FlexVec: Auto-vectorization for Irregular Loops

Sara Baghsorkhi, Nalini Vasudevan,
Youfeng Wu

Intel Labs
June 17, 2016

What is FlexVec?

- Optimistic compiler vectorization with dynamic detection of memory and control flow dependencies
- Targets 3 general patterns that don’t vectorize efficiently:

What is FlexVec?

- Optimistic compiler vectorization with dynamic detection of memory and control flow dependencies
- Targets 3 general patterns that don’t vectorize efficiently:

Early Loop Termination

```
for (i=0; i<N; i++){
  t = a[b[i]];
  if ( t < 3 )
    return t;
}
```

What is FlexVec?

- Optimistic compiler vectorization with dynamic detection of memory and control flow dependencies
- Targets 3 general patterns that don't vectorize efficiently:

Early Loop Termination

```
for (i=0; i<N; i++){
  t = a[b[i]];
  if ( t < 3 )
    return t;
}
```

What is FlexVec?

- Optimistic compiler vectorization with dynamic detection of memory and control flow dependencies
- Targets 3 general patterns that don't vectorize efficiently:

Early Loop Termination

```
for (i=0; i<N; i++){
  t = a[b[i]];
  if ( t < 3 )
    return t;
}
```

What is FlexVec?

- Optimistic compiler vectorization with dynamic detection of memory and control flow dependencies
- Targets 3 general patterns that don't vectorize efficiently:

Early Loop Termination

```
for (i=0; i<N; i++){
  t = a[b[i]];
  if ( t < 3 )
    return t;
}
```

Conditional Update

```
for (i=0; i<N; i++){
  t = b[i+x];
  if (t < 5)
    x = t;
}
```

What is FlexVec?

- Optimistic compiler vectorization with dynamic detection of memory and control flow dependencies
- Targets 3 general patterns that don't vectorize efficiently:

Early Loop Termination

```
for (i=0; i<N; i++){
  t = a[b[i]];
  if ( t < 3 )
    return t;
}
```

Conditional Update

```
for (i=0; i<N; i++){
  t = b[i+x];
  if (t < 5)
    x = t;
}
```

Runtime Memory Conflicts

```
for (i=0; i<N; i++) {
  x[y[i]] = x[z[i]];
}
```

What is FlexVec?

- Optimistic compiler vectorization with dynamic detection of memory and control flow dependencies
- Targets 3 general patterns that don't vectorize efficiently:

Early Loop Termination

```
for (i=0; i<N; i++){
  t = a[b[i]];
  if ( t < 3 )
    return t;
}
```

Conditional Update

```
for (i=0; i<N; i++){
  t = b[i+x];
  if (t < 5)
    x = t;
}
```

Runtime Memory Conflicts

```
for (i=0; i<N; i++) {
  x[y[i]] = x[z[i]];
}
```

- FlexVec identifies idioms and vector intrinsics that allow efficient defer of dependency resolution to run time

What is FlexVec?

- Optimistic compiler vectorization with dynamic detection of memory and control flow dependencies
- Targets 3 general patterns that don't vectorize efficiently:

Early Loop Termination

```
for (i=0; i<N; i++){
  t = a[b[i]];
  if ( t < 3 )
    return t;
}
```

Conditional Update

```
for (i=0; i<N; i++){
  t = b[i+x];
  if (t < 5)
    x = t;
}
```

Runtime Memory Conflicts

```
for (i=0; i<N; i++) {
  x[y[i]] = x[z[i]];
}
```

- FlexVec identifies idioms and vector intrinsics that allow efficient defer of dependency resolution to run time
  - software speculation

What is FlexVec?

- Optimistic compiler vectorization with dynamic detection of memory and control flow dependencies
- Targets 3 general patterns that don't vectorize efficiently:

Early Loop Termination

```
for (i=0; i<N; i++){
  t = a[b[i]];
  if ( t < 3 )
    return t;
}
```

Conditional Update

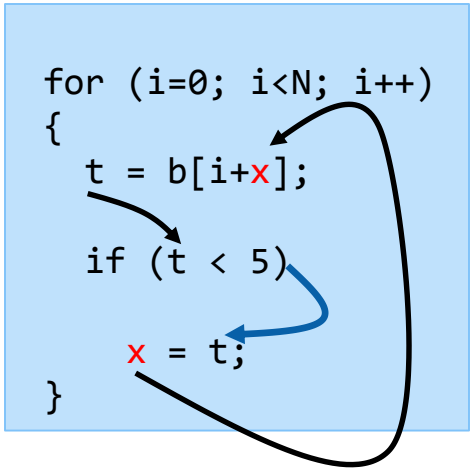
```
for (i=0; i<N; i++){
  t = b[i+x];
  if (t < 5)
    x = t;
}
```

Runtime Memory Conflicts

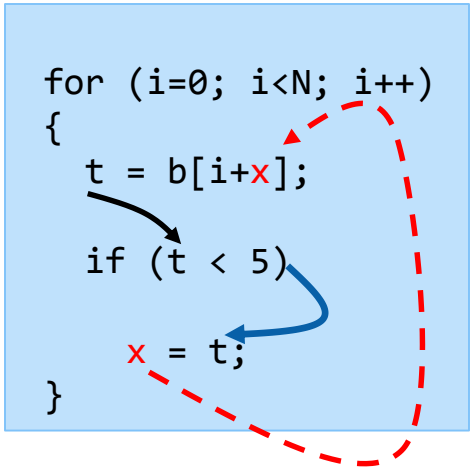
```
for (i=0; i<N; i++) {
  x[y[i]] = x[z[i]];
}
```

- FlexVec identifies idioms and vector intrinsics that allow efficient defer of dependency resolution to run time
  - software speculation
  - dynamic partitioning of vector iterations

Conditional Dependence Pattern



Conditional Dependence Pattern



Steady State

k_{todo} =	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
v_x =	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
v_t =	8	7	8	5	8	6	3	6	7	9	9	7	8	4	8	5
k_{stop} =	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0

Patch-up Code?

Conditional Dependence Pattern

```
for (i=0; i<N; i++)
{
  t = b[i+x];

  if (t < 5)

    x = t;
}
```

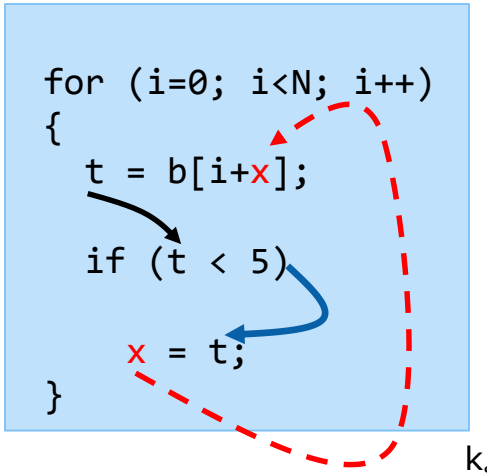
Steady State

k_{todo} =	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
v_x =	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
v_t =	8	7	8	5	8	6	3	6	7	9	9	7	8	4	8	5
k_{stop} =	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0

Patch-up Code?
More conservative version
Of the Steady State

```
while(k_stop){
   salvage work
   x = t;
   mark off processed lanes
   Steady state for rem lanes
}
```

Conditional Dependence Pattern



Steady State

k_{todo} =	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
v_x =	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
v_t =	8	7	8	5	8	6	3	6	7	9	9	7	8	4	8	5
k_{stop} =	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0

Patch-up Loop?

k_{safe} = **KFTM.INC**(k_{todo}, k_{stop}) = 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0

Set k_{todo} **enabled** bits of output mask k_{safe} until and including the first enabled set bit in k_{stop}.

Conditional Dependence Pattern

```
for (i=0; i<N; i++)
{
  t = b[i+x];

  if (t < 5)

    x = t;
}
```

Steady State

k_{todo} =	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
v_x =	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
v_t =	8	7	8	5	8	6	3	6	7	9	9	7	8	4	8	5
k_{stop} =	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0

Patch-up Loop?

k_{safe} = **KFTM.INC**(k_{todo}, k_{stop}) =	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0
v_x = **VPSLCTLAST**(k_{safe},v_t) =	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3

Select the last element in vector register v_t enabled by k_{safe} and broadcast the element to v_x.

Conditional Dependence Pattern

```
for (i=0; i<N; i++)
{
  t = b[i+x];

  if (t < 5)

    x = t;
}
```

Steady State

k_{todo} =	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
v_x =	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
v_t =	8	7	8	5	8	6	3	6	7	9	9	7	8	4	8	5
k_{stop} =	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0

Patch-up Loop?

k_{safe} = **KFTM.INC**(k_{todo}, k_{stop}) =	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0
v_x = **VPSLCTLAST**(k_{safe},v_t) =	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
k_{todo} =	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1
k_{stop} =	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0

Clear off vector elements processed: update **k_{todo}** and **k_{stop}**

Conditional Dependence Pattern

```
for (i=0; i<N; i++)
{
  t = b[i+x];

  if (t < 5)

    x = t;
}
```

Steady State

k_{todo} =	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
v_x =	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
v_t =	8	7	8	5	8	6	3	6	7	9	9	7	8	4	8	5
k_{stop} =	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0

Patch-up Loop?

k_{safe} = **KFTM.INC**(k_{todo}, k_{stop}) =	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0
v_x = **VPSLCTLAST**(k_{safe},v_t) =	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
k_{todo} =	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1
k_{stop} =	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
v_t =	8	7	8	5	8	6	3	7	9	9	7	8	6	8	9	7
k_{stop} =	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Re-execute statements affected by the cross-iteration dependence.

Conditional Dependence Pattern

```
for (i=0; i<N; i++)
{
  t = b[i+x];

  if (t < 5)

    x = t;
}
```

Steady State

k_{todo} =	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
v_x =	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
v_t =	8	7	8	5	8	6	3	6	7	9	9	7	8	4	8	5
k_{stop} =	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0

Patch-up Loop?

k_{safe} = **KFTM.INC**(k_{todo}, k_{stop}) =	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0
v_x = **VPSLCTLAST**(k_{safe},v_t) =	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
k_{todo} =	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1
k_{stop} =	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
v_t =	8	7	8	5	8	6	3	7	9	9	7	8	6	8	9	7
k_{stop} =	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Repeat the **Vector Partitioning Loop** till all vector elements are processed.

Conditional Dependence Pattern

```
for (i=0; i<N; i++)
{
  t = b[i+x];

  if (t < 5)

    x = t;
}
```

Steady State

k_{todo} =	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
v_x =	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
v_t =	8	7	8	5	8	6	3	6	7	9	9	7	8	4	8	5
k_{stop} =	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0

Patch-up Loop?

k_{safe} = **KFTM.INC**(k_{todo}, k_{stop}) =

v_x = **VPSLCTLAST**(k_{safe},v_t) =

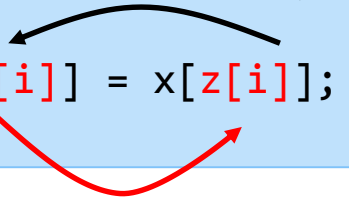
	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0
	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
k_{todo} =	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1
k_{stop} =	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0

v_t =	8	7	8	5	8	6	3	7	9	9	7	8	6	8	9	7
k_{stop} =	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

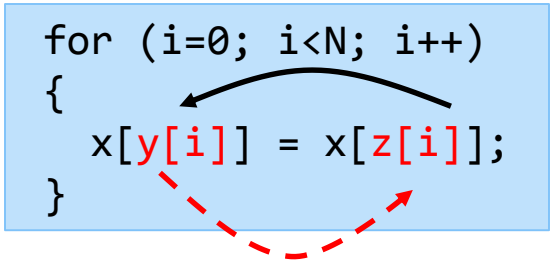
```
while(k_stop){
   salvage work
   x = t;
   mark off processed lanes
   Steady state for rem lanes
}
```

Runtime Memory Conflict Pattern

```
for (i=0; i<N; i++)
{
  x[y[i]] = x[z[i]];
}
```



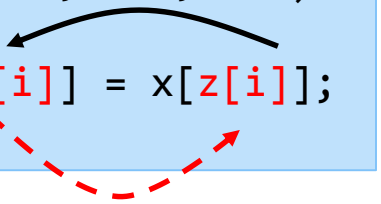
Runtime Memory Conflict Pattern




```
for (i=0; i<N; i+=8){
    …
    do{
        conflict detection
        salvage work
        mark off processed lanes
    }while(k_stop);
    …
}
```

Runtime Memory Conflict Pattern

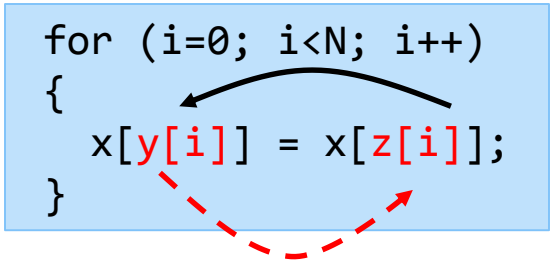
```
for (i=0; i<N; i++)
{
  x[y[i]] = x[z[i]];
}
```



