CS 526

Advanced Compiler Construction

https://charithm.web.illinois.edu/cs526/sp2022/
Compiler Auto-vectorization
Vector Machines from 1970s

Cray-I Supercomputer

Table 1. CRAY-1 CPU characteristics summary

<table>
<thead>
<tr>
<th>Computation Section</th>
<th>80 MHz clock</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalar and vector processing modes</td>
<td></td>
</tr>
<tr>
<td>12.5 nanosecond clock period operation</td>
<td></td>
</tr>
<tr>
<td>64-bit word size</td>
<td></td>
</tr>
<tr>
<td>Integer and floating-point arithmetic</td>
<td></td>
</tr>
<tr>
<td>Twelve fully segmented functional units</td>
<td></td>
</tr>
<tr>
<td>Eight 24-bit address (A) registers</td>
<td></td>
</tr>
<tr>
<td>Sixty-four 24-bit intermediate address (B) registers</td>
<td></td>
</tr>
<tr>
<td>Eight 64-bit scalar (S) registers</td>
<td></td>
</tr>
<tr>
<td>Sixty-four 64-bit intermediate scalar (T) registers</td>
<td></td>
</tr>
<tr>
<td>Eight 64-element vector (V) registers (64-bits per element)</td>
<td></td>
</tr>
<tr>
<td>Vector length and vector mask registers</td>
<td></td>
</tr>
<tr>
<td>One 64-bit real time clock (RT) register</td>
<td></td>
</tr>
<tr>
<td>Four instruction buffers of sixty-four 16-bit parcels each</td>
<td></td>
</tr>
<tr>
<td>128 basic instructions</td>
<td></td>
</tr>
<tr>
<td>Prioritized interrupt control</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory Section</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1,048,576 64-bit words (plus 8 check bits per word)</td>
<td></td>
</tr>
<tr>
<td>16 independent banks of 65,536 words each</td>
<td></td>
</tr>
<tr>
<td>4 clock period bank cycle time</td>
<td></td>
</tr>
<tr>
<td>1 word per clock period transfer rate for B, T, and V registers</td>
<td></td>
</tr>
<tr>
<td>1 word per 2 clock periods transfer rate for A and S registers</td>
<td></td>
</tr>
<tr>
<td>4 words per clock period transfer rate to instruction buffers (up to 16 instructions per clock period)</td>
<td></td>
</tr>
</tbody>
</table>


Courtesy silicon valley graphics
8 x 64-word (4096 bit) vector registers
Vector length register and mask register
Single Instruction Multiple Data (SIMD) model
Characteristics of Vector Machines

Large Vectors (4096 bit)
Unbounded Vector Computations (Vector Length)

• Pros:
  • Can exploit large amounts of parallelism
• Cons:
  • Has a high startup cost
  • Not suitable for programs with scattered parallel computations

Fig. 7. Scalar/vector timing.
COST (CLOCK PERIODS/RESULT)
Modern Short Vector Instructions


• Main characteristic:
  • fixed (no vector length register)
  • short width (e.g., 64-bit, 128-bit)

• Packed data formats
  • Allows multiple data types (e.g., int8, int16, int32, double, float)

• Startup cost similar to scalar loads / stores to registers
Modern Short Vector Instructions

- **32-bit scalar only**
  - 1997

- **64-bit vector (MMX)**
  - Intel Pentium II

- **128-bit vector (SSE2)**
  - 2000

- **256-bit vector (AVX2)**
  - Intel Haswell
  - 2011

- **512-bit vector (AVX512)**
  - Intel Xeon
  - 2016

Increase in bit-width

Diversity in Instruction Set
How do we use these hardware features?

```
for(int i=0; i< N; i++)
{
    av[i] = sqrt(bv[i]);
}
```

```
for(int i=0; i< N; i+=4)
{
    av[i] = _mm_sqrt_pd(bv[i]);
    av[i+2] = _mm_sqrt_pd(bv[i+2]);
}
```

- **Compiler Intrinsics**
- **Assembly**
- **Compiler Auto-vectorization**
Auto-vectorization introduction

- Why would we want the compiler to perform vectorization?
  - Exploit hardware features
  - Portable code
  - Support multiple hardware targets

