited to static-control, regular loop nests. Time has come to address these challenges on a much wider class of programs.

3. like PoCC [29] or CHiLL [8] — target code parts that exactly fit the affine constraints of the model. Only loop nests with affine bounds and conditional expressions can be translated to a polyhedral representation. The reason behind

If we only think about high-level task, we probaly don't have to solve so many complexities Then, polyhedral still could be a powerful tool for high-level task scheduler.

Traditional operational semantical representation vs. algebric semantical representation

↑ polyhedral

I think this also can be used to explain the design of tensorflow's controlflow primitives

Since the very first compilers, the internal representation of programs has been in direct correspondance with their operational semantics. In such abstract syntaxes, each statement appears only once even if it is executed many times. This representation has severe limitations. First of all, it may limit the accuracy of program analysis. For instance, if a statement in a loop has some data dependence relation with another statement, it will consider both of them as single entities while the dependence relation may involve only very few of the dynamic iterations of these statements. This is particularly common in loop-based programs accessing arrays. Next, it may limit program transformation applicability. For instance, loop transformations operate on *individual statement iterations*. Lastly, it limits the expressiveness of program transformations: the most impactful loop nest transformations cannot be expressed as structural, incremental updates of the loop tree structure [19].

Why polyhedral?

The power of the polyhedral model is not to achieve exact data dependence analysis, but to implement compositions of complex transformations as a single algebraic operation, and to model these transformations in a convex optimization space.

The polyhedral model is a semantical, algebraic representation which combines analysis power, transformation expressiveness and flexibility to design sophisticated optimization heuristics. It was born with the seminal work of Karp, Miller and Winograd on systems of uniform recurrence equations [23]. The polyhedral model is closer to the program execution than operational/syntactic representations because it operates on individual statement iterations, or *statement instances*. It has been the basis for major advances in automatic optimization and parallelization of programs [15, 6, 26, 20, 5]. After decades of research, pro-

this is what we are looking for

It seems that polyhedral is pefectly suitable for the analysis of recurrent computation from the first day it was born.

✔ What is SCop ?

A SCoP is defined as a maximal set of consecutive statements, where loop bounds and conditionals are affine functions of the surrounding loop iterators and the parameters (constants whose values are unknown at compilation time). The it-

parameters (constants whose values are unknown at compilation time). The iteration domain of these loops can always be specified thanks to a set of linear inequalities defining a polyhedron. The term polyhedron will be used to denote a set of points in a $\mathbb{Z}^n$ vector space bounded by affine inequalities:

$$\mathcal{D} = \{\boldsymbol{x} \mid \boldsymbol{x} \in \mathbb{Z}^n, A\boldsymbol{x} + \boldsymbol{a} \geq \mathbf{0}\}$$

constant vector

iteration vector

constant matrix

An example.

The iteration space can be defined in affine inequalities.

```
for (i = 1; i <= n; i++)
  for (j = 1; j <= n; j++)
    if (i <= n + 2 - j)
      S(i,j);
```

(a) Surrounding Control of S

$$\mathcal{D}_S = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \middle| \begin{pmatrix} i \\ j \end{pmatrix} \in \mathbb{Z}^2, \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \\ -1 & -1 \end{bmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} -1 \\ n \\ -1 \\ n \\ n+2 \end{pmatrix} \geq \mathbf{0} \right\}$$

(b) Iteration Domain of S

$$\begin{cases} i \geq 1 \\ i \leq n \\ j \geq 1 \\ j \leq n \\ i + j \leq n + 2 \end{cases} \longrightarrow \begin{cases} i - 1 \geq 0 \\ -i + n \geq 0 \\ j - 1 \geq 0 \\ -j + n \geq 0 \\ -i - j + n + 2 \geq 0 \end{cases}$$

normalized representation

Throughout this paper, we assume that any constraint we are manipulating has been normalized. A normalized constraint in one in which all the coefficients are integers and the greatest common divisor of the coefficients (not including a_0) is 1.

Normalizing

If a constraint with rational but not integer coefficients is given ⟶ scale the coefficient to produce integer coefficients.

