Auto-Vectorization of Interleaved Data for SIMD

Dorit Nuzman, Ira Rosen, Ayal Zaks (IBM Haifa Labs) – PLDI ’06

Presented by Bertram Schmitt – 2011/04/13
Motivation

- **SIMD** (Single Instruction Multiple Data) exploits the natural parallelism of many applications by executing the same instruction on multiple data elements.

- Most implementations of the SIMD model require data elements to be packed in **vector registers**.

- SIMD memory architecture typically provides access to **contiguous memory items** only.
Approach

- **Goal:** vectorization of computations that require data-reordering techniques

- Optimized compiler for SIMD targets that
  - solves data-reordering efficiently
  - exploits data reuse
  - is generic and multi-platform compatible
The Interleaving Problem

- **Example:** complex multiplication

```java
for(int i = 0; i < len; i++) {
    c[2i] = a[2i]*b[2i] - a[2i+1]*b[2i+1];
    c[2i+1] = a[2i]*b[2i+1] + a[2i+1]*b[2i];
}
```

- Multiple occurrences of the same operation across consecutive iterations can be grouped into single SIMD instructions
for(int i = 0; i < len; i+=VF) {
    vector abee = (a[2i] * b[2i], a[2i+2] * b[2i+2], ..., 
                   a[2i+2*(VF-1)] * b[2i+2*(VF-1)]);
    vector aboo = (a[2i+1] * b[2i+1], a[2i+3] * b[2i+3], ..., 
                   a[2i+2*(VF-1)+1] * b[2i+2*(VF-1)+1]);
    vector abeo = (a[2i] * b[2i+1], a[2i+2] * b[2i+3], ..., 
                   a[2i+2*(VF-1)]*b[2i+2*(VF-1)+1]);
    vector aboe = (a[2i+1] * b[2i], a[2i+3] * b[2i+2], ..., 
                   a[2i+2*(VF-1)+1]*b[2i+2*(VF-1)]);

    c[2i,2i+2,...,2i+2*(VF-1)] = abee - aboo;
    c[2i+1,2i+3,...,2i+2*(VF-1)+1] = abeo + aboe;
}

- **VF** is the **vectorization factor** (number of elements that fit in a vector register)
The Interleaving Problem

- Most SIMD architectures allow the loading and storing of multiple data from memory only if the addresses $a(i)$ are consecutive
  - addresses need to be in the form of $a(i) = b + ui$

- To allow vectorization, the memory accesses need to be reordered
Addresses transformed to form \( a(i) = b + \delta ui \)

- \( \delta \) is referred to as **stride** or **interleaving factor**

- **Non-unit stride** accesses with \( \delta \neq 1 \)
Vectorizer Overview

- GNU Compiler Collection (GCC) is used to implement the vectorization of computations that involve interleaved data.
- GCC uses multiple levels of Intermediate Languages (IL) to translate source code to assembly code.
- In this paper: focus on GIMPLE.
Vectorizer Overview

- GIMPLE supports **Static Single Assignment (SSA)**

- Translated code retains enough information to allow advanced data-dependence analysis and aggressive high-level optimizations (e.g. auto-vectorization)

- Statements are then translated to **Register Transfer Language (RTL)** to optimize instruction scheduling and register allocation
Vectorizer Overview

- GCC vectorizer follows the approach of loop- and dependence-based vectorization
- Applies an eight-step analysis to each loop, followed by the actual vector transformation
- Vectorizer of this paper modifies several steps to allow vectorization of interleaved data
Loop Analysis

- **Step 1:**
  - Identify inner-most, single basic block **countable loops**
    (countable = the number of iterations can be determined prior to entering the loop)

- **Step 2:**
  - Determine **vectorization factor VF**
    (VF represents the number of data elements in a vector and is also the strip-mining factor of the loop)
Loop Analysis

- **Step 3:**
  - Find all memory references in the loop and check if they are "analyzable" (i.e. if a function can be constructed to describe memory accesses across iterations)

- **Step 4:**
  - Examine dependence cycles that involve *scalar variables* (i.e. do not go through memory), such as reductions
Loop Analysis

Step 5:

Check that the dependence distance between every pair of data references in the loop is either zero (loop-independent dependence) or at least VF

Step 6:

Check that the addresses of all memory accesses are consecutively increased

(This must be changed to support interleaved accesses!)
Loop Analysis

- **Step 7:**
  - Make sure that the alignment of all data references in the loop can be supported (e.g. by loop peeling)

- **Step 8:**
  - Verify that every operation in the loop can be supported in **vector form** by the target architecture
**Extending the Vectorizer**

- **Goal:** partition the set of load and store instructions with non-unit stride access into groups, where each group contains useful spatial locality.

