StVEC: A Vector Instruction Extension for High Performance Stencil Computation

Naser Sedaghati    Renji Thomas    Louis-Noël Pouchet
Radu Teodorescu    P. Sadayappan

Department of Computer Science and Engineering
The Ohio State University

HPC Research Lab: barista.cse.ohio-state.edu
Computer Architecture Lab: arch.cse.ohio-state.edu

October 13th 2011
1. Introduction

2. Vectorization of Stencils

3. Enhancing Vector ISA with StVEC

4. Generating Code for StVEC

5. Evaluation

6. Summary
Stencil Computation

Repeat over TIME

- Sweep over a spatial grid
- Compute a point from neighbor points values
  - Same grid or multiple grids

Numerous application domains

- Finite difference methods for solving PDEs
- Image processing (e.g. MRI image pipeline)
- Computational electromagnetics, CFD, numerical relativity, etc.
2-D 5-point Jacobi

for (t = 0; t < TMAX; t++)
    for (i = 1; i < N - 1; i++)
        for (j = 1; j < M - 1; j++)
2-D 5-point Jacobi

for (t = 0; t < TMAX; t++)
    for (i = 1; i < N - 1; i++)
        for (j = 1; j < M - 1; j++)
Stencil Computation: An Example

2-D 5-point Jacobi

for (t = 0; t < TMAX; t++)
    for (i = 1; i < N - 1; i++)
        for (j = 1; j < M - 1; j++)
2-D 5-point Jacobi

for (t = 0; t < TMAX; t++)
    for (i = 1; i < N - 1; i++)
        for (j = 1; j < M - 1; j++)
Short-Vector SIMD

Identical computation on small chunks of data

- Independent operations
- Vector size (width) of 2 to 64
- Packing operations to form a vector (shuffle, extract, etc.)

SIMD performance

- Multiple SIMD units per CPU
- Maximum speedup equals the vector width

Ubiquitous features on modern processors

- x86 – SSE, AVX
- Power – VMX/VSX
- ARM – NEON
- Cell SPU
Vectorization: An Example

Vector width = 4, N divisible by 4

for (t = 0; t < T; t++)
    for (i = 4; i < N; i++)
Vectorization: An Example

Vector width = 4, N divisible by 4

\[
\text{for } (t = 0; t < T; t++) \\
\quad \text{for } (i = 4; i < N; i++) \\
\quad \quad A[i] = B[i] * B[i] ;
\]

1: ASM (MIPS-like)

\[
\text{for } (t = 0; t < T; t++) \\
\quad \text{for } (i = 4; i < N; i++)
\quad \{ \\
\quad \quad \text{LD } R1, \&B[i] \\
\quad \quad \text{MUL } R2, R1, R1 \\
\quad \quad \text{ST } R2, \&A[i] \\
\quad \}
\]
**Vectorization: An Example**

**Vector width = 4, N divisible by 4**

\[
\text{for } (t = 0; \ t < T; \ t++) \\
\quad \text{for } (i = 4; \ i < N; \ i++) \\
\qquad A[i] = B[i] \times B[i] \\
\]

1: ASM (MIPS-like)

\[
\text{for } (t = 0; \ t < T; \ t++) \\
\quad \text{for } (i = 4; \ i < N; \ i++) \\
\qquad \begin{align*}
\text{LD} & \quad R1, &B[i] \\
\text{MUL} & \quad R2, R1, R1 \\
\text{ST} & \quad R2, &A[i] \\
\end{align*}
\]

2: 4-way unroll + re-schedule

\[
\text{for } (t = 0; \ t < T; \ t++) \\
\quad \text{for } (i = 4; \ i < N; \ i+=4) \\
\qquad \begin{align*}
\text{LD} & \quad R1, &B[i] \\
\text{LD} & \quad R2, &B[i+1] \\
\text{LD} & \quad R3, &B[i+2] \\
\text{LD} & \quad R4, &B[i+3] \\
\text{MUL} & \quad R5, R1, R1 \\
\text{MUL} & \quad R6, R2, R2 \\
\text{MUL} & \quad R7, R3, R3 \\
\text{MUL} & \quad R8, R4, R4 \\
\text{ST} & \quad R5, &A[i] \\
\text{ST} & \quad R6, &A[i+1] \\
\text{ST} & \quad R7, &A[i+2] \\
\text{ST} & \quad R8, &A[i+3] \\
\end{align*}
\]

