Course Outline

1. Structure of a Compiler
2. Instruction Level Parallelism Optimizations (Josep Llosa)
   - Instruction Level Parallelism
   - Machine Independent Optimizations
   - Instruction Scheduling
   - Register Allocation
3. **Memory Hierarchy Optimizations (J.R. Herrero)**
   - **Basic Concepts** **Acknowledgement: Marta Jiménez**
   - **Basic Transformations**
   - **Loop Vectorization**
4. Thread Level Parallelism Optimizations (Marc González)
   - Thread level Parallelism
   - Analysis and detection of parallelism
   - Programming models
   - Parallel execution
   - Memory models

Loop Vectorization

- Microprocessor vector extensions (SIMD)
- Overcoming limitations to SIMD-Vectorization
  - Data Dependences
  - Data Alignment
  - Aliasing
  - Non-unit strides

Microprocessor vector extensions

- Loop vectorization transforms a program so that the same operation is performed at the same time on several vector elements

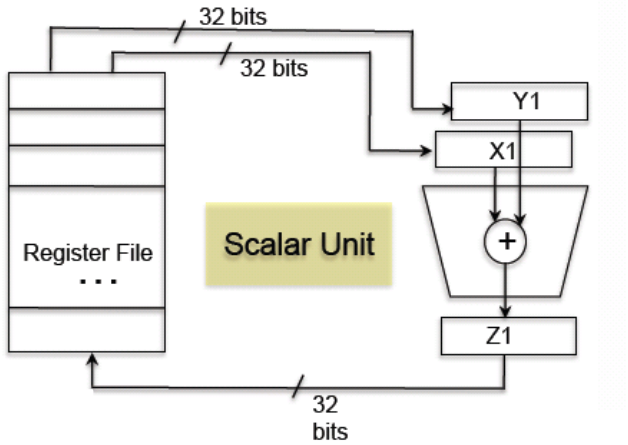
n times

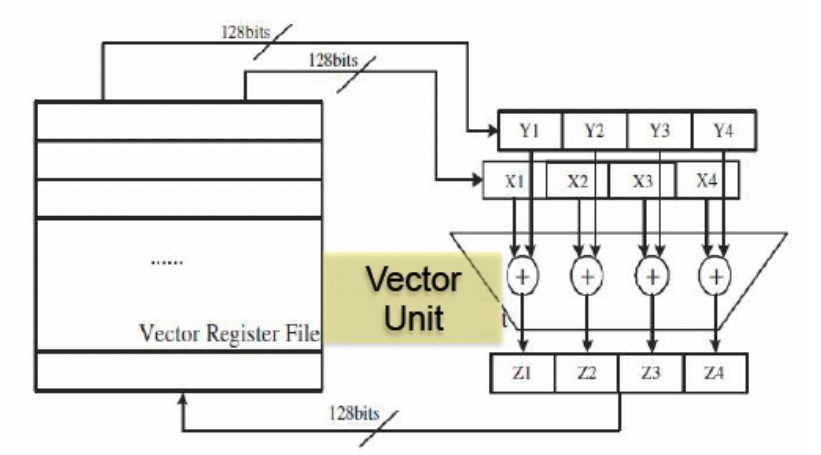
```
lw $t0, 0($a0)
lw  $t1, 0($a1)
add $t3, $t0, $t1
sw $t3, 0($a3)
```

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

n/4 times

```
lwv $vt0, 0($a0)
lwv $vt1, 0($a1)
addv $vt3, $vt0, $vt1
swv %vt3, $0($a3)
```