k_{todo} = | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

z[i] = v_z =	2	9	1	8	4	**5**	2	9	9	7	1
y[i] = v_y =	3	6	**5**	6	8	1	3	6	6	4	8

Runtime Memory Conflict Pattern



k_{todo} = 1 1 1 1 1 1 1 1 1 1 1 1

z[i] = v_z = 2 9 1 8 4 **5** 2 9 9 7 1

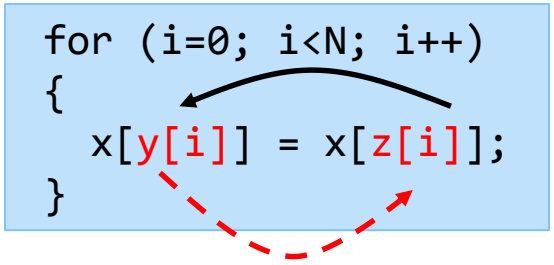
y[i] = v_y = 3 6 **5** 6 8 1 3 6 6 4 8

k_{stop} = **VPCONFLICTM**(k_{todo}, v_z, v_y) = 0 0 0 0 0 **1** 0 0 0 0 0

Compare each element of vector register v_z with all enabled previous elements of v_y, and set the bit to one in output mask k_{stop}.

Runtime Memory Conflict Pattern

```
for (i=0; i<N; i++)
{
  x[y[i]] = x[z[i]];
}
```



k_{todo} = 1 1 1 1 1 1 1 1 1 1 1

z[i] = v_z = 2 9 1 8 4 5 2 9 9 7 1

y[i] = v_y = 3 6 5 6 8 1 3 6 6 4 8

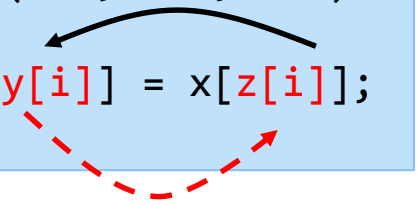
k_{stop} = **VPCONFLICTM**(k_{todo}, v_z, v_y) = 0 0 0 0 0 1 0 0 0 0 0

k_{safe} = **KFTM.EXC**(k_{todo}, k_{stop}) = 1 1 1 1 1 0 0 0 0 0 0

Set k_{todo} enabled bits of k_{safe} mask until but excluding the first enabled set bit in k_{stop}.

Runtime Memory Conflict Pattern

```
for (i=0; i<N; i++)
{
  x[y[i]] = x[z[i]];
}
```



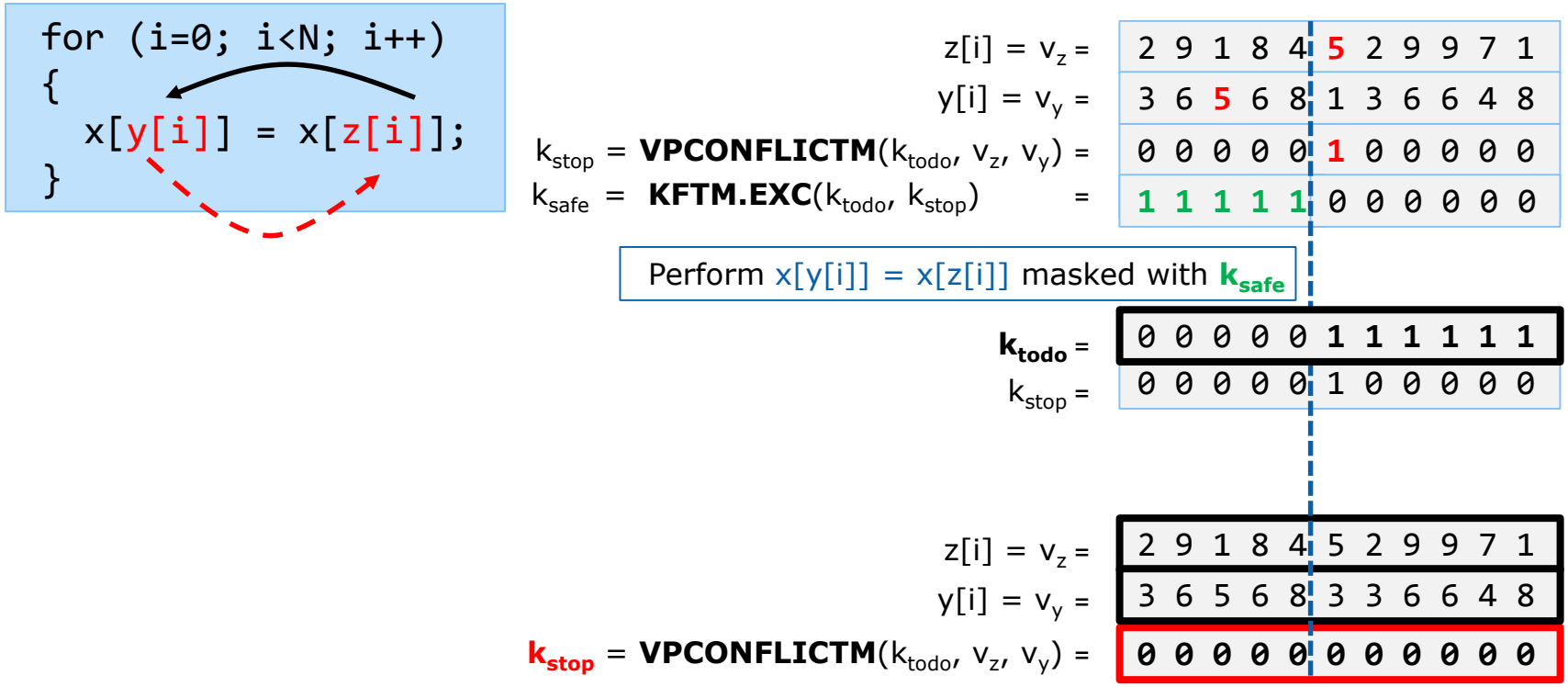
k_{todo} =	1	1	1	1	1	1	1	1	1	1	1
z[i] = v_z =	2	9	1	8	4	5	2	9	9	7	1
y[i] = v_y =	3	6	5	6	8	1	3	6	6	4	8
k_{stop} = **VPCONFLICTM**(k_{todo}, v_z, v_y) =	0	0	0	0	0	1	0	0	0	0	0
k_{safe} = **KFTM.EXC**(k_{todo}, k_{stop}) =	1	1	1	1	1	0	0	0	0	0	0

