Vectorization for SIMD Architectures with Alignment Constraints

Alexandre Eichenberger, Peng Wu, Kevin O’Brien

IBM T.J. Watson Research Center, Yorktown Heights, NY
Overview

- **Background on simdization**
  - SIMD units
  - prior compilation techniques

- **General approach**
  - 3 step approach
  - data reorganization graph
  - code generation

- **Performance evaluation**

- **Summary**
Traditional Execution of a Loop

- **Sequential execution of**
  
  ```
  for (i=0; i<100; i++) a[i+3] = b[i+1] + c[i+2];
  ```

  Memory streams
  (grey is 1st iteration)
Can Be Speeded Up using SIMD

- **Single Instruction Multiple Data (SIMD) units are popular**
  - typical SIMD vectors are 16 bytes
  - compute 2 double, 4 float/int, 8 short, 16 char results
  - available on most platforms:
    - VMX/AltiVec (IBM PowerPC, Apple G5)
    - MMX/SSE (Intel x86),...

- **Theoretical speedups are high**
  - e.g. factors of 2 → 16 depending on data types

- **Limited support for memory alignment**
  - some have limited support (SSE2 has slower misaligned load ops)
  - some have no hardware misalignment support (VMX/AltiVec, VIS)

- **SIMD is hard to program**
  - optimized code very dependent on data alignment
  - difficult both for programmers and compilers
SIMD May Suffer from Alignment Problem

- SIMD memory unit only load/store 16 byte chunk of 16 byte aligned data

```plaintext
<table>
<thead>
<tr>
<th>0x1000</th>
<th>0x1010</th>
<th>0x1020</th>
</tr>
</thead>
<tbody>
<tr>
<td>b0</td>
<td>b1</td>
<td>b2</td>
</tr>
<tr>
<td>b3</td>
<td>b4</td>
<td>b5</td>
</tr>
<tr>
<td>b6</td>
<td>b7</td>
<td>b8</td>
</tr>
<tr>
<td>b9</td>
<td>b10</td>
<td></td>
</tr>
</tbody>
</table>
```

16-byte boundaries

```plaintext
&b[1] = 0x1004
```

- byte offset 4 in register

⇒ refer to vectorization for SIMD units as ⇒ SIMDIZATION
Alignment Problem (cont.)

- Alignment matters in registers too, e.g. in our “b[i+1]+c[i+2]” example

```
<table>
<thead>
<tr>
<th>b0</th>
<th>b1</th>
<th>b2</th>
<th>b3</th>
<th>b4</th>
<th>b5</th>
<th>b6</th>
<th>b7</th>
<th>b8</th>
<th>b9</th>
<th>b10</th>
</tr>
</thead>
<tbody>
<tr>
<td>vload b[1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vload c[2]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vadd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

16-byte boundaries

Offset 4:
- b0  b1  b2  b3
- vload b[1]

Offset 8:
- c0  c1  c2  c3
- vload c[2]

```
<table>
<thead>
<tr>
<th>b0+</th>
<th>b1+</th>
<th>b2+</th>
<th>b3+</th>
</tr>
</thead>
<tbody>
<tr>
<td>c0</td>
<td>c1</td>
<td>c2</td>
<td>c3</td>
</tr>
</tbody>
</table>
```

This is not b[1]+c[2], ...
How to Align Data in Registers

- If you need a vector of 4 values starting at misaligned address b[1]

16-byte boundaries

vload b[1]

vload b[5]

vpermute

vpermute is a generic permutation op
it is supported by most ISA,
e.g. vec_permute bytes on VMX/AltiVec

Aligned data: 1st element is at offset 0
How to Align Data in Registers (cont.)

- When you need a stream of vectors starting at address b[1]

![Diagram showing vectorization process with vload and vpermute instructions.](image-url)
How to Align Data in Registers (cont.)

Back to our “b[i+1]+c[i+2]” example

16-byte boundaries

vload b[1],b[5] → vpermute

vload c[2],c[6] → vpermute

vadd

this is b[1]+c[2], ...

offset 0

b1 b2 b3 b4

c2 c3 c4 c5

offset 0

b0 b1 b2 b3 b4 b5 b6 b7 b8 b9 b10

c0 c1 c2 c3 c4 c5 c6 c7 c8 c9 c10
Prior Work Approach Evaluation

- **Initial approaches**
  - variations on “peel until all memory streams are aligned”

- **Vast Compiler**
  - one permute op per misaligned memory access
  - inefficient in the number of permute ops
  - which is often a critical resource, e.g. Mac G5:
    - 2 SIMD memory pipes
    - 1 SIMD permute pipe

- **Superword Level Parallelism**
  - for loops: “unroll and pack into SIMD”
  - exhibit the same tradeoffs as loop unrolling vs. modulo scheduling
    - backedge is a optimization barrier

- **Our focus is on loop dominated multimedia/gaming applications**
Paper’s Approach Overview

- **Loop based, with optimized data reorganization**
  1. Build data reorganization graph
  2. Place “shift streams” using optimizing policies
  3. Generate SIMD code

Data Reorganization Graph:

```
Code:
<simdized prologue>
for(i=0; i<100; i+=4)
  <simdized loop body>
<simdized epilogue>
```
Data Reorganization Graph

- **Streams:** values addressed over the lifetime of a loop
  - memory streams, e.g. \( b[i+1] \) (\( i=0..99 \) here)

  - memory streams, e.g. \( b[i+1] \) (\( i=0..99 \) here)

  - register streams, output of stream operations, e.g. \texttt{vload}(b[i+1])

- Alignment of stream can be uniquely described by offset of 1st value
Additional stream operation: \texttt{vshiftstream}(O_{\text{in}}, O_{\text{out}})

- shift input stream to offset $O_{\text{out}}$, e.g.

\begin{align*}
\text{offset} = 4 & \quad \text{vshiftstream}(4, 12) \\
\text{offset} = 12
\end{align*}
Data Reorganization Graph (cont.)

- `vload/vstore(addr(i))`
  - offset from memory alignment

![Diagram showing data reorganization with vload and vstore operations, including offsets for memory alignment.](image-url)
Data Reorganization Graph (cont.)

- \(v\text{load/vstore}(\text{addr}(i))\)
  - offset from memory alignment

- \(v\text{add}(in_1, in_2)\)
  - all offsets must be identical
Valid Data Reorganization Graph

- **vload/vstore(addr(i))**
  - offset from memory alignment

- **vadd(in_1, in_2)**
  - all offsets must be identical

- **Valid: no violation of offset definitions**
Valid Data Reorganization Graph (cont.)

- **vload/vstore(addr(i))**
  - offset from memory alignment

- **vadd(in_1,in_2)**
  - all offsets must be identical

- **Valid: no violation of offset defs**

- **Make valid by suitably inserting**

- **vshiftstream(O_{in}, O_{out})**
  - shift streams to offset $O_{out}$

---

Where to insert shift streams?  
First focus of our work.
Shift Stream Placement: Zero Policy

- Shifts all misaligned streams to/from offset zero
  - least optimized, used for runtime alignment

![Diagram showing the shift stream placement with offsets and operations like vload, vadd, vshiftstream, and vstore.](image-url)
Shift Stream Placement: Eager Policy

- Eagerly shifts to store offset
  - offset 12 is the store alignment here

3 → 2 compared to Zero-Shift
Shift Stream Placement: Lazy Policy

- **Lazily shifts to store offset**
  - if \(b[i+1]\) and \(c[i+1]\) have same alignment \(\Rightarrow\) delay shifting past add

![Diagram showing vector operations]

\[
\begin{align*}
&vload \ b[i+1] \\
&vload \ c[i+1] \\
&vadd \\
&vshiftstream(4,12) \\
&vstore \ a[i+3]
\end{align*}
\]

3 \(\rightarrow\) 1 compared to Zero-Shift
Lazily shifts to **dominant** offset

- offset 4 is dominant $\Rightarrow$ align to it instead of store
Code Generation

- **SIMD code is generated from a valid data reorganization graph**
  - simdized loop (steady state)
  - simdized prologue & epilogue with partial vector stores
  - no redundant load/computation is introduced

- **Handles**
  - statements with stride one accesses, loop invariants (more is being added)
  - multiple statements with arbitrary misalignments

- **With alignment known at**
  - compile time: arbitrary data reorganization graphs
  - runtime: Zero-shift policy only
    - limitation due to code generation issues
Code Generation with Multiple Statements

SIMD execution of:
for (i=0; i<100; i++) { a[i] = ... ; b[i+1] = ...; c[i+3] = ... }

Implicit loop skewing
Performance Study

- **Implementation**
  - algorithm implemented in the IBM’s production XL compiler

- **Compares different shift placement policies**
  - ZERO: shifts to/from offset zero
  - EAGER: eagerly shifts to store offset
  - LAZY: lazily shifts to store offset
  - DOM: lazily shifts to/from dominant offset
  - SEQ: no simdization

- **Code Gen. redundancy elimination**
  - sp: using “software pipelining” technique within simdization
  - cse+: using separate CSE phase, CSE among consecutive loop iterations
  - Ø no redundancy elimination

[Vast compiler]
Performance Study (cont.)

- **Initial experiments**
  - 16 byte SIMD units with 4 floats per vector register
  - loops with 6 loads, 1 store, offsets randomly selected
  - loaded values are simply added together
  \[ \Rightarrow \text{7 mem + 5 add = 12 ops} \]

- **Performance numbers**
  - operations per datum
  - harmonic mean over 50 loops with different random offsets

- **Reports**
  - provably lower bounds
  - operation count from a cycle accurate simulator, including
    - actual computations
    - address computations
    - branching overhead, spills, ...
Selecting Best Code Gen. Scheme  (7 Mem, 5 Add Loops)

50 loops with randomly generated offsets, harmonic mean operations per datum
Comparing Shift Policies (7 Mem, 5 Add Loops)

50 loops with randomly generated offsets, harmonic mean operations per datum
Comparing Shift Policies (Using Offset Reassociation)

50 loops with randomly generated offsets, harmonic mean operations per datum
Summary

- **General framework to optimize data reorganization**
  - stream concept to view values through lifetime of a loop
  - data reorganization graph to minimize reorganization due to misalignment
  - extendable

- **Efficient code generation**
  - support for arbitrarily optimized graph (for compile time)
  - simdization without redundant load/computations
  - simdized prologue/epilogue (important for short trip)

- **Good performance in presence of misaligned data**
  - with 75% or more of the data misaligned
  - speedup factor of **3.71** with 4 floats per register
  - speedup factor of **6.06** with 8 shorts per registers