TOPICS IN LOOP VECTORIZATION

Michael Voss, Principal Engineer
Software and Services Group, Intel

With material used by permission
from J.D. Patel, Intel
from “Program Optimization Through Loop Vectorization” lecture slides by
María Garzarán, Saeed Maleki, William Gropp and David Padua, University of Illinois at Urbana-Champaign
from “Low-level Performance Analysis,” lecture slides by Pablo Reble.

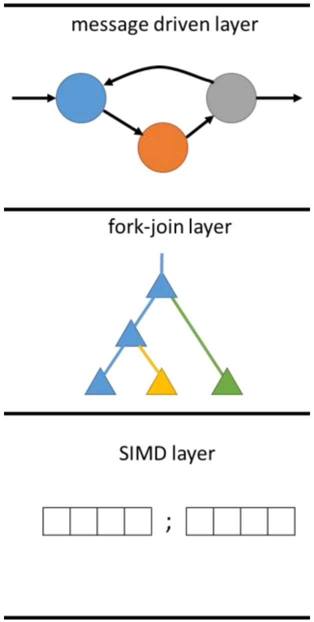
Outline

- What is vectorization and why is it important
- The different ways we can vectorize our code
- The two main challenges in vectorization
  - Determining that vectorization is legal (the results will be the same)
    - Dependence analysis
    - Obstacles to vectorization and how to deal with them
  - Optimizing performance
    - Memory issues (alignment, layout)
    - Telling the compiler what you know (about your code & about your platform)
- Using compiler intrinsics
- Using OpenMP simd pragmas
- A case study (after Spring Break)

Outline

- What is vectorization and why is it important
- The different ways we can vectorize our code
- The two main challenges in vectorization
  - Determining that vectorization is legal (the results will be the same)
    - Dependence analysis
    - Obstacles to vectorization and how to deal with them
  - Optimizing performance
    - Memory issues (alignment, layout)
    - Telling the compiler what you know (about your code & about your platform)
- Using compiler intrinsics
- Using OpenMP simd pragmas
- A case study

Hardware and software have evolved together



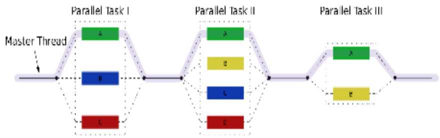
- There are different styles / models for expressing parallelism in applications
- These styles are often mixed in applications because they each best exploit a particular level of parallelism in the hardware
- For example MPI for message passing, OpenMP for fork-join parallelism and SIMD intrinsics for SIMD layer.

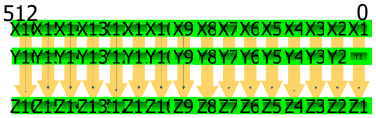
Arch D. Robison and Ralph E. Johnson. 2010. **Three layer cake for shared-memory programming**. In Proceedings of the 2010 Workshop on Parallel Programming Patterns (ParaPLoP '10). ACM, New York, NY, USA, , Article 5 , 8 pages. DOI=http://dx.doi.org/10.1145/1953611.1953616

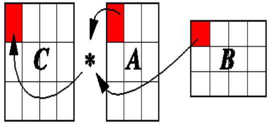
Different levels of parallelism in hardware

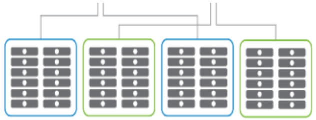
- Instruction Level Parallelism (**ILP**) -- Needs no user intervention
  - Micro-architectural techniques
    - Pipelined Execution
    - Super-scalar execution
    - Out-of/In-order execution
    - Branch prediction…
- Vector Level Parallelism (**VLP**)
  - Using **S**ingle **I**nstruction, **M**ultiple **D**ata (SIMD) vector processing instructions
    - Intel has introduced extensions over time: SSE, AVX/AVX2, AVX-512
    - SIMD registers width:
      - Intel CPUs: 64-bit (MMX) → 128-bit (SSE) → 256-bit (AVX,CORE-AVX2) → 512-bit (CORE-AVX512)
- Thread-Level Parallelism (**TLP**)
  - Multi/many-core architectures
  - Hyper threading (HT)
- Node Level Parallelism (**NLP**) (Distributed/Cluster/Grid Computing)

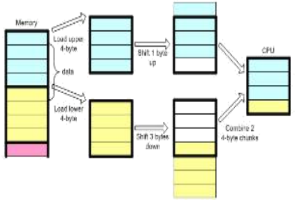
At Intel, we talk about “Modernized” Code





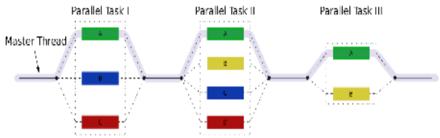


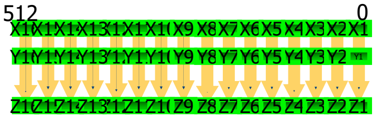


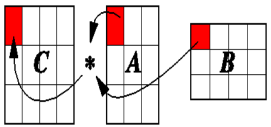


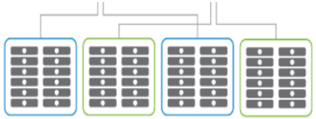
	What	Defined	Tools of the trade
1	**Thread Scaling**	Increase concurrent thread use across coherent shared memory	OpenMP, TBB, Cilk Plus
2	**Vector Scaling**	Use many wide-vector (512-bit) instructions	Vector loops, vector functions, array notations
3	**Cache Blocking**	Use algorithms to reduce memory bandwidth pressure and improve cache hit rate	Blocking algorithms
4	**Fabric Scaling**	Tune workload to increased node count	MPI
5	**Data Layout**	Optimize data layout for unconstrained performance	AoS→SoA, directives for alignment

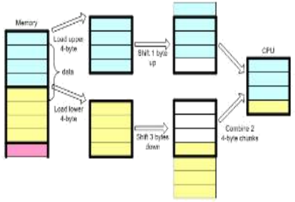
At Intel, we talk about “Modernized” Code





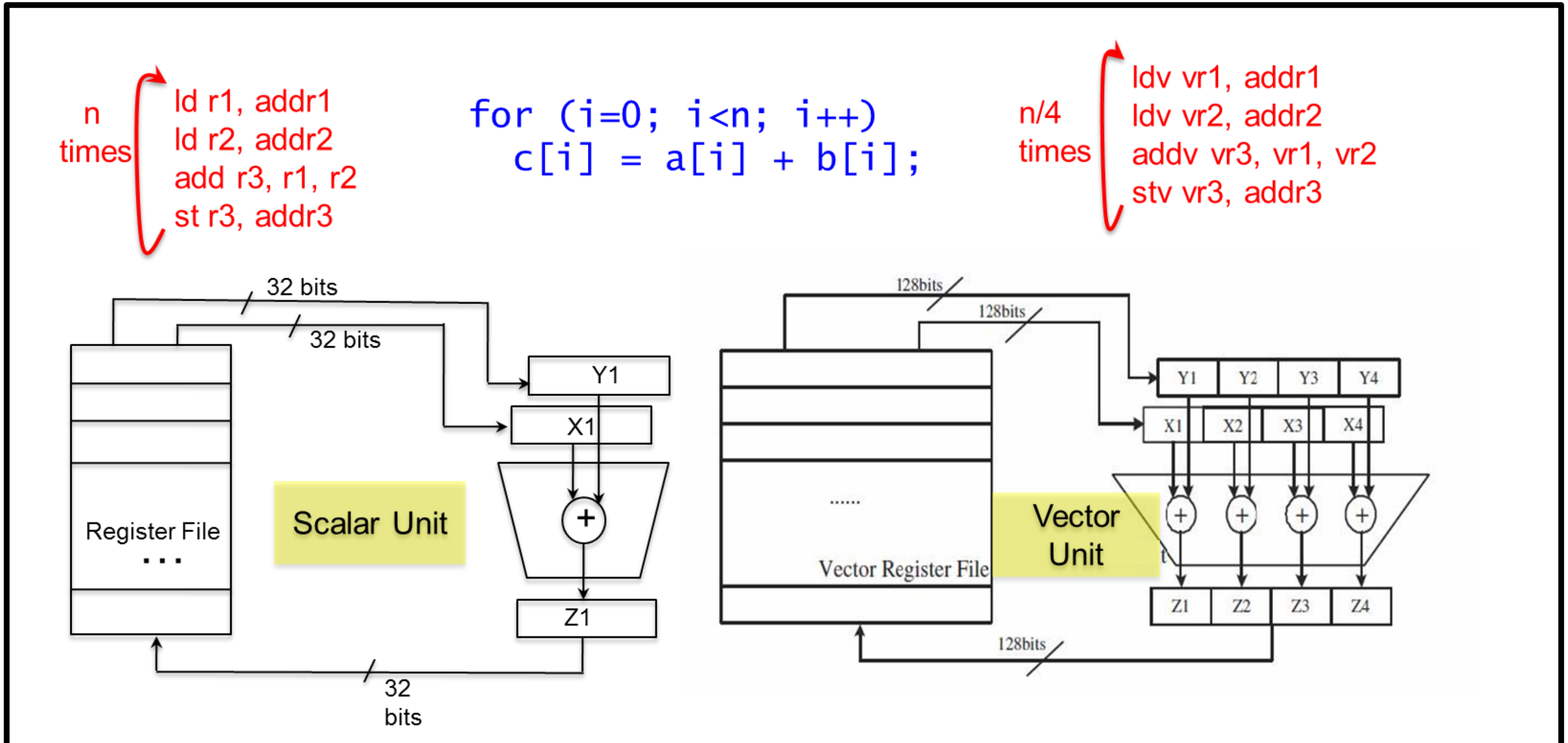






	What	Defined	Tools of the trade
1	**Thread Scaling**	Increase concurrent thread use across coherent shared memory	OpenMP, TBB, Cilk Plus
2	**Vector Scaling**	Use many wide-vector (512-bit) instructions	Vector loops, vector functions, array notations
3	**Cache Blocking**	Use algorithms to reduce memory bandwidth pressure and improve cache hit rate	Blocking algorithms
4	**Fabric Scaling**	Tune workload to increased node count	MPI
5	**Data Layout**	Optimize data layout for unconstrained performance	AoS→SoA, directives for alignment

Loop vectorization applies the same operation at the same time to several vector elements



Used by permission: María Garzarán, Saeed Maleki, William Gropp and David Padua

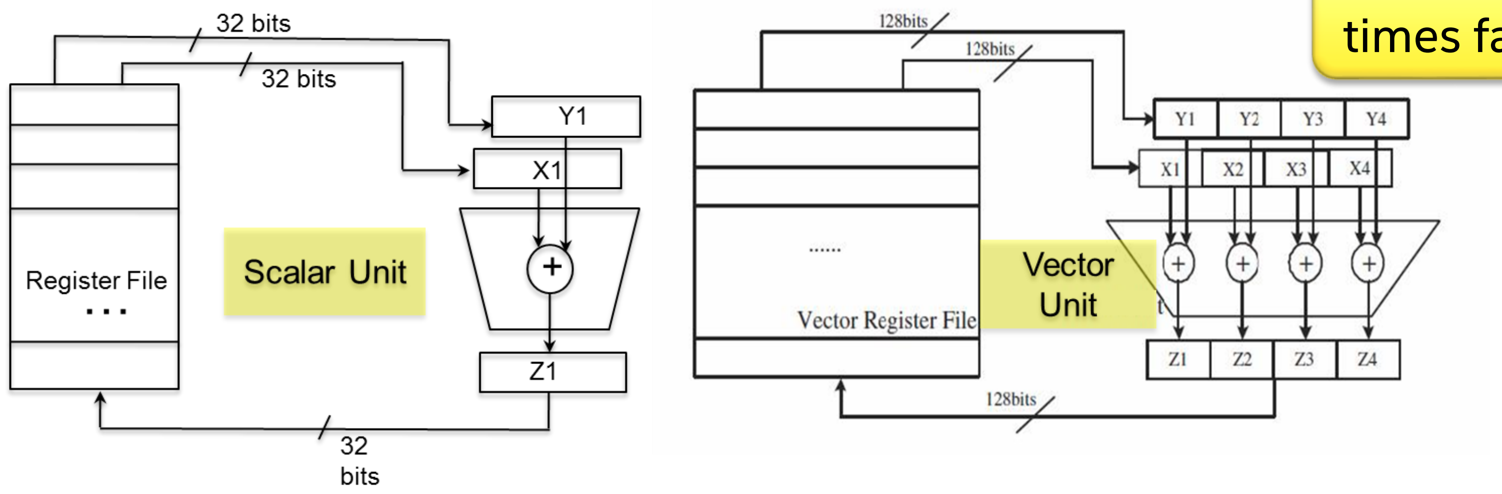
Loop vectorization applies the same operation at the same time to several vector elements

n times
```
ld r1, addr1
ld r2, addr2
add r3, r1, r2
st r3, addr3
```

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

n/4 times
```
ldv vr1, addr1
ldv vr2, addr2
addv vr3, vr1, vr2
stv vr3, addr3
```

Done 4 times faster!



