CS 526 Advanced Compiler Construction

https://charithm.web.illinois.edu/cs526/sp2024/
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>Computation Section</td>
<td></td>
</tr>
<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 128 basic instructions</td>
<td></td>
</tr>
<tr>
<td>Prioritized interrupt control</td>
<td></td>
</tr>
</tbody>
</table>

Memory Section

| 1,048,576 64-bit words (plus 8 check bits per word)                              |              |
| 16 independent banks of 65,536 words each                                        |              |
| 4 clock period bank cycle time                                                  |              |
| 1 word per clock period transfer rate for B, T, and V registers                 |              |
| 1 word per 2 clock periods transfer rate for A and S registers                  |              |
| 4 words per clock period transfer rate to instruction buffers (up to 16 instructions per clock period) | |
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.](image-url)
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**
  - Year: 1997

- **64-bit vector (MMX)**
  - Year: 2000

- **128-bit vector (SSE2)**
  - Year: 2011

- **256-bit vector (AVX2)**
  - Year: 2016

- **512-bit vector (AVX512)**

**Increase in bit-width**

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

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

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

```assembly
sqrtpd 80(%rdx,%rax), %xmm0
sqrtpd 96(%rdx,%rax), %xmm1
vmovdqu %xmm0, 40(%rdi,%rax)
vmovdqu %xmm1, 56(%rdi,%rax)
```
Auto-vectorization introduction

Why would we want the compiler to perform vectorization?

- Exploit hardware features
- Portable code
- Support multiple hardware targets

Scalar

\[
\text{for}(\text{int } i=0; i<N; i++) \quad a[i] = b[i] + c[i];
\]

Vector

\[
\text{for}(\text{int } i=0; i<N; i+=4) \quad a[i:i+4] = b[i:i+4] + c[i:i+4];
\]
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
Brief Announcements

• Progress Reports for Project 2 were due 04/09
• I will send a poll for scheduling meetings on 04/16
• I am traveling to ASPLOS 2024, so final exam dates may be shifted +1/-1 days
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];
```

Vectorize

```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];
```

Vectorize

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

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

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

- Well, not quite....
- If there are **no cycles** in the dependence graph, then the loop can be vectorized
- Is it always the case? and what if there are cycles?
Acyclic: Forward

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

Can you vectorize this loop? Yes

for(int i = 1; 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+3] + 1;
}
Acyclic: Backward

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

Can you \textbf{vectorize} this loop? \textbf{Yes}, since there are no cycles

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

We created new dependencies 😞
Acyclic: Backward

Reorder the statements

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

```
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];
}
```

Dependence Graph

S1 -> S2

Dependencies are preserved 😊
Cyclic: Example 1

for (int i = 2; 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
Cyclic: Example 1

for (int i = 2; 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;
}

Loop Distribution

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

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

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

Is this beneficial?
Cyclic: Example 2:

```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 the cycle? **Yes**
Cyclic: Example 2: Scalar Expansion

```java
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

```java
for (int i = 0; i < N; i++){
    S1: a[i] = b[i] + 1;
    S2: c[i] = a[i] + 2;
}
```
Cyclic: Example 3

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
Cyclic: Example 3: Focus on the innermost loop

```java
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)**

```java
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];
```

**Dependence Graph**

Vectorization can happen only in the innermost access level
Cyclic: Example 4

Can this loop be vectorized? If yes, what transformations need to happen?

```c
for (int i = 0; i < N; i++) {
    S1: b[i] = a[i] + 1.0;
    S2: a[i+1] = b[i] + 2.0;
}
```
Summary: Loop Vectorization

- In general, loops with no dependency cycles can be vectorized
- However, you may need to do code transformations to expose the parallelism
- Specifically, loop vectorizers,
  - **Acyclic** dependencies
    - **Forward**: Always vectorized
    - **Backward**: reorder and sometimes gets vectorized
  - **Cyclic** dependencies
    - Dependence transformations
    - Removing Dependencies
    - Changing the algorithm
    - Focusing only on inner loops
Advanced 1: Outer-loop Vectorization

Can you vectorize this loop? No

First need to expose the outer-loop parallelism

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

```
Unroll
```

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

```
Fuse (jam)
```

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

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

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

```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

\[
a[2i] \cdot b[2i] - a[2i+1] \cdot b[2i+1];
\]
Vectorizing interleaved data

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

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>

Trivia:

• Will this loop be vectorized by compilers?