SIMD Vectorization

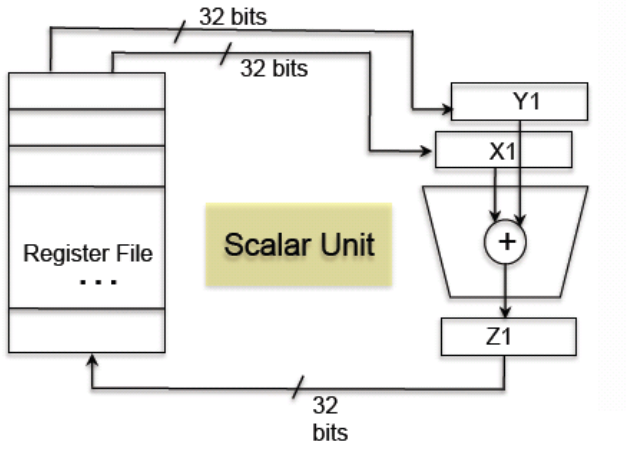
- The use of SIMD units can speed up the program
- Intel SSE and IBM Altivec have 128-bit vector registers and functional units
  - 4 32-bit single precision floating point numbers
  - 2 64-bit double precision floating point numbers
  - 4 32-bit integer numbers
  - 2 64-bit integer numbers
  - 8 16-bit integer or shorts
  - 16 8-bit bytes or char
- Assuming a single ALU, these SIMD units can execute 4 single precision or 2 double precision operations in the time it takes to do only one of these operations by a scalar unit.
- Newer processors: increasing vector length
  - Intel & AMD: AVX, AVX2 (256-bit vector registers)
  - Intel: AVX-512 (512-bit vector registers) [Beware of downclocking]
  - IBM Power 9 VSU (128-bits)
  - ARM: SVE -- vector length-agnostic programming model (128-bits -> 2048)
  - …

Executing our simple example

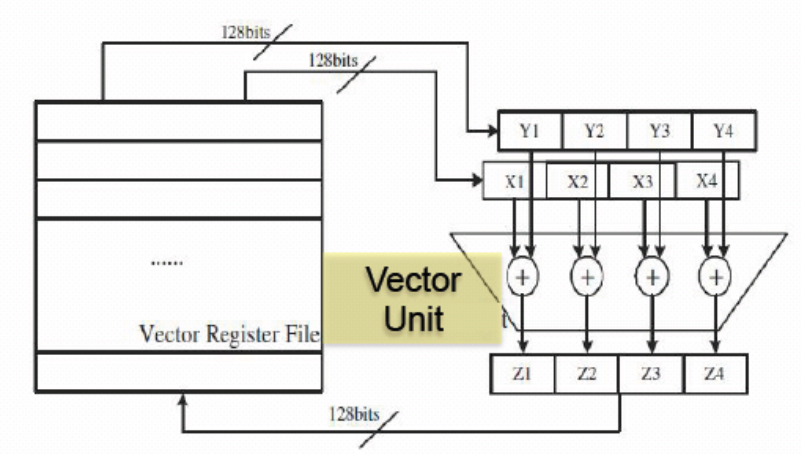
- Platform: Intel Nehalem (Intel Core i7, 2.67GHz)
  Intel ICC compiler, version 11.1

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

Exec. Time scalar code: 6.1



Exec. Time vector code: 3.2



Speedup: 1.8

How do we access the SIMD units?

- Three choices
  1. Assembly Language

```
..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
```

  2. Vector Intrinsics

```
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);
}}
```

  3. C code and a vectorizing compiler

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

Why use compiler vectorization?

- Easier
- Portable across vendors and machines
  - Although compiler directives differ across compilers
- Better performance of the compiler generated code
  - Compiler applies other transformations

Compilers make your codes (almost) machine independent

Compiler vectorization

- Compilers can vectorize for us, but they may fail:

1. Code cannot be vectorized due to data dependences: vectorization will produce incorrect results

2. Code can be vectorized, but the compiler fail to vectorize the code in its current form
   1. Programmer can use compiler directives to give the compiler the necessary information

   2. Programmer can transform the code

Example

```
void test (float* A, float* B, float* C,
float* D, float* E)
{
  for (i=0; i<LEN; i++)
    A[i] = B[i]+C[i]+D[i]+E[i];
}
```

```
void test (float* __restrict__ A,
float* __restrict__ B,
float* __restrict__ C,
float* __restrict__ D,
float* __restrict__ E)
{
  for (i=0; i<LEN; i++)
    A[i] = B[i]+C[i]+D[i]+E[i];
}
```

Intel Nehalem

Compiler report: Loop was not vectorized

Exec. Time scalar code: 5.6

Exec. Time vector code: --

Speedup: --

Intel Nehalem

Compiler report: Loop was vectorized

Exec. Time scalar code: 5.6

Exec. Time vector code: 2.2

Speedup: 2.5

The __restrict__ keyword is used in pointer declarations to say:

- for the lifetime of the pointer, only it (or a value directly derived from it) will be used to access the object to which it points

Loop Vectorization

- Microprocessor vector extensions (SIMD)
- Overcoming limitations to SIMD-Vectorization
  - Data Dependences
  - Data Alignment
  - Aliasing
  - Non-unit strides

