Vectorization Past Dependent Branches Through Speculation

Majedul Haque Sujon
R. Clint Whaley
Center for Computation & Technology (CCT),
Louisiana State University (LSU).
University of Texas at San Antonio (UTSA)*
&
Qing Yi
Department of Computer Science,
University of Colorado – Colorado Springs (UCCS).

*part of the research work had been done when the authors were there
Outline

• Motivation
• Speculative Vectorization
• Integration within Our Framework
• Experimental Results
• Related Work
• Conclusions
Motivation

• SIMD vectorization is required to attain high performance on modern computers
• Many loops cannot be vectorized by existing techniques
  – Only 18–30% loops from two benchmarks can be auto-vectorized – Maleki et al.[PACT’11]
  – A key inhibiting factor is control hazard
→ We introduce a new technique for vectorization past dependent branches --- a major source where existing techniques fail
Example: SSQ Loop

```c
for(i=1; i<=N; i++)
{
    ax = X[i];
    ax = ABS & ax;
    if (ax > scal)
    {
        t0 = scal/ax;
        t0 = t0*t0;
        ssq = 1.0+t1;
        scal = ax;
    }
    else
    {
        t0 = ax/scal;
        ssq += t0*t0;
    }
}
SSQ Loop (NRM2)
```
Variable Analysis (1)

scal: defined
ssq: used before defined

ax = X[i];
ax = ABS & ax;
if (ax > scal)
GOTO L2;

L2:
t0 = scal/ax;
t0 = t0*t0;
ssq = 1.0+t1;
scal = ax;

Path-1

Path-2

t0 = ax/scal;
ssq += t0*t0;

scal: Recurrent variable [unvectorizable pattern]

ssq: Recurrent variable [unvectorizable pattern]

Statements that operate on scal are not vectorizable
Variable Analysis (2)

\[ \text{ax} = X[i]; \]
\[ \text{ax} = \text{ABS} \& \text{ax}; \]
\[ \text{if (ax > scal)} \]
\[ \text{GOTO L2;} \]

Path-2

L2:
\[ t0 = \text{scal}/\text{ax}; \]
\[ t0 = t0\times t0; \]
\[ \text{ssq} = 1.0+t1; \]
\[ \text{scal} = \text{ax}; \]

Path-1

\[ t0 = \text{ax}/\text{scal}; \]
\[ \text{ssq} += t0\times t0; \]

\textbf{scal}: Recurrent variable
[unvectorizable pattern]

\textbf{ssq}: Recurrent variable
[unvectorizable pattern]

\textbf{ssq}: reduction but defined in the other path

\textbf{ssq} is defined again

considering both paths, statements that operate on \textbf{ssq} are not vectorizable
Analysis of Path–1

Path–1:

\[ ax = X[i]; \]
\[ ax = \text{ABS} & ax; \]
\[ \text{if} \ (ax > \text{scal}) \]
\[ \text{GOTO L2;} \]

\[ t0 = ax/\text{scal}; \]
\[ \text{ssq} += t0 \times t0; \]

\text{ssq}: reduction variable (vectorizable)

\text{scal}: Invariant
\text{ssq}: Reduction
\text{ABS}: Invariant
\text{t0}, \text{ax}: private variable

Path–1: Vectorizable
Speculative Vectorization

Vectorize past branches using speculation:
1. Vectorize a chosen path —-- *speculate* it will be taken in *consecutive* loop iterations (e.g. vector length iterations).

2. When speculation fails, re-evaluate mis- vectorized iterations using scalar operations [*Scalar Restart*].
Vectorized Loop Structure

Scalar Restart

Vector Path

- vector-prologue (initialization)
- vector-backup (if needed)
- vector-body
- vector-loop-update
- vector-epilogue (Reduction)
Vectorized Loop Structure

Scalar Restart

vector-restore (if needed)

vector-to-scalar (reduction)

scalar loop of vector-length # of iterations

scalar-to-vector update

vector-prologue (initialization)

vector-backup (if needed)

vector-body

vector-loop-update

Vector Path

vector-epilogue (Reduction)
Example Vectorized Code (SSQ)

