Transformation Selection for Good Vectorization

Louis-Noël Pouchet
pouchet@cse.ohio-state.edu

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

November 2010

888.11
The Problem of Efficient Vectorization

- A loop is SIMDizable if it is sync-free parallel
  - If it is not, how to transform the code to make the inner loop(s) SIMDizable?

- But how many vector instructions are required to load/store data?
  - **Stride** of accesses is critical
  - Best scenario: stride is \{-1, 0, 1\} for all accesses
Stride-1 Memory Access

- Stride-1 implies 1 vector load per 4 elements to be accessed
- Non stride-1 implies up to 4 vector load per 4 elements

Focus on inner-most loops:
- stride: "distance" in memory of data accessed by two consecutive iterations
- Array size must be constant (but may be parametric)
Example

Original code

```c
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];
```

Task 1: make the inner-most loop parallel
Example

Permute(k,i)

```
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];
```

Strides (assume all arrays are of size $N \times N$):

- **C**: $C[i][j]$ stride is $N$
- **A**: $A[i][k]$ stride is $N$
- **B**: $B[k][j]$ stride is 0
Example

Permute(k,i) + PermuteLayout(C) + PermuteLayout(A)

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

Strides (assume all arrays are of size $N \times N$):

- **C**: $C[i][j]$ stride is 1
- **A**: $A[i][k]$ stride is 1
- **B**: $B[k][j]$ stride is 0
Example

Permute(k,i) + Permute(i’,j)

Example

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

Strides (assume all arrays are of size $N \times N$):

- **C**: $C[i][j]$ stride is 1
- **A**: $A[i][k]$ stride is 0
- **B**: $B[k][j]$ stride is 1
Stride of Memory Accesses:

Polyhedral Compilation Classes

Stride-1 with Data Layout Permutation

- Simply transpose the array in memory
- Requires to transpose the access functions to this array

Pros:
- Always legal transformation (1-to-1 mapping)
- Allow to work individually on each array

Cons:
- All memory references to this array must be transposed in the entire program (may kill stride-1 somewhere else)
- Array declaration not necessarily accessible
Stride of Memory Accesses:

### Stride-1 with Loop Permutation

- Permute loops in a loop nest (aka interchange)
- The access function gets permuted to mirror the loop permutation change

**Pros:**
- Allow to work locally on an inner-most loop
- Flexible: different permutations possible for different loops

**Cons:**
- Not always legal!
- Spans at once all references in the inner-most loop
A (Slightly) More Complex Example

Original code

```
for (k = 0; k < N; ++k)
    for (i = 0; i < N; ++i)
        for (j = 0; j < N; ++j)
            C[i][j] += A[i][k] * B[k][j] / D[j][i];
    for (j = 0; j < N / 2; ++j)
        D[k][j] += F[k][j];
```

Strides (assume all arrays are of size $N \times N$):

- **C**: $C[i][j]$ stride is 1
- **A**: $A[i][k]$ stride is 0
- **B**: $B[k][j]$ stride is 1
- **D**: $D[j][i]$ stride is $N$
- **D**: $D[k][j]$ stride is 1
A (Slightly) More Complex Example

PermuteLayout(D)

Example

```c
for (k = 0; k < N; ++k)
    for (i = 0; i < N; ++i)
        for (j = 0; j < N; ++j)
            C[i][j] += A[i][k] * B[k][j] / D[i][j];
    for (j = 0; j < N / 5; ++j)
        D[j][k] += F[k][j];
```

Strides (assume all arrays are of size $N \times N$):

- **C**: $C[i][j]$ stride is 1
- **A**: $A[i][k]$ stride is 0
- **B**: $B[k][j]$ stride is 1
- **D**: $D[j][i]$ stride is 1
- **D**: $D[k][j]$ stride is $N$
Observations From the Example

- Is it profitable to permute the layout of $D$?
  - Maybe: there are 5 times less accesses to $D[j][k]$
  - Depends on the architecture / vector implementation

- Is this loop order the best?

- Is there any loop transformation which could help here?
  - What about loop distribution?
  - Impact of distribution-enabling transformations?

We need a systematic cost model!
Cost Model for Vectorization

Trifunovic et al., PACT’09

▶ Search space: loop permutations
▶ In a nutshell:
  ▶ To each possible permutation corresponds transformed access functions
  ▶ Compute a vectorization cost for all possibilities
  ▶ Select the best one, implement the corresponding permutation

▶ Cost model:
  ▶ Naive execution time estimate
  ▶ Non stride-1: needs multiple loads per vector register
  ▶ Stride-0: needs splat
  ▶ Stride-1: 1 load per vector register
Cost Estimation

Definition (Cost estimation for a polyhedral statement)

\[
\text{cost}(\mathcal{D}_S, \Theta^S) = \frac{|\mathcal{D}_S|}{VF} \cdot \sum c_{\text{vector_numerical_ops}} \\
+ \sum_{m \in \mathcal{W}_S} \left( c_a + \frac{|\mathcal{D}_S|}{VF} \cdot (c_{\text{vectstore}}) \right) \\
+ \sum_{m \in \mathcal{R}_S} \left( c_a + \frac{|\mathcal{D}_S|}{VF} \cdot (c_{\text{vectload}} + c_s) \right)
\]

Where \( VF \) is the vector length, and the different \( c \) are vector costs.
Cost of Non Stride-1 Loads

- It is a function of the stride of the access, noted $\delta_dv$
- Captured in the $c_s$ term:

$$c_s = \begin{cases} 
  c_0 & : \delta_dv = 0 \\
  0 & : \delta_dv = 1 \\
  \delta_dv \cdot c_1 + (\delta_dv - 1) \cdot c_2 & : \delta_dv > 1 
\end{cases}$$

- $c_1$ is the cost of a vector load
- $c_2$ is the cost of a vector extract (odd or even)
Different Cost Components

- Scheduling-invariant metrics:
  - $c_a$: cost of unaligned operations
  - $c_{\text{vector\_numerical\_ops}}$: cost of vector numerical operations
  - $c_{\text{vectstore}}, c_{\text{vectload}}$: cost of an individual load/store op

- Scheduling-sensitive metrics:
  - $c_S$ (aka stride load factor)

- Code generation-dependent metrics:
  - None here
Observations

Limitations:

▶ What about reuse?

▶ What about data locality estimation?

▶ What about coupling with other transformations?
  ▶ How to integrate fusion/distribution?
  ▶ What about complementary transformations for fusion?
  ▶ A real research problem here :-)}