Limitations of traditional polyhedral representation

However, because of its strong mathematical constraints, any irregularity in the code splits the program into several smaller SCoPs. Furthermore, irregularities inside a loop nest will result in SCoPs with lower dimensionality (only the inner regular loops may be considered) [19]. For instance, let us consider the Outer Product Kernel shown in Figure 4(a): because of the irregular conditional, existing polyhedral frameworks can only consider the two *innermost* loops *separately*, or, to the contrary, consider the whole `if-else` statements as an atomic block, hence with a significantly reduced potential impact.

Relaxing the constraints:

The program model we target in this paper is general functions where the only control statements are `for` loops, `while` loops and `if` conditionals. This means function calls have to be inlined and `goto`, `continue` and `break` statements have been removed thanks to some preprocessing. To move from static control

not all recursive (tail recursion → for)

Reconstruct the program using polyhedral model is a three steps framework.

First, the Program Analysis phase aims at translating high level codes to their polyhedral representation and to provide data dependence analysis based on this representation. Second, some optimizing or parallelizing algorithm use the analysis to restructure the programs in the polyhedral model. This is the Program Transformation step. Lastly, the Code Generation step returns back from the polyhedral representation to a high level program. Targeting full functions requires revisiting the whole framework, from analysis to code generation.

Goal: simply and portably exploit parallism.

↓ the base step to this

Approach: data dependence analysis

↓

challenge: → exploit parallelism can change the order of memory operations

↓

constraint:

↳ resultant reordering does not change the sequential senatics of the program

✔ determine whether a dependence exists between two array references can be solved by an integer programming algorithm.

> A fundamental analysis step in a parallelizing compiler (as well as many other software tools) is data dependence analysis for arrays: deciding if two references to an array can refer to the same element and if so, under what conditions. This information is used to determine allowable program transformations and optimizations. For example, we can

1. data dependency problems are equivalent to deciding whether there exists an integer solution to a set of linear equalities an inequalities. a form of integer programming. → a NPC problem

2. dependence analysis is often structured as a decision problem: tests simply yes or no.

The above example can be formulated as an integer problem.

how to automatically get this formulation.

```
for i = 1 to 100 do
  for j = i to 100 do
    A[i, j+1] = A[100,j]
```

$$\begin{cases} 1 \le i \le j \le 100 & \text{read} \\ 1 \le i' \le j' \le 100 & \text{write} \\ i = 100 \\ j + 1 = j' \end{cases}$$

i, j : values of loop variables at the time the write is performed.

i', j' : values of loop variables at the time the read is performed.

What is "Omega test"

- The omega test determines whether there is an integer solution to an arbitrary set of linear equalities and inequalities. ← It is a "yes" or "no" test.

- Inputs: a set of linear equalities and inequalities

$$\begin{cases} \sum_{1 \le i \le n} a_i x_i = c \\ \sum_{1 \le i \le n} a_i x_i \ge c \end{cases}$$

↓ to simplify the representation we define $x_0 = 1$

$$\begin{cases} \sum_{0 \le i \le n} a_i x_i = 0 \\ \sum_{0 \le i \le n} a_i x_i \ge 0 \end{cases}$$

← affine constraints

$V = \{ i \mid 0 \le i \le n \}$ ← an integer set

V denote the set of indices of the variables being manipulated.

To normalize a constraint:

1. Normalize coefficients

① compute the greatest common divisor g of the coefficients $a_1, \ldots a_n$.

② Then divide all the coeffients by g.

a. The constraint is an equality constraint, and g does not evenly divide a_0

↓

The constraint is unsatisfiable.

b. The constraint is an inequality constraint

↓

take the floor when dividing

2. Eliminate all equality constraints, producing a new problem of inequality constraints.

↑

└ the new problem has integer solutions if and only if the original problem had integer solutions.

in the process. we might decide that the problem has no integer solutions regardless of inequality constraints.

increasingly relying on parallelism to improve performance.

↓ We're witnessing the introduction of massively parallel machines with multi-level parallelism.