Perform x[y[i]] = x[z[i]] masked with **k_{safe}**

k_{todo} =	0	0	0	0	0	1	1	1	1	1	1
k_{stop} =	0	0	0	0	0	1	0	0	0	0	0

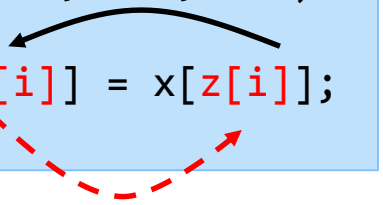
Runtime Memory Conflict Pattern

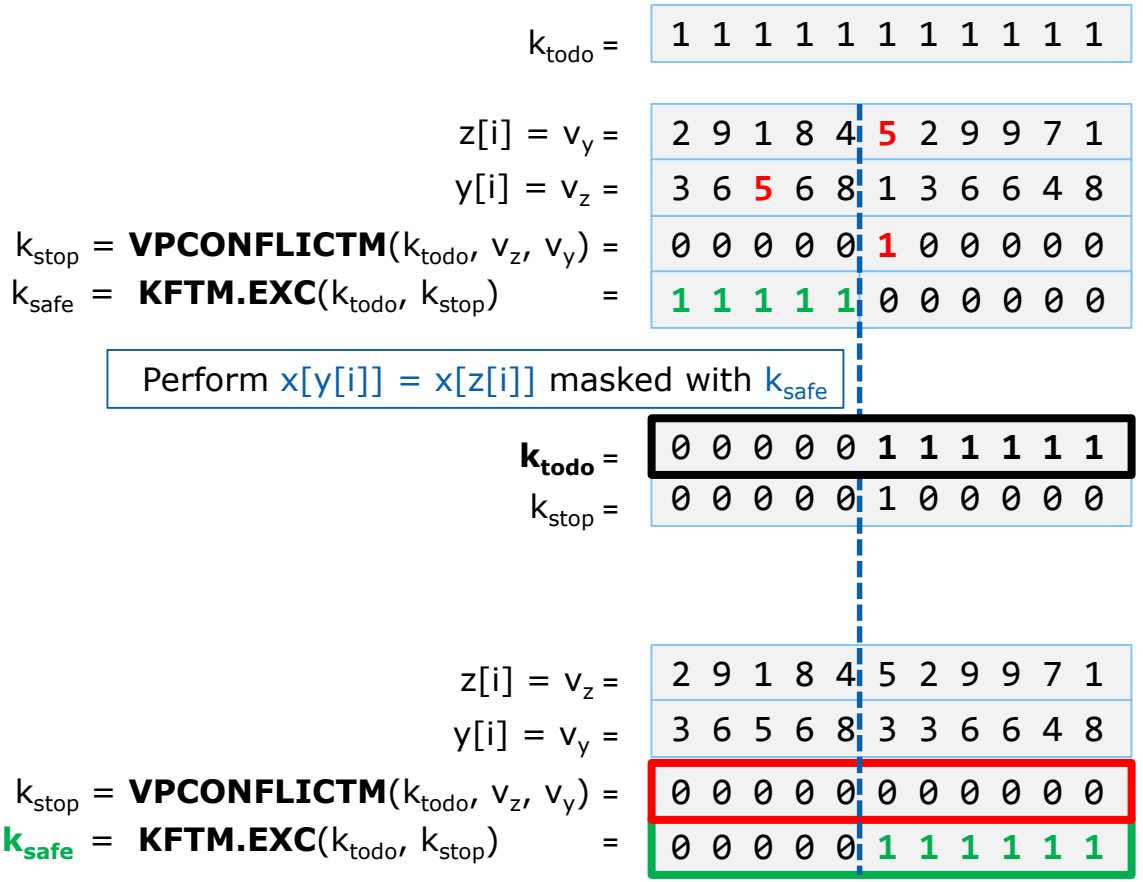
```
for (i=0; i<N; i++)
{
  x[y[i]] = x[z[i]];
}
```