Scalar

```c
for(int i=0; i<N; i++)
a[i] = b[i] + c[i];
```

Compiler

Vector

```c
for(int i=0; i<N; i+=4)
a[i:i+4] = b[i:i+4] + c[i:i+4];
```
Auto-vectorization introduction

- **SIMD** (Single Instruction Multiple Data)
  - Single Scalar Instruction applied to Multiple Data items at the same time
  - Fine grained parallelism
  - Use vectorization to exploit to vector units
  - Easier to find with low to none startup cost

- **SPMD** (Single Program Multiple Data)
  - A chunk of program code applied to Multiple Data items at the same time
  - Coarse grained parallelism
  - Use auto-parallelization to exploit multiple processing units (multi-threaded programs)
  - Difficult to automatically find profitable chunks of code
Auto-vectorization approaches

Two main approaches in the literature. Can you guess which came first?

**Loop Vectorization (Loops)**

```c
for(int i=0; i<N; i++)
a[i] = b[i] + c[i];
```

```c
for(int i=0; i<N; i+=4)
a[i:i+4] = b[i:i+4] + c[i:i+4];
```

**SLP Vectorization (Basic Blocks*)**

```c
a[0] = b[0] + c[0];
a[1] = b[1] + c[1];
```

```c
a[0:2] = b[0:2] + c[0:2];
```

*many works have extended this to work with Control Flow
Loop Vectorization (origins)

• Kennedy and Allen, “Automatic Translation of Fortran Programs to Vector Form”, 1987 ACM

• Fortran to Fortran 8x Source to Source Compiler focused on Vectorization

• **Main Idea**: You can vectorize statements with no self-dependence

```plaintext
do 20 i = 1, 100
   ki = i
   do 10 j = 1, 300, 3
      ki = ki + 2
   s3 = u(j) = u(j) * w(ki)
   s4 = v(j + 3) = v(j) + w(ki)
10 continue
20 continue
```

Can we vectorize S3, S4?
Loop Vectorization

\[ S_1 \]
\[
\begin{align*}
\text{DO} 20 & \text{ I} = 1, 100 \\
& \text{KI} = I \\
\text{DO} 10 & \text{ J} = 1, 300, 3 \\
& \text{KI} = \text{KI} + 2 \\
S_2 & \quad U(J) = U(J) \cdot W(\text{KI}) \\
S_3 & \quad V(J + 3) = V(J) + W(\text{KI}) \\
10 & \quad \text{CONTINUE} \\
20 & \quad \text{CONTINUE}
\end{align*}
\]

\[ S_4 \]
\[
\begin{align*}
\text{DO} 20 & \text{ I} = 1, 100 \\
S_5 & \quad U(1:298:3) = U(1:298:3) \cdot W(2:1 + 200:2) \\
\text{DO} 10 & \text{ j} = 1, 100 \\
S_6 & \quad V(3 \cdot j + 1) = V(3 \cdot j - 2) + W(1 + 2 \cdot j) \\
10 & \quad \text{CONTINUE} \\
20 & \quad \text{CONTINUE}
\end{align*}
\]

- Other pre-processing transformations needed for this example:
  - Loop normalization
  - Induction variable substitution

\[ S_3 \quad \text{Can vectorize} \]
\[ S_4 \quad \text{Cannot vectorize} \]
Loop Vectorization

- Now, let’s see more extensions!
- Some of the slides are inspired by slides used by Maleki, Gropp and Padua
- **Main Idea:** You can vectorize statements with no self-dependence
- Vectorization Factor (VF): The number of elements in a vector register of a particular type
- For the rest of the examples, we assume VF=4;
Loop Vectorization

• What if you have self-dependencies, can you still vectorize?
• You can perform (partial) loop vectorization
  • Dependence transformations
  • Removing Dependencies
  • Changing the algorithm
  • Focusing only on inner loops
• Other Advanced Loop Vectorization techniques
  • Outer-loop vectorization
  • Vectorizing interleaved data accesses
  • Vectorizing with vector permutation instructions
Kinds of Data Dependence

**Direct Dependence**

S1: \( X = \ldots \)
S2: \( \ldots = X + \ldots \)

**Anti-dependence**

S1: \( \ldots = X \)
S2: \( X = \ldots \)

**Output Dependence**

S1: \( X = \ldots \)
S2: \( X = \ldots \)
Dependence Graph for Loops

(Repeat) A dependence graph is a graph with:

• one node per statement, and

• a directed edge from S1 to S2 if there is a data dependence between S1 and S2 (where the instance of S2 follows the instance of S1 in the relevant execution).