The lack of data-flow information on array elements is the key limitation to exploit parallelism.

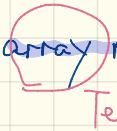
↓ inline

✔ Most of the parallelism found in scientific programs is present in loops.

how to define "deep learning" programs?

✔ The key to find parallelism in scientific programs is analyzing array references. which is called data dependence analysis.

✔ parallelize loops require compiler to analyze array reference.



What kind of information is required to parallel loops.

↳ We must be sure the parallelized version will retain the sequential semantics of the original version.

Import concepts.

1. Two array references is dependent.

> Two array references a and a′ are said to be *dependent* if any of the locations accessed by reference a are also accessed by reference a′ [53]. Otherwise, the two references are *independent*. The existence of a dependence implies that there exist two iterations of the loop, $\vec{i}$ and $\vec{i}'$, within the loop bounds, such that the location accessed by reference a in iteration $\vec{i}$ is the same as the location accessed by reference a′ in iteration $\vec{i}'$. We introduce the term *dependent iteration pairs* to describe the set of such pairs, $(\vec{i},\vec{i}')$.

two array references a and a′ acess the same memory location.

2. dependent iteration pairs.

within the loop bounds, there exist two iterations of the loop, $\vec{i}$ and $\vec{i}'$ that they access the same memory location.

① If all array reference pairs in a loop are independent,

→ we can run all the iterations of loop in parallel

② If some pairs are dependent, it might still be possible.

This is most favorable. → redundancy ← Transformer

RNN

Any dependent iteration pair such that $\vec{i} = \vec{i}'$ is a *loop-independent dependence*, while a dependent iteration pair such that $\vec{i} \neq \vec{i}'$ is a *loop-carried dependence* [55]. All the

- loop-independent dependence → data is not transferred between iterations
- loop-carried dependence → data is transferred between iterations.

All the iterations of a loop nesting can be run in parallel if and only if there are no loop-carried dependences between any two references in the loop

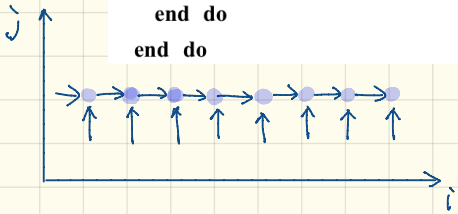
```
do i = 11 to 20
  a[i] = a[i]+3
end do
```

→ loop-independent dependence.

To parallelize a subset of the loop nestings, we need to know more than if there are loop-carried dependences. For example:

```
do i = 11 to 20
  do j = 11 to 20
    a[i][j] = a[i-1][j-1]+3
  end do
end do
```

← direction is missing.



1. all the dependences in this loop nesting are loop-carried

↓↓

we can not run all the iterations in parallel.

2. if we run loop i sequentially, we can run the j loop in parallel.

To fully exploit all the parallelism inherently present in a loop, we need to calculate all the dependent iteration pairs.

↓

infeasible and unnecessary for most optimizations.

↓

representation of distance vectors and direction vectors

3. distance vectors and direction vectors

distance vectors represent the vector difference between the two iteration elements in a dependent iteration pair.

direction vectors represent the sign of the distance vectors.

that allow us to summarize the set of dependent iteration pairs [55]. Distance vectors represent the vector difference between the two iteration elements in a dependent iteration pair. Two references are dependent with distance vector $\vec{d}$ if there exists a dependent iteration pair, $(\vec{i}, \vec{i'})$, such that $\vec{i'} - \vec{i} = \vec{d}$. In the above example, there is a dependence from the write to the read with distance (1,1). Direction vectors represent the sign of the distance vectors. We replace each component distance with its sign; $+$, $-$ or 0. We use a $*$ as a short hand notation to represent the direction when the sign of the distance is unknown or when there are dependent iteration pairs with all the possible directions. In the above example, there is a dependence from the write to the read with direction (+,+). It can be shown that since there is no dependence with direction (0,+), it is possible to run loop j in parallel as long as we sequentialize loop i. We use the term *dependence vector* to refer to either distance or direction vectors.

standard compiler systems require parallelization to preserve the order between (1) all write operations and all (2) read/write operations to the same locations.

