# CS 526

**A**dvanced
**C**ompiler
**C**onstruction

https://charithm.web.illinois.edu/cs526/sp2024/

# Compiler Auto-vectorization

# Vector Machines from 1970s

## Cray-I Supercomputer

Courtesy silicon valley graphics

R. Russell, "The CRAY 1 Computer System", CACM 1978

Table I. CRAY-1 CPU characteristics summary

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

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)

80 MHz clock

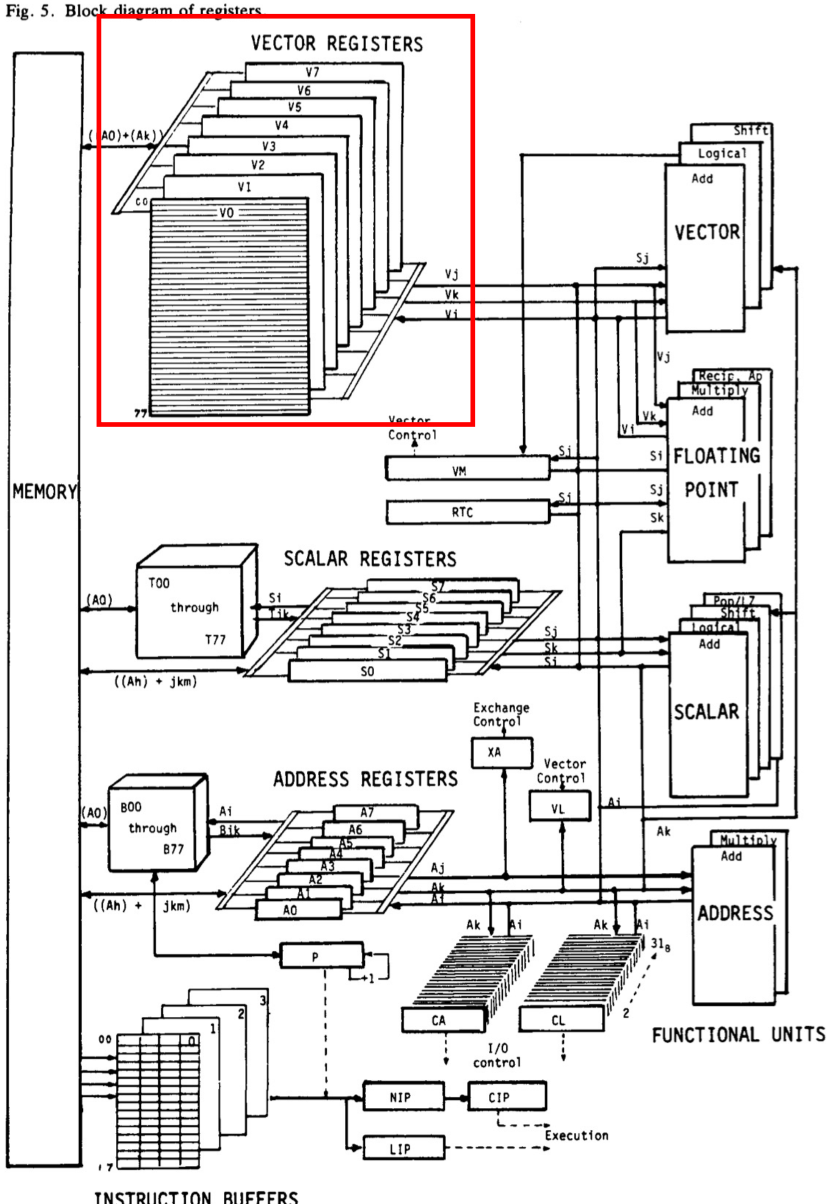
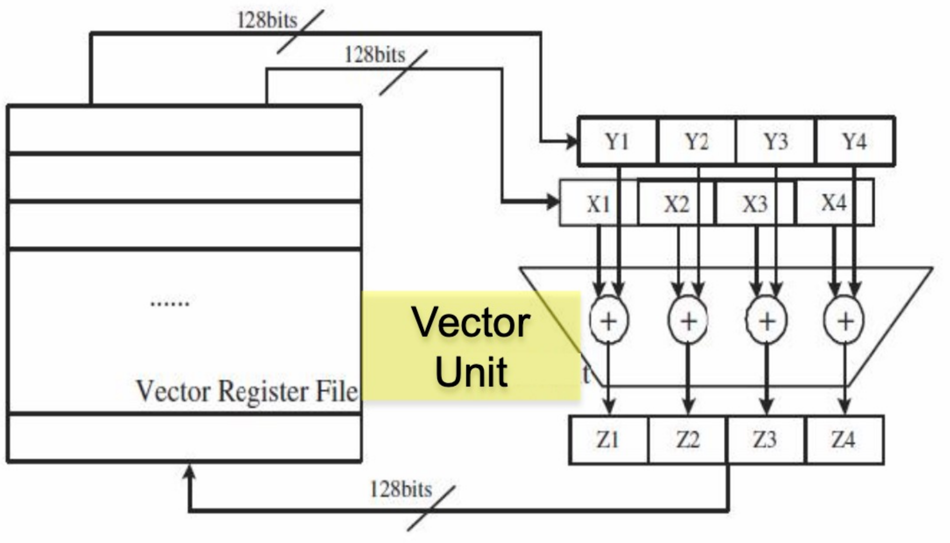


Fig. 5. Block diagram of registers.

