An Integrated Simdization Framework
Using Virtual Vectors

Peng Wu, Alexandre Eichenberger
IBM T.J. Watson Research Center

Amy Wang, Peng Zhao,
IBM Toronto Laboratory

International Conference on Supercomputing, Tuesday, June 21, 2005
Single Instruction Multiple Data (SIMD) Computation

Process multiple “b[i]+c[i]” data per operations

16-byte boundaries

R1

R2

R3
SIMD Units

- Ubiquitous
  - from supercomputers: BlueGene/L
  - to PC: MMX / SSE / 3DNow on x86, Altivec / VMX on PowerPC
  - to game machines: PlayStation 3, XBox 360

- Diverse
  - varying supports for char to double computations
  - varying support for aligned / misaligned memory accesses
  - varying support for byte permutations

- Can be hard to program
  - compiler support to automatically generate SIMD codes
Successful Simdization

Extract Parallelism

loop level
for (i=0; i<256; i++)
a[i] =

basic-block level
a[i+0] =
a[i+1] =
a[i+2] =
a[i+3] =

entire short loop
for (i=0; i<8; i++)
a[i] =

Satisfy Constraints

alignment constraints

data size conversion

multiple targets

How can we integrate all this?
Multiple Sources of SIMD Parallelism

- **Loop level**
  - SIMD for a single statement across consecutive iterations
  - successful at:
    - efficiently handling misaligned data
    - pattern recognition (reduction, linear recursion)
    - leverage loop transformations in most compilers
    - amortize overhead (versioning, alignment handling) and employ cost models

References:

[Bik et al, IJPP 2002]
[VAST compiler, 2004]
[Eichenberger et al, PLDI 2004] [Wu et al, CGO 2005]
[Naishlos, GCC Developer's Summit 2004]
Multiple Sources of SIMD Parallelism (cont.)

- **Basic-block level**
  - SIMD across multiple isomorphic operations
  - successful at
    - handling unrolled loops (manually or by compiler)
    - extracting SIMD parallelism within structs, e.g.
      
      \[
      \begin{align*}
      a[i].x &= \\
      a[i].y &= \\
      a[i].z &= \\
      \end{align*}
      \]
    - extracting SIMD parallelism within a statement
      
      \[
      s += a(i)\times b(i) + a(i+1)\times b(i+1) + a(i+2)\times b(i+2) + a(i+3)\times b(i+3) + a(i+4)\times b(i+4)
      \]

      [Larsen et al, PLDI 2000]
      [Shin et al, PACT 2002]
Multiple Sources of SIMD Parallelism (cont.)

- **Short-loop level**
  - SIMD across entire loop iterations
  - effectively collapse innermost loop
  - we can now extract SIMD at the next loop level
  - e.g. FIR

```c
for (k=0; k<248; k++)
    for (i=0; i<8; i++)
        res[k] += in[k+i] * coef[k+i];
```

```
for (i=0; i<256; i++)
a[i] =
```
Multiple SIMD Hardware Constraints

- **Alignment in SIMD units matters**
  - when alignments within inputs do not match
  - must realign the data

```
R1:  b1 b2 b3 b4
R2:  c0 c1 c2 c3
    +
```

- **vpermute**

```
[16-byte boundaries]
```

---

**alignment constraints**

```
[16-byte boundaries]
```

**data size conversion**

```
[16-byte boundaries]
```

**multiple targets**

```
GENERIC VMX SPU BG/L
```

---

**16-byte boundaries**

```
[16-byte boundaries]
```

---

**Alignment in SIMD units matters**
- when alignments within inputs do not match
- must realign the data

---

**alignment constraints**

```
[16-byte boundaries]
```

**data size conversion**

```
[16-byte boundaries]
```

**multiple targets**

```
GENERIC VMX SPU BG/L
```
Multiple SIMD Hardware Constraints (cont.)

- **Size of data in SIMD registers matters**
  - a SIMD register that fits 8 short integers (2 bytes)
  
<table>
<thead>
<tr>
<th>R1</th>
<th>b1</th>
<th>b2</th>
<th>b3</th>
<th>b4</th>
<th>b5</th>
<th>b6</th>
<th>b7</th>
<th>d8</th>
</tr>
</thead>
</table>
  
  - will only fit 4 integers (4-bytes) in the same space
  
<table>
<thead>
<tr>
<th>R2</th>
<th>b1</th>
<th>b2</th>
<th>b3</th>
<th>b4</th>
</tr>
</thead>
</table>
  
  - thus when converting from short to integer,
  - we must issue 2x integer SIMD operations

- **Alignment constraints**

- **Data size conversion**

- **Multiple targets**

- **GENERIC VMX SPU BG/L**
Different platforms have varying SIMD support
- e.g. VMX / SPU have SIMD permute instructions
- e.g. BG/L has cross-path arithmetic operations
Can We Support All Cross-Product Interactions?

