Exploiting Superword Level Parallelism (SLP) with Multimedia Instruction Sets

Larsen, S., & Amarasinghe, S. (2000). Exploiting Superword Level Parallelism with Multimedia Instruction Sets. *SIGPLAN Not., 35(5), 145–156.*

Presented by:

Group 16 - Joel Harrison, Owen Hoffend, Rohan Naik, Daniel Wan

Table of Contents

Introduction - SIMD & Vectorization

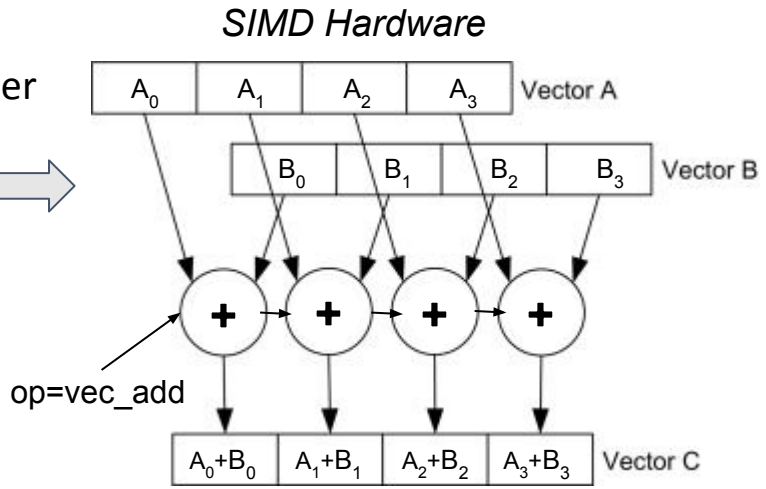
- SIMD Instructions: Single Instruction, Multiple Data
  - Hardware feature for parallel execution (ex: Intel's Advanced Vector Extensions, AVX)
  - Execute same op on a vector of values, write back a vector a values
- Vectorization:
  - Process of exposing SIMD execution opportunities
  - Auto-vectorization: Vectorization performed by compiler

Sequential Code

$C_0 = A_0 + B_0$
$C_1 = A_1 + B_1$
$C_2 = A_2 + B_2$
$C_3 = A_3 + B_3$

Vectorized Code

$C_{[0:3]} = A_{[0:3]} +_{vec} B_{[0:3]}$



Introduction - Why SLP?

Auto-Vectorization was failing because...

- Vectorization was done on a case-by-case basis
- Often failed to vectorize complex code
- Required complex loop optimizations

SLP Vectorization set out to...

- Generalize the vectorization process
- Identify more parallelization opportunities
- Keep the algorithm simple and robust
- Operate at the Basic Block level

Original Loop:

```
for (i=0; i<16; i++) {
  localdiff = ref[i] - curr[i];
  diff += abs(localdiff);
}
```

Loop after scalar expansion and loop fission

```
for (i=0; i<16; i++) {
  T[i] = ref[i] - curr[i];
}
```

Vectorizable

```
for (i=0; i<16; i++) {
  diff += abs(T[i]);
}
```

NOT Vectorizable

SLP Example: Uncovering SLP via Loop Unrolling

Original Loop:

```
for (i=0; i<16; i+=4){
  localdiff = ref[i] - curr[i];
  diff += abs(localdiff);
}
```

After Unrolling:

```
for (i=0; i<16; i+=4){
  localdiff0 = ref[i+0] - curr[i+0];
  localdiff1 = ref[i+1] - curr[i+1];
  localdiff2 = ref[i+2] - curr[i+2];
  localdiff3 = ref[i+3] - curr[i+3];

  diff0 += abs(localdiff0);
  diff1 += abs(localdiff1);
  diff2 += abs(localdiff2);
  diff3 += abs(localdiff3);
}
```

- SLP alg. operates at BB level, but loop unrolling helps
  - Unrolled instrs tend to have the same ops
  - Also tends to expose adjacent references
- Can pack groups of instructions w/ same ops
- Pack operands using special HW instructions pack()/unpack()