3: Vectorize

\[
\text{for } (t = 0; \ t < T; \ t++) \\
\quad \text{for } (i = 4; \ i < N; \ i+=4) \\
\qquad \begin{align*}
\text{VLD} & \quad VR1, &B[i] \\
\text{VMUL} & \quad VR2, VR1, VR1 \\
\text{VST} & \quad VR2, &A[i] \\
\end{align*}
\]

Observation

Aligned memory referencing (i.e. $B[i]$) helps vectorization!
Vectorization: An Example

Vector width = 4, N divisible by 4

for (t = 0; t < T; t++)
    for (i = 4; i < N; i++)

1: ASM (MIPS-like)
for (t = 0; t < T; t++)
    for (i = 4; i < N; i++){
        LD  R1, &B[i]
        MUL R2, R1, R1
        ST  R2, &A[i]
    }

2: 4-way unroll + re-schedule
for (t = 0; t < T; t++)
    for (i = 4; i < N; i+=4){
        LD  R1, &B[i]
        LD  R2, &B[i+1]
        LD  R3, &B[i+2]
        LD  R4, &B[i+3]
        MUL R5, R1, R1
        MUL R6, R2, R2
        MUL R7, R3, R3
        MUL R8, R4, R4
        ST  R5, &A[i]
        ST  R6, &A[i+1]
        ST  R7, &A[i+2]
        ST  R8, &A[i+3]
    }

3: Vectorize
for (t = 0; t < T; t++)
    for (i = 4; i < N; i+=4){
        VLD VR1, &B[i]
        VMUL VR2, VR1, VR1
        VST VR2, &A[i]
    }

Observation
Aligned memory referencing (i.e. B[i]) helps vectorization!

Naser Sedaghati (CSE @ Ohio State)
StVEC: A Vector Instruction Extension
Vectorization: An Example

Vector width = 4, N divisible by 4

\[
\text{for (t = 0; t < T; t++)} \\
\text{for (i = 4; i < N; i++)} \\
\text{A[i] = B[i] * B[i] ;}
\]

1: ASM (MIPS-like)

```
for (t = 0; t < T; t++)
    for (i = 4; i < N; i++){
        LD R1, &B[i]
        MUL R2, R1, R1
        ST R2, &A[i]
    }
```

2: 4-way unroll + re-schedule

```
for (t = 0; t < T; t++)
    for (i = 4; i < N; i+=4){
        LD R1, &B[i]
        LD R2, &B[i+1]
        LD R3, &B[i+2]
        LD R4, &B[i+3]
        MUL R5, R1, R1
        MUL R6, R2, R2
        MUL R7, R3, R3
        MUL R8, R4, R4
        ST R5, &A[i]
        ST R6, &A[i+1]
        ST R7, &A[i+2]
        ST R8, &A[i+3]
    }
```

3: Vectorize

```
for (t = 0; t < T; t++)
    for (i = 4; i < N; i+=4){
        VLD VR1, &B[i]
        VMUL VR2, VR1, VR1
        VST VR2, &A[i]
    }
```

Observation

- Aligned memory referencing (i.e. B[i]) helps vectorization!
Vectorization of Stencils
Vectorizing Stencil Computation

for (t = 0; t < T; t++)
   for (i = 4; i < N; i++)
      A[i] += B[i-1] * B[i];
Vectorization of Stencils

Vectorizing Stencil Computation

for (t = 0; t < T; t++)
    for (i = 4; i < N; i++)
        A[i] += B[i-1] * B[i];

Solution 1: load + shuffle

B[ ] in XMM Registers

<table>
<thead>
<tr>
<th>xmm</th>
<th>127</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>h</td>
<td>g</td>
</tr>
<tr>
<td>2</td>
<td>h</td>
<td>g</td>
</tr>
<tr>
<td>i</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>d</td>
<td>c</td>
</tr>
</tbody>
</table>