Used by permission: María Garzarán, Saeed Maleki, William Gropp and David Padua

SIMD => **S**ingle **I**nstruction **M**ultiple **D**ata

VLP / Vectorization

Vectorization is the process of transforming a scalar operation acting on single data elements at a time (Single Instruction Single Data – SISD), to an operation acting on multiple data elements at once (Single Instruction Multiple Data – SIMD)

SIMD extensions	Width (bits)	DP (64-bit) calculations	FP (32-bit) calculations	Years introduced
SSE2/SSE3/SSE4	**128**	**2**	**4**	**~2001-2007**
AVX/AVX2	**256**	**4**	**8**	**~2011/2015**
AVX-512	**512**	**8**	**18**	**~2017**

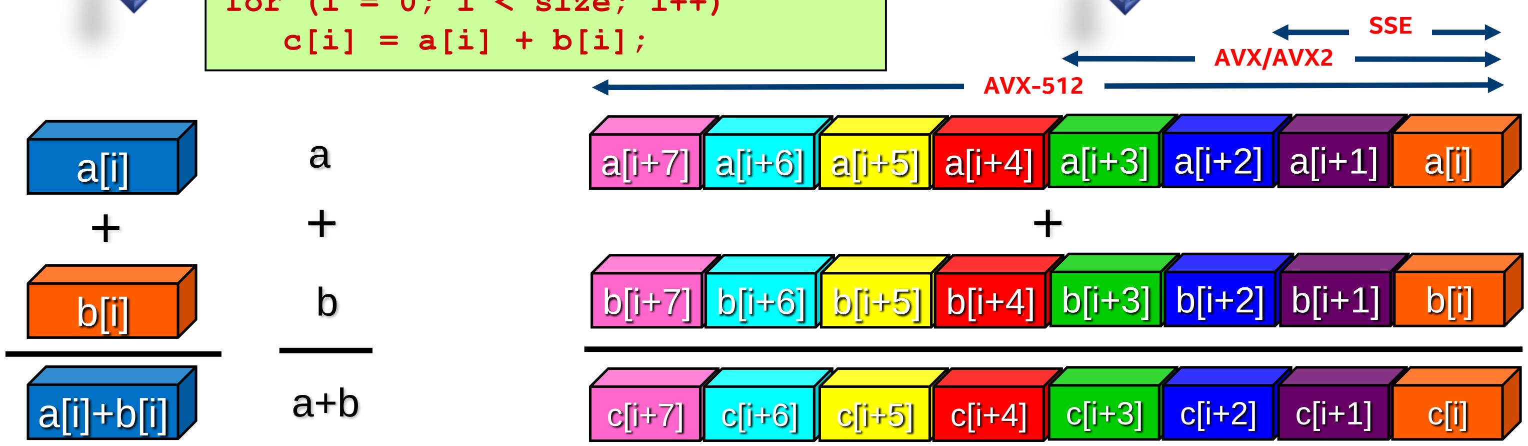
These are the Intel supported ISA extensions. Other platforms that support SIMD have different extensions.

SIMD => **S**ingle **I**nstruction **M**ultiple **D**ata

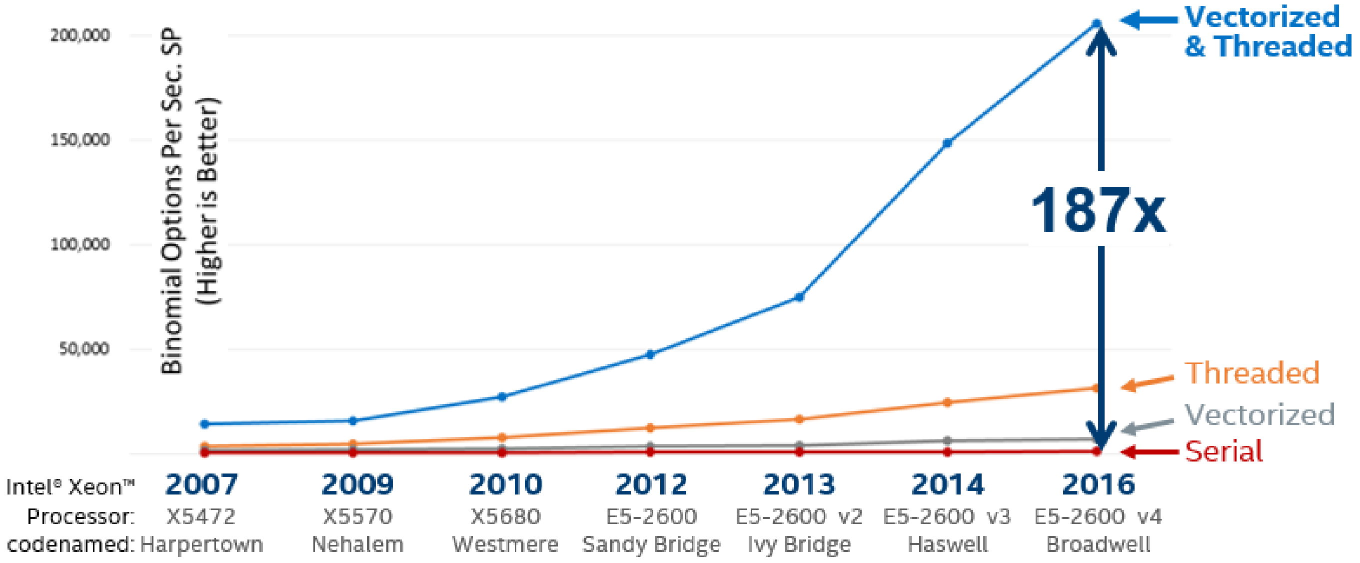
VLP / Vectorization

- **Scalar mode**
  - one instruction produces one result
  - e.g. vadd**sd** / vadd**ss** (s => scalar)
- **SIMD processing**
  - one instruction can produce multiple results (SIMD)
  - e.g. vadd**pd** / vadd**ps** (p => packed)

```
double *a,*b,*c; …
for (i = 0; i < size; i++)
   c[i] = a[i] + b[i];
```



The combined effect of vectorization and threading



The Difference Is Growing With Each New Generation of Hardware

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more information go to http://www.intel.com/performance Configurations at the end of this presentation.

Outline

- What is vectorization and why is it important
- The different ways we can vectorize our code
- The two main challenges in vectorization
  - Determining that vectorization is legal (the results will be the same)
    - Dependence analysis
    - Patterns that inhibit vectorization and how to deal with them
  - Optimizing performance
    - Memory issues (alignment, layout)
    - Telling the compiler what you know (about your code & about your platform)
- Using compiler intrinsics
- Using OpenMP simd pragmas
- A case study

How to write code to use the SIMD units

Hardest to use / Most Control

```
..B8.5
  movaps     a(,%rdx,4), %xmm0
  addps      b(,%rdx,4), %xmm0
  movaps     %xmm0, c(,%rdx,4)
  addq       $4, %rdx
  cmpq       $rdi, %rdx
  jl         ..B8.5
```

Assembly Language

```
void example(){
__m128 rA, rB, rC;
 for (int i = 0; i <LEN; i+=4){
    rA = _mm_load_ps(&a[i]);
    rB = _mm_load_ps(&b[i]);
    rC = _mm_add_ps(rA,rB);
   _mm_store_ps(&C[i], rC);
}}
```

Macros / Intrinsics

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

Vectorizing Compiler

Easiest to use / Least Control

Code snippets used by permission: María Garzarán, Saeed Maleki, William Gropp and David Padua

How to write code to use the SIMD units?

Hardest to use / Most Control

↓

Easiest to use / Least Control

1. Inline Assembly Language support
   - Most control but much harder to learn, code, debug, maintain...
2. SIMD intrinsics
   - Access to low level details similar to assembler but same issues
3. Compiler based Vectorization
   The fastest & easiest way; recommended for most cases
   - **Auto-Vectorization**
     - No code-changes; compiler vectorizes automatically for specified processor(s)
   - **Semi-Auto-Vectorization***
     - Use simple pragmas to guide compiler for missed auto-vectorization opportunities
     - Still hints to compiler, NOT mandatory!
   - **Explicit Vector Programming**
     - OpenMP SIMD-pragma, SIMD functions w/ powerful clauses… express code behavior better
     - Go after the performance opportunities that are missed by auto and semi-auto vectorization

Or, use a library that exploits the SIMD capabilities for you
e.g. the Intel® Math Kernel Library (Intel® MKL)

How to write code to use the SIMD units?

Hardest to use /
Most Control

Easiest to use /
Least Control

1. Inline Assembly Language support
   – Most control but much harder t[o] ... [ma]intain...
2. SIMD intrinsics
   – Access to low level details similar to assembler but same issues
3. Compiler based Vectorization
   The fastest & easiest way; recomme[n]ded for most cases
   – **Auto-Vectorization**
     – No code-changes; compiler vectorizes automatica[lly f]or specified processor(s)
   – **Semi-Auto-Vectorization***
     – Use simple pragmas to guide compiler for missed auto-ve[ctorization]
     – Still hints to compiler, NOT mandatory!
   – **Explicit Vector Programming**
     – OpenMP SIMD-pragma, SIMD functions w/ powerful clauses... express code behavior better
     – Go after the performance opportunities that're missed by auto and semi-auto vectorization

Will talk about this briefly

Main focus

Or, use a library that exploits the SIMD capabilities for you
e.g. the Intel® Math Kernel Library (Intel® MKL)

Outline

- What is vectorization and why is it important
- The different ways we can vectorize our code
- The two main challenges in vectorization
  - Determining that vectorization is legal (the results will be the same)
    - Dependence analysis
    - Obstacles to vectorization and how to deal with them
  - Optimizing performance
    - Memory issues (alignment, layout)
    - Telling the compiler what you know (about your code & about your platform)
- Using compiler intrinsics
- Using OpenMP simd pragmas
- A case study

Some slides are taken from:

Program Optimization Through Loop Vectorization

María Garzarán, Saeed Maleki
William Gropp and David Padua

Department of Computer Science
University of Illinois at Urbana-Champaign

Data dependences

- The notion of dependence is the foundation of the process of vectorization.
- It is used to build a calculus of program transformations that can be applied manually by the programmer or automatically by a compiler.

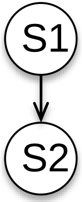
Definition of Dependence

- A statement S is said to be data dependent on statement T if
  - T executes before S in the original sequential/scalar program
  - S and T access the same data item
  - At least one of the accesses is a write.

Data Dependence

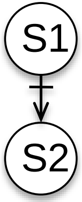
Flow dependence (True dependence)

S1: X = A+B
S2: C= X+A



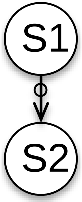
Anti dependence

S1: A = X + B
S2: X= C + D



Output dependence

S1: X = A+B
S2: X= C + D



Data Dependence

- Dependences indicate an execution order that must be honored.
- Executing statements in the order of the dependences guarantee correct results.
- Statements not dependent on each other can be reordered, executed in parallel, or coalesced into a vector operation.

Dependences in Loops (I)

- Dependences in loops are easy to understand if the loops are unrolled. Now the dependences are between statement “executions”.

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

Dependences in Loops (I)

- Dependences in loops are easy to understand if loops are unrolled. Now the dependences are between statement "executions"

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

i=0	i=1	i=2
S1: a[0] = b[0] + 1 S2: c[0] = a[0] + 2	S1: a[1] = b[1] + 1 S2: c[1] = a[1] + 2	S1: a[2] = b[2] + 1 S2: c[2] = a[2] + 2

Dependences in Loops (I)

- Dependences in loops are easy to understand if loops are unrolled. Now the dependences are between statement “executions”

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

i=0	i=1	i=2
S1: a[0] = b[0] + 1	S1: a[1] = b[1] + 1	S1: a[2] = b[2] + 1
S2: c[0] = a[0] + 2	S2: c[1] = a[1] + 2	S2: c[2] = a[2] + 2

Dependences in Loops (I)

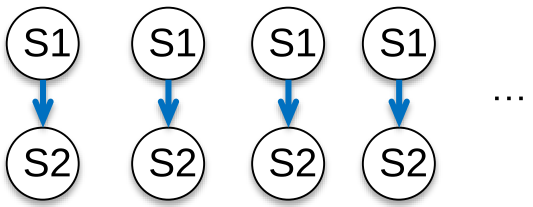
- Dependences in loops are easy to understand if loops are unrolled. Now the dependences are between statement "executions"

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

iteration: 0 1 2 3 …

instances of S1:

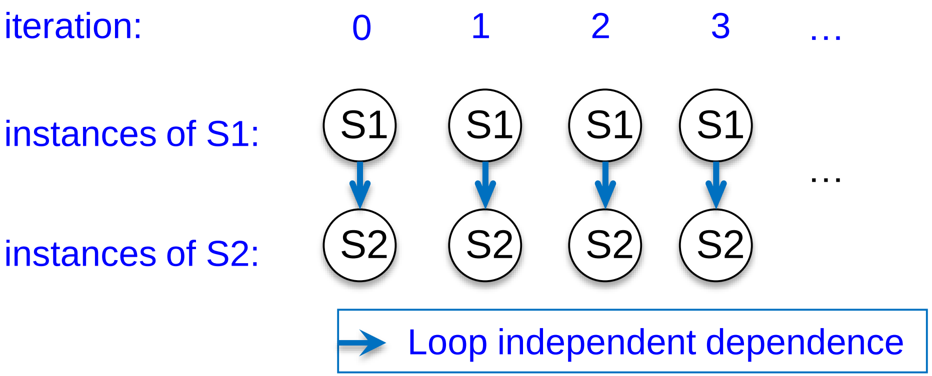
instances of S2:



Dependences in Loops (I)

- Dependences in loops are easy to understand if loops are unrolled. Now the dependences are between statement “executions”

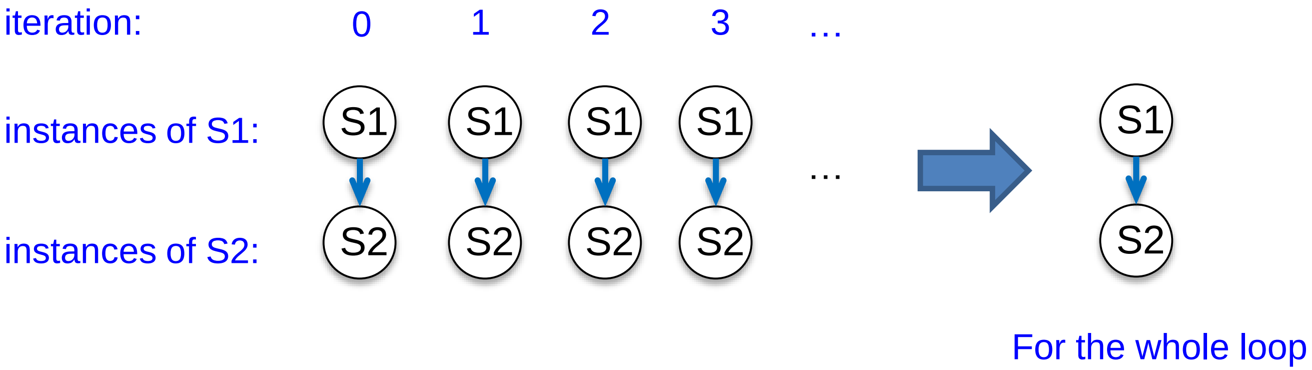
```
   for (i=0; i<n; i++){
S1   a[i] = b[i] + 1;
S2   c[i] = a[i] + 2;
   }
```



Dependences in Loops (I)

- Dependences in loops are easy to understand if loops are unrolled. Now the dependences are between statement "executions"