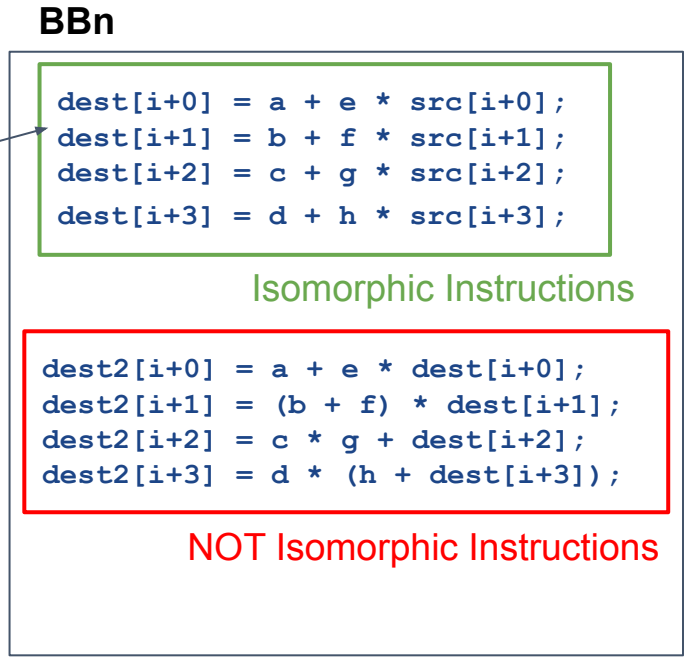
Vectorized Pseudocode

```
for (i=0; i<16; i+=4){
  curr_v = pack(curr[i+0:i+3]);
  ref_v = pack(ref[i+0:i+3]);
  localdiff_v = curr_v - ref_v; //SIMD
  diff_v += abs(localdiff_v);   //SIMD (couldn't do this before)
}
```

SLP Vectorization Algorithm Overview

Overall algorithm for performing SLP vectorization is:

1. **Perform Loop Unrolling**
2. **Memory Alignment Analysis** for each BB
   - Want to find adjacent references: A[i], A[i+1]
   - Often profitable to vectorize these (data locality)
   - Create a "*PackSet*" that holds pairs of adjacent refs in BB
3. **Extend** the *PackSet*:
   - Merge pairs of adjacent refs
   - Add independent **isomorphic instrs** from the BB
   - Isomorphic = **Same ops, same order**. Can be packed.
4. **Schedule** vector instrs from the set following DU chains



Building the PackSet

- Within a BB: Find the set of pairs of adjacent isomorphic instructions, s.t. each instr appears at most **twice** (once as a "top" and again as a "bottom")
  - Register-only ops are always adjacent
- Combine pairs into larger groups
- "Top" of one pair must equal "Bottom" of other pair

BBn (instrs)

```
b = a[i]
d = b+c
e = a[i+1]
g = e+f
h = a[i+2]
k = h+j
```

BBn PackSet (pairs)

```
(1)  b = a[i]      (3)  d = b+c
     e = a[i+1]         g = e+f

(2)  e = a[i+1]    (4)  g = e+f
     h = a[i+2]         k = h+j
```

(BBn) PackSet Groups

```
(1)  b = a[i]      (2)  d = b+c
     e = a[i+1]         g = e+f
     h = a[i+2]         k = h+j
```

Scheduling

- Packing and unpacking SIMD vectors is **costly**
- Rough speedup measure:

$$speedup_{SLP} = \frac{t_{seq}}{t_{pack} + t_{unpack} + t_{SIMD}}$$

- **Key Idea: Schedule groups following DU chains**
- Reduces the required number of packs/upacks.
  - Results of previous SIMD ops will be ready in a SIMD register