For loops: dependence graph is a summary of unrolled dependencies for different iterations

• Some (detailed) information may be lost
Example 1:

```c
for (int i =0; i < N; i++){
  S1: a[i] = b[i] + c[i];
  S2: c[i] = a[i] + 2;
}
```

Can we **vectorize**?  Yes

```c
for (int i =0; i < N; i+=4){
  S1: a[i:i+4] = b[i:i+4] + c[i:i+4];
  S2: c[i:i+4] = a[i:i+4] + 2;
}
```

Are these **loop independent (LI)** or **loop carried (LC)** dependencies?

**Loop independent**
Example 2:

```java
for (int i = 0; i < N; i++) {
    S1: b[i] = b[i] + c[i];
    S2: a[i] = a[i-1]*a[i-2] + b[i];
    S3: c[i] = a[i] + 1;
}
```

Can we vectorize? No

Can we vectorize partially? Yes

Are these loop independent (LI) or loop carried (LC) dependencies?
Example 2:

for (int i =0; i < N; i++){  
    S1: b[i] = b[i] + c[i];  
    S2: a[i] = a[i-1]*a[i-2] + b[i];  
    S3: c[i] = a[i] + 1;  
}

for (int i =0; i < N; i++){  
    S1: b[i] = b[i] + c[i];  
}

for (int i =0; i < N; i++){  
    S2: a[i] = a[i-1]*a[i-2] + b[i];  
}

for (int i =0; i < N; i++){  
    S3: c[i] = a[i] + 1;  
}
Example 2: Loop distribution

for (int i = 0; i < N; i++){
    S1: b[i] = b[i] + c[i];
    S2: a[i] = a[i-1]*a[i-2] + b[i];
    S3: c[i] = a[i] + 1;
}

Is this beneficial?

for (int i = 0; i < N; i+=4){
    S1: b[i:i+4] = b[i:i+4] + c[i:i+4];
}

for (int i = 0; i < N; i++){
    S2: a[i] = a[i-1]*a[i-2] + b[i];
}

for (int i = 0; i < N; i+=4){
    S3: c[i:i+4] = a[i:i+4] + 1;
}
Example 3:

```c
for (int i = 0; i < N; i++){
    S1: a = b[i] + 1;
    S2: c[i] = a + 2;
}
```

Can we vectorize? **No**

Can we remove self-dependencies? **Yes**

Are these loop independent (LI) or loop carried (LC) dependencies?
Example 3: Scalar Expansion

for (int i =0; i < N; i++){  
    S1: a = b[i] + 1;  
    S2: c[i] = a + 2;  
}

In general, we can eliminate false dependence by adding more storage

for (int i =0; i < N; i++){  
    S1: a[i] = b[i] + 1;  
    S2: c[i] = a[i] + 2;  
}
Example 4:

```c
for(int i = 0; i < N; i++){
    for (int j =0; j < N; j++){
        S1: a[i][j] = a[i][j] + a[i-1][j];
    }
}
```

Can we vectorize?  **No**

What if we focus one loop level at a time?  **Yes**
Example 4: Focus on the innermost loop

```c
for(int i = 0; i < N; i++){
    for (int j =0; j < N; j++){
        S1: a[i][j] = a[i][j] + a[i-1][j];
    }
}
```

Vectorize Loop (j)

```c
for(int i = 0; i < N; i++)
    for (int j =0; j < N; j+=4)
        S1: a[i][j:j+4] = a[i][j:j+4] + a[i-1][j:j+4];
```

Vectorization can happen only in the innermost access level
Can you vectorize this?

for(int i = 0; i < N; i++){
    \[ S1: \ a[i] = b[i] + c[i]; \]
    \[ S2: \ d[i] = a[i+1] + 1; \]
}

Can you vectorize this loop? **Yes**, since there are no self-dependencies

for(int i = 0; i < N; i+=4){
    \[ S1: \ a[i:i+4] = b[i:i+4] + c[i:i+4]; \]
    \[ S2: \ d[i:i+4] = a[i+1:i+5] + 1; \]
}

We created new dependencies 😞
Can you vectorize this?