Runtime Memory Conflict Pattern

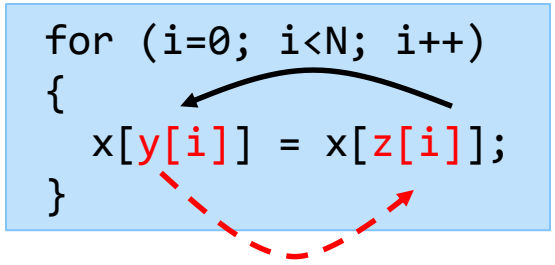
```
for (i=0; i<N; i++)
{
  x[y[i]] = x[z[i]];
}
```





Runtime Memory Conflict Pattern

```
for (i=0; i<N; i++)
{
  x[y[i]] = x[z[i]];
}
```



k_{todo} =	1	1	1	1	1	1	1	1	1	1	1
z[i] = v_y =	2	9	1	8	4	5	2	9	9	7	1
y[i] = v_z =	3	6	5	6	8	1	3	6	6	4	8
k_{stop} = **VPCONFLICTM**(k_{todo}, v_z, v_y) =	0	0	0	0	0	1	0	0	0	0	0
k_{safe} = **KFTM.EXC**(k_{todo}, k_{stop}) =	1	1	1	1	1	0	0	0	0	0	0

Perform x[y[i]] = x[z[i]] masked with k_{safe}

k_{todo} =	0	0	0	0	0	1	1	1	1	1	1
k_{stop} =	0	0	0	0	0	1	0	0	0	0	0

z[i] = v_z =	2	9	1	8	4	5	2	9	9	7	1
y[i] = v_y =	3	6	5	6	8	3	3	6	6	4	8
k_{stop} = **VPCONFLICTM**(k_{todo}, v_z, v_y) =	0	0	0	0	0	0	0	0	0	0	0
k_{safe} = **KFTM.EXC**(k_{todo}, k_{stop}) =	0	0	0	0	0	1	1	1	1	1	1

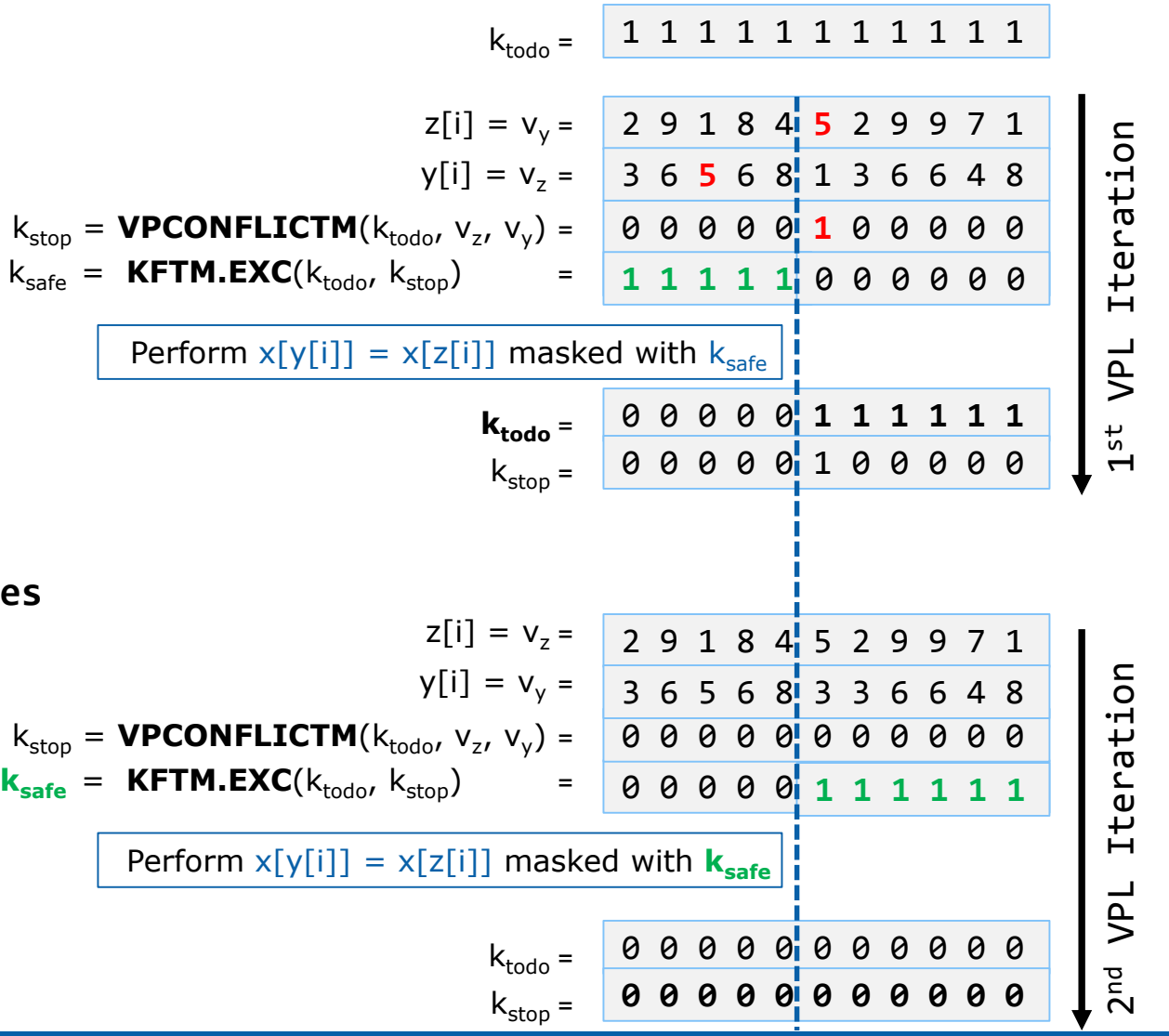
Perform x[y[i]] = x[z[i]] masked with **k_{safe}**

k_{todo} =	0	0	0	0	0	0	0	0	0	0	0
k_{stop} =	0	0	0	0	0	0	0	0	0	0	0

Runtime Memory Conflict Pattern

```
for (i=0; i<N; i++)
{
  x[y[i]] = x[z[i]];
}
```

```
do{
  conflict detection
  salvage work
  mark off processed lanes
}while(k_stop);
```

