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 $\Theta^R$ is an affine form on the outer loop iterators $\vec{x}_S$ and the global parameters $\vec{n}$. It is written:

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

- A schedule assigns a timestamp to each executed instance of a statement
- If $T$ is a vector, then $\Theta$ is a one-dimensional schedule
- If $T$ is a matrix, then $\Theta$ is a multidimensional schedule
Program Transformations

Original Schedule

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

Θ_{S1} \vec{x}_{S1} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} i \\ j \\ n \\ 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];
    }

Θ_{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} i \\ j \\ k \\ n \\ 1 \end{pmatrix}

- 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

\[
\Theta^{S_1} \vec{x}_{S_1} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 \end{pmatrix} \cdot \begin{pmatrix} i \\ j \\ n \\ 1 \end{pmatrix}
\]

\[
\Theta^{S_2} \vec{x}_{S_2} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} i \\ j \\ k \\ n \\ 1 \end{pmatrix}
\]

- 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];

Θ\text{S1}. \vec{x}_{S1} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} i \\ j \\ n \\ 1 \end{pmatrix}

for (i = 0; i < n; ++i)
  for (j = 0; j < n; ++j)
    C[i][j] = 0;
  for (i = n; i < 2\cdot 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];

Θ\text{S2}. \vec{x}_{S2} = \begin{pmatrix} 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} i \\ j \\ k \\ n \\ 1 \end{pmatrix}

for (i = 0; i < n; ++i)
  for (j = 0; j < n; ++j)
    C[i][j] = 0;
  for (i = n; i < 2\cdot 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];
```

\[
\Theta^{S_1} \vec{x}_{S_1} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} i \\ j \\ n \end{pmatrix}
\]

\[
\Theta^{S_2} \vec{x}_{S_2} = \begin{pmatrix} 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} i \\ j \\ k \\ n \end{pmatrix}
\]

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

▶ 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];
}

\[ \Theta^{S_1} x_{S_1} = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} i \\ j \\ n \end{pmatrix} \]
```

```
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 \( \Theta^{S_1} \) and \( \Theta^{S_2} \)
Program Transformations

Implicit fine-grain parallelism

\[ \Theta_{S1} \cdot \vec{x}_{S1} = \begin{pmatrix} 1 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} i \\ j \\ n \\ 1 \end{pmatrix} \]

\[ \Theta_{S2} \cdot \vec{x}_{S2} = \begin{pmatrix} 0 & 0 & 1 & 1 & 0 \end{pmatrix} \begin{pmatrix} i \\ j \\ k \\ n \\ 1 \end{pmatrix} \]

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

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

▶ Number of rows of \( \Theta \) ↔ 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];

Θ.\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} \cdot (i \ j \ i \ j \ k \ n \ n \ 1 \ 1)^T

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];
Program Transformations

Representing a schedule

\[ \Theta^{S_1} \bar{x}_{S_1} = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} i \\ j \\ n \\ 1 \end{pmatrix} \]

for \( i = 0; i < n; ++i \)
for \( j = 0; j < n; ++j \)
\[ S_1: C[i][j] = 0; \]
for \( k = 0; k < n; ++k \)
\[ S_2: C[i][j] += A[i][k] \cdot B[k][j]; \]

\[ \Theta^{S_2} \bar{x}_{S_2} = \begin{pmatrix} 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} i \\ j \\ k \\ n \\ 1 \end{pmatrix} \]

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

\[ \Theta.\bar{x} = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} i \\ j \\ i \\ j \\ k \\ n \\ n \\ 1 \\ 1 \end{pmatrix}^T \]
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];
  }

Θ_{S1} \vec{x}_{S1} = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} i \\ j \\ n \\ 1 \end{pmatrix}

for (i = n; i < 2*n; ++i)
  for (j = 0; j < n; ++j)
    C[i][j] = 0;

Θ_{S2} \vec{x}_{S2} = \begin{pmatrix} 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} i \\ j \\ k \\ n \\ 1 \end{pmatrix}

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];

<table>
<thead>
<tr>
<th>Transformation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\vec{i}$ reversal</td>
<td>Changes the direction in which a loop traverses its iteration range</td>
</tr>
<tr>
<td>skewing</td>
<td>Makes the bounds of a given loop depend on an outer loop counter</td>
</tr>
<tr>
<td>interchange</td>
<td>Exchanges two loops in a perfectly nested loop, a.k.a. permutation</td>
</tr>
<tr>
<td>$\vec{p}$ fusion</td>
<td>Fuses two loops, a.k.a. jamming</td>
</tr>
<tr>
<td>distribution</td>
<td>Splits a single loop nest into many, a.k.a. fission or splitting</td>
</tr>
<tr>
<td>$c$ peeling</td>
<td>Extracts one iteration of a given loop</td>
</tr>
<tr>
<td>shifting</td>
<td>Allows to reorder loops</td>
</tr>
</tbody>
</table>
Example of 2 extended dependence graphs
Checking the Legality of a Schedule

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

\[ D_1 : \begin{bmatrix} 1 & 1 & 0 & 0 & -1 \\
1 & -1 & 0 & 1 & -1 \\
1 & 0 & 1 & 0 & 1 \\
1 & 0 & -1 & 1 & -1 \\
0 & 1 & -1 & 0 & 1 \\
1 & -1 & 1 & 0 & -1 \end{bmatrix} \cdot \begin{pmatrix} eq \\ i_s \\ i'_s \\ n \\ 1 \end{pmatrix} \]

\[ D_2 : \begin{bmatrix} 1 & 1 & 0 & 0 & -1 \\
1 & -1 & 0 & 1 & -1 \\
1 & 0 & 1 & 0 & 1 \\
1 & 0 & -1 & 1 & -1 \\
0 & 1 & -1 & 0 & 1 \\
1 & -1 & 1 & 0 & -1 \end{bmatrix} \cdot \begin{pmatrix} eq \\ i_s \\ i'_s \\ n \\ 1 \end{pmatrix} \]

1. Θ = i
2. Θ = −i
Checking the Legality of a Schedule

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

\[ D_1 : \begin{bmatrix} 1 & 1 & 0 & 0 & -1 \\ 1 & -1 & 0 & 1 & -1 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 1 & -1 \\ 0 & 1 & -1 & 0 & 1 \\ 1 & -1 & 1 & 0 & -1 \end{bmatrix} \cdot \begin{pmatrix} eq \\ i_S \\ i'_S \\ n \\ 1 \end{pmatrix} \]
\[ D_2 : \begin{bmatrix} 1 & 1 & 0 & 0 & -1 \\ 1 & -1 & 0 & 1 & -1 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 1 & -1 \\ 0 & 1 & -1 & 0 & 1 \\ 1 & -1 & 1 & 0 & -1 \end{bmatrix} \cdot \begin{pmatrix} eq \\ i_S \\ i'_S \\ n \\ 1 \end{pmatrix} \]

1. \( \Theta = i \)
2. \( \Theta = -i \)

Solution: check for the emptiness of the polyhedron

\[ P : \begin{bmatrix} D \\ i_S > i'_S \end{bmatrix} \cdot \begin{pmatrix} i_S \\ i'_S \\ n \\ 1 \end{pmatrix} \]

where:

- \( i_S > i'_S \) gets the consumer instances scheduled after the producer ones
- For \( \Theta = -i \), it is \( -i_S > -i'_S \), 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
Pre-Vectorization in the Polyhedral Model:

Stride-1 Access

Definition (Data Distance Vector between two references)

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

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 $\forall i$, $\delta(i, i + 1)_{f_A, f_A} = (0, \ldots, 0, 1)$. 
Detecting Stride-1 Access

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