Efficient SIMD Code Generation
for
Runtime Alignment & Length Conversion

Peng Wu, Alexandre Eichenberger

Amy Wang

IBM T.J. Watson Research Center

IBM Toronto Laboratory
Overview

- Background on SIMD code generation
- Contribution #1: efficient alignment handling
- Contribution #2: support for data size conversion
- Measurements
Objectives

- **Single Instructions Multiple Data (SIMD):**
  - may yield 2x -16x speedup
  - moderate hardware cost
  - available in most platform: x86, PowerPC 970, CELL, BlueGene/L

- **Target applications:**
  - data intensive: multimedia, DSP, game, numerical
  - rich in data types: chars, shorts, ints, float,…
  - rich in data alignments, often runtime

- **Goal:**
  - produce SIMD code with low overhead for the type of code found in target apps
Sequential Execution of a Loop

- for (i=0; i<100; i++) a[i+2] = b[i+1] + c[i+3];

1st original i=0 loop iteration in yellow
Access only a 16-byte chunk of 16-byte aligned data*

VLOAD b[1]  
\&b[1] = 0x1004

byte offset 4 in register

* AltiVec/VMX and others; SSE supports unaligned access, but less efficiently
SIMD Load/Store Preserve Memory Alignment

- Access only a 16-byte chunk of 16-byte aligned data*

\[ \text{VLOAD } b[1] \]
\[ \&b[1] = 0x1004 \]

1st original \( i=0 \) loop iteration in yellow
1st SIMD loop iteration in grey

* Altivec/VMX and others; SSE supports unaligned access, but less efficiently
Erroneous SIMD Execution

- for (i=0; i<100; i++) a[i+2] = b[i+1] + c[i+3];

b[1] and c[3] are not aligned
=> wrong results*

Correct SIMD Execution (Zero-Shift) [VAST 04]

- for (i=0; i<100; i++) a[i+2] = b[i+1] + c[i+3];

<table>
<thead>
<tr>
<th>4 byte boundary</th>
<th>16-byte boundaries</th>
</tr>
</thead>
<tbody>
<tr>
<td>b0   b1   b2   b3</td>
<td>b4   b5   b6   b7</td>
</tr>
<tr>
<td>b8   b9   b10  b11</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>12-byte boundaries</th>
</tr>
</thead>
<tbody>
<tr>
<td>c0   c1   c2   c3</td>
</tr>
<tr>
<td>c4   c5   c6   c7</td>
</tr>
<tr>
<td>c8   c9   c10  c11</td>
</tr>
<tr>
<td>c12  c13  c14</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>8-byte boundary</th>
</tr>
</thead>
<tbody>
<tr>
<td>a0   a1   a2   a3</td>
</tr>
<tr>
<td>a4   a5   a6   a7</td>
</tr>
<tr>
<td>a8   a9   a10  a11</td>
</tr>
</tbody>
</table>
How to Align Data in Registers

Load 4 values from misaligned address b[1]

VLOAD b[1]

VLOAD b[5]

VPERMUTE*

* AltiVec/VMX and most other SIMD ISA have support for shuffling data of 2 registers

LOAD/STORE & SHIFT are expensive
=> want to minimize them
Better SIMD Execution (Lazy-Shift) [Eichen 04]

for (i=0; i<100; i++) a[i+2] = b[i+1] + c[i+3];

VLOAD & SHIFT b[1]

VLOAD & SHIFT C[3]

VADD

VSTORE a[2]
Contribution #1: Handling Runtime Alignment Efficiently

- **Runtime alignment is frequent:**
  - algorithmic characteristics: e.g. “a[i-1] + a[i] + a[i+1]”
  - coding style: pointers, libraries

- **Previous work does not support**
  - runtime alignment for minimized numbers of shifts (e.g. lazy-shift policy)
  - reason: code generation issue

- **We propose:**
  - an additional phase that “normalizes” the computation to eliminate this issue
Problem with Runtime Alignment

- **Shift c[3] left:**
  - VLOAD c[3]
  - VPERM-LEFT
  - Shift in data from the NEXT vector

- **Shift b[1] right:**
  - VLOAD b[1]
  - VPERM-RIGHT
  - Shift in data from the PREV vector