- read → producer
- write → consumer

keep this in mind

The only dependences that inherently limit parallelism are dependences between a write operation and a read operation where the read uses a value produced by an earlier write.

there are loop-carried dependence in both loops
we can not run either loop in parallel without modification.

```
Loop1:
do i = 11 to 20
  a[i] = a[i-1]+3
end do
```

data is transferred across iterations
the value being written in iteration i
is being consumed in the next iteration i+1.
The loop is inherently sequential

```
Loop2:
do i = 11 to 20
  a[i] = a[i+1]+3
end do
```

the location being read in iteration i
is being overwritten in the next iteration i+1.

there is no data being transferred across iterations of the loop.

transfer into 2 parallelizable loops

```
do i = 12 to 21
  b[i] = a[i]
end do
do i = 11 to 20
  a[i] = b[i+1]+3
end do
```

distance vectors is sufficient to discover that loop1 is inherently sequential. but it can not discover parallelism in loop2.

Loop1. there is a dependence from read and write statement with a distance vector of (1).

Loop2. there is a dependence from read and write statement with a distance vector of (-1).

True dependence: For any pair of write and read references with a positive distance vector from the write to read i.e $\vec{i_r} - \vec{i_w} > 0$, the write operation occurs dynamically before the read.

Anti-dependence: Any pair for which the distance vector is negative.

Anti-dependence are artifacts of aliasing and can always be eliminated.

artifacts of aliasing and can always be eliminated. Note that if the distance vector is 0, the dependence is a true dependence if the write comes lexically before the read and an anti-dependence otherwise.

Only true dependences inherently limit parallelism, not all true dependences are inherently harmful.

```
do i = 11 to 20
  do j = 11 to 20
    a[j] = ...        write
  do j = 11 to 20
    ... = a[j]        read
```

the value read in iteration i was in fact written in the same iteration array.

the same location is write and then read

The array a is in fact being used as a temporary array.

there is no loop-carried data-flow dependences in this example.

↳ privatizing the array → give each processor its own copy of a.

data-flow dependence: a read reference to be data-flow dependent on a write reference if the write reference writes a value that is read by an instance of the read reference.

Data-flow dependences are a subset of true dependences.

Only data-flow dependences inherently limit parallelism.

A fact is: dependence vectors do not contain sufficient information to distinguish true dependences from data-flow dependences.

a new representation → data-flow dependence vectors

Why Affine constraints?

Ideally, we would always like to know the exact dependence and data-flow dependence vectors between any two references.

Unfortunately, even simpler problems are decidable at compile time in the general case.

1. static vs. dynamic dependences.

```
read (n)        ← n is unknown at the compile-time.
do i = 11 to 20
  a[i] = a[i − n]+3
end do
```

Even ignoring the static versus dynamic dependency issue, the problem of deciding whether two references are dependent can be made arbitrarily difficult.

```
if n > 2 then
  if a > 0 then
    if b > 0 then
      if c > 0 then
        x[a^n] = x[b^n + c^n]+3
      end if
    end if
  end if
end if
```

→ Fermat's last Theorem

✔ Affine Memory Disambiguation

Affine memory access is the traditional approach to restrict the data dependence domain to simplify the analysis. Pairs of references that cannot be proved independent in this domain are assumed dependent.

A memory disambiguator does not distinguish true dependences from data-flow dependences. A memory disambiguator calculates whether two references refer to overlapping locations; it does not calculate the flow of values through a program.

An affine data dependence solver only utilizes information from loop bounds that are integer linear functions of more outwardly nested loop indices and from array reference functions that are integer linear functions of the loop indices. This is frequently extended to allow constant symbolic terms as well.

Affine systems allow more general loops and array references than we have shown in the previous examples. They can handle multi-dimensional arrays and nested loops. The loops need not be rectangular; they can be triangular or trapezoidal. For example: