Vectorization in the Polyhedral Model

Louis-Noël Pouchet

pouchet@cse.ohio-state.edu

Dept. of Computer Science and Engineering, the Ohio State University

October 2010

888.11

Overview

Vectorization:

- Detection of parallel loops
- Vectorization in Pluto
- Vectorization in PoCC
- Alignment issues

Vectorization

Pre-transformation

- Exhibit inner-most parallel loops
- Ensure (if needed) stride-1 access
- Peel/shift for better alignment

Code generation

- Generate vector instruction for vectorizable loops
- Hardware considerations:
  - Speed of different instructions
  - Alignment constraints

Vectorization in the Polyhedral Model

Main consideration: pre-transformation

- Find a transformation (scheduling) for inner parallelism
- Complete the transformation for alignment

- Detection vs. transformations
  - Detect a loop is parallel, permutable, aligned, etc.
  - Transform: move parallel loops inwards, create parallel dimensions

Affine Scheduling

Definition (Affine schedule)

Given a statement S, a p-dimensional affine schedule Θ^R is an affine form on the outer loop iterators $\vec{x}_S$ and the global parameters $\vec{n}$. It is written:

$$\Theta^S(\vec{x}_S) = \mathbf{T}_S \begin{pmatrix} \vec{x}_S \\ \vec{n} \\ 1 \end{pmatrix}, \quad \mathbf{T}_S \in \mathbb{K}^{p \times dim(\vec{x}_S) + dim(\vec{n}) + 1}$$

- **A schedule assigns a timestamp to each executed instance of a statement**
- If T is a vector, then Θ is a one-dimensional schedule
- If T is a matrix, then Θ is a multidimensional schedule

Program Transformations

Original Schedule

```
 for (i = 0; i < n; ++i)
  for (j = 0; j < n; ++j){
S1: C[i][j] = 0;
    for (k = 0; k < n; ++k)
S2:   C[i][j] += A[i][k]*
                 B[k][j];
  }
```

$$\Theta^{S1}.\vec{x}_{S1} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} . \begin{pmatrix} \mathbf{i} \\ \mathbf{j} \\ \mathbf{n} \\ \mathbf{1} \end{pmatrix}$$

$$\Theta^{S2}.\vec{x}_{S2} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix} . \begin{pmatrix} \mathbf{i} \\ \mathbf{j} \\ \mathbf{k} \\ \mathbf{n} \\ \mathbf{1} \end{pmatrix}$$

```
for (i = 0; i < n; ++i)
 for (j = 0; j < n; ++j){
  C[i][j] = 0;
  for (k = 0; k < n; ++k)
   C[i][j] += A[i][k]*
              B[k][j];
 }
```

- Represent Static Control Parts (control flow and dependences must be statically computable)
- Use code generator (e.g. CLooG) to generate C code from polyhedral representation (provided iteration domains + schedules)

Program Transformations

Original Schedule

```
 for (i = 0; i < n; ++i)
  for (j = 0; j < n; ++j){
S1: C[i][j] = 0;
    for (k = 0; k < n; ++k)
S2:   C[i][j] += A[i][k]*
                 B[k][j];
  }
```

$$\Theta^{S1}.\vec{x}_{S1} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} . \begin{pmatrix} \mathbf{i} \\ \mathbf{j} \\ \mathbf{n} \\ \mathbf{1} \end{pmatrix}$$

$$\Theta^{S2}.\vec{x}_{S2} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix} . \begin{pmatrix} \mathbf{i} \\ \mathbf{j} \\ \mathbf{k} \\ \mathbf{n} \\ \mathbf{1} \end{pmatrix}$$

```
for (i = 0; i < n; ++i)
 for (j = 0; j < n; ++j){
  C[i][j] = 0;
  for (k = 0; k < n; ++k)
   C[i][j] += A[i][k]*
              B[k][j];
 }
```

- Represent Static Control Parts (control flow and dependences must be statically computable)
- Use code generator (e.g. CLooG) to generate C code from polyhedral representation (provided iteration domains + schedules)

Program Transformations

Original Schedule

```
  for (i = 0; i < n; ++i)
   for (j = 0; j < n; ++j){
S1: C[i][j] = 0;
    for (k = 0; k < n; ++k)
S2:   C[i][j] += A[i][k]*
                 B[k][j];

   }
```

$$\Theta^{S1}.\vec{x}_{S1} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} . \begin{pmatrix} \mathbf{i} \\ \mathbf{j} \\ \mathbf{n} \\ \mathbf{1} \end{pmatrix}$$

$$\Theta^{S2}.\vec{x}_{S2} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix} . \begin{pmatrix} \mathbf{i} \\ \mathbf{j} \\ \mathbf{k} \\ \mathbf{n} \\ \mathbf{1} \end{pmatrix}$$

```
for (i = 0; i < n; ++i)
 for (j = 0; j < n; ++j){
  C[i][j] = 0;
  for (k = 0; k < n; ++k)
   C[i][j] += A[i][k]*
              B[k][j];

 }
```

- Represent Static Control Parts (control flow and dependences must be statically computable)
- Use code generator (e.g. CLooG) to generate C code from polyhedral representation (provided iteration domains + schedules)

Program Transformations

Distribute loops

```
 for (i = 0; i < n; ++i)
  for (j = 0; j < n; ++j){
S1: C[i][j] = 0;
   for (k = 0; k < n; ++k)
S2:  C[i][j] += A[i][k]*
                B[k][j];

  }
```

$$\Theta^{S1}.\vec{x}_{S1} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} . \begin{pmatrix} \mathbf{i} \\ \mathbf{j} \\ \mathbf{n} \\ \mathbf{1} \end{pmatrix}$$

$$\Theta^{S2}.\vec{x}_{S2} = \begin{pmatrix} 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix} . \begin{pmatrix} \mathbf{i} \\ \mathbf{j} \\ \mathbf{k} \\ \mathbf{n} \\ \mathbf{1} \end{pmatrix}$$

```
for (i = 0; i < n; ++i)
 for (j = 0; j < n; ++j)
  C[i][j] = 0;
for (i = n; i < 2*n; ++i)
 for (j = 0; j < n; ++j)
  for (k = 0; k < n; ++k)
   C[i-n][j] += A[i-n][k]*
                B[k][j];
```

- **All instances of S1 are executed before the first S2 instance**

Program Transformations

Distribute loops + Interchange loops for S2

```
 for (i = 0; i < n; ++i)
  for (j = 0; j < n; ++j){
S1: C[i][j] = 0;
   for (k = 0; k < n; ++k)
S2:  C[i][j] += A[i][k]*
                B[k][j];
  }
```

$$\Theta^{S1}.\vec{x}_{S1} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} . \begin{pmatrix} \mathbf{i} \\ \mathbf{j} \\ \mathbf{n} \\ \mathbf{1} \end{pmatrix}$$

$$\Theta^{S2}.\vec{x}_{S2} = \begin{pmatrix} 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{pmatrix} . \begin{pmatrix} \mathbf{i} \\ \mathbf{j} \\ \mathbf{k} \\ \mathbf{n} \\ \mathbf{1} \end{pmatrix}$$

```
for (i = 0; i < n; ++i)
 for (j = 0; j < n; ++j)
  C[i][j] = 0;
for (k = n; k < 2*n; ++k)
 for (j = 0; j < n; ++j)
  for (i = 0; i < n; ++i)
   C[i][j] += A[i][k-n]*
              B[k-n][j];
```

- **The outer-most loop for S2 becomes k**

Program Transformations

Illegal schedule

```
 for (i = 0; i < n; ++i)
  for (j = 0; j < n; ++j){
S1: C[i][j] = 0;
   for (k = 0; k < n; ++k)
S2:  C[i][j] += A[i][k]*
                B[k][j];

  }
```

$$\Theta^{S1}.\vec{x}_{S1} = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} . \begin{pmatrix} \mathbf{i} \\ \mathbf{j} \\ \mathbf{n} \\ \mathbf{1} \end{pmatrix}$$

$$\Theta^{S2}.\vec{x}_{S2} = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{pmatrix} . \begin{pmatrix} \mathbf{i} \\ \mathbf{j} \\ \mathbf{k} \\ \mathbf{n} \\ \mathbf{1} \end{pmatrix}$$

```
for (k = 0; k < n; ++k)
 for (j = 0; j < n; ++j)
  for (i = 0; i < n; ++i)
   C[i][j] += A[i][k]*
              B[k][j];
for (i = n; i < 2*n; ++i)
 for (j = 0; j < n; ++j)
  C[i-n][j] = 0;
```

- **All instances of S1 are executed after the last S2 instance**

Program Transformations

A legal schedule

```
 for (i = 0; i < n; ++i)
  for (j = 0; j < n; ++j){
S1: C[i][j] = 0;
   for (k = 0; k < n; ++k)
S2:  C[i][j] += A[i][k]*
                B[k][j];

  }
```

$$\Theta^{S1}.\vec{x}_{S1} = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} . \begin{pmatrix} \mathbf{i} \\ \mathbf{j} \\ \mathbf{n} \\ \mathbf{1} \end{pmatrix}$$

$$\Theta^{S2}.\vec{x}_{S2} = \begin{pmatrix} 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{pmatrix} . \begin{pmatrix} \mathbf{i} \\ \mathbf{j} \\ \mathbf{k} \\ \mathbf{n} \\ \mathbf{1} \end{pmatrix}$$

```
for (i = n; i < 2*n; ++i)
 for (j = 0; j < n; ++j)
  C[i][j] = 0;
for (k= n+1; k<= 2*n; ++k)
 for (j = 0; j < n; ++j)
  for (i = 0; i < n; ++i)
   C[i][j] += A[i][k-n-1]*
              B[k-n-1][j];
```

- **Delay the S2 instances**
- Constraints must be expressed between Θ^{S1} and Θ^{S2}

Program Transformations

Implicit fine-grain parallelism

```
 for (i = 0; i < n; ++i)
  for (j = 0; j < n; ++j){
S1: C[i][j] = 0;
   for (k = 0; k < n; ++k)
S2:  C[i][j] += A[i][k]*
               B[k][j];
  }
```

$$\Theta^{S1}.\vec{x}_{S1} = \begin{pmatrix} 1 & 0 & 0 & 0 \end{pmatrix} . \begin{pmatrix} \mathbf{i} \\ \mathbf{j} \\ \mathbf{n} \\ \mathbf{1} \end{pmatrix}$$

$$\Theta^{S2}.\vec{x}_{S2} = \begin{pmatrix} 0 & 0 & 1 & 1 & 0 \end{pmatrix} . \begin{pmatrix} \mathbf{i} \\ \mathbf{j} \\ \mathbf{k} \\ \mathbf{n} \\ \mathbf{1} \end{pmatrix}$$

```
for (i = 0; i < n; ++i)
 pfor (j = 0; j < n; ++j)
  C[i][j] = 0;
for (k = n; k < 2*n; ++k)
 pfor (j = 0; j < n; ++j)
  pfor (i = 0; i < n; ++i)
   C[i][j] += A[i][k-n]*
              B[k-n][j];
```

- **Number of rows of Θ $\leftrightarrow$ number of outer-most sequential loops**

Program Transformations

Representing a schedule

```
 for (i = 0; i < n; ++i)
  for (j = 0; j < n; ++j){
S1: C[i][j] = 0;
    for (k = 0; k < n; ++k)
S2:   C[i][j] += A[i][k]*
                 B[k][j];

  }
```

$$\Theta^{S1}.\vec{x}_{S1} = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} . \begin{pmatrix} i \\ j \\ n \\ 1 \end{pmatrix}$$

$$\Theta^{S2}.\vec{x}_{S2} = \begin{pmatrix} 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{pmatrix} . \begin{pmatrix} i \\ j \\ k \\ n \\ 1 \end{pmatrix}$$

```
for (i = n; i < 2*n; ++i)
 for (j = 0; j < n; ++j)
  C[i][j] = 0;
for (k= n+1; k<= 2*n; ++k)
 for (j = 0; j < n; ++j)
  for (i = 0; i < n; ++i)
   C[i][j] += A[i][k-n-1]*
              B[k-n-1][j];
```

$$\Theta.\vec{x} = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} . \begin{pmatrix} i & j & i & j & k & n & n & 1 & 1 \end{pmatrix}^T$$

Program Transformations

Representing a schedule

```
 for (i = 0; i < n; ++i)
  for (j = 0; j < n; ++j){
S1: C[i][j] = 0;
    for (k = 0; k < n; ++k)
S2:   C[i][j] += A[i][k]*
                 B[k][j];

  }
```

$$\Theta^{S1}.\vec{x}_{S1} = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} . \begin{pmatrix} \mathbf{i} \\ \mathbf{j} \\ \mathbf{n} \\ \mathbf{1} \end{pmatrix}$$

$$\Theta^{S2}.\vec{x}_{S2} = \begin{pmatrix} 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{pmatrix} . \begin{pmatrix} \mathbf{i} \\ \mathbf{j} \\ \mathbf{k} \\ \mathbf{n} \\ \mathbf{1} \end{pmatrix}$$

```
for (i = n; i < 2*n; ++i)
 for (j = 0; j < n; ++j)
  C[i][j] = 0;
for (k= n+1; k<= 2*n; ++k)
 for (j = 0; j < n; ++j)
  for (i = 0; i < n; ++i)
   C[i][j] += A[i][k-n-1]*
              B[k-n-1][j];
```

$$\Theta.\vec{x} = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} . \begin{pmatrix} \underbrace{\mathbf{i} \ \mathbf{j}}_{} & \underbrace{\mathbf{i} \ \mathbf{j} \ \mathbf{k}}_{\vec{\imath}} & \underbrace{\mathbf{n} \ \mathbf{n}}_{\vec{p}} & \underbrace{\mathbf{1} \ \mathbf{1}}_{\mathbf{c}} \end{pmatrix}^T$$

Program Transformations

Representing a schedule

```
for (i = 0; i < n; ++i)
 for (j = 0; j < n; ++j){
S1: C[i][j] = 0;
   for (k = 0; k < n; ++k)
S2:  C[i][j] += A[i][k]*
                 B[k][j];

 }
```

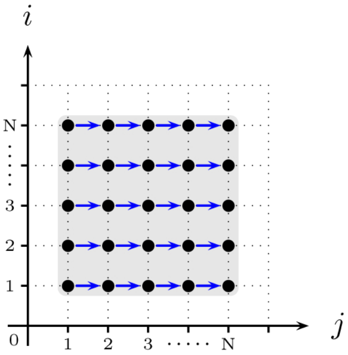
$$\Theta^{S1}.\vec{x}_{S1} = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} . \begin{pmatrix} i \\ j \\ n \\ 1 \end{pmatrix}$$

$$\Theta^{S2}.\vec{x}_{S2} = \begin{pmatrix} 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{pmatrix} . \begin{pmatrix} i \\ j \\ k \\ n \\ 1 \end{pmatrix}$$

```
for (i = n; i < 2*n; ++i)
 for (j = 0; j < n; ++j)
  C[i][j] = 0;
for (k= n+1; k<= 2*n; ++k)
 for (j = 0; j < n; ++j)
  for (i = 0; i < n; ++i)
   C[i][j] += A[i][k-n-1]*
              B[k-n-1][j];
```

	Transformation	Description
$\vec{i}$	`reversal`	Changes the direction in which a loop traverses its iteration range
	`skewing`	Makes the bounds of a given loop depend on an outer loop counter
	`interchange`	Exchanges two loops in a perfectly nested loop, a.k.a. `permutation`
$\vec{p}$	`fusion`	Fuses two loops, a.k.a. `jamming`
	`distribution`	Splits a single loop nest into many, a.k.a. `fission` or `splitting`
c	`peeling`	Extracts one iteration of a given loop
	`shifting`	Allows to reorder loops

Pictured Example



Example of 2 extended dependence graphs

Checking the Legality of a Schedule

Exercise: given the dependence polyhedra, check if a schedule is legal

$$\mathcal{D}_1 : \begin{bmatrix} 1 & 1 & 0 & 0 & -1 \\ 1 & -1 & 0 & 1 & -1 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 1 & -1 \\ 0 & 1 & -1 & 0 & 1 \\ 1 & -1 & 1 & 0 & -1 \end{bmatrix} . \begin{pmatrix} eq \\ i_S \\ i'_S \\ n \\ 1 \end{pmatrix} \qquad \mathcal{D}_2 : \begin{bmatrix} 1 & 1 & 0 & 0 & -1 \\ 1 & -1 & 0 & 1 & -1 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 1 & -1 \\ 0 & 1 & -1 & 0 & 1 \\ 1 & -1 & 1 & 0 & -1 \end{bmatrix} . \begin{pmatrix} eq \\ i_S \\ i'_S \\ n \\ 1 \end{pmatrix}$$