Reorder the statements

```c
for(int i = 0; i < N; i++){
    S2: d[i] = a[i+1] + 1;
    S1: a[i] = b[i] + c[i];
}
```

Dependencies are preserved 😊

```c
for(int i = 0; i < N; i+=4){
    S2: d[i:i+4] = a[i+1:i+5] + 1;
    S1: a[i:i+4] = b[i:i+4] + c[i:i+4];
}
```
Summary: Loop Vectorization

• Statements without (true) self-dependencies can be vectorized
• However, you may need to do code transformations to expose the parallelism
• In general loop vectorizers,
  • **Acyclic** dependencies
    • **Forward**: Always vectorized
    • **Backward**: reorder and sometimes gets vectorized
  • **Cyclic** dependencies
    • **Forward**: false self-dependencies vectorized
    • **Backward**: Usually not vectorized
  • **Partially vectorize** non self-dependent statements
Outer-loop Vectorization

Can you vectorize this loop?  **No**

First need to expose the outer-loop parallelism

```java
for(int i = 0; i < N; i++){
    for (int j = 0; j < N; j++){
        a[i] = a[i] + b[j];
    }
}
```

**Unroll**

```java
for(int i = 0; i < N; i+=2){
    for (int j = 0; j < N; j++){
        a[i] = a[i] + b[j];
    }
    for (int j = 0; j < N; j++){
        a[i+1] = a[i+1] + b[j];
    }
}
```

**Fuse (jam)**

```java
for(int i = 0; i < N; i+=2){
    for (int j = 0; j < N; j++){
        a[i+1] = a[i+1] + b[j];
    }
}
```
Vectorizing interleaved data

```c
for(int i = 0; i < len; i++){
    c[2i] = a[2i]*b[2i] - a[2i+1]*b[2i+1];
    c[2i+1] = a[2i]*b[2i+1] + a[2i+1]*b[2i];
}
```

Let's visualize data accesses

What's wrong with this?
Vectorizing interleaved data

```c
for(int i = 0; i < len; i++){
    c[2i]   = a[2i]*b[2i] - a[2i+1]*b[2i+1];
    c[2i+1] = a[2i]*b[2i+1] + a[2i+1]*b[2i];
}
```

What’s wrong with this?

Let’s visualize data accesses
Vectorizing interleaved data

\[ a[2i] \times b[2i] - a[2i+1] \times b[2i+1] ; \]

**Interleaved storage**
Vectorizing interleaved data

\[ c[2i] = a[2i]*b[2i] - a[2i+1]*b[2i+1]; \]
\[ c[2i+1] = a[2i]*b[2i+1] + a[2i+1]*b[2i]; \]

---

Nuzman et. al. “Auto-vectorization of interleaved data for SIMD”, PLDI 2006
### Vectorization Capability

<table>
<thead>
<tr>
<th>Loops</th>
<th>Compiler</th>
<th>XLC</th>
<th>ICC</th>
<th>GCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td></td>
<td>159</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vectorized</td>
<td></td>
<td>74</td>
<td>75</td>
<td>32</td>
</tr>
<tr>
<td>Not vectorized</td>
<td></td>
<td>85</td>
<td>84</td>
<td>127</td>
</tr>
<tr>
<td>Average Speed Up</td>
<td></td>
<td>1.73</td>
<td>1.85</td>
<td>1.30</td>
</tr>
</tbody>
</table>

![Venn Diagram](image)

Maleki et. al, “An Evaluation of Vectorizing Compilers”, PACT 2011
Limitations of Loop Vectorization

• Works only on Loops; misses finer grain parallelism in BBs
• Fragile in the presence of control flow
• Highly susceptible to data layouts
• Partial Vectorization requires co-operation of multiple loop transformations
• More suited for large vector machines with abundant parallelism
SLP Vectorization

**Independent** and **Isomorphic** statements can be vectorized

Scalar Code

\[
\begin{align*}
    a[0] &= b[0] + c[0] \\
\end{align*}
\]

Vector Code

**Single Instruction Multiple Data (SIMD)**

\[
\{a[0], a[1]\} = \{b[0], b[1]\} + \{c[0], c[1]\}
\]