SSE Assembly (N=1024)

LOOP:
movaps 16+B(...), %xmm2
movaps %xmm2, %xmm1
palignr $12, B(...), %xmm1
mulps %xmm2, %xmm1
addps 16+A(...), %xmm1
movaps %xmm1, 16+A(...) 
addq $4, %rdx
cmpq $1020, %rdx
jb LOOP

Naser Sedaghati (CSE @ Ohio State)  StVEC: A Vector Instruction Extension  PACT'11
Vectorizing Stencil Computation

for (t = 0; t < T; t++)
  for (i = 4; i < N; i++)
    A[i] += B[i-1] * B[i];

**Solution 1:** load + shuffle

**Solution 2:** unaligned load

---

B[ ] in XMM Registers

<table>
<thead>
<tr>
<th>xmm</th>
<th>127</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>g</td>
<td>f</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>h</td>
<td>g</td>
</tr>
</tbody>
</table>

SSE Assembly (N=1024)

```assembly
LOOP:
  movups 12+B(...), %xmm1
  mulps 16+B(...), %xmm1
  addps 16+A(...), %xmm1
  movaps %xmm1, 16+A(....)
  addq $4, %rdx
  cmpq $1020, %rdx
  jb LOOP
```
Vectorizing Stencil Computation

for (t = 0; t < T; t++)
    for (i = 4; i < N; i++)
        A[i] += B[i-1] * B[i];

Solution1: load + shuffle
Solution2: unaligned load
Our Solution: StVEC (no shuffle, no unaligned load)

Naser Sedaghati (CSE @ Ohio State)
StVEC: A Vector Instruction Extension
PACT’11
Enhancing Vector ISA with StVEC
Building Unaligned Vector Operands

Idea: build an unaligned operand during register read

- Only one unaligned operand suffice for stencils
Building Unaligned Vector Operands

Idea: build an unaligned operand during register read

- Only one unaligned operand suffice for stencils

- Build the unaligned operand (i.e. VOPR_x) with two source regs
  - base and extension
Building Unaligned Vector Operands

Idea: build an unaligned operand during register read

- Only one unaligned operand suffice for stencils
- Build the unaligned operand (i.e. VOPR_x) with two source regs
  - base and extension

16x128-bit vector register file

<table>
<thead>
<tr>
<th>VR0</th>
<th>VR1</th>
<th>VR2</th>
<th>VR14</th>
<th>VR15</th>
</tr>
</thead>
<tbody>
<tr>
<td>X_0,0</td>
<td>X_0,1</td>
<td>X_0,2</td>
<td>X_1,0</td>
<td>X_1,3</td>
</tr>
<tr>
<td>X_1,0</td>
<td>X_1,1</td>
<td>X_1,2</td>
<td>X_14,0</td>
<td>X_14,3</td>
</tr>
<tr>
<td>X_2,0</td>
<td>X_2,1</td>
<td>X_2,2</td>
<td>X_14,1</td>
<td>X_15,2</td>
</tr>
<tr>
<td>X_3,0</td>
<td>X_3,1</td>
<td>X_3,2</td>
<td>X_14,2</td>
<td>X_15,3</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>X_15,0</td>
<td>X_15,1</td>
<td>X_15,2</td>
<td>X_15,3</td>
<td>...</td>
</tr>
</tbody>
</table>

base = VR_1, extension = VR_{14}

source | offset | VOPR_x
## Building Unaligned Vector Operands

**Idea:** build an unaligned operand during register read

- Only one unaligned operand suffice for stencils
- Build the unaligned operand (i.e. $VOPR_x$) with two source regs
  - base and extension

### 16x128-bit vector register file

<table>
<thead>
<tr>
<th>VR0</th>
<th>VR1</th>
<th>VR2</th>
<th>VR14</th>
<th>VR15</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X_{0,0}$</td>
<td>$D$</td>
<td>$X_{2,3}$</td>
<td>$X_{14,3}$</td>
<td>$X_{15,3}$</td>
</tr>
<tr>
<td>$X_{0,1}$</td>
<td>$C$</td>
<td>$X_{2,2}$</td>
<td>$X_{14,2}$</td>
<td>$X_{15,2}$</td>
</tr>
<tr>
<td>$X_{0,2}$</td>
<td>$B$</td>
<td>$X_{2,1}$</td>
<td>$X_{14,1}$</td>
<td>$X_{15,1}$</td>
</tr>
<tr>
<td>$X_{0,3}$</td>
<td>$A$</td>
<td>$X_{2,0}$</td>
<td>$X_{14,0}$</td>
<td>$X_{15,0}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>source</th>
<th>offset</th>
<th>$VOPR_x$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$VR_1$</td>
<td>0</td>
<td>$X_{1,0:4}$ (aligned)</td>
</tr>
</tbody>
</table>
Enhancing Vector ISA with StVEC

Execution Model

Building Unaligned Vector Operands

Idea: build an unaligned operand during register read

- Only one unaligned operand suffice for stencils
- Build the unaligned operand (i.e. VOPR_x) with two source regs
  - base and extension

16x128-bit vector register file

<table>
<thead>
<tr>
<th>VR0</th>
<th>VR1</th>
<th>VR2</th>
<th>VR14</th>
<th>VR15</th>
</tr>
</thead>
<tbody>
<tr>
<td>X_0,3</td>
<td>C</td>
<td>X_2,3</td>
<td>X_14,3</td>
<td></td>
</tr>
<tr>
<td>X_0,2</td>
<td>B</td>
<td>X_2,2</td>
<td>X_14,2</td>
<td></td>
</tr>
<tr>
<td>X_0,1</td>
<td>A</td>
<td>X_2,1</td>
<td>X_14,1</td>
<td></td>
</tr>
<tr>
<td>X_0,0</td>
<td></td>
<td>X_2,0</td>
<td>X_14</td>
<td>D</td>
</tr>
</tbody>
</table>

base = VR_1, extension = VR_{14}

<table>
<thead>
<tr>
<th>source</th>
<th>offset</th>
<th>VOPR_x</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR_1</td>
<td>0</td>
<td>X_{1,0:4} (aligned)</td>
</tr>
<tr>
<td>VR_1, VR_{14}</td>
<td>1</td>
<td>X_{1,1:3} X_{14,0:1}</td>
</tr>
</tbody>
</table>
Building Unaligned Vector Operands

Idea: build an unaligned operand during register read

- Only one unaligned operand suffice for stencils
- Build the unaligned operand (i.e. VOPR_x) with two source regs
  - base and extension

16x128-bit vector register file

<p>| base = VR_1, extension = VR_{14} |</p>
<table>
<thead>
<tr>
<th>source</th>
<th>offset</th>
<th>VOPR_x</th>
</tr>
</thead>
<tbody>
<tr>
<td>VR_1</td>
<td>0</td>
<td>X_{1,0:4} (aligned)</td>
</tr>
<tr>
<td>VR_1, VR_{14}</td>
<td>1</td>
<td>X_{1,1:3} X_{14,0:1}</td>
</tr>
<tr>
<td>VR_1, VR_{14}</td>
<td>2</td>
<td>X_{1,2:2} X_{14,0:2}</td>
</tr>
</tbody>
</table>
Building Unaligned Vector Operands

Idea: build an unaligned operand during register read

- Only one unaligned operand suffice for stencils

- Build the unaligned operand (i.e. \( \text{VOPR}_x \)) with two source regs
  - \textit{base} and \textit{extension}

16x128-bit vector register file

\[
\begin{array}{cccc}
VR_0 & X_{0,0} & X_{0,1} & X_{0,2} \\
VR_1 & A & X_{1,0} & X_{1,1} \\
VR_2 & X_{2,0} & X_{2,1} & X_{2,2} \\
VR_14 & X_{14,0} & D & C \\
VR_15 & X_{15,0} & X_{15,1} & X_{15,2}
\end{array}
\]

\[
\begin{array}{c|c|c}
\text{base} = VR_1, \text{ extension} = VR_{14} \\
\hline
\text{source} & \text{offset} & \text{VOPR}_x \\
\hline
VR_1 & 0 & X_{1,0:4} \text{ (aligned)} \\
VR_1, VR_{14} & 1 & X_{1,1:3}X_{14,0:1} \\
VR_1, VR_{14} & 2 & X_{1,2:2}X_{14,0:2} \\
VR_1, VR_{14} & 3 & X_{1,3:1}X_{14,0:3}
\end{array}
\]
StVEC Instructions

StVEC operands

Target: register-register vector instructions

- src1 and dst: unchanged
- src2: expanded to: offset, base and extension
StVEC Instructions

StVEC operands

Target: register-register vector instructions

- **src1** and **dst**: unchanged
- **src2**: expanded to: offset, base and extension

SSE translation to StVEC (vector width: \( W = 4 \))

<table>
<thead>
<tr>
<th>SSE</th>
<th>StVEC</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mulps VR_x, VR_y</code></td>
<td><code>stmulps offset, VR_x, VR_z, VR_y</code></td>
</tr>
</tbody>
</table>
StVEC Instructions

StVEC operands
Target: register-register vector instructions
- src1 and dst: unchanged
- src2: expanded to: offset, base and extension

SSE translation to StVEC (vector width: $W = 4$)

SSE
- $\text{mulps } VR_x, VR_y$
- $VR_y = VR_x \cdot VR_y$

StVEC
- $\text{stmulps offset, } VR_x, VR_z, VR_y$
- $VR_y = VR_x\{\text{offset : } W - \text{offset}\} VR_z\{0 : \text{offset}\} \cdot VR_y$

```plaintext
LOOP:
movaps 16+B(...), %xmm2
stmulps $3, %xmm1, %xmm2, %xmm2
addps 16+A(...), %xmm2
movaps %xmm2, 16+A(...)
; circulate the buffer(s)
addq $4, %rdx
cmpq $1020, %rdx
jb LOOP
```
Modified Vector Register File (StVRF)

Modifications to the baseline VRF (BVRF)

- separate register address for each bank
- vector register adjustment (VRA) logic w/ offset

\[ T_{StVRF} \approx T_{BVRF} + T_{VRA} \]
Modified Vector Register File (StVRF)

Modifications to the baseline VRF (BVRF)

- separate register address for each bank
- vector register adjustment (VRA) logic w/ offset

Example: $stmulps\$1, \%xmm1, \%xmm2, \%xmm3$

- vector width = 4
- offset = 1
- base = \%xmm1
- extension = \%xmm2
- src1 = dst = \%xmm3
- src2 = \%xmm1{1:3}\%xmm2{0:1}
- OP = vector multiply
Modified Vector Register File (StVRF)

Modifications to the baseline VRF (BVRF)

- separate register address for each bank
- vector register adjustment (VRA) logic w/ offset

Example: \textit{stmulps} $1, \%xmm1, \%xmm2, \%xmm3$

- vector width $= 4$
- offset $= 1$
- base $= \%xmm1$
- extension $= \%xmm2$
- src1 $= \%xmm3$
- src2 $= \%xmm1 \{1:3\} \%xmm2 \{0:1\}$
- OP $= \text{vector multiply}$
**Modified Vector Register File (StVRF)**

**Modifications to the baseline VRF (BVRF)**

- separate register address for each bank
- vector register adjustment (VRA) logic w/ offset

**Example:** \texttt{stmulps $1, \%xmm1, \%xmm2, \%xmm3}

- vector width = 4
- offset = 1
- base = \%xmm1
- extension = \%xmm2
- src1 = dst = \%xmm3
- src2 = \%xmm1{1:3}\%xmm2{0:1}
- OP = vector multiply
Modified Vector Register File (StVRF)

Modifications to the baseline VRF (BVRF)

- separate register address for each bank
- vector register adjustment (VRA) logic w/ offset

Example: `stmulps $1, %xmm1, %xmm2, %xmm3`

- vector width = 4
- offset = 1
- base = %xmm1
- extension = %xmm2
- src1 = dst = %xmm3
- src2 = %xmm1{1:3}%xmm2{0:1}
- OP = vector multiply
Generating Code for StVEC
The Code Generation Procedure

Input: Abstract syntax tree (AST) of a vectorizable innermost loop

1. generate basic intrinsics

2. perform StVEC code generation

Output: vectorized loop with StVEC intrinsics
StVEC Code Generation

**Input:** basic intrinsics loop

**The proposed algorithm**

1. replace every unaligned reference by two aligned loads
2. find offset values and promote to StVEC insts when possible
StVEC Code Generation

**Input:** basic intrinsics loop

**The proposed algorithm**

1. replace every unaligned reference by two aligned loads

2. find offset values and promote to StVEC insts when possible

Some additional optimizations:
- dead-code elimination
- 3-stage software-pipelining
StVEC Code Generation

**Input:** basic intrinsics loop

**The proposed algorithm**

1. replace every unaligned reference by two aligned loads
2. find offset values and promote to StVEC insts when possible

Some additional optimizations:

- dead-code elimination
- 3-stage software-pipelining

**Properties**

- can be emulated w/ existing vector ISA
- unaligned loads can be eliminated
Evaluation Methodology
Emulating StVEC Instructions on Real Vector ISA

\[ \text{StVRF} \approx \text{BVRF} + \text{VRA} \]

**st-opt**: StVEC inst models delay of 1 SIMD inst

\[ \text{StVRF} \leq T_{\text{cycle}} \]

**st-pes**: StVEC inst models delay of 2 dependent SIMD insts

\[ \text{StVRF} \leq 2 \times T_{\text{cycle}} \]

Naser Sedaghati (CSE @ Ohio State)
Emulating StVEC Instructions on Real Vector ISA

\[ T_{\text{StVRF}} \approx T_{\text{BVRF}} + T_{\text{VRA}} \]

- **st-opt**: StVEC inst models delay of 1 SIMD inst
- **st-pes**: StVEC inst models delay of 2 dependent SIMD insts

\[ T_{\text{StVRF}} \leq T_{\text{cycle}} \]

- Stencil Kernel
- Intrinsics
- Unrolling
- SWP

baseline
Emulating StVEC Instructions on Real Vector ISA

\[ T_{\text{StVRF}} \approx T_{\text{BVRF}} + T_{\text{VRA}} \]

\[ T_{\text{StVRF}} \leq T_{\text{cycle}} \]

\[ T_{\text{StVRF}} \leq 2 \times T_{\text{cycle}} \]

- Instrinsics
- Unrolling
- SWP

StVEC Code-Gen

baseline
Emulating StVEC Instructions on Real Vector ISA

\[ T_{StVRF} \approx T_{BVRF} + T_{VRA} \]

**st-opt**: StVEC inst models delay of 1 SIMD inst

\[ T_{StVRF} \leq T_{cycle} \]
Emulating StVEC Instructions on Real Vector ISA

- $T_{StVRF} \approx T_{BVRF} + T_{VRA}$
- **st-opt**: StVEC inst models delay of 1 SIMD inst
  - $T_{StVRF} \leq T_{cycle}$
- **st-pes**: StVEC inst models delay of 2 dependent SIMD insts
  - $T_{StVRF} \leq 2 \times T_{cycle}$
Emulating StVEC Instructions on Real Vector ISA

- \( T_{StVRF} \approx T_{BVRF} + T_{VRA} \)
- **st-opt**: StVEC inst models delay of 1 SIMD inst
  - \( T_{StVRF} \leq T_{cycle} \)
- **st-pes**: StVEC inst models delay of 2 dependent SIMD insts
  - \( T_{StVRF} \leq 2 \times T_{cycle} \)
Setup

Running on x86 architectures
- Intel Core i7 Nehalem, Intel Sandy Bridge, Intel Core2 Quad
- AMD Phenom (K10)

Stencil Benchmarks
- 1D: Jacobi (2-, 3-, 5- and 7-point)
- 2D: Jacobi (5- and 9-point), POP, FDTD 2D, Rician Denoise 2D
- 3D: Jacobi (27-point), Heatut 3D

L1-resident problem size
- assume tiling was performed beforehand if necessary

Compilers
- ICC 12 (w/ -fast) and GCC 4.4.4 (w/ -O3)
Experimental Results
Average (Geometric Mean) Speedup with StVEC

Single-precision (average across all 12 benchmarks)

Normalized to baseline (intrinsics + unrolling + SWP)
Average (Geometric Mean) Speedup with StVEC

Single-precision (average across all 12 benchmarks)

Normalized to baseline (intrinsics + unrolling + SWP)
Average (Geometric Mean) Speedup with StVEC

Single-precision (average across all 12 benchmarks)

- Normalized to **baseline** (intrinsics + unrolling + SWP)
- Single-precision: 7% to 2.26x for st-pes and 20% to 2.47x for st-opt
- Double-precision: 30% to 65% for st-opt
Speedup with ICC Across Machines

Single-precision

Normalized to **baseline** (intrinsics + unrolling + SWP)
Speedup with ICC Across Machines

Single-precision

- Normalized to **baseline** (intrinsics + unrolling + SWP)
StVEC Hardware Overhead

StVRF access time in 45nm CMOS

- $T_{BVRF}$: SRAM model in CACTI
- $T_{VRA}$: circuit synthesis/layout by Synopsys Design Compiler

<table>
<thead>
<tr>
<th>#Regs.</th>
<th>#Banks</th>
<th>Reg. size</th>
<th>BVRF</th>
<th>StVRF</th>
</tr>
</thead>
<tbody>
<tr>
<td>128 (st-opt)</td>
<td>4</td>
<td>128-bit</td>
<td>0.24 ns</td>
<td>0.30 ns</td>
</tr>
<tr>
<td>256 (st-pes)</td>
<td>8</td>
<td>256-bit</td>
<td>0.37 ns</td>
<td>0.50 ns</td>
</tr>
</tbody>
</table>

- $T_{StVRF} \approx T_{BVRF} + T_{VRA}$
- **st-opt**: StVEC inst models delay of 1 SIMD inst
  - $T_{StVRF} \leq T_{cycle}$
- **st-pes**: StVEC inst models delay of 2 dependent SIMD inst
  - $T_{StVRF} \leq 2 \times T_{cycle}$

Ex1: $F = 3$GHz, $T_{cycle} = 0.33$ns $\Rightarrow T_{StVRF} < T_{cycle} \Rightarrow$ No overhead!

Ex2: $F = 3$GHz, $T_{cycle} = 0.33$ns $\Rightarrow T_{StVRF} > T_{cycle} \Rightarrow$ Extra cycle overhead!

Ex3: $F = 2$GHz, $T_{cycle} = 0.50$ns $\geq T_{StVRF} \Rightarrow$ No overhead!
StVEC Hardware Overhead

StVRF access time in 45nm CMOS

- $T_{BVRF}$: SRAM model in CACTI
- $T_{VRA}$: circuit synthesis/layout by Synopsys Design Compiler

<table>
<thead>
<tr>
<th>#Regs.</th>
<th>#Banks</th>
<th>Reg. size</th>
<th>BVRF</th>
<th>StVRF</th>
</tr>
</thead>
<tbody>
<tr>
<td>128 (st-opt)</td>
<td>4</td>
<td>128-bit</td>
<td>0.24 ns</td>
<td>0.30 ns</td>
</tr>
<tr>
<td>256 (st-pes)</td>
<td>8</td>
<td>256-bit</td>
<td>0.37 ns</td>
<td>0.50 ns</td>
</tr>
</tbody>
</table>

- $T_{StVRF} \approx T_{BVRF} + T_{VRA}$
- **st-opt**: StVEC inst models delay of 1 SIMD inst
  - $T_{StVRF} \leq T_{cycle}$
  - **Ex1**: $F = 3$GHz, $T_{cycle} = 0.33$ns $> T_{StVRF} \Rightarrow$ No overhead!
- **st-pes**: StVEC inst models delay of 2 dependent SIMD inst
  - $T_{StVRF} \leq 2 \times T_{cycle}$
StVEC Hardware Overhead