- Pair of loads or stores $x, y$

  $$a_x(i) = b_x + \delta_x u_x i \quad \text{and} \quad a_y(i) = b_y + \delta_y u_y i$$

  exhibit spatial locality if

  $$\delta_x u_x = \delta_y u_y \quad \text{and} \quad |b_x - b_y| \text{ is “sufficiently” small}$$
Analyzing Interleaved Accesses

- A new data structure to represent a group of load (or store) instructions with non-unit stride accesses is introduced.

- All members of a group access the same array and have the same stride $\delta$.

- Each member $x$ of a group is assigned an integer $j_x$ (possibly negative), called the index, such that $j_x - j_y = b_x - b_y$ for any two members $x, y$ of a group.
Analyzing Interleaved Accesses

- The member with the smallest index (if $\delta > 0$, or the largest if $\delta < 0$) in a group is called the leader of the group.

- The leader is later responsible (at the transformation stage) for loading or storing the data for the group.

- The base address $b$ of the leader determines the overall starting address.
Analyzing Interleaved Accesses

- All other members use the **difference** between their index and the leader’s index to determine their address.

- For each load and store instruction, a **pointer** to its group and a **field** for storing its index is recorded.

- These pointers enable group members to quickly navigate to their leader and their index.
Analyzing Interleaved Accesses

- Previous example:

```c
for(int i = 0; i < len; i++) {
    c[2i] = a[2i]*b[2i] - a[2i+1]*b[2i+1];
    c[2i+1] = a[2i]*b[2i+1] + a[2i+1]*b[2i];
}
```

- Leads to three groups:
  - loads from a[2i], a[2i+1]
  - loads from b[2i], b[2i+1]
  - stores to c[2i], c[2i+1]
Extensions to Transformation Phase

- When reaching the first member of a group having stride $\delta$, a total of $\delta$ load or store statements starting from the address of the leader are created.

- A set of $\delta \log_2 \delta$ data reordering statements is generated:
  - `extract_even/odd` (for loads)
  - `interleave_low/high` (for stores)

- Statements can handle strides that are powers of 2.
Extensions to Transformation Phase

- Example: \( a(i) = b + 4i \) (assuming \( b \) is aligned)

```
vector b1 = load(b,b+1,...,b+VF-1);
vector b2 = load(b+VF,b+VF+1,...,b+2VF-1);
vector b3 = load(b+2VF,b+2VF+1,...,b+3VF-1);
vector b4 = load(b+3VF,b+3VF+1,...,b+4VF-1);
vector b12e = extract evens(b1,b2);
vector b34e = extract evens(b3,b4);
vector b1234ee = extract evens(b12e,b34e);
```

- Each member of the group is then connected to the appropriate resultant `extract_odd/even` statement, according to its index relative to the index of the group leader.
Estimated Profitability

- A group of $\delta$ scalar-loads accessing $\delta$ data-elements is interleaved by factor $\delta$

- Vectorization transforms this into a group of $\delta$ vector-loads, accessing $\delta VF$ data elements followed by a tree of $\delta \log_2 \delta$ extract operations

- There are therefore $\delta(1+\log_2 \delta)$ vector operations compared to $\delta VF$ scalar operations (that correspond to $VF$ scalar loop iterations)
  
  $\rightarrow$ leads to a factor of $VF / (1 + \log_2 \delta)$
### Estimated Profitability

<table>
<thead>
<tr>
<th>$\delta$</th>
<th>VF = 4</th>
<th>VF = 8</th>
<th>VF = 16</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>1.3</td>
<td>2.6</td>
<td>5.3</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>16</td>
<td>0.8</td>
<td>1.6</td>
<td>3.2</td>
</tr>
<tr>
<td>32</td>
<td>0.6</td>
<td>1.2</td>
<td>2.4</td>
</tr>
</tbody>
</table>

Estimated improvement in number of instructions:

$$VF / (1 + \log_2 \delta)$$
Experimental Results

- Many tests were conducted, based on synthetic tests and real-world computations
- Speed-ups were measured in cases where data was
  - aligned/unaligned
  - had gaps/no gaps (= fully interleaved)
- Speed-ups up to 3.7 for strides as high as 8
  (including increased overhead created by vectorization)
Experimental Results

Vectorization Speedup Factors on Interleaved Data

Benchmarks
- floating-point and 32-bit integer kernels (VF = 4)
- short data type kernels (VF = 8)
- char type kernels (no gaps) (VF = 16)
- char type kernels (gaps) (VF = 16)

Auto-Vectorization of Interleaved Data for SIMD
Summary

- Extension of loop-based vectorizer to handle computation with non-unit stride accesses
- Strides are powers of 2
- First step towards a hybrid **loop-aware SLP** (Superword Level Parallelism) vectorizer, which can exploit parallelism across loop iterations as well as inside loops
Thank you for your attention!