(BBn) PackSet Groups

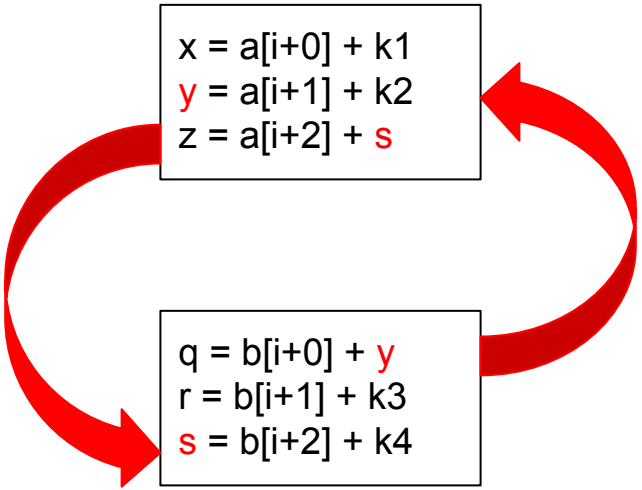
```
(1)  b = a[i]      (2)  d = b+c
     e = a[i+1]         g = e+f
     h = a[i+2]         k = h+j
```

(BBn) Vectorized Schedule

```
a_v = pack(a[i+0:i+2])
beh_v = a_v //SIMD op
cfj_v = pack(c, f, j)
dgk_v = beh_v + cfj_v //SIMD op
d, g, k = unpack(dgk_v)
```

Scheduling Continued

- Schedule statements by original order (when possible)
  - Only after all dependencies have been fulfilled
- For cycle, split apart group with earliest statement



Simple Vectorizing Compiler

- Simplified version of SLP
- Limit packing to unrolled version of same statement
    - Only one possible grouping for each statement
- Not great for long vector architectures
    - Unroll factor has to be consistent with vector size
    - Unrolling could make basic blocks that overwhelm the analysis and code generator

Results

- **Measured % of dynamic instructions eliminated** for a theoretical datapath
  - Need *n* - *1* instructions to move *n* values into SIMD register
  - Measured by instrumenting code w/counters
- **Benchmarks consisted of both scientific and multimedia applications**
  - SPEC95fp suite for scientific benchmarks
  - Variety of kernels for multimedia benchmarks, including matrix multiplication and RGB to YUV conversion
- **20-70% of dynamic instructions eliminated** for most benchmarks, fairly evenly distributed

Results

- Significant correlation between instructions eliminated and code w/vector components
  - Most instructions saved were vector operations, fewer instructions saved w/analysis of general loop parallelism
- Multimedia generally showed more instruction elimination
  - Most obvious is RGB to YUV conversion

$$\begin{pmatrix} Y \\ U \\ V \end{pmatrix} = \begin{pmatrix} c_1 & c_2 & c_3 \\ c_4 & c_5 & c_6 \\ c_7 & c_8 & c_9 \end{pmatrix} \cdot \begin{pmatrix} R \\ G \\ B \end{pmatrix}$$

Results

- Also measured performance on a microprocessor supporting the AltiVec instruction set
  - Many benchmarks required double precision floats, which the processor did not have support for
  - Limited set of benchmarks for real-world performance
- Measured performance speedup with parallelized vs unparalleled code

Strengths/Weaknesses & Future Work

Weaknesses:

- Paper written in 2000 (SSE2 introduced in 2000)
    - Support for double precision floats
- Inflexible - SIMD viewed as something mostly for multimedia
    - Lots of matrix multiplication
    - No support for moving data between register files
    - Not all HW supports the required packing/unpacking instructions
- Compiler is still not robust
    - Small source modifications greatly affect parallelism extracted
    - Hard to determine safety
        - Loop-carried dependencies, especially for memory
- Some optimizations are language-specific

Strengths/Weaknesses & Future Work

Strengths:

- Achieved substantial dynamic instruction count reduction on many benchmarks
- Much better support for SIMD today
  - Many instructions in SSE/AVX that support packing of different sizes
  - Support for more data types

Future Work:

- Hard to extend SLP parallelism beyond basic blocks
  - Couldn't find in LLVM - Our project focuses on this issue
  - **Key idea:** Use profile data to build superblocks, then SLP vectorize within them

Questions?