```c
void loop(int * a, int * b, int * c){
    for (int i =0; i < N; i++)
        a[i]  = b[i] + c[i];
}
```
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 (Superword Level Parallelism) Vectorization

Independent and Isomorphic statements can be vectorized

Scalar Code

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

Vector Packs

Vector Code

Single Instruction Multiple Data (SIMD)

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

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 \{s1,…,sn\} where s1, s2,…, sn are independent isomorphic statements in a basic block.

• **Packset:** is a set of Packs

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

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 $P = \{S_1, \ldots, S_N\}$
    
    $\text{vec\_cost}(P) = \sum_{i=1}^{N} \text{scalar\_cost}(S_i)$

• **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 $1^{st}$ scalar value from pack $P$.

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

```c
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

S4 : A4 = L[1] - A2

Possible Initial packs
{L[5],L[6]}
{L[6],L[7]}

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

Assume VF=2

For brevity load statements are not shown. Refer to the ordering for how they are loaded.
Load and store packs

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}
Def-use chain packs

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>S4</td>
<td>A4 = L[1] - A2</td>
</tr>
</tbody>
</table>

Instruction Breakdown

Scalar = 7 + 3 + 3 = 13

Vector code

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>SV1</td>
<td>{A1,A2} = {L[5],L[6]} / {L[2],L[3]}</td>
</tr>
<tr>
<td>SU1</td>
<td>L[3] = unpack({L[2],L[3]},2)</td>
</tr>
<tr>
<td>SP1</td>
<td>{L[3],L[1]} = pack(L[3],L[1])</td>
</tr>
<tr>
<td>SV2</td>
<td>{A6,A4} = {L[3],L[1]} - {A1,A2}</td>
</tr>
<tr>
<td>SU2</td>
<td>L[2] = unpack({L[2],L[3]},1)</td>
</tr>
</tbody>
</table>

Instruction Breakdown

4 vector packing
2 unpacking
4 scalar
**Larsen & Amarasinghe algorithm**

<table>
<thead>
<tr>
<th>Schedule</th>
<th>Ordering 1</th>
<th>Ordering 2</th>
</tr>
</thead>
</table>

Algorithm highly susceptible to the statement ordering 😞
Holistic SLP Vectorizer

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

\[
\begin{align*}
S4 : A4 &= L[1] - A2 \\
\end{align*}
\]
## Holistic SLP Vectorizer

### Scalar code

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

### Vector code

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

### Instruction Breakdown

**Scalar** = 7 + 3 + 3 = 13

**Vector** = 0

0 vector

0 packing

0 unpacking

0

- vector

- packing

- unpacking

---

52
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}

Non-isomorphic

Instruction Breakdown

scalar = 7 + 3 + 3 = 13

Instruction Breakdown

4 vector
0 packing
0 unpacking
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
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}
SU1 : A1 = unpack(SV1, 1)

SP1 : {A3, A1} = pack(A3, A1)
S4 : A4 = L[1] - A2

Instruction Breakdown

scalar = 7 + 3 + 3 = 13

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)
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

Vector code

Instruction Breakdown

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

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

#### Scalar code

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>S4</td>
<td>A4 = L[1] - A2</td>
</tr>
</tbody>
</table>

#### 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

Scalar code: scalar = 7 + 3 + 3 = 13

Vector code: 5 vector

58
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”)
Subtraction on the even lanes!
Diversity of Instruction Sets

 Synopsis
__m512i_mm512_madd_epi16 (__m512i a, __m512i b)

#include <immintrin.h>
Instruction: vpnnaddw 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:i] := SignExtend32(a[i+31:i+16]+b[i+31:i+16]) + SignExtend32(a[i+31:i+16]*b[i+31:i+16])
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

X
VeGen Approach

VeGen: A Vectorizer Generator for SIMD and Beyond

Scalar Program

```c
int16_t A[4], B[4];
int32_t C[2];
void dot_prod() {
}
```

Target-specific Vectorizer

Vector Assembly

```
vmovd xmm0, [A]
vmovd xmm1, [B]
pmadwd xmm0, xmm1, xmm0
vmovd [C], xmm0
```

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