StVRF access time in 45nm CMOS

- $T_{BVRF}$: SRAM model in CACTI
- $T_{VRA}$: circuit synthesis/layout by Synopsys Design Compiler

<table>
<thead>
<tr>
<th>#Regs.</th>
<th>#Banks</th>
<th>Reg. size</th>
<th>BVRF</th>
<th>StVRF</th>
</tr>
</thead>
<tbody>
<tr>
<td>128 (st-opt)</td>
<td>4</td>
<td>128-bit</td>
<td>0.24 ns</td>
<td>0.30 ns</td>
</tr>
<tr>
<td>256 (st-pes)</td>
<td>8</td>
<td>256-bit</td>
<td>0.37 ns</td>
<td>0.50 ns</td>
</tr>
</tbody>
</table>

- $T_{StVRF} \approx T_{BVRF} + T_{VRA}$
- **st-opt:** StVEC inst models delay of 1 SIMD inst
  - $T_{StVRF} \leq T_{cycle}$
  - Ex1: $F = 3$GHz, $T_{cycle} = 0.33$ns $> T_{StVRF} \Rightarrow$ No overhead!
- **st-pes:** StVEC inst models delay of 2 dependent SIMD inst
  - $T_{StVRF} \leq 2 \times T_{cycle}$
  - Ex2: $F = 3$GHz, $T_{cycle} = 0.33$ns $< T_{StVRF} \Rightarrow$ Extra cycle overhead!
StVEC Hardware Overhead

StVRF access time in 45nm CMOS

- $T_{BVRF}$: SRAM model in CACTI
- $T_{VRA}$: circuit synthesis/layout by Synopsys Design Compiler

<table>
<thead>
<tr>
<th>#Regs.</th>
<th>#Banks</th>
<th>Reg. size</th>
<th>BVRF</th>
<th>StVRF</th>
</tr>
</thead>
<tbody>
<tr>
<td>128 (st-opt)</td>
<td>4</td>
<td>128-bit</td>
<td>0.24 ns</td>
<td>0.30 ns</td>
</tr>
<tr>
<td>256 (st-pes)</td>
<td>8</td>
<td>256-bit</td>
<td>0.37 ns</td>
<td>0.50 ns</td>
</tr>
</tbody>
</table>

- $T_{StVRF} \approx T_{BVRF} + T_{VRA}$
- **st-opt**: StVEC inst models delay of 1 SIMD inst
  - $T_{StVRF} \leq T_{cycle}$
  - Ex1: $F = 3$GHz, $T_{cycle} = 0.33$ns $> T_{StVRF} \Rightarrow$ No overhead!
- **st-pes**: StVEC inst models delay of 2 dependent SIMD inst
  - $T_{StVRF} \leq 2 \times T_{cycle}$
  - Ex2: $F = 3$GHz, $T_{cycle} = 0.33$ns $< T_{StVRF} \Rightarrow$ Extra cycle overhead!
  - Ex3: $F = 2$GHz, $T_{cycle} = 0.50$ns $\geq T_{StVRF} \Rightarrow$ No overhead!
Conclusion

Take-home Message

- Vectorization of stencils is expensive
  - Previous solutions: unaligned loads or shuffle instructions!

- **Our solution: StVEC** – new vector instruction extension
  - Fast execution of stencils
  - Small hardware changes
  - Eliminating unaligned loads
## Conclusion

**Take-home Message**

- Vectorization of stencils is expensive
  - Previous solutions: unaligned loads or shuffle instructions!

- **Our solution: StVEC** – new vector instruction extension
  - Fast execution of stencils
  - Small hardware changes
  - Eliminating unaligned loads

- Performance evaluation with existing x86 vector ISAs
  - Optimistic (1 StVEC inst $\approx$ 1 SIMD inst): 20% to 2.47x
  - Pessimistic (1 StVEC $\approx$ 2 dependent SIMD insts): 7% to 2.26x

- Best fit for 128-bit wide vector computations
  - May require additional pipeline stage(s) for wider vectors
Questions?