Vector Packs

Larsen & Amarasinghe “Exploiting Superword Level Parallelism with Multimedia Instruction Sets” [PLDI’00]
SLP Vectorization

- **Independent**: No use-def relationship between the statements
- **Isomorphic**: Same operation and operands of the same type
- **Pack**: tuple \(<s_1, \ldots, s_n>\) where \(s_1, s_2, \ldots, s_n\) are independent isomorphic statements in a basic block.
- **Packset**: is a set of Packs

\[
\begin{align*}
a[0] &= b[0] + c[0]; & \checkmark \\
a[1] &= b[1] + c[1]; & \checkmark \\
a[0] &= b[0] + c[0]; & \times \\
a[1] &= b[1] \times c[1]; & \times \\
a[0] &= b[0] + c[0]; & \times \\
a[1] &= a[0] + c[1]; & \times
\end{align*}
\]

Larsen & Amarasinghe “Exploiting Superword Level Parallelism with Multimedia Instruction Sets” [PLDI’00]
Costs and Benefits

• **Vector Savings**
  • Executing one vectorized statement is beneficial compared to executing multiple scalar statements.
  • For a pack $P = \langle S_1, \ldots, S_N \rangle$  
    $$\text{vec\_cost}(P) = \sum_{i=1}^{N} \text{scalar\_cost}(S_i)$$

• **Packing Cost**
  • Explicitly pack non-vectorizable statements into a vector register using overhead packing instructions
  • e.g., $P = \text{pack}(s1,s2)$; packs $s1$ and $s2$ into vector form $P$.

• **Unpacking Cost**
  • Explicitly unpack values in a vector register to its scalar components.
  • e.g., $s1 = \text{unpack}(P,1)$; unpack 1st scalar value from pack $P$.

• **Goal of any SLP vectorization algorithm is to find a profitable set of packs**
SLP advantages over Loop Vectorization

```
struct Color{int r,g,b;}
Color color[len];

for(int i = 0; i < len; i++){
    color[i].r = color[i].r + 1;
    color[i].g = color[i].g + 1;
    color[i].b = color[i].b + 1;
}
```

Can loop vectorization vectorize this? **No**

Can SLP vectorization vectorize this? **Yes**

* Can use interleaved data vectorization
Larsen & Amarasinghe algorithm

- Perform loop unrolling (expose parallelism inside loops)
- Create the initial packset with packs of adjacent memory loads and stores (pairs of statements at a time)
- Until packset does not change
  - Follow use-def chains of existing packs and create new packs of operands
  - Follow def-use chains of existing packs and create new packs of uses
  - Add these packs to the packset only if their addition is profitable
- Combine packs until vector length is full
- Schedule the packs in the final packset
  - Schedule packs in a top-down manner in the order of their dependencies
Larsen & Amarasinghe algorithm

Initial packs
{L[5], L[6]}  {L[1], L[2]}
{L[6], L[7]}  {L[2], L[3]}
{L[3], L[4]}

Extend packs (use-def and def-use chains)
{A1, A2}  {A6, A4}
{A2, A3}  {A3, A1}
{A4, A5}
{A5, A6}

Assume VF=2

For brevity load statements are not shown. Refer to the ordering for how they are loaded.
Larsen & Amarasinghe algorithm

Schedule

S4 : A4 = L[1] - A2

Assume VF=2

Scalar

packs

{L[5], L[6]}
{L[6], L[7]}
{L[1], L[2]}
{L[2], L[3]}
{L[3], L[4]}

{A1, A2}
{A2, A3}
{A4, A5}
{A5, A6}
{A6, A4}
{A3, A1}
Larsen & Amarasinghe algorithm

Scalar code

S1 : \( A_1 = \frac{L[5]}{L[2]} \)
S2 : \( A_2 = \frac{L[6]}{L[3]} \)
S3 : \( A_3 = \frac{L[7]}{L[4]} \)
S4 : \( A_4 = A_1 - A_2 \)
S5 : \( A_5 = L[2] - A_3 \)
S6 : \( A_6 = L[3] - A_1 \)