SCALAR_RESTART:

```c
/* vector-to-scalar */
ssq = sum(Vssq[0:3]);

/* scalar loop */
for(j=0; j<4; j++)
{
    ax = X[i];
    ax = ABS & ax;
    if (ax > scal)
    {
        t0 = scal/ax;
        t0 = t0*t0;
        ssq = 1.0+t1;
        scal = ax;
    }
    else
    {
        t0 = ax/scal;
        ssq += t0*t0;
    }
}

/* scalar-to-vector */
Vssq=[ssq,0.0,0.0,0.0,0.0];
Vscal=[scal,scal,scal,scal,scal];
```

LOOP:

```c
/* vector-body */
Vax = X[i:i+3];
Vax = VABS & Vax;
if(VEC_ANY_GT(Vax,Vscal)
    GOTO SCALAR_RESTART;
Vt0 = Vax/Vscal;
Vssq += Vt0*Vt0;

/* vector-loop-update */
i+=4;
if(i<=N4) GOTO LOOP;

/* vector-to-scalar */
Vssq = [ssq,0.0,0.0,0.0,0.0];
Vscal = [scal,scal,scal,scal,scal];
```

September 12, 2013
Integration within the iFKO framework

- iFKO (Iterative Floating Point Kernel Optimizer)

Why necessary:
- To find the best path to speculate for SV
- To apply SV only when profitable
Results: SV vs Scalar

AVX: float:8, double: 4
Data: in-L2, random [-0.5,0.5], sin/cos [0, 2π]
SV & Scalar : auto tuned

Machine: Intel Xeon CPU E5–2620

September 12, 2013
Results: SV vs Scalar

6.8 x / 3.4 x

Speedup of AMAX/IAMAX: float 6.8x, double 3.4x

September 12, 2013

PACT'2013
Results: SV vs Scalar

NRM2: Not vectorizable by prior methods
4.18x (float), 2.08x (double)
Results: SV vs Scalar

![Bar chart showing speedup over scalar for different operations.]

- **AMAX**: Single 6.81, Double 6.83
- **IAMAX**: Single 3.46, Double 3.47
- **NRM2 (SSQ)**: Single 2.08, Double 4.18
- **ASUM**: Single 0.93, Double 0.92
- **IRK1AMAX**: Single 5.96, Double 5.16
- **IRK2AMAX**: Single 5.86, Double 3.2
- **IRK3AMAX**: Single 5.43, Double 3.01
- **SIN**: Single 1.01, Double 0.08
- **COS**: Single 1.01

- **September 12, 2013**
- **PACT’2013**
Results: SV vs Scalar

Slowdown up to 8% for ASUM and COS
Vectorization Strategies in iFKO

- VMMR (Vectorization after Max/Min Reduction):
  - Eliminating Max/Min conditionals with vmax/vmin instruction
- VRC (Vectorization with Redundant Computation):
  - Redundant computation with select/blend operation
  - Only effective if all paths are vectorizable in our implementation
→ SV (Speculative Vectorization):
  - at least one path is vectorizable
Comparing Vectorization Strategies with AMAX

- **VMMR**: only one branch to find max
- **VRC**: minimum redundant operation
- **SV**: strong directionality

AVX: float:8, double: 4
Intel Xeon CPU E5–2620

<table>
<thead>
<tr>
<th></th>
<th>Single</th>
<th>Double</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speedup over scalar</td>
<td>7.08</td>
<td>3.5</td>
</tr>
<tr>
<td></td>
<td>6.46</td>
<td>3.13</td>
</tr>
<tr>
<td></td>
<td>6.81</td>
<td>3.48</td>
</tr>
</tbody>
</table>
Related Work

• If Conversion: J.R. Allen [POPL’83]
  - Control dependence to data dependence

• Bit masking to combine different values from if–else branches: Bik et al.[int. J. PP’02]

• Formalize predicated execution with select/blend operation: Shin et al.[CGO’05]
  - General approach
Conclusions

- Impressive speedup can be achieved when control-flow is directional.
  - Can vectorize some loops effectively when other methods can’t.
    - SSQ (NRM2): 4.18x (float), 2.08x (double)
    - AMAX/IAMAX: 6.8x (float), 3.6 (double)
  - Complimentary to and can be combined with existing other vectorization methods (e.g., VRC)
  - Specialize hardware is not needed
- Future work
  - Investigate combining vectorization strategies
  - Try under-speculation as veclen increases
  - Speculative vectorization of multiple paths
  - Loop specialization: switch to scalar loop when mispeculation is frequent