8 x 64-word (4096 bit) vector registers

Vector length register and mask register

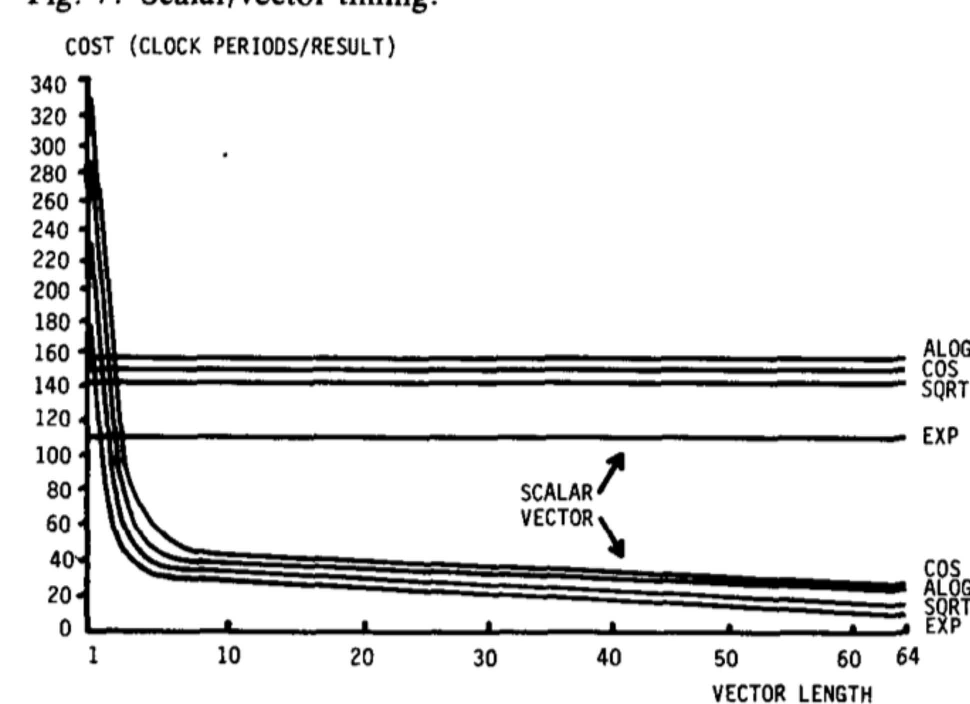
Single Instruction Multiple Data (SIMD) model



# 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



# Modern Short Vector Instructions

- Story of Intel MMX: https://www.intel.com/content/dam/www/public/us/en/documents/research/1997-vol01-iss-3-intel-technology-journal.pdf
- 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

**64**-bit vector (MMX)

**1997**

**128**-bit vector (SSE2)

**2000**

**256**-bit vector (AVX2)

**2011**

**512**-bit vector (AVX512)

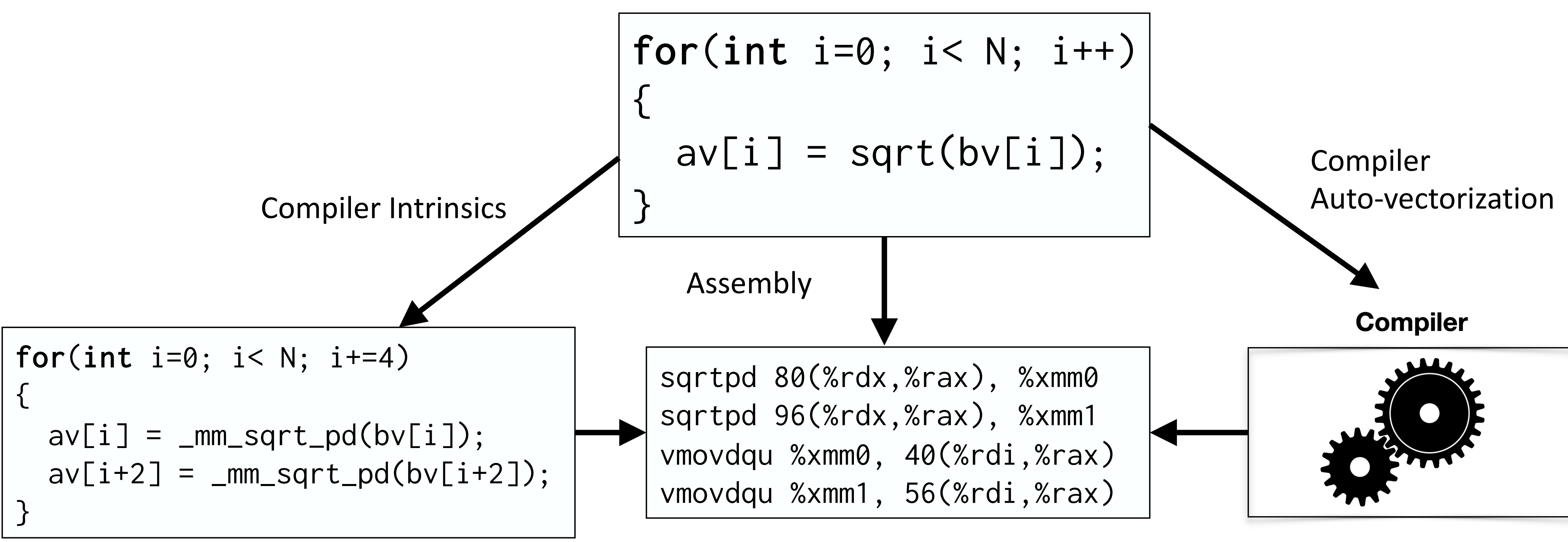
**2016**

Increase in bit-width

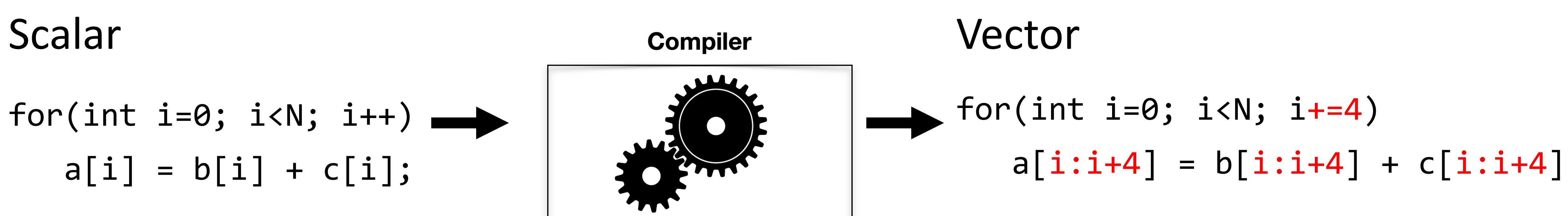
→

Diversity in Instruction Set

# How do we use these hardware features?



# Auto-vectorization introduction




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

# 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)**

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

Vectorize

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

**SLP Vectorization (Basic Blocks*)**

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

Vectorize

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

* many works have extended this to work with Control Flow

# Loop Vectorization

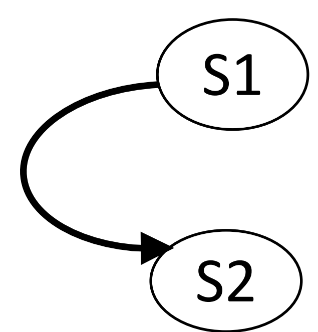
**Main Idea:** You can vectorize statements with no self-dependence

- 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;
}
```

Dependence Graph



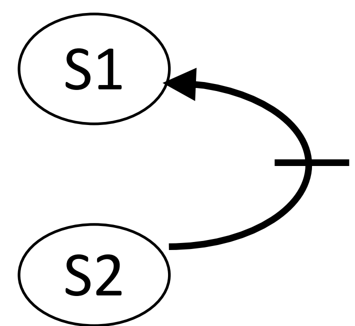

Can you vectorize this loop? **Yes**

```
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

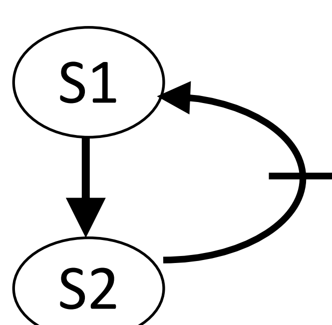
Dependence Graph

```
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 cycles

```
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;
}
```



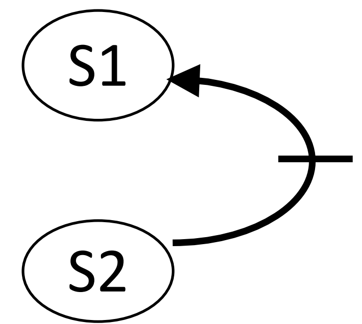

**We created new dependencies ☹**

# Acyclic: Backward

**Reorder the statements**

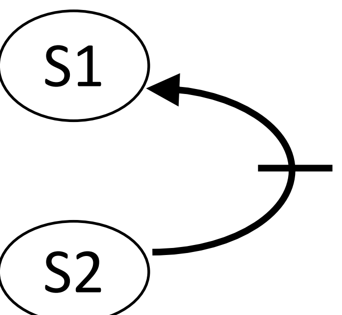
Dependence Graph

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




**Dependencies are preserved ☺**

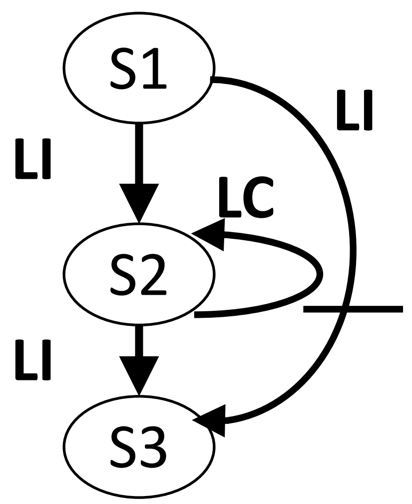
```
for(int i = 0; i < N; i+=4){
   S2: d[i:i+4] = a[i+1:i+5] + 1;
   S1: a[i:i+4] = b[i:i+4] + c[i:i+4];
}
```



# 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;
}
```

Dependence Graph




Can we vectorize? **No**

Can we vectorize partially? **Yes**

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

# 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

```
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 →

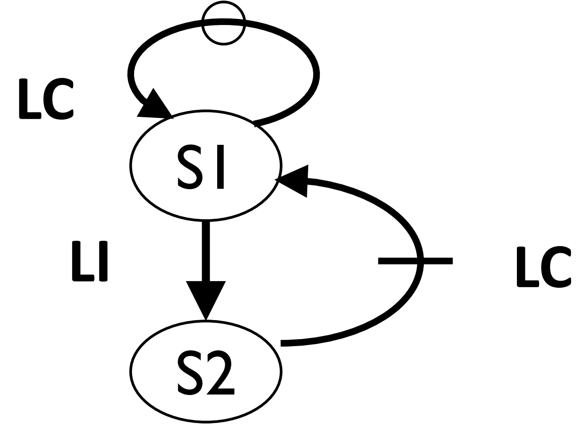
```
for (int i =0; i < N; i+=4){
  S1: b[i:i+4] = b[i:i+4] +
                           c[i:i+4];
}

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;
}
```

Is this beneficial?

# Cyclic: Example 2:

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

Dependence Graph



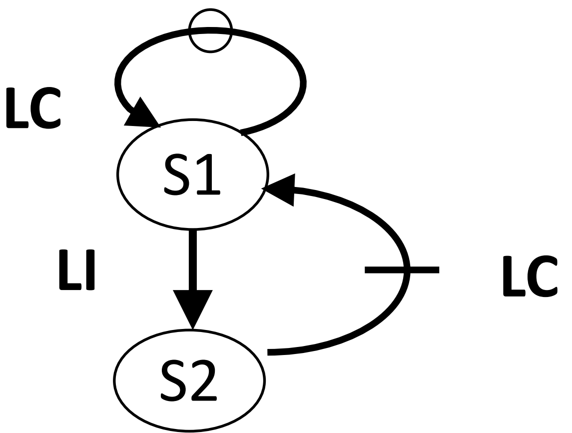

Can we vectorize? **No**

Can we remove the cycle? **Yes**

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

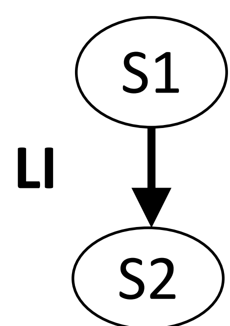
# Cyclic: Example 2: Scalar Expansion

```
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

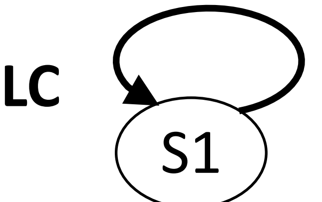
```
for (int i =0; i < N; i++){
    S1: a[i] = b[i] + 1;
    S2: c[i] = a[i] + 2;
}
```



# Cyclic: Example 3

Dependence Graph

```
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];
  }
}
```



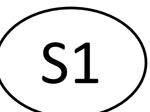

Can we vectorize? **No**

What if we focus one loop level at a time? **Yes**

# Cyclic: Example 3: Focus on the innermost loop

```
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];
  }
}
```

Dependence Graph




Vectorize Loop (j)

```
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

# Cyclic: Example 4

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

```
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

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

Can you vectorize this loop? **No**

First need to expose the outer-loop parallelism

Unroll

```
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)

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

# Advanced 2: Vectorizing interleaved data

What’s wrong with this?

```
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

a[0] a[1] a[2] a[3] a[4] a[5] a[6]

b[0] b[1] b[2] b[3] b[4] b[5] b[6]