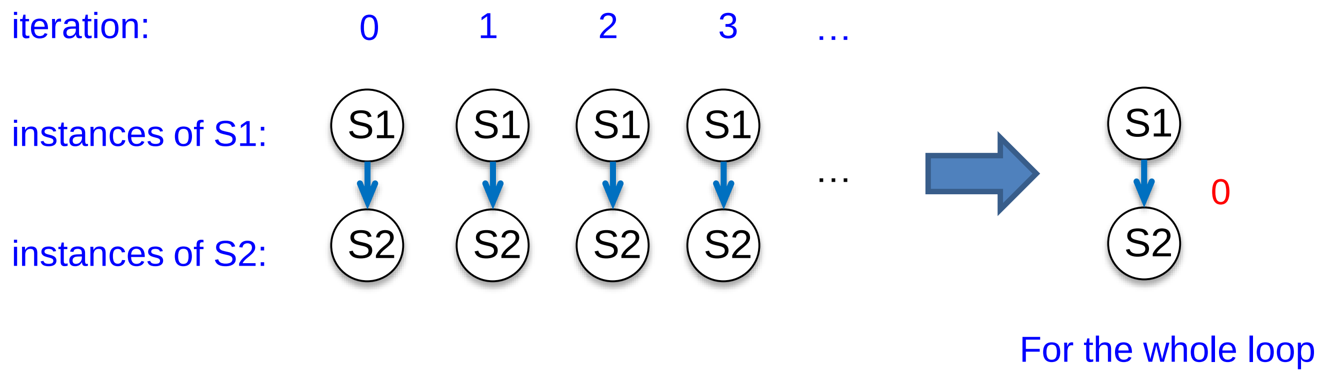
```
   for (i=0; i<n; i++){
S1   a[i] = b[i] + 1;
S2   c[i] = a[i] + 2;
   }
```



Dependences in Loops (I)

- Dependences in loops are easy to understand if loops are unrolled. Now the dependences are between statement “executions”

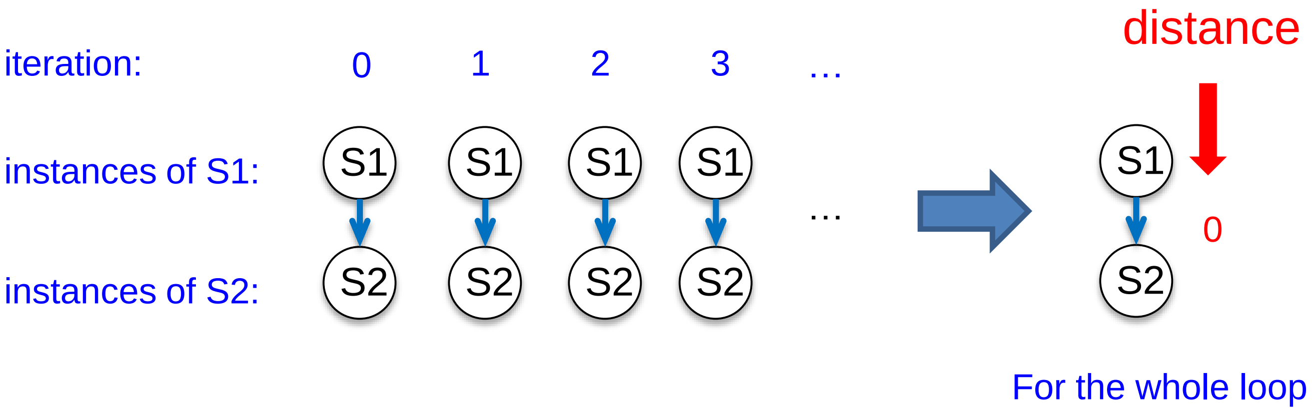
```
      for (i=0; i<n; i++){
S1      a[i] = b[i] + 1;
S2      c[i] = a[i] + 2;
      }
```



Dependences in Loops (I)

- Dependences in loops are easy to understand if loops are unrolled. Now the dependences are between statement “executions”

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



Dependences in Loops (I)

- Dependences in loops are easy to understand if loops are unrolled. Now the dependences are between statement "executions"

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

For the dependences shown here, we assume that arrays do not overlap in memory (no aliasing). Compilers must know that there is no aliasing in order to vectorize.

Dependences in Loops (II)

- Dependences in loops are easy to understand if loops are unrolled. Now the dependences are between statement “executions”

```
   for (i=1; i<n; i++){
S1   a[i] = b[i] + 1;
S2   c[i] = a[i-1] + 2;
   }
```

Dependences in Loops (II)

- Dependences in loops are easy to understand if loops are unrolled. Now the dependences are between statement "executions"

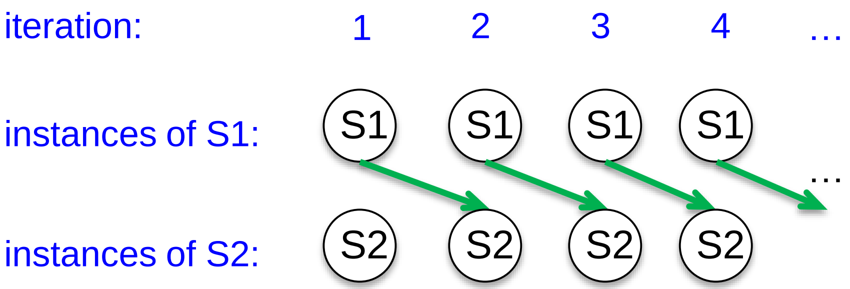
```
     for (i=1; i<n; i++){
S1     a[i] = b[i] + 1;
S2     c[i] = a[i-1] + 2;
     }
```

i=1	i=2	i=3
S1: a[1] = b[1] + 1	S1: a[2] = b[2] + 1	S1: a[3] = b[3] + 1
S2: c[1] = a[0] + 2	S2: c[2] = a[1] + 2	S2: c[3] = a[2] + 2

Dependences in Loops (II)

- Dependences in loops are easy to understand if loops are unrolled. Now the dependences are between statement "executions"

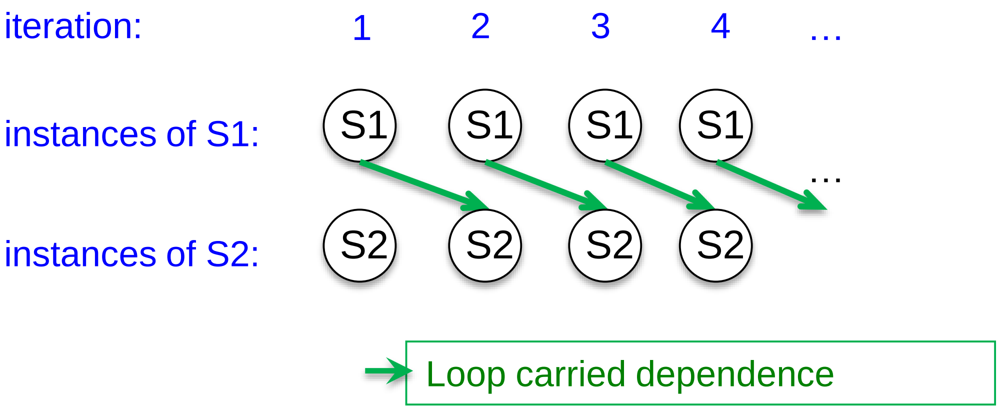
```
   for (i=1; i<n; i++){
S1   a[i] = b[i] + 1;
S2   c[i] = a[i-1] + 2;
   }
```



Dependences in Loops (II)

- Dependences in loops are easy to understand if loops are unrolled. Now the dependences are between statement “executions”

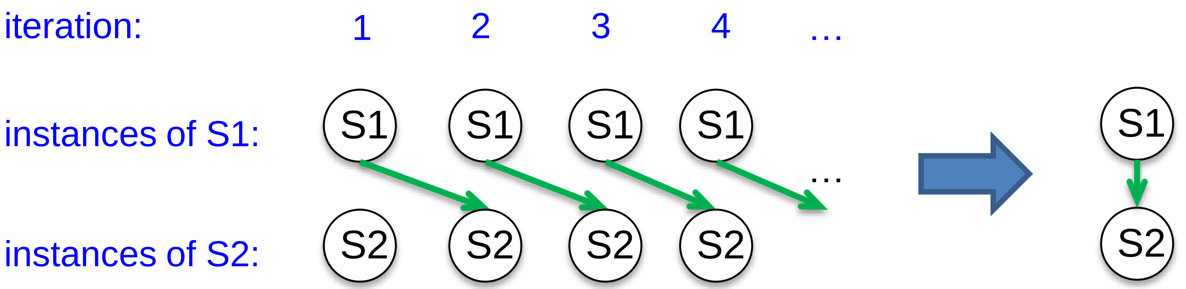
```
     for (i=1; i<n; i++){
S1     a[i] = b[i] + 1;
S2     c[i] = a[i-1] + 2;
     }
```



Dependences in Loops (II)

- Dependences in loops are easy to understand if loops are unrolled. Now the dependences are between statement "executions"

```
     for (i=1; i<n; i++){
S1     a[i] = b[i] + 1;
S2     c[i] = a[i-1] + 2;
     }
```

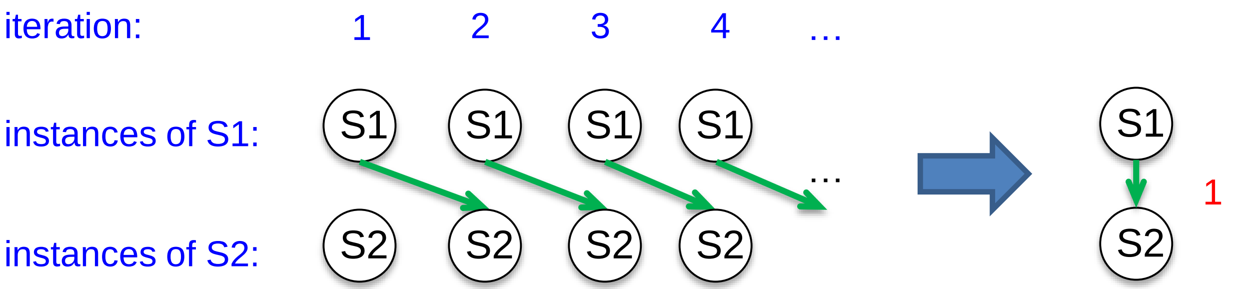


For the whole loop

Dependences in Loops (II)

- Dependences in loops are easy to understand if loops are unrolled. Now the dependences are between statement "executions"

```
     for (i=1; i<n; i++){
S1     a[i] = b[i] + 1;
S2     c[i] = a[i-1] + 2;
     }
```

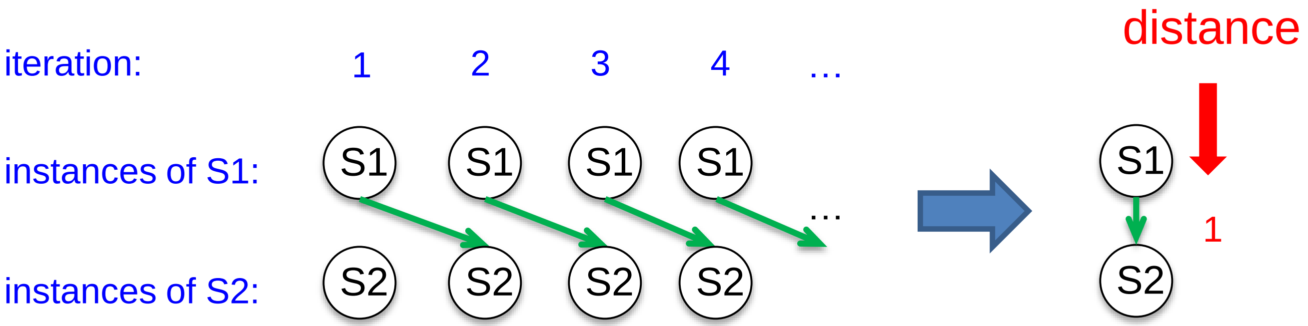


For the whole loop

Dependences in Loops (II)

- Dependences in loops are easy to understand if loops are unrolled. Now the dependences are between statement "executions"

```
     for (i=1; i<n; i++){
S1     a[i] = b[i] + 1;
S2     c[i] = a[i-1] + 2;
     }
```



For the whole loop

Dependences in Loops (III)

- Dependences in loops are easy to understand if loops are unrolled. Now the dependences are between statement “executions”

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

Dependences in Loops (III)

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

i=0	i=1	i=2
S1: a= b[0] + 1	S1: a = b[1] + 1	S1: a = b[2] + 1
S2: c[0] = a + 2	S2: c[1] = a + 2	S2: c[2] = a+ 2

Dependences in Loops (III)

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

i=0

S1: a= b[0] + 1
S2: c[0] = a + 2

i=1

S1: a = b[1] + 1
S2: c[1] = a + 2

i=2

S1: a = b[2] + 1
S2: c[2] = a+ 2

Loop independent dependence
Loop carried dependence

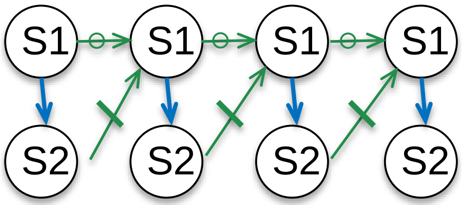
Dependences in Loops (III)

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

iteration: 0 1 2 3 …

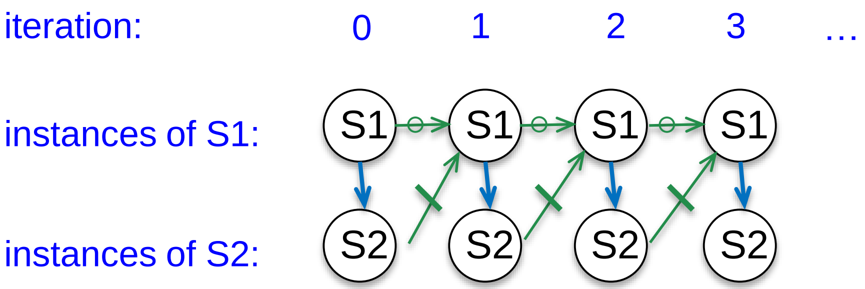
instances of S1:

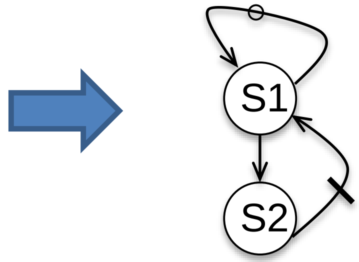
instances of S2:



Dependences in Loops (III)

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





Dependences in Loops (IV)

- Doubly nested loops

```
 for (i=1; i<n; i++) {
  for (j=1; j<n; j++) {
S1  a[i][j]=a[i][j-1]+a[i-1][j];
 }}
```

Dependences in Loops (IV)

```
    for (i=1; i<n; i++) {
     for (j=1; j<n; j++) {
S1    a[i][j]=a[i][j-1]+a[i-1][j];
    }}
```

	i=1	i=2
j=1	a[1][1] = a[1][0] + a[0][1]	a[2][1] = a[2][0] + a[1][1]
j=2	a[1][2] = a[1][1] + a[0][2]	a[2][2] = a[2][1] + a[1][2]
j=3	a[1][3] = a[1][2] + a[0][3]	a[2][3] = a[2][2] + a[1][3]
j=4	a[1][4] = a[1][3] + a[0][4]	a[2][4] = a[2][3] + a[1][4]

Loop carried dependences

Dependences in Loops (IV)

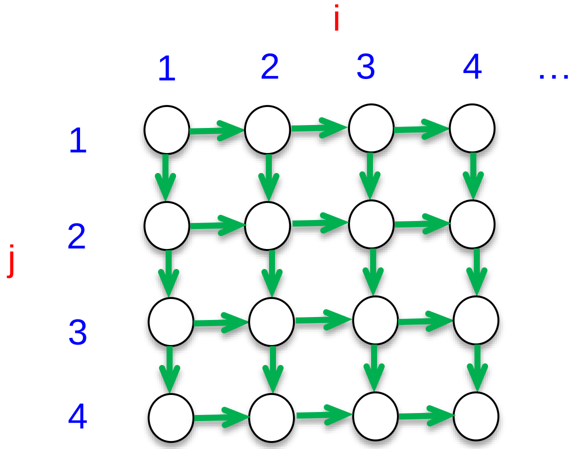
```
 for (i=1; i<n; i++) {
  for (j=1; j<n; j++) {
S1  a[i][j]=a[i][j-1]+a[i-1][j];
 }}
```

	i=1	i=2
j=1	a[1][1] = a[1][0] + a[0][1]	a[2][1] = a[2][0] + a[1][1]
j=2	a[1][2] = a[1][1] + a[0][2]	a[2][2] = a[2][1] + a[1][2]
j=3	a[1][3] = a[1][2] + a[0][3]	a[2][3] = a[2][2] + a[1][3]
j=4	a[1][4] = a[1][3] + a[0][4]	a[2][4] = a[2][3] + a[1][4]

Loop carried dependences

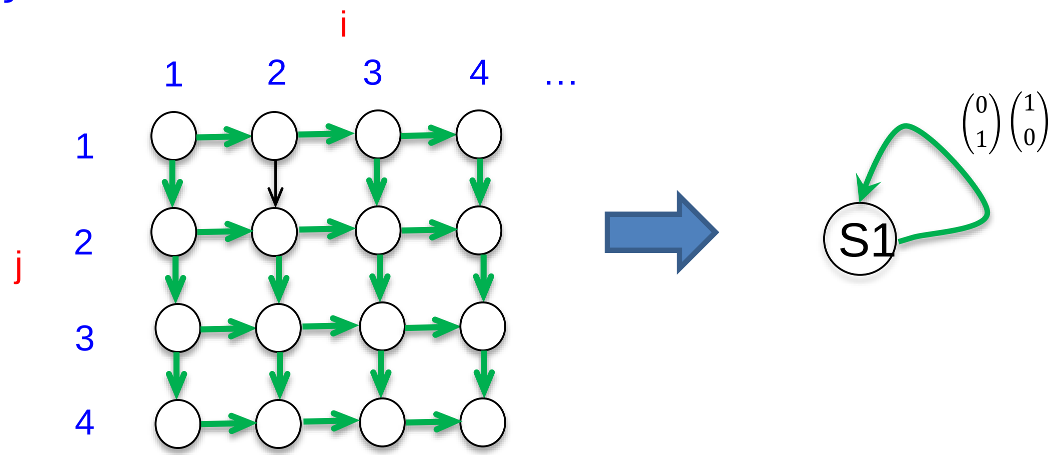
Dependences in Loops (IV)

```
 for (i=1; i<n; i++) {
  for (j=1; j<n; j++) {
S1  a[i][j]=a[i][j-1]+a[i-1][j];
 }}
```



Dependences in Loops (IV)

```
  for (i=1; i<n; i++) {
   for (j=1; j<n; j++) {
S1  a[i][j]=a[i][j-1]+a[i-1][j];
  }}
```



Data dependences and vectorization

- Loop dependences guide vectorization
- **Main idea:** A statement inside a loop which is not in a cycle of the dependence graph can be vectorized.

```
     for (i=0; i<n; i++){
S1     a[i] = b[i] + 1;
     }
```

```
a[0:n-1] = b[0:n-1] + 1;
```

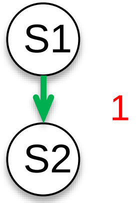
S1

Data dependences and vectorization

- **Main idea:** A statement inside a loop which is not in a cycle of the dependence graph can be vectorized.

```
   for (i=1; i<n; i++){
S1   a[i] = b[i] + 1;
S2   c[i] = a[i-1] + 2;
   }
```

```
a[1:n] = b[1:n] + 1;
c[1:n] = a[0:n-1] + 2;
```



Stripmining

- Stripmining is a simple transformation.

```
for (i=1; i<n; i++){
   …
}
```

```
/* n is a multiple of q */
for (k=1; k<n; k+=q){
   for (i=k; i<k+q-1; i++){
   …
   }
}
```

- It is typically used to improve locality.

Stripmining (cont.)

- Stripmining is often used when vectorizing

```
for (i=1; i<n; i++){
  a[i] = b[i] + 1;
  c[i] = a[i-1] + 2;
}
```

stripmine

```
for (k=1; k<n; k+=q){
/* q could be size of vector register */
  for (i=k; i < k+q; i++){
     a[i] = b[i] + 1;
     c[i] = a[i-1] + 2;
  }
}
```

vectorize

```
for (i=1; i<n; i+=q){
  a[i:i+q-1] = b[i:i+q-1] + 1;
  c[i:i+q-1] = a[i-1:i+q] + 2;
}
```

Outline

- What is vectorization and why is it important
- The different ways we can vectorize our code
- The two main challenges in vectorization
  - Determining that vectorization is legal (the results will be the same)
    - Dependence analysis
    - Obstacles to vectorization and how to deal with them
  - Optimizing performance
    - Memory issues (alignment, layout)
    - Telling the compiler what you know (about your code & about your platform)
- Using compiler intrinsics
- Using OpenMP simd pragmas
- A case study

Factors that Impact Compiler Vectorization

Acyclic dependencies

```
for (i = 1; i < N: ++i) {
  a[i] = b[i] + c[i];
  d[i] = a[i] + (float)1.0;
}
```

Loop-carried dependencies

```
for (i = 1; i < N: ++i) {
   a[i] = a[i-1] + b[i];
}
```

Function calls

```
for (i = 1; i < nx; i++) {
  x = x0 + i * h;
  sumx = sumx + func(x, y, xp);
}
```

Pointer aliasing

```
void scale(int *a, int *b, int size)
{
   for (int i = 0; i < size; i++)
       b[i] = z * a[i];
}
```

Unknown/aliased loop iteration count

```
struct _x { int d; int bound; };

void doit(int *a, struct _x *x)
{
  for(int i = 0; i < x->bound; i++)
    a[i] = 0;
}
```

Indirect memory access

```
for (i = 0; i < N; ++i) {
     a[b[i]] += c[i]*d[i];
}
```

And many More ……

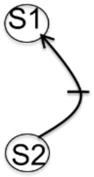
Factors that Impact Compiler Vectorization

Acyclic dependencies

```
for (i = 1; i < N: ++i) {
   a[i] = b[i] + c[i];
   d[i] = a[i] + (float)1.0;
}
```

- Forward dependences are typically handled by a compiler
- Backward dependences may need reordering, but compiler may do it:

```
for (i = 1; i < N: ++i) {
   a[i] = b[i] + c[i];
   d[i] = a[i+1] + (float)1.0;
}
```



reordered

```
for (i = 1; i < N: ++i) {
   d[i] = a[i+1] + (float)1.0;
   a[i] = b[i] + c[i];
}
```

Factors that Impact Compiler Vectorization

Loop-carried dependencies

```
for (i = 1; i < N: ++i) {
   a[i] = a[i-1] + b[i];
}
```

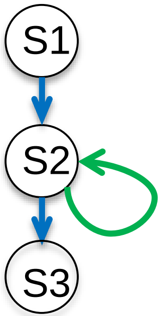
- There should be no loop-carried dependencies.
- For example, the loop must not require statement of iteration 1 to be executed before statement of iteration 2 for correct results.
- This allows consecutive iterations of the original loop to be executed simultaneously in a single iteration of the unrolled, vectorized loop.

Data dependences and transformations

- When cycles are present, vectorization can be achieved by:
  - Separating (distributing) the statements not in a cycle
  - Removing dependences
  - Freezing loops
  - Changing the algorithm

Distributing

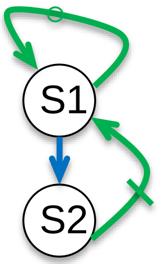
```
  for (i=1; i<n; i++){
S1  b[i] = b[i] + c[i];
S2  a[i] = a[i-1]*a[i-2]+b[i];
S3  c[i] = a[i] + 1;
  }
```



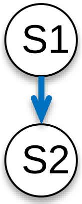

```
b[1:n-1] = b[1:n-1] + c[1:n-1];
for (i=1; i<n; i++){
     a[i] = a[i-1]*a[i-2]+b[i];
}
c[1:n-1] = a[1:n-1] + 1;
```

Removing dependences (Scalar Expansion)

```
    for (i=0; i<n; i++){
S1    a = b[i] + 1;
S     c[i] = a + 2;
2   }
```




```
    for (i=0; i<n; i++){
S1    a'[i] = b[i] + 1;
S     c[i] = a'[i] + 2;
2   }
    a=a'[n-1]
```



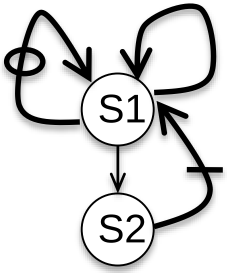

```
S1    a'[0:n-1] = b[0:n-1] + 1;
S     c[0:n-1] = a'[0:n-1] + 2;
2     a=a'[n-1]
```

Removing dependences (Induction variables)

- Induction variable is a variable that can be expressed as a function of the loop iteration variable

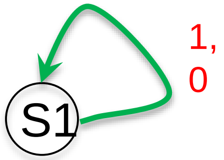
```
float s = (float)0.0;
for (int i=0;i<LEN;i++){
  s += (float)2.;
  a[i] = s * b[i];
}
```

```
for (int i=0;i<LEN;i++){
  a[i] = (float)2.*(i+1)*b[i];
}
```



Freezing Loops

```
for (i=1; i<n; i++) {
  for (j=1; j<n; j++) {
    a[i][j]=a[i][j]+a[i-1][j];
  }
}
```



Ignoring (freezing) the outer loop:

```
for (j=1; j<n; j++) {
    a[i][j]=a[i][j]+a[i-1][j];
}
```


```
for (i=1; i<n; i++) {
    a[i][1:n-1]=a[i][1:n-1]+a[i-1][1:n-1];
}
```

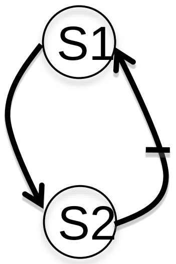
There are Many Different Kinds of Loop Transformations that can Enable Vectorization:

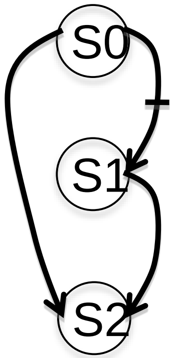
- Loop Distribution or loop fission
- Reordering Statements
- Node Splitting
- Scalar expansion
- Loop Peeling
- Loop Fusion
- Loop Unrolling
- Loop Interchanging

Node Splitting

```
  for (int i=0;i<LEN-1;i++){
S1  a[i]=b[i]+c[i];
S2  d[i]=(a[i]+a[i+1])*(float)0.5;
  }
```

```
  for (int i=0;i<LEN-1;i++){
S0    temp[i]=a[i+1];
S1    a[i]=b[i]+c[i];
S2    d[i]=(a[i]+temp[i])*(float) 0.5;
  }
```



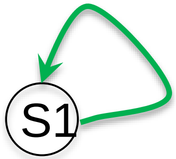


Changing the algorithm

- When there is a recurrence, it is necessary to change the algorithm in order to vectorize.
- Compilers use pattern matching to identify the recurrence and then replace it with a parallel version.
- Examples or recurrences include:
  - Reductions (`S+=A[i]`)
  - Linear recurrences (`A[i]=B[i]*A[i-1]+C[i]`)
  - Boolean recurrences (`if (A[i]>max) max = A[i]`)

Changing the algorithm (cont.)

S1 a[0]=b[0];
for (i=1; i<n; i++)
S2 a[i]=a[i-1]+b[i];



a[0:n-1]=b[0:n-1];
for (i=0;i<k;i++) /* n = 2^k */
a[2**i:n-1]=a[2**i:n-1]+b[0:n-2**i];

Factors that Impact Compiler Vectorization

Function calls

```
for (i = 1; i < nx; i++) {
    x = x0 + i * h;
    sumx = sumx + func(x, y, xp);
}
```

- There should be no special operators and no function or subroutine calls, unless these are inlined, either manually or automatically by the compiler, or they are SIMD (vectorized) functions.
- Intrinsic math functions such as sin(), log(), fmax(), etc. ***may be*** allowed, since the compiler runtime library ***may*** contain SIMD (vectorized) versions of these functions.

Factors that Impact Compiler Vectorization

Pointer aliasing

```
void scale(int *a, int *b, int size)
{
  for (int i = 0; i < size; i++)
        b[i] = z * a[i];
}
```

- Sometimes the compiler cannot safely vectorize a loop if there is even a potential dependency. The compiler must ask itself whether, for some iteration i, b[i] might refer to the same memory location of a[i] for a different iteration.
- For example, if a[i] pointed to the same memory location as b[i-1], there would be a read-after-write dependency as in the earlier example.
- If the compiler cannot exclude this possibility, it will not vectorize the loop (at least, not without help, we might help by using **#pragma ivdep** or the **restrict** keyword.

Factors that Impact Compiler Vectorization

Unknown/aliased loop iteration count

```
struct _x { int d; int bound; };

void doit(int *a, struct _x *x)
{
   for(int i = 0; i < x->bound; i++)
       a[i] = 0;
}
```

- The loop must be countable, i.e. the number of iterations must be known before the loop starts to execute, though it need not be known at compile time.
- Consequently, there must be no data-dependent exit conditions.

Factors that Impact Compiler Vectorization

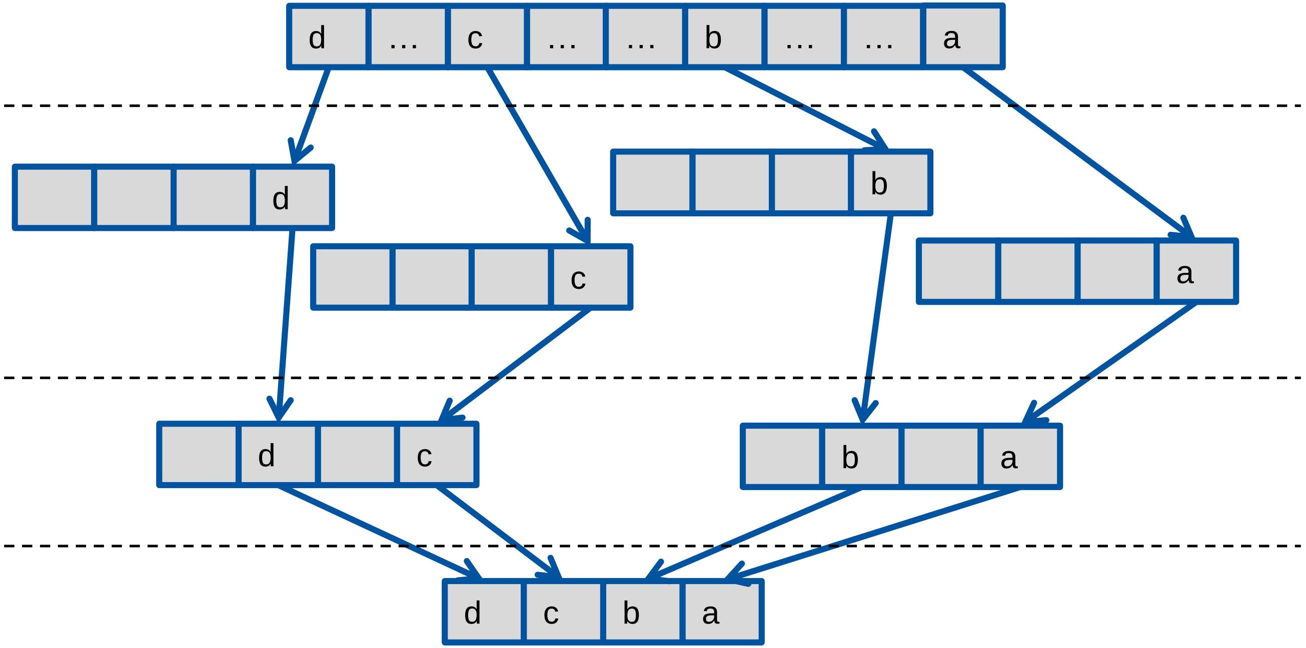
Indirect memory access

```
for (i = 0; i < N; ++i) {
        a[b[i]] += c[i]*d[i];
}
```

- The following do not always prevent vectorization, but frequently either prevent it or cause the compiler to decide that vectorization would not be worthwhile.
- Four consecutive ints or floats, or two consecutive doubles, may be loaded directly from memory in a single SSE instruction. But if the four ints are not adjacent, they must be loaded separately using multiple instructions, which is considerably less efficient.
- The most common example of non-contiguous memory access are loops with non-unit stride or with indirect addressing, as in the example. The typical message from the vectorization report is "vectorization possible but seems inefficient".
- Although indirect addressing may also result in "Existence of vector dependence".

Example: Load 4 float from arbitrary memory

- SSE2 version:



Used by permission of Pablo Reble

Outline

- What is vectorization and why is it important
- The different ways we can vectorize our code
- The two main challenges in vectorization
    - Determining that vectorization is legal (the results will be the same)
        - Dependence analysis
        - Obstacles to vectorization and how to deal with them
    - Optimizing performance
        - Memory issues (alignment, layout)
        - Telling the compiler what you know (about your code & about your platform)
- Using compiler intrinsics
- Using OpenMP simd pragmas
- A case study

Vectorization needs to be legal *and* profitable

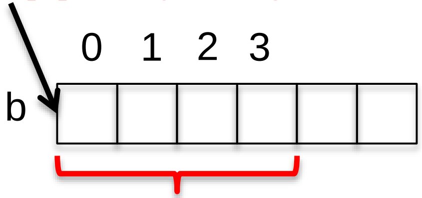
- Eliminating dependences make it legal but not necessarily profitable
- Memory issues are a big source of extra cost that can impact profitability
- The main issues are
  - alignment (16 bytes for SSE, 32 bytes for AVX/AVX2, 64 bytes for AVX 512)
  - aliasing
  - and non-consecutive layout in memory (non-unit strides)

Data Alignment

- SSE Vector loads/stores 128 consecutive bits to/from a vector register.
- Data addresses need to be 16-byte (128 bits) aligned to be loaded/stored for SSE, 32-byte aligned for AVX/AVX2 and 64-byte aligned for AVX512
  - Intel platforms support aligned and unaligned load/stores, but unaligned is slower

```
void test1(float *a,float *b,float *c)
{
     for (int i=0;i<LEN;i++){
       a[i] = b[i] + c[i];
}
```

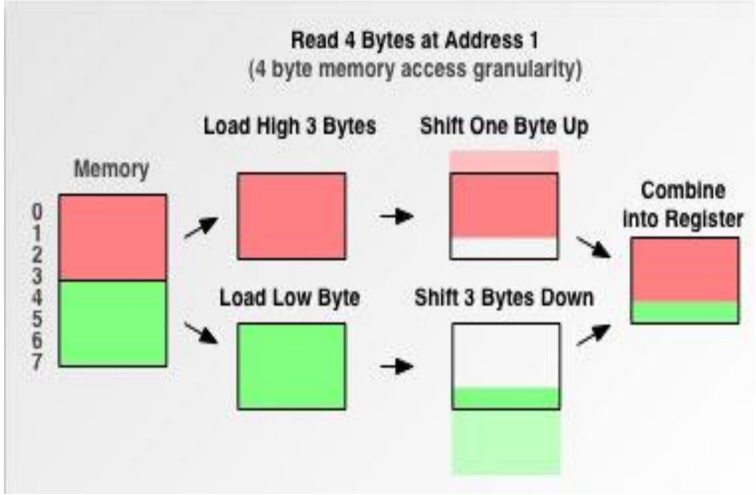
Is &b[0] 16-byte aligned?



vector load loads b[0] … b[3]

Why data alignment may improve efficiency

- Vector load/store from aligned data requires one memory access
- Vector load/store from unaligned data requires multiple memory accesses and some shift operations



Reading 4 bytes from address 1
requires two loads

Data Alignment

- To know if a pointer is 16-byte aligned, the last digit of the pointer address in hex must be 0.
- Note that if &b[0] is 16-byte aligned, and is a single precision array, then &b[4] is also 16-byte aligned

```
__attribute__ ((aligned(16))) float  B[1024];

int main(){
  printf("%p, %p\n", &B[0], &B[4]);
}
```

Output:
0x7fff1e9d8580, 0x7fff1e9d8590

Data Alignment

- In many cases, the compiler cannot statically know the alignment of the address in a pointer
- The compiler assumes that the base address of the pointer is 16-byte aligned and adds a run-time checks for it
  - if the runtime check is false, then it uses another code (which may be scalar)

Data Alignment

- Manual 16-byte alignment can be achieved by forcing the base address to be a multiple of 16.

```
__attribute__ ((aligned(16))) float b[N];
float* a = (float*) memalign(16,N*sizeof(float));
```

- When the pointer is passed to a function, the compiler should be aware of where the 16-byte aligned address of the array starts.

```
void func1(float *a, float *b, float *c)
{
  __assume_aligned(a, 16);
  __assume_aligned(b, 16);
  __assume_aligned(c, 16);
  for int (i=0; i<LEN; i++) {
    a[i] = b[i] + c[i];
  }
```

Aligning Data in C/C++

```
void* _mm_malloc(int size, int n)
int posix_memaligned(void **p, size_t n, size_t size)
```

`__attribute__((aligned(n))) var_name` or

`__declspec(align(n)) var_name` (Windows)

No need to do ➔ `new (_mm_malloc(sizeof(X), alignof(X))) X`

Instead ➔

```
#include <aligned_new>
```

Then ➔ `void *operator new (size_t, align_val_t);`

Or ➔ `void *operator new[] (size_t, align_val_t);`

AND TELL the compiler at use…

`#pragma vector aligned` or `#pragma simd aligned` or `#pragma omp simd aligned`

`__assume_aligned(array, n)`

- Compiler may assume array is aligned to n byte boundary
- May cause fault if data are not aligned

n=64 for AVX-512, n=32 for AVX/AVX2, n=16 for SSE

http://software.intel.com/en-us/articles/data-alignment-to-assist-vectorization

Data Alignment - Example

```
float A[N] __attribute__((aligned(16)));
float B[N] __attribute__((aligned(16)));
float C[N] __attribute__((aligned(16)));

void test(){
for (int i = 0; i < N; i++){
  C[i] = A[i] + B[i];
}}
```

Data Alignment - Example

```
float A[N] __attribute__((aligned(16)));
float B[N] __attribute__((aligned(16)));
float C[N] __attribute__((aligned(16)));

void test1(){
__m128 rA, rB, rC;
 for (int i = 0; i < N; i+=4){
  rA = _mm_load_ps(&A[i]);
  rB = _mm_load_ps(&B[i]);
  rC = _mm_add_ps(rA,rB);
  _mm_store_ps(&C[i], rC);
}}
```

```
void test2(){
__m128 rA, rB, rC;
for (int i = 0; i < N; i+=4){
  rA = _mm_loadu_ps(&A[i]);
  rB = _mm_loadu_ps(&B[i]);
  rC = _mm_add_ps(rA,rB);
  _mm_storeu_ps(&C[i], rC);
}}
```

```
void test3(){
__m128 rA, rB, rC;
for (int i = 1; i < N-3; i+=4){
  rA = _mm_loadu_ps(&A[i]);
  rB = _mm_loadu_ps(&B[i]);
  rC = _mm_add_ps(rA,rB);
  _mm_storeu_ps(&C[i], rC);
}}
```

Nanosecond per iteration		
	Core 2 Duo	Intel i7
Aligned	0.577	0.580
Aligned (unaligned ld)	0.689	0.581
Unaligned	2.176	0.629

Alignment in a struct

```
struct st{
  char A;
  int B[64];
  float C;
  int D[64];
};

int main(){
  st s1;
  printf("%p, %p, %p, %p\n", &s1.A, s1.B, &s1.C, s1.D);}
```

Output:
0x7fffe6765f00, 0x7fffe6765f04, 0x7fffe6766004, 0x7fffe6766008

- Arrays B and D are not 16-bytes aligned (see the address)

Alignment in a struct

```
struct st{
  char A;
  int B[64] __attribute__ ((aligned(16)));
  float C;
  int D[64] __attribute__ ((aligned(16)));
};

int main(){
  st s1;
  printf("%p, %p, %p, %p\n", &s1.A, s1.B, &s1.C, s1.D);}
```

Output:
0x7fff1e9d8580, 0x7fff1e9d8590, 0x7fff1e9d8690, 0x7fff1e9d86a0

- Arrays A and B are aligned to 16-byes (notice the 0 in the 4 least significant bits of the address)
- Compiler automatically does padding

Aliasing

- Can the compiler vectorize this loop?

```
void func1(float *a,float *b, float *c){
   for (int i = 0; i < LEN; i++) {
        a[i] = b[i] + c[i];
}
```

Aliasing

- Can the compiler vectorize this loop?

```
float* a  = &b[1];
 …
void func1(float *a,float *b, float *c)
{
   for (int i = 0; i < LEN; i++)
       a[i] = b[i] + c[i];
}
```

b[1]= b[0] + c[0]
b[2] = b[1] + c[1]

Aliasing

- Can the compiler vectorize this loop?

```
float* a  = &b[1];
 …
void func1(float *a,float *b, float *c)
{
   for (int i = 0; i < LEN; i++)
       a[i] = b[i] + c[i];
}
```

a and b are aliasing
There is a self-true dependence
Vectorizing this loop would be illegal

Aliasing

- To vectorize, the compiler needs to guarantee that the pointers are not aliased.
- When the compiler does not know if two pointer are alias, it still vectorizes, but needs to add up-to $O(n^2)$ run-time checks, where *n* is the number of pointers

  When the number of pointers is large, the compiler may decide to not vectorize

```
void func1(float *a, float *b, float *c){
for (int i=0; i<LEN; i++)
  a[i] = b[i] + c[i];
}
```

Aliasing

- Two solutions can be used to avoid the run-time checks

1. static and global arrays
2. `__restrict__` attribute

Aliasing

1. Static and Global arrays

```
__attribute__ ((aligned(16))) float a[LEN];
__attribute__ ((aligned(16))) float b[LEN];
__attribute__ ((aligned(16))) float c[LEN];

void func1(){
for (int i=0; i<LEN; i++)
  a[i] = b[i] + c[i];
}


int main() {
…
   func1();
}
```

Aliasing

1. `__restrict__` keyword

```
void func1(float* __restrict__ a,float* __restrict__ b, float*
__restrict__ c) {
   __assume_aligned(a, 16);
   __assume_aligned(b, 16);
   __assume_aligned(c, 16);
   for int (i=0; i<LEN; i++)
         a[i] = b[i] + c[i];
}
int main() {
    float* a=(float*) memalign(16,LEN*sizeof(float));
    float* b=(float*) memalign(16,LEN*sizeof(float));
    float* c=(float*) memalign(16,LEN*sizeof(float));
    …
    func1(a,b,c);
}
```

Aliasing – Multidimensional arrays

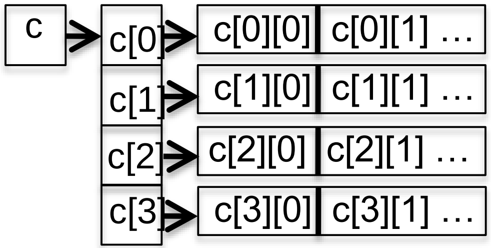
- Example with 2D arrays: pointer-to-pointer declaration.

```
void func1(float** __restrict__ a,float**
__restrict__ b, float** __restrict__ c) {
for (int i=0; i<LEN; i++)
    for (int j=1; j<LEN; j++)
       a[i][j] = b[i][j-1] * c[i][j];

}
```

Aliasing – Multidimensional arrays

- Example with 2D arrays: pointer-to-pointer declaration.

```
void func1(float** __restrict__ a,float** __restrict__ b,
float** __restrict__ c) {
for (int i=0; i<LEN; i++)
   for (int j=1; j<LEN; j++)
       a[i][j] = b[i][j-1] * c[i][j];
}
```



__restrict__ only qualifies the first dereferencing of c;

Nothing is said about the arrays that can be accessed through c[i]

Aliasing – Multidemensional Arrays

- Three solutions when __restrict__ does not enable vectorization

1. Static and global arrays

2. Linearize the arrays and use __restrict__ keyword

3. Use compiler directives

Aliasing – Multidimensional arrays

1. Static and Global declaration

```
__attribute__ ((aligned(16))) float a[N][N];
void t(){

    a[i][j]….
}

int main() {

   …
   t();
}
```

Aliasing – Multidimensional arrays

2. Linearize the arrays

```
void t(float* __restrict__ A){
    //Access to Element A[i][j] is now A[i*128+j]
    ….
}


int main() {
   float* A = (float*) memalign(16,128*128*sizeof(float));
   …
   t(A);
}
```

Aliasing – Multidimensional arrays

3. Use compiler directives:

```
#pragma ivdep (Intel compiler)
```

```
void func1(float **a, float **b, float **c) {
  for (int i=0; i<m; i++) {
     #pragma ivdep
     for (int j=0; j<LEN; j++)
       c[i][j] = b[i][j] * a[i][j];
}}
```

#pragma ivdep

- To ensure correctness, the correct treats an assumed dependence as a proven dependence, which can prevent vectorization
- Also, a compiler may decide a loop is not profitable to vectorize
- In either case, using #pragma ivdep overrides the compilers decision

```
void ignore_vec_dep(int *a, int k, int c, int m) {
  #pragma ivdep
  for (int i = 0; i < m; i++)
    a[i] = a[i + k] * c;
}
```

```
#pragma ivdep
for (j=0; j<n; j++) {
    a[b[j]] = a[b[j]] + 1;
}
```

We know there is no loop carried dependence, since k >= 0

#pragma ivdep

- To ensure correctness, the correct treats an assumed dependence as a proven dependence, which can prevent vectorization
- Also, a compiler may decide a loop is not profitable to vectorize
- In either case, using #pragma ivdep overrides the compilers decision

```
void ignore_vec_dep(int *a, int k, int c, int m) {
  #pragma ivdep
  for (int i = 0; i < m; i++)
    a[i] = a[i + k] * c;
}
```

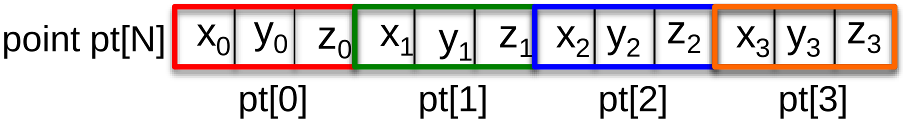
```
#pragma ivdep
for (j=0; j<n; j++) {
    a[b[j]] = a[b[j]] + 1;
}
```

We know there is no loop carried dependence, since we know the contents of b do not allow that

Non-unit Stride – Example I

- Array of a struct

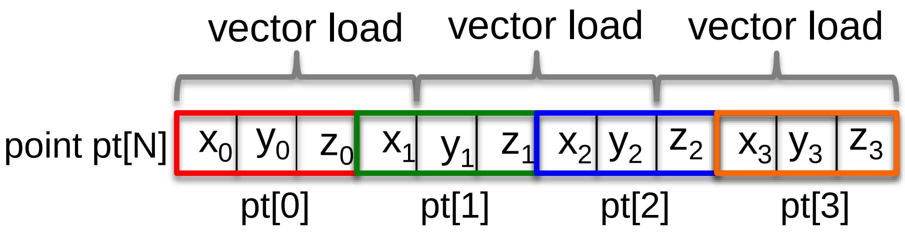
```
typedef struct{int x, y, z} point;
point pt[LEN];

for (int i=0; i<LEN; i++) {
  pt[i].y *= scale;
}
```



Non-unit Stride – Example I

- Array of a struct

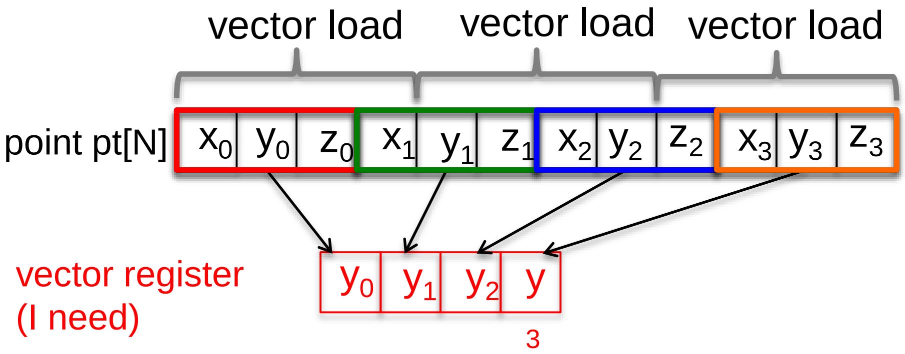
```
typedef struct{int x, y, z} point;
point pt[LEN];

for (int i=0; i<LEN; i++) {
  pt[i].y *= scale;
}
```



Non-unit Stride – Example I

- Array of a struct

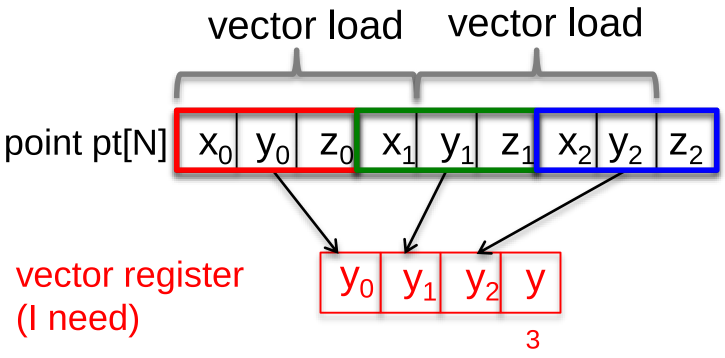
```
typedef struct{int x, y, z} point;
point pt[LEN];

for (int i=0; i<LEN; i++) {
  pt[i].y *= scale;
}
```



Non-unit Stride – Example I

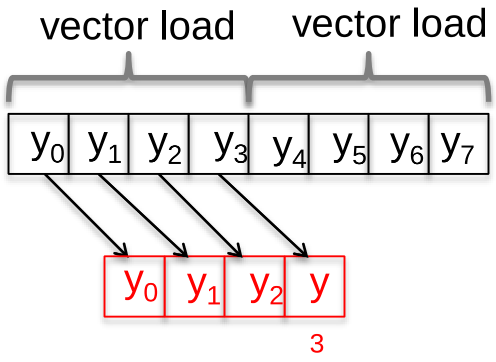
- Array of a struct

```
typedef struct{int x, y, z} point;
point pt[LEN];

for (int i=0; i<LEN; i++) {
  pt[i].y *= scale;
}
```



- Arrays

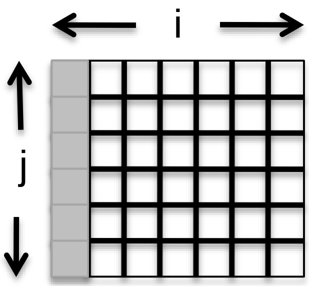
```
int ptx[LEN], int pty[LEN],
int ptz[LEN];

for (int i=0; i<LEN; i++) {
  pty[i] *= scale;
}
```



Non-unit Stride – Example II

```
for (int i=0;i<LEN;i++){
  sum = 0;
  for (int j=0;j<LEN;j++){
     sum += A[j][i];
  }
  B[i] = sum;
}
```

```
for (int i=0;i<size;i++){
  sum[i] = 0;
  for (int j=0;j<size;j++){
     sum[i] += A[j][i];
  }
  B[i] = sum[i];
}
```



Loop interchange…

Compiler Directives

- Compiler vectorizes many loops, but many more can be vectorized if the appropriate directives are used

Compiler Hints for Intel ICC	Semantics
#pragma ivdep	Ignore assume data dependences
#pragma vector always	override efficiency heuristics
#pragma novector	disable vectorization
__restrict__	assert exclusive access through pointer
__attribute__ ((aligned(int-val)))	request memory alignment
memalign(int-val,size);	malloc aligned memory
__assume_aligned(exp, int-val)	assert alignment property

Outline

- What is vectorization and why is it important
- The different ways we can vectorize our code
- The two main challenges in vectorization
  - Determining that vectorization is legal (the results will be the same)
    - Dependence analysis
    - Obstacles to vectorization and how to deal with them
  - Optimizing performance
    - Memory issues (alignment, layout)
    - Telling the compiler what you know (about your code & about your platform)
- Using compiler intrinsics
- Using OpenMP simd pragmas
- A case study

Access the SIMD through intrinsics

- Intrinsics are vendor/architecture specific
- We will focus on the Intel vector intrinsics
- Intrinsics are useful when
  - the compiler fails to vectorize
  - when the programmer thinks it is possible to generate better code than the one produced by the compiler

Where to get detailed info:

- The Intel® 64 and IA-32 Architectures Software Developer Manuals: https://software.intel.com/en-us/articles/intel-sdm
- The Intel® Intrinsics Guide: https://software.intel.com/sites/landingpage/IntrinsicsGuide/

The Intel SSE intrinsics Header file

- SSE can be accessed using intrinsics.

- You must use one of the following header files:

```
#include <xmmintrin.h> (for SSE)
#include <emmintrin.h> (for SSE2)
#include <pmmintrin.h> (for SSE3)
#include <smmintrin.h> (for SSE4)
```

- These include the prototypes of the intrinsics.

Intel SSE intrinsics Data types

- We will use the following data types:

  `__m128` packed single precision (vector XMM register)

  `__m128d` packed double precision (vector XMM register)

  `__m128i` packed integer (vector XMM register)

- Example

```
#include <xmmintrin.h>
int main ( ) {
  ...
  __m128 A, B, C;  /* three packed s.p. variables */
  ...
}
```

Intel SSE intrinsic Instructions

- Intrinsics operate on these types and have the format:

```
_mm_instruction_suffix(…)
```

- Suffix can take many forms. Among them:

  `ss` scalar single precision

  `ps` packed (vector) singe precision

  `sd` scalar double precision

  `pd` packed double precision

  `si#` scalar integer (8, 16, 32, 64, 128 bits)

  `su#` scalar unsigned integer (8, 16, 32, 64, 128 bits)

Intel SSE intrinsics Instructions – Examples

- Load four 16-byte aligned single precision values in a vector:

```
float a[4]={1.0,2.0,3.0,4.0};//a must be 16-byte aligned
__m128 x = _mm_load_ps(a);
```

- Add two vectors containing four single precision values:

```
__m128 a, b;
__m128 c = _mm_add_ps(a, b);
```

Intrinsics (SSE)

```
#define n 1024
__attribute__ ((aligned(16)))
float a[n], b[n], c[n];


int main() {
for (i = 0; i < n; i++) {
  c[i]=a[i]*b[i];
 }
}
```

```
#include <xmmintrin.h>
#define n 1024
__attribute__((aligned(16))) float a[n], b[n], c[n];

int main() {
__m128 rA, rB, rC;
for (i = 0; i < n; i+=4) {
  rA = _mm_load_ps(&a[i]);
  rB = _mm_load_ps(&b[i]);
  rC= _mm_mul_ps(rA,rB);
  _mm_store_ps(&c[i], rC);
}}
```

Intrinsics (SSE)

```c
#define n 1024
__attribute__ ((aligned(16)))
float a[n], b[n], c[n];


int main() {
for (i = 0; i < n; i++) {
  c[i]=a[i]*b[i];
 }
}
```

Header file

```c
#include <xmmintrin.h>
#define n 1024
__attribute__((aligned(16))) float a[n], b[n], c[n];

int main() {
__m128 rA, rB, rC;
for (i = 0; i < n; i+=4) {
  rA = _mm_load_ps(&a[i]);
  rB = _mm_load_ps(&b[i]);
  rC= _mm_mul_ps(rA,rB);
  _mm_store_ps(&c[i], rC);
}}
```

Intrinsics (SSE)

```c
#define n 1024
__attribute__ ((aligned(16)))
float a[n], b[n], c[n];


int main() {
for (i = 0; i < n; i++) {
  c[i]=a[i]*b[i];
 }
}
```

Declare 3 vector registers

```c
#include <xmmintrin.h>
#define n 1024
__attribute__((aligned(16))) float a[n], b[n], c[n];

int main() {
__m128 rA, rB, rC;
for (i = 0; i < n; i+=4) {
  rA = _mm_load_ps(&a[i]);
  rB = _mm_load_ps(&b[i]);
  rC= _mm_mul_ps(rA,rB);
  _mm_store_ps(&c[i], rC);
}}
```

Intrinsics (SSE)

```
#define n 1024
__attribute__ ((aligned(16)))
float a[n], b[n], c[n];


int main() {
for (i = 0; i < n; i++) {
  c[i]=a[i]*b[i];
 }
}
```

Execute vector statements

```
#include <xmmintrin.h>
#define n 1024
__attribute__((aligned(16))) float a[n], b[n], c[n];

int main() {
__m128 rA, rB, rC;
for (i = 0; i < n; i+=4) {
  rA = _mm_load_ps(&a[i]);
  rB = _mm_load_ps(&b[i]);
  rC= _mm_mul_ps(rA,rB);
  _mm_store_ps(&c[i], rC);
}}
```

Outline

- What is vectorization and why is it important
- The different ways we can vectorize our code
- The two main challenges in vectorization
  - Determining that vectorization is legal (the results will be the same)
    - Dependence analysis
    - Obstacles to vectorization and how to deal with them
  - Optimizing performance
    - Memory issues (alignment, layout)
    - Telling the compiler what you know (about your code & about your platform)
- Using compiler intrinsics
- Using OpenMP simd pragmas
- A case study

Ways to Write Vectorizable Code

Auto-Vectorization

```
for(i = 0; i < num_elem; i++){
  A[i] = B[i] + C[i];
}
```

Semi-Auto-Vectorization*

```
#pragma ivdep
for(i = 0; i < num_elem; i++){
  A[i] = B[i] + C[i];
}
```

Explicit vector programming using OpenMP

SIMD Pragma/Directive

```
#pragma omp simd
for(i = 0 i < num_elem; i++) {
  A[i] = B[i] + C[i];
}
```

Clauses to help recognize and vectorize idioms... examples:

Compress/Expand
Reduction
Search
Histogram ...

SIMD Function

```
#pragma omp declare simd
float work(float b, float c)
{
  return b + c;
}
…
#pragma omp simd aligned(A,B,C)
for(i = 0; i < num_elem; i++) {
  A[i] = work(B[i],C[i]);
}
```

How to write code to use the SIMD units?

Hardest to use / Most Control

1. Inline Assembly Language support
   – Most control but much harder to learn, code, debug, maintain…
2. SIMD intrinsics
   – Access to low level details similar to assembler but same issues
3. Compiler based Vectorization
   The fastest & easiest way; recommended for most cases
   – **Auto-Vectorization**
     – No code-changes; compiler vectorizes automatically for specified processor(s)
   – **Semi-Auto-Vectorization***
     – Use simple pragmas to guide compiler for missed auto-vectorization opportunities
     – Still hints to compiler, NOT mandatory!
   – **Explicit Vector Programming**
     – OpenMP SIMD-pragma, SIMD functions w/ powerful clauses… express code behavior better
     – Go after the performance opportunities that're missed by auto and semi-auto vectorization

Easiest to use / Least Control

Or, use a library that exploits the SIMD capabilities for you
e.g. the Intel® Math Kernel Library (Intel® MKL)

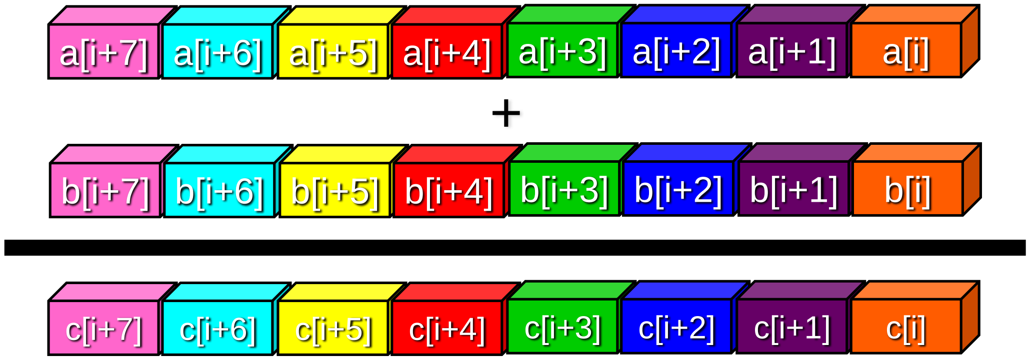
Semi-Auto-Vectorization* Example

Guiding compiler to help vectorize w/o multiversioning

```
void work( float* a, float *b, float *c, int num_elem) {
   #pragma ivdep
    for (int i=0; i<num_elem; i++)
       c[i] = a[i] + b[i];
}
```

$ icpc -c -xAVX -qopt-report:1 -qopt-report-phase:vec -qopt-report-file:stdout work.cpp

remark #15300: LOOP WAS VECTORIZED



Semi-Auto-Vectorization* – Black Scholes

Using hint *#pragma ivdep* to help auto-vectorize

// This sample is derived from code published by Bernt Arne Odegaard http://finance.bi.no/~bernt/gcc_prog/recipes/recipes/

```
static double N(const double& z) {
  return (1.0/sqrt(2.0*PI))*exp(-0.5*z*z);
}
double option_price_call_black_scholes(
         double S, double K, double r, double sigma, double time) {
   double time_sqrt = sqrt(time);
   double d1 = (log(S/K)+r*time)/(sigma*time_sqrt)+0.5*sigma*time_sqrt;
   double d2 = d1-(sigma*time_sqrt);
   return S*N(d1) - K*exp(-r*time)*N(d2);
}
void test_option_price_call_black_scholes(
         double S[], double K[], double r, double sigma, double time[],
         double call[], int num_options) {
#pragma ivdep
  for (int i=0; i < num_options; i++) {
    call[i] = option_price_call_black_scholes(S[i],K[i],r,sigma,time[i]);
  }
}
```

```
$ icpc -c -xAVX -qopt-report:5 BlackScholes.cpp
 remark #15300: LOOP WAS VECTORIZED
```

BUT… what if invoked functions in loop are in different files and not inlined?

How to write code to use the SIMD units?

Hardest to use / Most Control

Easiest to use / Least Control

1. Inline Assembly Language support
   – Most control but much harder to learn, code, debug, maintain…
2. SIMD intrinsics
   – Access to low level details similar to assembler but same issues
3. Compiler based Vectorization
   The fastest & easiest way; recommended for most cases
   – **Auto-Vectorization**
     – No code-changes; compiler vectorizes automatically for specified processor(s)
   – **Semi-Auto-Vectorization***
     – Use simple pragmas to guide compiler for missed auto-vectorization opportunities
     – Still hints to compiler, NOT mandatory!
   – **Explicit Vector Programming**
     – OpenMP SIMD-pragma, SIMD functions w/ powerful clauses… express code behavior better
     – Go after the performance opportunities that're missed by auto and semi-auto vectorization

Or, use a library that exploits the SIMD capabilities for you
e.g. the Intel® Math Kernel Library (Intel® MKL)

OPENMP* SIMD PROGRAMMING

Explicit Vector Programming

The OpenMP* API (www.openmp.org)

Has been an industry standard API for parallel programming since 1997

Defines pragmas for shared-memory parallel programming, including parallel regions, parallel loops, tasks, etc… (this will be covered in the threading part of the course)

Defines pragmas for offload to accelerators

And defines pragmas for vectorization

The OpenMP* API (www.openmp.org)

Pragmas for vectorization

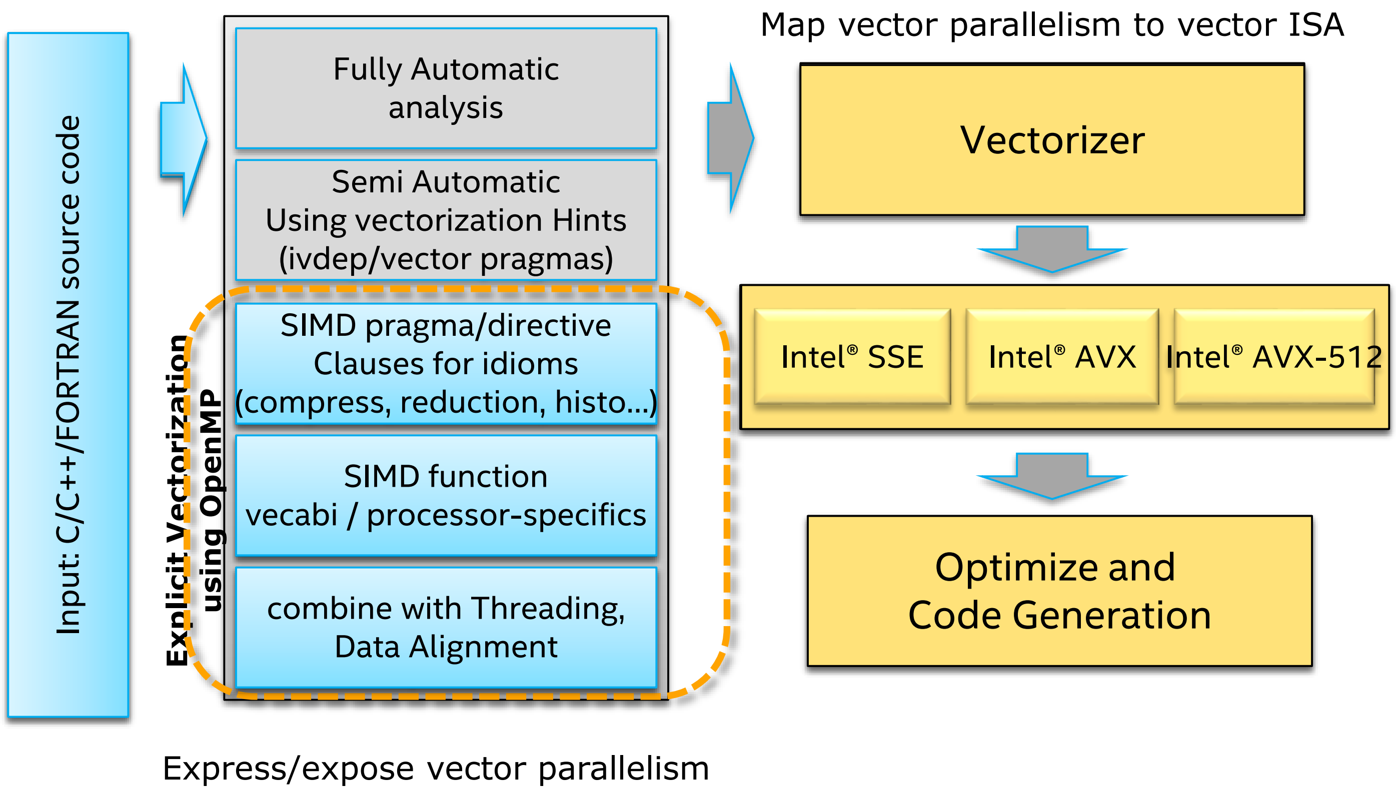
Pragmas are commands to the compiler, not hints

- E.g. #pragma omp simd
- Compiler does no dependency and cost-benefit analysis !!
- **Programmer is responsible for correctness**
  - Available in OpenMP since version 4.0 (2013) $\Rightarrow$ portable
- -qopenmp or -qopenmp-simd to enable

We will discuss some clauses, but everything is described in the OpenMP standard

Explicit Vector Programming

using OpenMP SIMD for C/C++ & Fortran



OpenMP* SIMD pragma

Use #pragma omp simd with -qopenmp-simd

```
void addit(double* a, double* b, int m, int n, int x)
{
  for (int i = m; i < m+n; i++)   {
        a[i] = b[i] + a[i-x];
  }
}
```

loop was NOT vectorized:
existence of vector
dependence.

```
void addit(double* a, double * b, int m, int n, int x)
{
#pragma omp simd // I know x<0
  for (int i = m; i < m+n; i++)   {
        a[i] = b[i] + a[i-x];
  }
}
```

SIMD LOOP WAS VECTORIZED.

Use when you **KNOW** that a given loop is safe to vectorize

The Intel® Compiler will vectorize if at all possible

- (ignoring dependency or efficiency concerns)
- Minimizes source code changes needed to enforce vectorization

Clauses for OMP SIMD directives

The programmer (i.e. you!) is responsible for correctness

- Just like for race conditions in loops with OpenMP* threading that will discuss later

Available clauses:

- private (variables that can be privatized, e.g. scalar expansion)
- lastprivate (private but last value is needed)
- reduction (ok to use associativity of operation)
- collapse (combine nested loops)
- linear (used to describe induction variables)
- simdlen (preferred number of iterations to execute concurrently)
- safelen (max iterations that can be executed concurrently)
- aligned (tells compiler about data alignment)

See www.openmp.org for details

Why use OpenMP* simd instead of intrinsics?

- OpenMP is portable
- Intrinsics are compiler / architecture specific
- With OpenMP, you do not select an ISA (i.e. SSE, AVX, etc..)
- With OpenMP, you describe the properties of the loop and instruct the compiler to vectorize it, but in a portable fashion
- You therefore do not need to modify your code every time you move to a different machine / compiler

Explicit Vector Programming with OpenMP #pragma omp simd

Programmer asserts:

**p* is loop invariant

A[] not aliased with *B[]*, *C[]*, & *sum*

sum not aliased with *B[]* and *C[]*

+ operator is associative

(compiler can reorder for better vectorization)

Vectorized code generated even if efficiency heuristic does not indicate a gain

```
float add( float* A, float* B, float* C, int* p) {
  float sum = 0.0f;
#pragma omp simd reduction(+:sum)
  for(int i = 0; i < *p; i++) {
    A[i] = B[i] * C[i];
    sum = sum + A[i];
  }
  return sum;
}
```

```
icpc -c -xAVX -qopenmp -qopt-report:5 add-simd.cpp
remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
```

Explicit Vector Programming lets you express what you mean!

#pragma omp simd using different clauses

NO #pragma omp simd → depending on auto-vectorization!