**Loop Level**
for (i=0; i<256; i++)
a[i] =

**Basic-Block Level**
a[i+0] =
a[i+1] =
a[i+2] =
a[i+3] =

**Entire Short Loop**
for (i=0; i<8; i++)
a[i] =

**Alignment Constraints**

**Data Size Conversion**

**Multiple Targets**
- GENERIC
- VMX
- SPU
- BG/L
Key Abstraction: Virtual SIMD Vector

- **Virtual SIMD Vector**
  - has arbitrary length
  - has no alignment constraints

- **Extraction of SIMD Parallelism**
  - use virtual vector as representation
  - abstract away all the hardware complexity

- **Progressive “de-virtualization” of the virtual vector**
  - until each vector in the loop satisfies all hardware constraints
  - or revert vectors back to scalars (if too much overhead)
Integrated SIMDization Framework

Characteristics:
- Modular
- Integrated Analysis
- Global Pointer Analysis
- Constant Propagation
- Dependence Elimination
- Idiom Recognition
- Data Layout Optimization

SIMD Extraction
- Virtual Vectors
  - arbitrary length
  - arbitrary alignment
- Transition
  - alignment handling
  - short length handling
- Physical Vectors
  - 16 bytes long
  - machine alignment

Machine Specific
Example 1: Basic-Block & Loop Level Aggregation

Original loop

```
for (i=0; i<256; i++) {
    a[i].x = a[i].y = a[i].z = b[i+1] =
}
```

Value streams

```
a0.x a0.y a0.z a1.x a1.y a1.z a2.x a2.y a2.z a3.x a3.y...
```

4 iterations shown here for the purpose of illustration

- 1st iter
- 2nd iter
- 3rd iter
- 4th iter
Phase 1: Basic-Block Level Aggregation

Original loop

```
for (i=0; i<256; i++) {
    a[i].x =
    a[i].y =
    a[i].z =
    b[i+1] =
}
```

Value streams

```
Value streams:  
a0.x  a0.y  a0.z  a1.x  a1.y  a1.z  a2.x  a2.y  a2.z  a3.x  a3.y...

Basic-Block level aggregation

- pack a[i].x, a[i].y, a[i].z into a vector of 3 elements
- pack regardless of alignment

```

An Integrated Simdization Framework using Virtual Vectors, ICS05
Alexandre Eichenberger
Phase 2: Loop-Level Aggregation

BB-aggregated loop

for (i=0; i<256; i++) {
    (a[i].x,y,z) = a0.x a0.y a0.z
    b[i+1] = a1.x a1.y a1.z
}

pack with self

Loop-level aggregation

- pack each statement with itself across consecutive iterations
- final vector lengths must be multiple of 16 bytes

- scalar “b[i]” or vector “(a[i].x,y,z)” are treated alike
- pack regardless of alignment
Phase 3: Alignment Devirtualization

Loop-aggregated

for (i=0; i<256; i+=4) {
    (a[i].x,...,a[i+3].z) =
    (b[i+1],...,b[i+4]) =
}

Value streams

Align access

Alignment *

- shift misaligned streams
- skew the computations so that loop computes (b[i+4]...b[i+7])

* Arrays (e.g. &a[0], &b[0],...) are assumed here 16-byte aligned.
**Phase 4: Length Devirtualization**

**Aligned loop**

\[(b[1]...b[3]) =\]

\[
\text{for } (i=0; \ i<252; \ i+=4) \ \{ \\
(a[i].x,...,a[i+3].z) = \\
(b[i+4],...,b[i+7]) = \\
\} \\
(a[252].x,...,a[255].z) = \\
b[256] =
\]

**Value streams**

**Length**

- break into 16-byte chunks
Example II: Data-Size Conversion and Misalignment

Original loop

for (i=0; i<256; i++) {
    a[i] += (int) b[i+1]
}

- short computations
- integer computations

Load b[i+1]

Convert

Add

Store a[i]

Short (2 bytes)

Integer (4 bytes)
Phase 1: Loop-Level Aggregation

Original loop

```c
for (i=0; i<256; i++) {
    a[i]  +=  (int) b[i+1]
}
```

Loop-level aggregation

- pack each statement with itself across consecutive iterations
- virtual vectors have uniform number of elements, even when
  - vector of 8 integer = 32 bytes of data
  - vector of 8 short = 16 bytes of data
Phase 2: Alignment Devirtualization

Original loop

for (i=0; i<256; i++) {
    a[i] += (int) b[i+1]
}

Align b[i+1]

Alignment

- shift misaligned streams
- easy to do as we are still dealing with long vectors
Phase 3: Length Devirtualization

Original loop

```
for (i=0; i<256; i++) { 
  a[i] += (int) b[i+1] 
}
```

Length

- 8 shorts fit into a 16-byte register
- 8 integers do not fit; must replicate integer registers and associated instructions
Measurements for Kernels

Machine: PowerPC 970 with VMX (16 byte SIMD unit)
Measurements for Applications

Machine: PowerPC 970 with VMX (16 byte SIMD unit)
Conclusions

- **Novel SIMD framework**
  - use virtual vectors to abstract machine constraints away
  - enable us to support “cross-product” interactions
    - basic-block & loop-level extraction of SIMD parallelism
    - misalignment & data-size conversion

- **Other interactions successfully handled are described in the paper**
  - extraction of SIMD parallelism in short loops
  - mixed mode exploitation of SIMD parallelism
    - some operations on SIMD units
    - others on scalar units

- **Demonstrated encouraging speedups**
  - up to 8x on kernels
  - up to 2x on applications
  - on a machine with generally 2 scalar units for 1 SIMD unit (PPC970)
Questions