Vector code

SV1 : \( \{A_1, A_2\} = \{L[5], L[6]\} / \{L[2], L[3]\} \)
SU1 : \( A_1 = \text{unpack}(SV1, 1) \)
SU2 : \( A_2 = \text{unpack}(SV1, 2) \)
SP1 : \( \{A_2, A_3\} = \text{pack}(A_2, A_3) \)
SV2 : \( \{A_4, A_5\} = \{L[1], L[2]\} - \{A_2, A_3\} \)
S4 : \( A_6 = L[3] - A_1 \)

Instruction Breakdown

Scalar = 7 + 3 + 3 = 13

Instruction Breakdown

4 vector
1 packing 12
2 unpacking
5 scalar
Larsen & Amarasinghe algorithm

Schedule

Ordering 1

S4 : A4 = L[1] - A2

Ordering 2

L[5]  
L[6]  
A1  
A2  
L[3]  
L[4]  
A3  
A4  
L[1]  
L[2]  
L[7]  
A1  
A2  
A3  
A4  
A5  
A6  
L[5]  
L[6]  
L[7]  
L[1]  
L[2]  
L[3]  
L[4]  
L[5]  
L[6]  
L[7]  
A1  
A2  
A3  
A4  
A5  
A6

Algorithm highly susceptible to the statement ordering 😞
Holistic SLP Vectorizer

• Choose which packs to materialize based on pack reuse; not ordering

<table>
<thead>
<tr>
<th>S1</th>
<th>A1 = L[5] / [L[2]]</th>
</tr>
</thead>
<tbody>
<tr>
<td>S4</td>
<td>A4 = L[1] - A2</td>
</tr>
<tr>
<td>S5</td>
<td>A5 = [L[2]] - A3</td>
</tr>
<tr>
<td>S6</td>
<td>A6 = [L[3]] - A1</td>
</tr>
</tbody>
</table>
Holistic SLP Vectorizer

Scalar code

S4 : A4 = L[1] - A2

Vector code

S4 : A4 = L[1] - A2

Instruction Breakdown

scalar = 7 + 3 + 3 = 13

Instruction Breakdown

0 vector
0 packing
0 unpacking
Holistic SLP Vectorizer

Scalar code

S4 : A4 = L[1] - A2

Vector code

SV1 : \{A1, A2\} = \{L[5], L[6]\} / \{L[2], L[3]\}
SV2 : \{A5, A6\} = \{L[2], L[3]\} - \{A3, A1\}

Instruction Breakdown

scalar = 7 + 3 + 3 = 13

Instruction Breakdown

4 vector
0 packing
0 unpacking

Non-isomorphic
Holistic SLP Vectorizer

Scalar code

S4 : A4 = L[1] - A2

Instruction Breakdown

scalar = 7 + 3 + 3 = 13

Vector code

SV1 : \{A1, A2\} = \{L[5], L[6]\} / \{L[2], L[3]\}
SU1 : A1 = unpack(SV1, 1)
S4 : A4 = L[1] - A2
SV2 : \{A5, A6\} = \{L[2], L[3]\} - \{A3, A1\}

Instruction Breakdown

4 vector
0 packing
1 unpacking
Scalar code

S4 : A4 = L[1] - A2

Vector code

SV1 : {A1,A2} = {L[5],L[6]} / {L[2],L[3]}
SU1 : A1 = unpack(SV1,1)

SP1 : {A3,A1} = pack(A3,A1)
S4 : A4 = L[1] - A2
SV2 : {A5,A6} = {L[2],L[3]} - {A3,A1}

Instruction Breakdown

Scalar code

scalar = 7 + 3 + 3 = 13

Vector code

4 vector
1 packing
1 unpacking
Holistic SLP Vectorizer

Scalar code

| S4 | A4 = L[1] - A2 |

Vector code

| SV1 | {A1, A2} = {L[5], L[6]} / {L[2], L[3]} |
| SU1 | A1 = unpack(SV1, 1) |
| SP1 | {A3, A1} = pack(A3, A1) |
| S4  | A4 = L[1] - A2 |
| SV2 | {A5, A6} = {L[2], L[3]} - {A3, A1} |

Instruction Breakdown

- 4 vector
- 1 packing
- 1 unpacking
Holistic SLP Vectorizer

Scalar code

S4 : A4 = L[1] - A2

Vector code

SV1 : {A1,A2} = {L[5],L[6]} / {L[2],L[3]}
SU1 : A1 = unpack(SV1,1)
SU2 : A2 = unpack(SV1,2)
SP1 : {A3,A1} = pack(A3,A1)
S4  : A4 = L[1] - A2
SV2 : {A5,A6} = {L[2],L[3]} - {A3,A1}