1. $\Theta = i$
2. $\Theta = -i$

Checking the Legality of a Schedule

Exercise: given the dependence polyhedra, check if a schedule is legal

$$\mathcal{D}_1 : \begin{bmatrix} 1 & 1 & 0 & 0 & -1 \\ 1 & -1 & 0 & 1 & -1 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 1 & -1 \\ 0 & 1 & -1 & 0 & 1 \\ 1 & -1 & 1 & 0 & -1 \end{bmatrix} \cdot \begin{pmatrix} eq \\ i_S \\ i'_S \\ n \\ 1 \end{pmatrix} \qquad \mathcal{D}_2 : \begin{bmatrix} 1 & 1 & 0 & 0 & -1 \\ 1 & -1 & 0 & 1 & -1 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 1 & -1 \\ 0 & 1 & -1 & 0 & 1 \\ 1 & -1 & 1 & 0 & -1 \end{bmatrix} \cdot \begin{pmatrix} eq \\ i_S \\ i'_S \\ n \\ 1 \end{pmatrix}$$

1. $\Theta = i$
2. $\Theta = -i$

Solution: check for the emptiness of the polyhedron

$$\mathcal{P} : \begin{bmatrix} \mathcal{D} \\ i_S \succ i'_S \end{bmatrix} \cdot \begin{pmatrix} i_S \\ i'_S \\ n \\ 1 \end{pmatrix}$$

where:

- $i_S \succ i'_S$ gets the consumer instances scheduled after the producer ones
- For $\Theta = -i$, it is $-i_S \succ -i'_S$, which is non-empty

Detecting Parallel Dimensions

Exercise:

Write an algorithm which detects if an inner-most loop is parallel

Limitation of Operating on Dimensions

- As soon as there is one non-parallel iteration, the dimension is not parallel
- Fusion/distribution impacts parallelism
- After fusion/distribution:
  - On the generated code, some inner loop may be parallel
  - The schedule for the program may not show the whole dimension as parallel

Exercise: Find a program where all schedule dimensions are sequential, but there are inner-most parallel loops

Pluto's Approach for Pre-Vectorization

1. Maximize the number of outer-most parallel/permutable **dimension**
2. An outer parallel dimension can be moved inwards
3. Proceed from the inner-most dimension, push inwards the "closest" parallel dimension

4. Missing considerations:
   - Alignment / stride-1 is not considered
   - Unable to model partially parallel dimensions (eg, those parallel only for some loop nests and not all)

PoCC's Approach for Pre-Vectorization

Very simple: decouple the problem

- Let Pluto transform the code for tiling, parallelism, etc.
- Generate the transformed code
- Re-analyze the transformed code, to extract its polyhedral representation
- Operate on each loop nest individually
  - Not limited to have a full dimension as parallel (local to a loop nest now)
  - Simple model to detect parallel loops with good alignment
  - Different cost models can be used
  - Possible pre-transformations for vectorization:
    - All of them!
    - However, limit to shift+peel+permute

Stride-1 Access

Definition (Data Distance Vector between two references)

Consider two access functions f_A^1 and f_A^2 to the same array A of dimension n. Let $\imath$ and $\imath\prime$ be two iterations of the innermost loop. The data distance vector is defined as an n-dimensional vector $\delta(\imath,\imath\prime)_{f_A^1,f_A^2} = f_A^1(\imath) - f_A^2(\imath\prime)$.

Definition (Stride-one memory access for an access function)

Consider an access function f_A surrounded by an innermost loop. It has stride-one access if $\forall\imath$, $\delta(\imath,\imath+1)_{f_A,f_A} = (0,\ldots,0,1)$.

Detecting Stride-1 Access

Exercise:

Write an algorithm which detects if an inner-most loop has stride-1 access for all memory references