# 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];
}
```

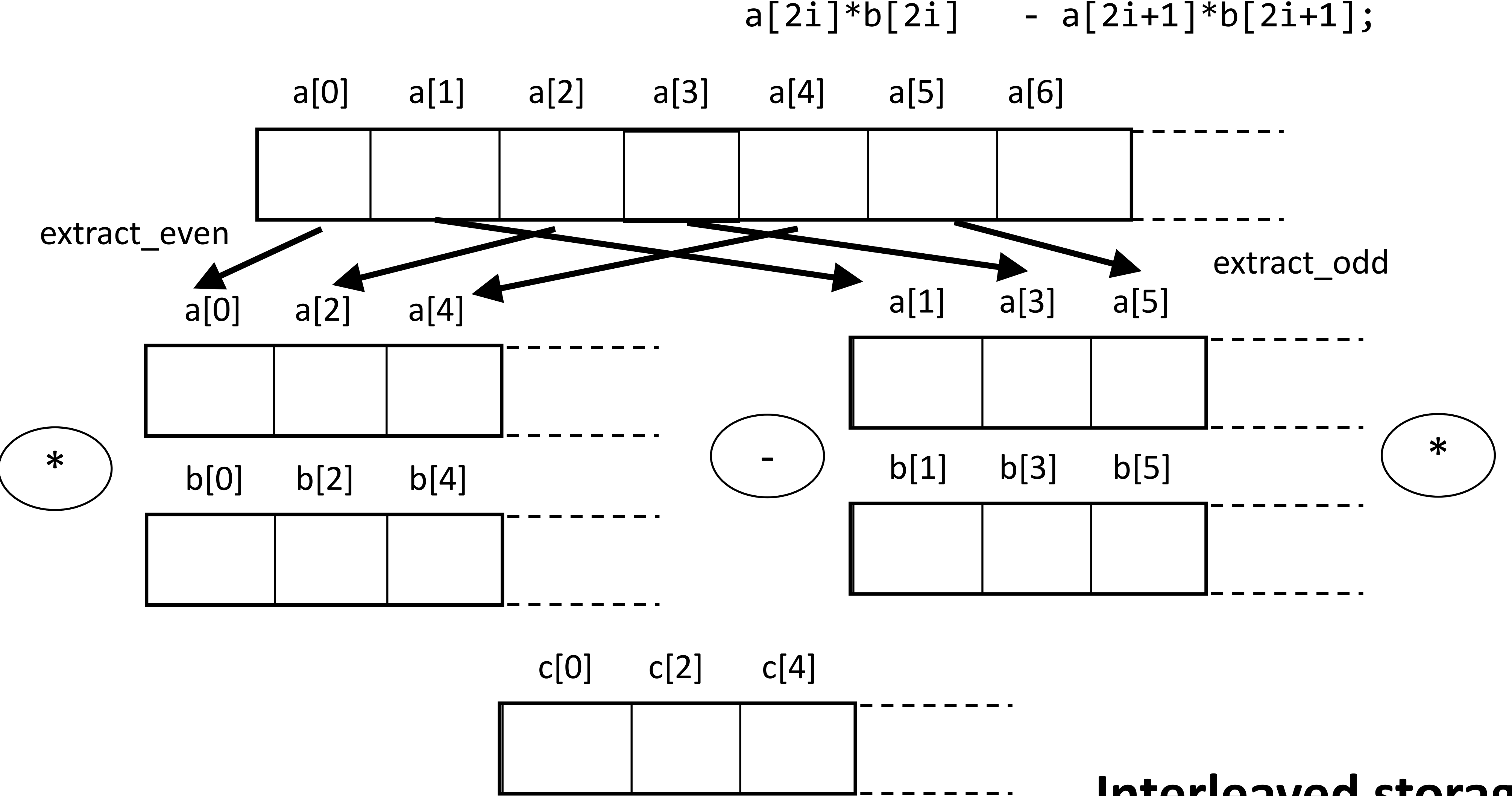
What's wrong with this?

Let's visualize data accesses

a[0] a[1] a[2] a[3] a[4] a[5] a[6]

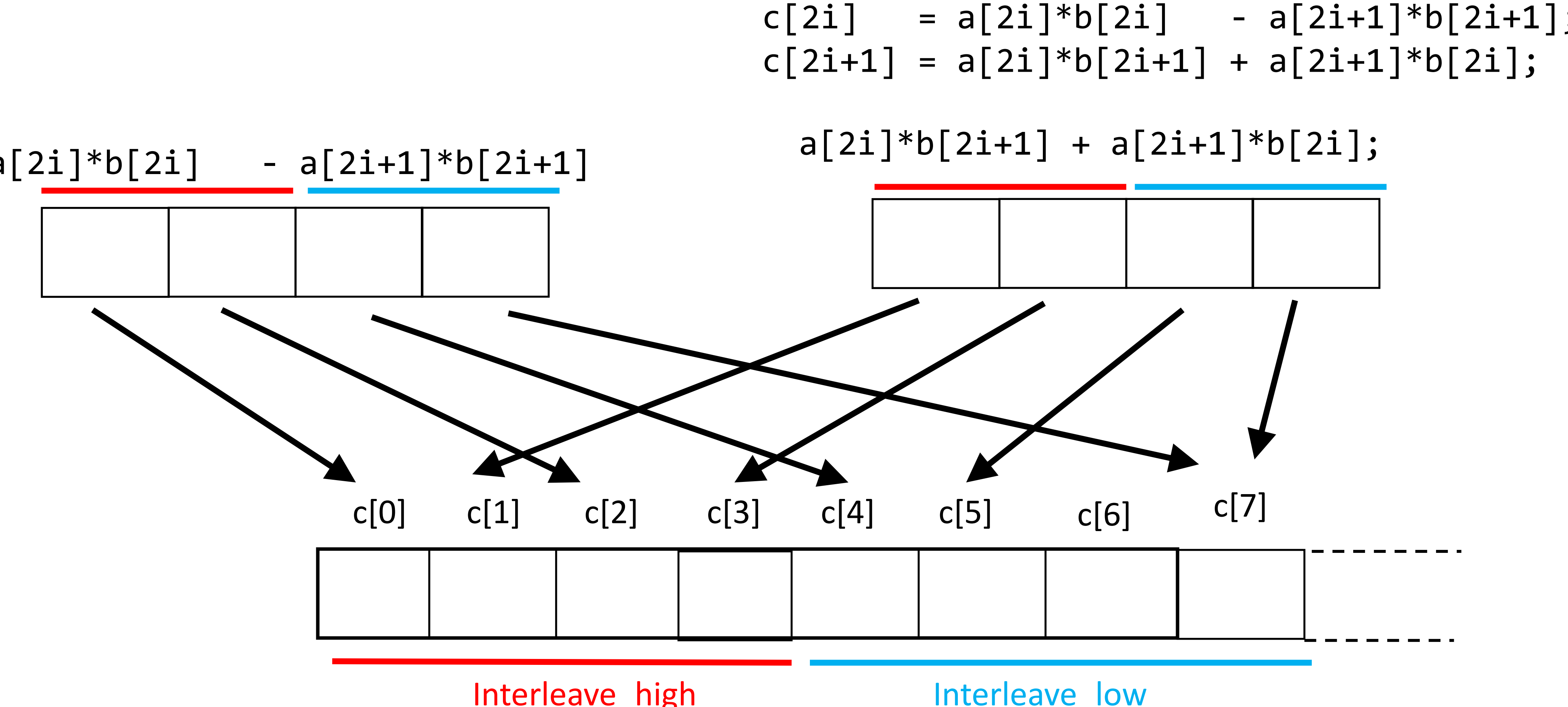
b[0] b[1] b[2] b[3] b[4] b[5] b[6]

# Vectorizing interleaved data




**Interleaved storage ☹**

# Vectorizing interleaved data

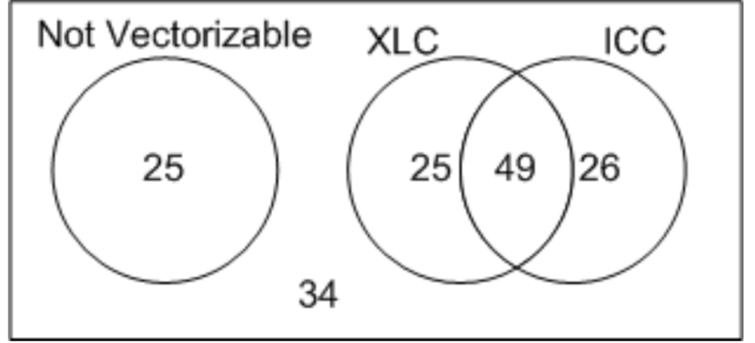



Nuzman et. al. "Auto-vectorization of interleaved data for SIMD", PLDI 2006

# Vectorization Capability

| Loops \ Compiler | XLC | ICC | GCC |
|---|---|---|---|
| Total | 159 | | |
| Vectorized | 74 | 75 | 32 |
| Not vectorized | 85 | 84 | 127 |
| Average Speed Up | 1.73 | 1.85 | 1.30 |

| Loops \ Compiler | XLC but not ICC | ICC but not XLC |
|---|---|---|
| Vectorized | 25 | 26 |




Maleki et. al, "An Evaluation of Vectorizing Compilers", PACT 2011

# Trivia:

- Will this loop be vectorized by compilers?

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

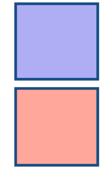
# Limitations of Loop Vectorization

- 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

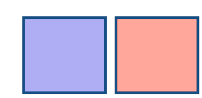


```
a[0] = b[0] + c[0]
a[1] = b[1] + c[1]
```

**Vector Packs**

Vector Code
**S**ingle **I**nstruction **M**ultiple **D**ata (SIMD)

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



Sam Larsen Parallelism (SLP)

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

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

✓

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

✗

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

✗

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 = $\{S_1, \ldots, S_N\}$

$$\texttt{vec_cost}(P) - \sum_{i=1}^{N} \texttt{scalar_cost}(S_i)$$

- **Packing Cost**
  - Explicitly pack non-vectorizable statements into a vector register using overhead packing instructions
  - e.g., P = 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 = 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

```
S1 : A1 = L[5] / L[2]
S2 : A2 = L[6] / L[3]
S3 : A3 = L[7] / L[4]
S4 : A4 = L[1] - A2
S5 : A5 = L[2] - A3
S6 : A6 = L[3] - A1
```

Assume VF=2

Possible Initial packs

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

Possible Extend packs (use-def and def-use chains)

```
{A1,A2}
{A2,A3}
{A4,A5}
{A5,A6}
{A6,A4}
{A1,A3}
```

Ordering

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

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

# Load and store packs

Schedule

```
S1 : A1 = L[5] / L[2]
S2 : A2 = L[6] / L[3]
S3 : A3 = L[7] / L[4]
S4 : A4 = L[1] - A2
S5 : A5 = L[2] - A3
S6 : A6 = L[3] - A1
```

Assume VF=2

Scalar

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

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

```
S1 : A1 = L[5] / L[2]
S2 : A2 = L[6] / L[3]
S3 : A3 = L[7] / L[4]
S4 : A4 = L[1] - A2
S5 : A5 = L[2] - A3
S6 : A6 = L[3] - A1
```

Assume VF=2

Scalar

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

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 : A1 = L[5] / L[2]
S2 : A2 = L[6] / L[3]
S3 : A3 = L[7] / L[4]
S4 : A4 = L[1] - A2
S5 : A5 = L[2] - A3
S6 : A6 = L[3] - A1
```

Vector code

```
SV1 : {A1,A2} = {L[5],L[6]} / {L[2],L[3]}
S3  : A3 = L[7] / L[4]
SU1 : L[3] = unpack({L[2],L[3]},2)
SP1 : {L[3],L[1]} = pack(L[3],L[1])
SV2 : {A6,A4} = {L[3],L[1]} - {A1,A2}
SU2 : L[2] = unpack({L[2],L[3]},1)
S4  : A5 = L[2] - A3
```

Instruction Breakdown

scalar = 7 + 3 + 3 = 13

Instruction Breakdown

4 vector
1 packing
2 unpacking
4 scalar

11

# Larsen & Amarasinghe algorithm

Schedule

```
S1 : A1 = L[5] / L[2]
S2 : A2 = L[6] / L[3]
S3 : A3 = L[7] / L[4]
S4 : A4 = L[1] - A2
S5 : A5 = L[2] - A3
S6 : A6 = L[3] - A1
```

Ordering 1

4 vector
1 packing
2 unpacking
4 scalar

11

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

Scalar

Ordering 2

5 vector
2 packing
5 unpacking
3 scalar

15

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

```
S1 : A1 = L[5] / L[2]
S2 : A2 = L[6] / L[3]
S3 : A3 = L[7] / L[4]
S4 : A4 = L[1] - A2
S5 : A5 = L[2] - A3
S6 : A6 = L[3] - A1
```

Liu et.al "A compiler framework for extracting superword level parallelism", PLDI 2011

# Holistic SLP Vectorizer

Scalar code

```
S1 : A1 = L[5] / L[2]
S2 : A2 = L[6] / L[3]
S3 : A3 = L[7] / L[4]
S4 : A4 = L[1] - A2
S5 : A5 = L[2] - A3
S6 : A6 = L[3] - A1
```

Instruction Breakdown

scalar = 7 + 3 + 3 = 13

Vector code

```
S1  : A1 = L[5] / L[2]
S2  : A2 = L[6] / L[3]

S3  : A3 = L[7] / L[4]

S4  : A4 = L[1] - A2
S5  : A5 = L[2] - A3
S6  : A6 = L[3] - A1
```

Instruction Breakdown

0 vector
0 packing
0 unpacking

# Holistic SLP Vectorizer

Scalar code

```
S1 : A1 = L[5] / L[2]
S2 : A2 = L[6] / L[3]
S3 : A3 = L[7] / L[4]
S4 : A4 = L[1] - A2
S5 : A5 = L[2] - A3
S6 : A6 = L[3] - A1
```

Vector code

```
SV1 : {A1,A2} = {L[5],L[6]} / {L[2],L[3]}

S3  : A3 = L[7] / L[4]

S4  : A4 = L[1] - A2
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

```
S1 : A1 = L[5] / L[2]
S2 : A2 = L[6] / L[3]
S3 : A3 = L[7] / L[4]
S4 : A4 = L[1] - A2
S5 : A5 = L[2] - A3
S6 : A6 = L[3] - A1
```

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)

S3  : A3 = L[7] / L[4]

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

```
S1 : A1 = L[5] / L[2]
S2 : A2 = L[6] / L[3]
S3 : A3 = L[7] / L[4]
S4 : A4 = L[1] - A2
S5 : A5 = L[2] - A3
S6 : A6 = L[3] - A1
```

Vector code

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

S3  : A3 = L[7] / L[4]
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

Instruction Breakdown

4 vector
1 packing
1 unpacking

# Holistic SLP Vectorizer

Scalar code

```
S1 : A1 = L[5] / L[2]
S2 : A2 = L[6] / L[3]
S3 : A3 = L[7] / L[4]
S4 : A4 = L[1] - A2
S5 : A5 = L[2] - A3
S6 : A6 = L[3] - A1
```

Vector code

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

S3  : A3 = L[7] / L[4]
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

```
S1 : A1 = L[5] / L[2]
S2 : A2 = L[6] / L[3]
S3 : A3 = L[7] / L[4]
S4 : A4 = L[1] - A2
S5 : A5 = L[2] - A3
S6 : A6 = L[3] - A1
```

Vector code

```
SV1 : {A1,A2} = {L[5],L[6]} / {L[2],L[3]}
SU1 : A1 = unpack(SV1,1)
SU2 : A2 = unpack(SV1,2)
S3  : A3 = L[7] / L[4]
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

Instruction Breakdown

4 vector
1 packing
2 unpacking
5 scalar

12

# Optimal Strategy (goSLP)

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

Scalar code

```
S1 : A1 = L[5] / L[2]
S2 : A2 = L[6] / L[3]
S3 : A3 = L[7] / L[4]
S4 : A4 = L[1] - A2
S5 : A5 = L[2] - A3
S6 : A6 = L[3] - A1
```

Vector code

```
SV1 : {A2,A3} = {L[6],L[7]} / {L[3],L[4]}
SU1 : L[2] = unpack(SLV1,2)
S1  : A1 = L[5] / L[2]
SU2 : L[3] = unpack(SLV2,1)
SV2 : {A4,A5} = {L[1],L[2]} - {A2,A3}
S6  : A6 = L[3] - A1
```

Instruction Breakdown

scalar = 7 + 3 + 3 = 13

Instruction Breakdown

5 vector
0 packing
2 unpacking
3 scalar

10

# Other Extensions

- Control Flow (Shin et. al. “Superword-Level Parallelism in the Presence of Control Flow”, CGO 2005)
- Vectorizing non-isomorphic chains (Porpodas et. al. “PSLP: Padded SLP Automatic Vectorization”, CGO 2015)
- 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

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

**Synopsis**

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

**Description**

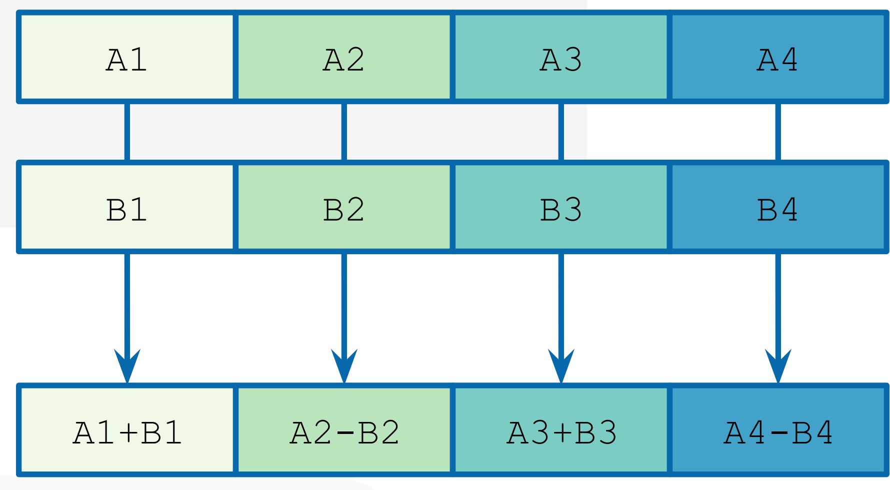
Alternatively add and subtract packed double-precision (64-bit) floating-point elements in a to/from packed elements in b, and store the results in dst.