Different code sequences for shift LEFT & RIGHT.
With Runtime alignment, we don’t know which direction we are shifting to.
=> Hence the compiler cannot generate this code statically.
Insight: Convert Arbitrary Shifts into Left-Shifts

- Instead of shifting $b[1]$ RIGHT

By changing the reference point, i.e. by pre-pending 8 bytes, the shift RIGHT becomes a shift LEFT.
Approach

- “Normalize” each shift into a shift LEFT
  - prepend input stream to transform shift into a shift left
  - prepend amount is mechanically derived from alignment
  - for runtime alignment, prepend amount is only known at runtime

- details and proofs are in the paper
**Contribution #2: Handling Data Conversion Efficiently**

- **Data conversion is frequent**
  - multimedia often store in small data type, operate on larger data types
  - data conversion often occurs jointly with misalignment

- **Previous work does not support**
  - data conversion and misalignment for minimized numbers of shifts
  - reason: code generation issue

- **We extend the our approach to handle such cases**
Data Size Conversion

What happens when converting between data of different sizes?

2 vectors of integers

b0 b1 b2 b3

b4 b5 b6 b7

VPACK

VPACK perform the packing part of the int=>short conversion

becomes 1 vector of short

To fully utilize the SIMD bandwidth, we must compute over

- 2 vectors of 4 integers
- 1 vector of 8 shorts

*The logical part of the conversion (e.g. sign extension) has no SIMD impact and is ignored here
Alignment of Data is Impacted by Conversion

This become offset $12/2 = 6$ as $b[3]$ gets packed into shorts

$b[3]$ reside at offset 12
Alignment of Data is Impacted by Conversion (cont.)

Because of conversions, we must keep track of the truncation factors as data flows through the graph.

VLOAD b[3]

As it gets packed into shorts
offset < 8

VPACK

b0 b1 b2 b3 b4 b5 b6 b7

VSTORE a[3]

Truncates at 16 bytes, thus offset < 16

Store still truncates at 16 bytes

integers
shorts
Approach

- **Additional steps in presence of data conversion:**
  - propagate offsets through data conversion (vpack/vunpack)
  - keep track of truncation factors
  - must satisfy some constraints on truncation factors

- more details in the paper
Measurements

- **Target:**
  - Mac G5 with PowerPC 970 with VMX (Altivec)

- **Compiler:**
  - IBM’s XL compiler
  - comparison performed at identical level of optimizations

- **Metric:**
  - speedup factor (SIMD vs. scalar)
Measurements for Alignment Handling

- Synthetic benchmark (avg over 50 loops) with random align for 2 loads/1 store

Our contribution here

Optimized Alignment Handling
(Compile time only) [Eichen 04]
Measurements for Kernels

- With/without proposed runtime alignment handling, conversion support

![Speedup Bar Chart]

- Speedup Factor
- Kernels: numerical.saxpy, numerical.swim, tcp/ip.checksum, video.alphablending
- Techniques: zero-shift (prior work), lazy-shift, conversion + zero-shift, conversion + lazy-shift

- Speedup Factors:
  - numerical.saxpy: 0.71, 1.08, 1.08
  - numerical.swim: 1.21, 1.38, 1.21, 1.38
  - tcp/ip.checksum: 1.00, 1.00, 2.52, 2.92
  - video.alphablending: 1.00, 1.00, 6.14, 4.97

- Highest Speedup: 8.25
- Average Speedup: 2.24
Measurements for Benchmarks

![Bar Chart]

- **spec2k_base**: 1.02
- **spec2k_base**: 1.02
- **spec2k_base**: 1.02
- **SPEC5.20**: 1.05
- **SPEC5.20**: 1.06
- **SPEC5.20**: 1.20
- **SPEC5.20**: 1.41
- **SPEC5.20**: 1.62
- **semht**: 2.16
Conclusions

- **Realistic benchmarks have:**
  - runtime alignment
  - data size conversion

- **We propose here a technique that support both**
  - in an integrated fashion
  - fully compatible with optimized number of shifts

- **Demonstrated good speedups**
  - 1.1 – 6.1 x for kernels
  - up to 2.2 x for benchmarks
  - more integration with full compiler needed for better performance