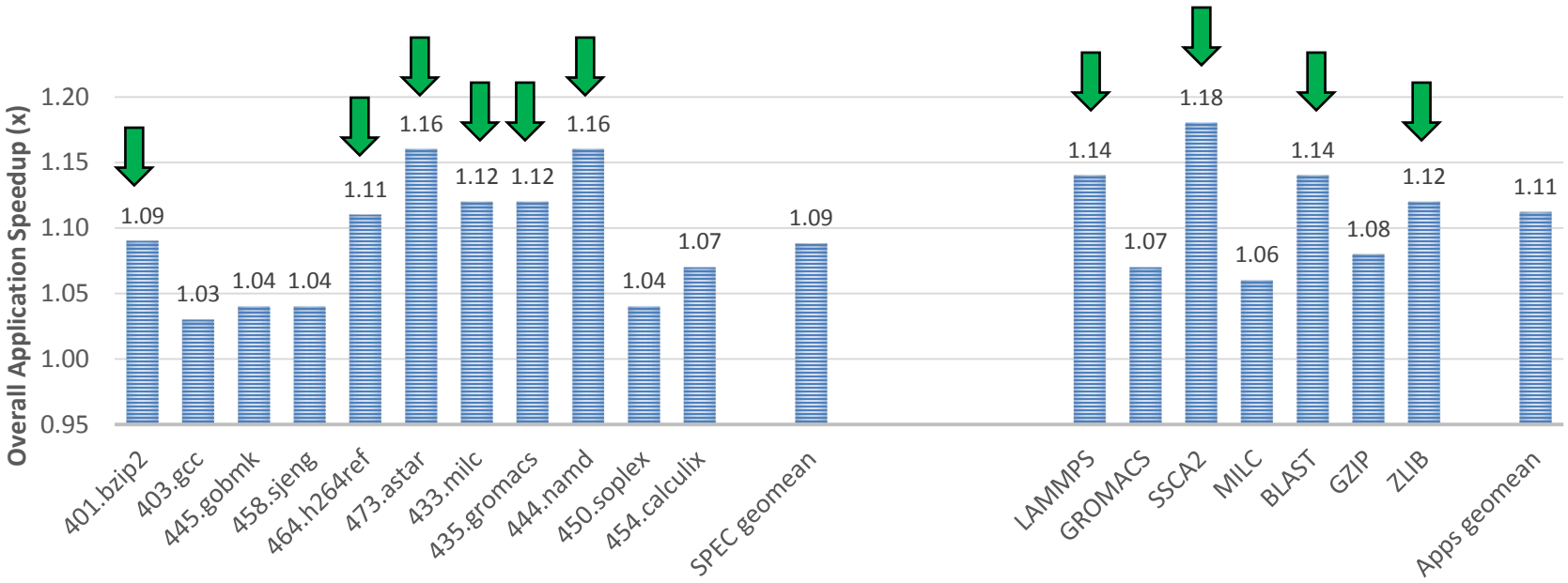


Experiment Setup

- Compiler ICC with AVX512 as baseline
- Simulation samples of hot spots around the vectorized loops
- LIT Trace tool for collecting simulation checkpoints
- rdtsc time stamp to measure the coverage of hot regions and scale down the region speedups
- Aggressive OOO as baseline for simulation
- FlexVec's vector intrinsics implemented following AVX512

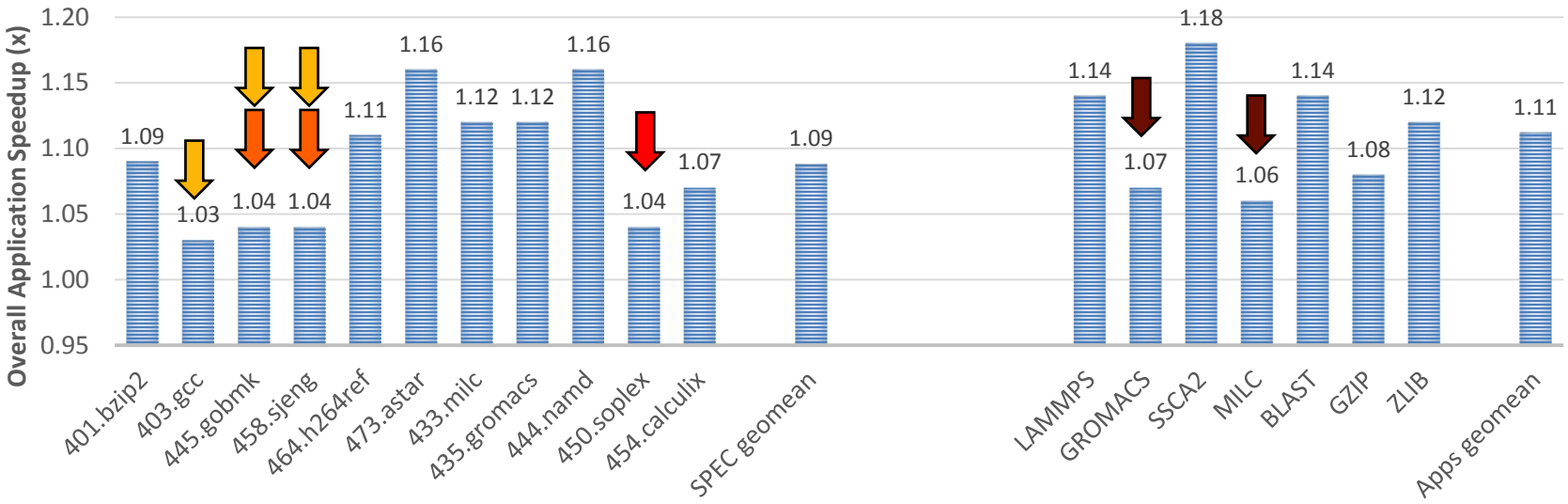
FlexVec Instruction	Latency(cycles), Throughput
KFTMINC/KFTMEXC	2, 1
VPSLCTLAST	3, 1
VPGATHERFF and VMOVFF	1 cycle AGU latency, 2 loads per cycle[2]
VPCONFLICTM	20 cycles, 2

Results: High Performers



- Baseline is AVX512 vector code
- High trip counts, compute intensiveness, and having a high coverage

Results: Low Performers



- Low two digit trip counts limits an OOO's processor capability in exploiting vector ILP.
- Nested branches in loops reduce the effective vector length
- Low coverage
- Memory bound

Conclusions

- New code generation dynamically adapts SIMD vector length to accommodate applications' cross-iteration dependencies.
- Identified idioms and vector intrinsics required to capture and to communicate data and control flow relationships for efficient partial vector code generation
- Noticeable performance benefits across a wide range of applications, missed by existing vectorizing techniques.
- Check out AVX512!