**Operation**

```
FOR j := 0 to 3
	i := j*64
	IF ((j & 1) == 0)
		dst[i+63:i] := a[i+63:i] - b[i+63:i]
	ELSE
		dst[i+63:i] := a[i+63:i] + b[i+63:i]
	FI
ENDFOR
dst[MAX:256] := 0
```

**Performance**

| Architecture | Latency | Throughput (CPI) |
|---|---|---|
| Icelake | 4 | 0.5 |
| Skylake | 4 | 0.5 |
| Broadwell | 3 | 1 |
| Haswell | 3 | 1 |




Subtraction on the even lanes!

X

# Diversity of Instruction Sets

```
__m512i _mm512_madd_epi16 (__m512i a, __m512i b)
```

vpmaddwd

**Synopsis**

```
__m512i _mm512_madd_epi16 (__m512i a, __m512i b)
#include <immintrin.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:i] := SignExtend32(a[i+31:i+16]*b[i+31:i+16]) + SignExtend32
ENDFOR
dst[MAX:512] := 0
```

**Performance**

| Architecture | Latency | Throughput (CPI) |
|---|---|---|
| Icelake | - | 1 |
| Skylake | 5 | 0.5 |

A1 A2 A3 A4 A5 A6 A7 A8

B1 B2 B3 B4 B5 B6 B7 B8

A1*B1+ A2*B2 | A3*B3+ A4*B4 | A5*B5+ A6*B6 | A7*B8+ A7*B8

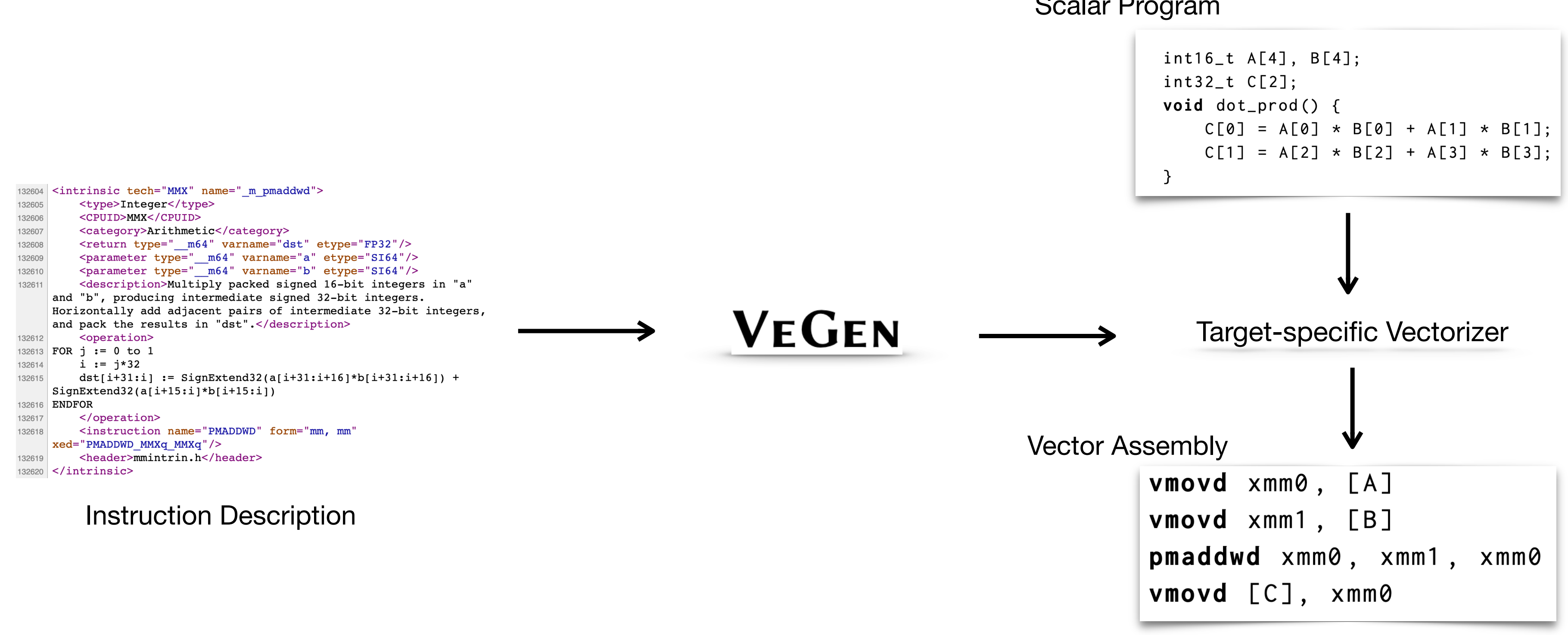
Cross lane operation

Complex operation

x

# Vegen Approach

## VeGen: A Vectorizer Generator for SIMD and Beyond




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