<Peeled loop for vectorization, Multiversioned v1>
<Multiversioned v1>
  remark #15300: LOOP WAS VECTORIZED
 remark #15478: estimated potential speedup: **3.760**
<Remainder loop for vectorization, Multiversioned v1>
<Multiversioned v2>
  remark #15304: loop was not vectorized: non-vectorizable loop instance from multiversioning
<Remainder, Multiversioned v2>

#pragma omp simd

<Peeled loop for vectorization>
 remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
 remark #15478: estimated potential speedup: **3.760**
<Remainder loop for vectorization>

129

#pragma omp simd using different clauses

#pragma omp simd reduction(+:sum)

<Peeled loop for vectorization>
remark #15388: vectorization support: reference A has aligned access
remark #15389: vectorization support: reference B has unaligned access
remark #15389: vectorization support: reference C has unaligned access
remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
remark #15478: estimated potential speedup: **4.310**
remark #15301: REMAINDER LOOP WAS VECTORIZED

#pragma omp simd reduction(+:sum) aligned(A,B,C)

remark #15388: vectorization support: reference A has aligned access
remark #15388: vectorization support: reference B has aligned access
remark #15388: vectorization support: reference C has aligned access
remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
remark #15478: estimated potential speedup: **7.560**
remark #15301: REMAINDER LOOP WAS VECTORIZED

OPENMP* SIMD FUNCTIONS

A way to vectorize loops containing calls to functions that can't be inlined

Loops Containing Function Calls

Function calls can have side effects that introduce a loop-carried dependency, preventing vectorization

Possible remedies:

- Inlining
  - best for small functions
  - Must be in same source file, or else use -ipo
- OMP SIMD pragma or directive to vectorize rest of loop, while preserving scalar calls to function (last resort)
- SIMD-enabled functions
  - Good for large, complex functions and in contexts where inlining is difficult
  - Call from regular “for”

Clauses for OMP declare simd

Asks compiler to create a vectorized version of a function

- i.e. parameters become vector registers

Again, the programmer (i.e. you!) is responsible for correctness

Available clauses:

- Same as #pragma omp simd plus…
- notinbranch, inbranch (generate or do not generate masking code)
- uniform (constants, i.e. non vector arguments)

See www.openmp.org for details

SIMD-enabled Function

Compiler generates SIMD-enabled (vector) version of a scalar function that can be called from a vectorized loop:

```
#pragma omp declare simd uniform(y,z,xp,yp,zp)
float func(float x, float y, float z, float xp, float yp, float zp)
{
float denom = (x-xp)*(x-xp) + (y-yp)*(y-yp) + (z-zp)*(z-zp);
  denom = 1./sqrtf(denom);
  return denom;
}
...
#pragma omp simd  private(x)  reduction(+:sumx)
for (i=1; i<nx; i++) {
    x = x0 + (float) i * h;
    sumx = sumx + func(x, y, z, xp, yp, zp);
  }
```

y, z, xp, yp and zp are constant, x can be a vector

FUNCTION WAS VECTORIZED with ...

These clauses are required for correctness, just like for OpenMP*

SIMD LOOP WAS VECTORIZED.

#pragma omp simd   may not be needed in simpler cases

SPECIAL IDIOMS

Compiler must recognize to handle apparent dependencies

Special Idioms

Dependency on an earlier iteration usually makes vectorization unsafe

- Some special patterns can still be handled by the compiler
  - Provided the compiler recognizes them (auto-vectorization)
    - Often works only for simple, 'clean' examples
  - Or the programmer tells the compiler (explicit vector programming)
    - May work for more complex cases
  - Examples: reduction, compress/expand, search, histogram/scatter, minloc
- Sometimes, the main speed-up comes from vectorizing the rest of a large loop, more than from vectorization of the idiom itself

Reduction – simple example

```
double reduce(double a[], int na) {
/*   sum all positive elements of a  */
     double sum = 0.;
     for (int ia=0; ia <na; ia++)  {
        if (a[ia] > 0.)  sum += a[ia];   // sum causes cross-iteration dependency
     }
     return sum;
}
```

Auto-vectorizes with any instruction set:

```
icc -std=c99 -O2 -qopt-report-phase=loop,vec -qopt-report-file=stderr reduce.c;
...
    LOOP BEGIN at reduce.c(17,6))
      remark #15300: LOOP WAS VECTORIZED
```

Reduction – when auto-vectorization doesn't work

icc -std=c99 -O2 **-fp-model precise** -qopt-report-phase=loop,vec -qopt-report-file=stderr reduce.c;

…

LOOP BEGIN at reduce.c(17,6))

remark #15331: loop was not vectorized: precise FP model implied by the command line or a directive prevents vectorization. Consider using fast FP model [reduce.c(18,26)

Vectorization would change order of operations, and hence the result

- Can use a SIMD pragma to override and vectorize:

```
#pragma omp simd reduction(+:sum)
    for (int ia=0; ia <na; ia++)  {
        sum += …
```

icc -std=c99 -O2 -fp-model precise –qopenmp-simd -qopt-report-file=stderr reduce.c;

LOOP BEGIN at reduce.c(18,6)

remark #15301: OpenMP SIMD LOOP WAS VECTORIZED

ANOTHER OPENMP EXAMPLE WITH OPTIMIZATION REPORTS

Example of Optimization Report - 1

$ icpc -c -qopt-report=4 -qopt-report-phase=vec -qopt-report-file=stderr foo.cpp

```
LOOP BEGIN at foo.cpp(4,3)
<Peeled loop for vectorization, Multiversioned v1>
LOOP END

LOOP BEGIN at foo.cpp(4,3)
<Multiversioned v1>
  remark #15388: vectorization support: reference theta[i] has aligned access   [ foo.cpp(5,21) ]
  remark #15388: vectorization support: reference sth[i] has aligned access   [ foo.cpp(5,8) ]
  remark #15305: vectorization support: vector length 4
  remark #15309: vectorization support: normalized vectorization overhead 0.094
  remark #15417: vectorization support: number of FP up converts: single precision to double precision 1   [ foo.cpp(5,17) ]
  remark #15418: vectorization support: number of FP down converts: double precision to single precision 1   [ foo.cpp(5,8) ]
  remark #15300: LOOP WAS VECTORIZED
  remark #15442: entire loop may be executed in remainder
  remark #15448: unmasked aligned unit stride loads: 1
  remark #15449: unmasked aligned unit stride stores: 1
  remark #15475: --- begin vector cost summary ---
  remark #15476: scalar cost: 112
  remark #15477: vector cost: 40.000
  remark #15478: estimated potential speedup: 2.730
  remark #15482: vectorized math library calls: 1
  remark #15487: type converts: 2
  remark #15488: --- end vector cost summary ---
LOOP END

LOOP BEGIN at foo.cpp(4,3)
<Alternate Alignment Vectorized Loop, Multiversioned v1>
LOOP END

LOOP BEGIN at foo.cpp(4,3)
<Remainder loop for vectorization, Multiversioned v1>
LOOP END

LOOP BEGIN at foo.cpp(4,3)
<Multiversioned v2>
   remark #15304: loop was not vectorized: non-vectorizable loop instance from multiversioning
LOOP END
```

```
#include <cmath>

void foo (float * theta, float * sth, int count)  {
  for (int i = 0; i < count; i++)
       sth[i] = sin(theta[i]+3.1415927);
}
```

- Note multiversioning

Example of New Optimization Report - 2

$ icpc -c -qopenmp -qopt-report=4 -qopt-report-phase=vec -qopt-report-file=stderr foo.cpp

```
LOOP BEGIN at foo.cpp(5,3)
<Peeled loop for vectorization>
LOOP END

LOOP BEGIN at foo.cpp(5,3)
   remark #15388: vectorization support: reference theta[i] has aligned access   [ foo.cpp(6,21) ]
   remark #15388: vectorization support: reference sth[i] has aligned access   [ foo.cpp(6,8) ]
   remark #15305: vectorization support: vector length 4
   remark #15309: vectorization support: normalized vectorization overhead 0.094
   remark #15417: vectorization support: number of FP up converts: single precision to double precision 1   [ foo.cpp(6,17) ]
   remark #15418: vectorization support: number of FP down converts: double precision to single precision 1   [ foo.cpp(6,8) ]
   remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
   remark #15442: entire loop may be executed in remainder
   remark #15448: unmasked aligned unit stride loads: 1
   remark #15449: unmasked aligned unit stride stores: 1
   remark #15475: --- begin vector cost summary ---
   remark #15476: scalar cost: 112
   remark #15477: vector cost: 40.000
   remark #15478: estimated potential speedup: 2.730
   remark #15482: vectorized math library calls: 1
   remark #15487: type converts: 2
   remark #15488: --- end vector cost summary ---
LOOP END

LOOP BEGIN at foo.cpp(5,3)
<Alternate Alignment Vectorized Loop>
LOOP END

LOOP BEGIN at foo.cpp(5,3)
<Remainder loop for vectorization>
LOOP END
```

```cpp
#include <cmath>

void foo (float * theta, float * sth, int count)  {
#pragma omp simd
  for (int i = 0; i < count; i++)
      sth[i] = sin(theta[i]+3.1415927);
}
```

- OMP SIMD take care of multiversioning
- Next focus on FP converts

Example of New Optimization Report - 3

$ icpc -c -qopenmp -qopt-report=4 -qopt-report-phase=vec -qopt-report-file=stderr foo.cpp

```
LOOP BEGIN at foo.cpp(5,3)
<Peeled loop for vectorization>
LOOP END

LOOP BEGIN at foo.cpp(5,3)
  remark #15388: vectorization support: reference theta[i] has aligned access   [ foo.cpp(6,21) ]
  remark #15388: vectorization support: reference sth[i] has aligned access   [ foo.cpp(6,8) ]
  remark #15305: vectorization support: vector length 4
  remark #15309: vectorization support: normalized vectorization overhead 0.190
  remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
  remark #15442: entire loop may be executed in remainder
  remark #15448: unmasked aligned unit stride loads: 1
  remark #15449: unmasked aligned unit stride stores: 1
  remark #15475: --- begin vector cost summary ---
  remark #15476: scalar cost: 109
  remark #15477: vector cost: 19.750
  remark #15478: estimated potential speedup: 5.190
  remark #15482: vectorized math library calls: 1
  remark #15488: --- end vector cost summary ---
LOOP END

LOOP BEGIN at foo.cpp(5,3)
<Alternate Alignment Vectorized Loop>
LOOP END

LOOP BEGIN at foo.cpp(5,3)
<Remainder loop for vectorization>
LOOP END
```

```
#include <cmath>

void foo (float * theta, float * sth, int count)  {
#pragma omp simd
 for (int i = 0; i < count; i++)
     sth[i] = sin(theta[i]+3.1415927f);
}
```

- FP Pi takes care of FP converts
- Next focus on vector length 4 (using SSE)

Example of New Optimization Report – 4

$ icpc -c -xCORE-AVX2 -qopenmp -qopt-report=4 -qopt-report-phase=vec -qopt-report-file=stderr foo.cpp

```
LOOP BEGIN at foo.cpp(5,3)
<Peeled loop for vectorization>
LOOP END

LOOP BEGIN at foo.cpp(5,3)
   remark #15389: vectorization support: reference theta[i] has unaligned access   [ foo.cpp(6,21) ]
   remark #15389: vectorization support: reference sth[i] has unaligned access   [ foo.cpp(6,8) ]
   remark #15381: vectorization support: unaligned access used inside loop body
   remark #15305: vectorization support: vector length 8
   remark #15309: vectorization support: normalized vectorization overhead 0.175
   remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
   remark #15442: entire loop may be executed in remainder
   remark #15450: unmasked unaligned unit stride loads: 1
   remark #15451: unmasked unaligned unit stride stores: 1
   remark #15475: --- begin vector cost summary ---
   remark #15476: scalar cost: 109
   remark #15477: vector cost: 10.000
   remark #15478: estimated potential speedup: 7.780
   remark #15482: vectorized math library calls: 1
   remark #15488: --- end vector cost summary ---
LOOP END

LOOP BEGIN at foo.cpp(5,3)
<Remainder loop for vectorization>
LOOP END
```

```
#include <cmath>

void foo (float * theta, float * sth, int count)  {
#pragma omp simd
  for (int i = 0; i < count; i++)
      sth[i] = sin(theta[i]+3.1415927f);
}
```

- CORE-AVX2 target takes vector length to 8
- Next focus on data alignment

Example of New Optimization Report - 5

$ icpc -c -xCORE-AVX2 -qopenmp -qopt-report=4 -qopt-report-phase=vec -qopt-report-file=stderr foo.cpp

```
LOOP BEGIN at foo.cpp(5,3)
   remark #15388: vectorization support: reference theta[i] has aligned access   [ foo.cpp(6,21) ]
   remark #15388: vectorization support: reference sth[i] has aligned access   [ foo.cpp(6,8) ]
   remark #15305: vectorization support: vector length 8
   remark #15309: vectorization support: normalized vectorization overhead 0.013
   remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
   remark #15448: unmasked aligned unit stride loads: 1
   remark #15449: unmasked aligned unit stride stores: 1
   remark #15475: --- begin vector cost summary ---
   remark #15476: scalar cost: 109
   remark #15477: vector cost: 9.870
   remark #15478: estimated potential speedup: 9.730
   remark #15482: vectorized math library calls: 1
   remark #15488: --- end vector cost summary ---
LOOP END

LOOP BEGIN at foo.cpp(5,3)
<Remainder loop for vectorization>
LOOP END
```

```
#include <cmath>

void foo (float * theta, float * sth, int count)  {
#pragma omp simd aligned(theta,sth:64)
  for (int i = 0; i < count; i++)
      sth[i] = sin(theta[i]+3.1415927f);
}
```

- OMP aligned clause helps
- **Overall speedup 2.73x → 9.73x**

Basic Optimizations with Intel C/C++ compiler

-O0 no optimization; sets -g for debugging

-O1 scalar optimizations

- Excludes optimizations tending to increase code size

-O2 **default** for icc & ifort (except with -g)

- includes **vectorization**; some loop transformations such as unrolling; inlining within source file;
- Start with this (after initial debugging at -O0)

-O3 more aggressive loop optimizations

- Including cache blocking, loop fusion, loop interchange, …
- May not help all applications; need to test

High-Level Optimizations (HLO)

- Enabled with –O3 (/O3 on Windows)
  - With auto-vectorization does more aggressive data dependency analysis than at /O2
  - Exploits properties of source code (loops & arrays)
  - Best chance for performing loop transformations

Loop optimizations:

Automatic vectorization‡	(use of packed SIMD instructions)
Loop interchange ‡	(for more efficient memory access)
Loop unrolling‡	(more instruction level parallelism)
Prefetching	(for patterns not recognized by h/w prefetcher)
Cache blocking	(for more reuse of data in cache)
Loop versioning ‡	(for loop count; data alignment; runtime dependency tests)
Memcpy recognition ‡	(call Intel's fast memcpy, memset)
Loop splitting ‡	(facilitate vectorization)
Loop fusion	(more efficient vectorization)
Scalar replacement‡	(reduce array accesses by scalar temps)
Loop rerolling	(enable vectorization)
Loop peeling ‡	(allow for misalignment)
Loop reversal	(handle dependencies)
etc.	

‡ all or partly enabled at -O2

Legal Disclaimer & Optimization Notice

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more complete information visit www.intel.com/benchmarks.

INFORMATION IN THIS DOCUMENT IS PROVIDED "AS IS". NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

intel®

Software