Instruction Breakdown

scalar = 7 + 3 + 3 = 13

4 vector
1 packing
2 unpacking
5 scalar
Optimal Strategy (goSLP)

Mendis & Amarasinghe “goSLP: Globally Optimized Superword Level Parallelism Framework”, OOPSLA 2018

Scalar code

S4 : A4 = L[1] - A2

Instruction Breakdown

scalar = 7 + 3 + 3 = 13

Vector code

SV1 : \{A2,A3\} = \{L[6],L[7]\} / \{L[3],L[4]\}
SU1 : L[2] = unpack(SLV1,2)
SU2 : L[3] = unpack(SLV2,1)
SV2 : \{A4,A5\} = \{L[1],L[2]\} - \{A2,A3\}

Instruction Breakdown

5 vector
0 packing
2 unpacking
3 scalar
Other Extensions

• **Control Flow** (Shin et. al. “Superword-Level Parallelism in the Presence of Control Flow”, CGO 2005)


• **Adaptive Vector Width** (Porpodas et. al. “VW-SLP: auto-vectorization with adaptive vector width”, PACT 2018)

• **Better unrolling heuristics** (Rocha et. al. “Vectorization-aware loop unrolling with seed forwarding”, CC 2020)

• **Commutative Operators** (Porpodas et. al. “Look-ahead SLP: auto-vectorization in the presence of commutative operations, CGO 2018)
Diversity of Instruction Sets

```c
__m256d _mm256_addsub_pd (__m256d a, __m256d b)

Synopsis
__m256d _mm256_addsub_pd (__m256d a, __m256d b)
#include <immintrin.h>
Instruction: vaddsubpd ymm, ymm, ymm
CPUID Flags: AVX

Description
Alternatively add and subtract packed double-precision (64-bit) floating-point elements in a to/from packed elements in b, and store the results in dst.

Operation
FOR j := 0 to 3
  i := j*64
  IF ((j & 1) == 0)
  ELSE
  FI
ENDFOR
dst[MAX:256] := 0
```

Performance

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Latency</th>
<th>Throughput (CPI)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ice Lake</td>
<td>4</td>
<td>0.5</td>
</tr>
<tr>
<td>Skylake</td>
<td>4</td>
<td>0.5</td>
</tr>
<tr>
<td>Broadwell</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Haswell</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

Subtraction on the even lanes!
Diversity of Instruction Sets

Synopsis
__m512i __mm512_madd_epi16 (__m512i a, __m512i b)

#include <imintrin.h>
Instruction: vpmaddwd zmm, zmm, zmm
CPUID Flags: AVX512BW

Description
Multiply packed signed 16-bit integers in a and b, producing intermediate signed 32-bit integers. Horizontally add adjacent pairs of intermediate 32-bit integers, and pack the results in dst.

Operation
FOR j := 0 to 15
  i := j*32
  dst[i+31:1] := SignExtend32(a[i+31:i+16]+b[i+31:i+16]) + SignExtend32
ENDFOR
dst[MAX:512] := 0

Performance
<table>
<thead>
<tr>
<th>Architecture</th>
<th>Latency</th>
<th>Throughput (CPI)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Icelake</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>Skylake</td>
<td>5</td>
<td>0.5</td>
</tr>
</tbody>
</table>

Cross lane operation

Complex operation
Vegen Approach

VeGen: A Vectorizer Generator for SIMD and Beyond

Scalar Program

```c
void dot_prod()

int16_t A[4], B[4];
int32_t C[2];

void dot_prod()
{
}
```

Target-specific Vectorizer

Vector Assembly

- `vmovd` xmm0, [A]
- `vmovd` xmm1, [B]
- `pmaddwd` xmm0, xmm1, xmm0
- `vmovd` [C], xmm0

Instruction Description

Chen et. al. “VeGen: a vectorizer generator for SIMD and beyond”, ASPLOS 2021
Can we unify SLP and Loop Vectorization?

All you need is Superword-Level Parallelism: Systematic Control-Flow Vectorization with SLP
Yishen Chen, Charith Mendis and Saman Amarasinghe
PLDI 2022