Vectorization is not always legal

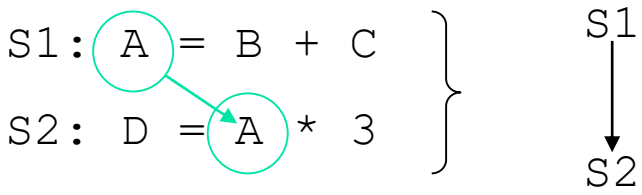
- Vectorization of some codes could produce incorrect results

- Compilers (and programmers) should study data dependences to determine if a program can be vectorized

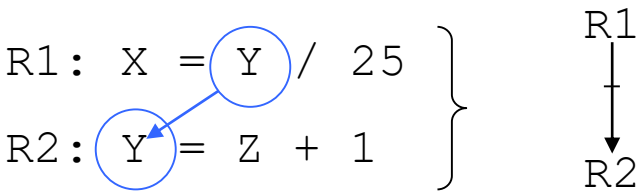
Definition of dependence

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

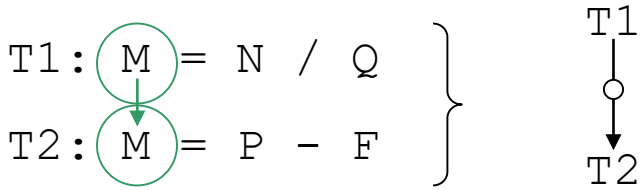
True dependence



Antidependence



Output Dependence

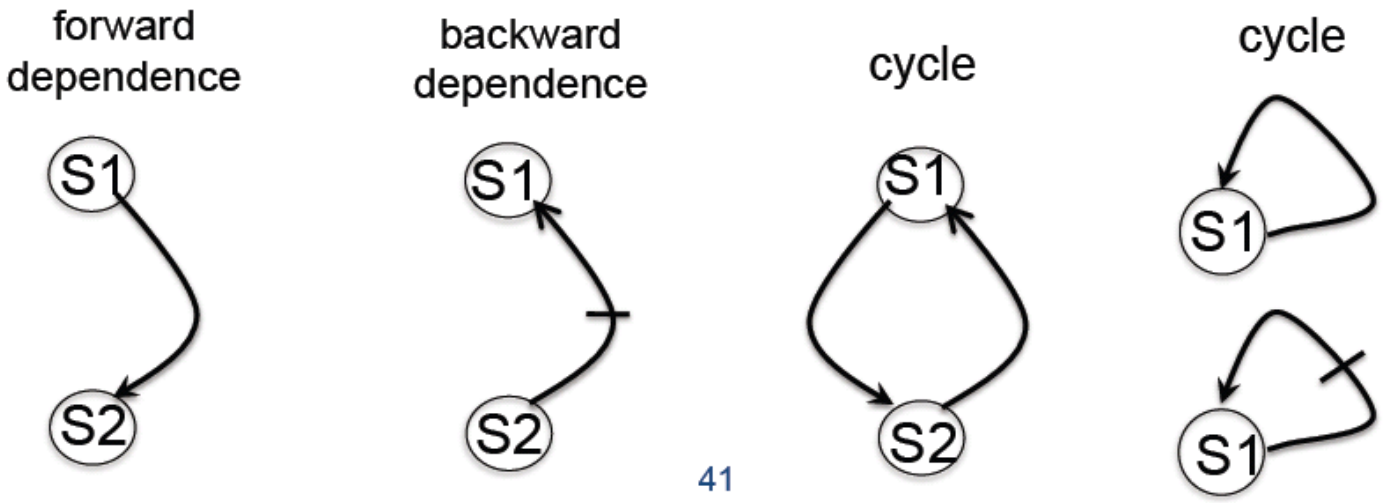


Data dependences

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

Loop vectorization

- Loop vectorization is not always a legal transformation
  - Compilers can vectorize when there are only forward dependences
  - Compilers cannot vectorize when there is a cycle in the data dependences graph (with the exception of a self-antidependence), unless a transformation is applied to remove the cycle
  - Codes with only backward dependences can be vectorized, but need to be transformed



41

Loop vectorization

- When vectorizing a loop with several statements the compiler need to strip-mine the loop and then apply loop distribution

```
for (i=0; i<LEN; i++) {
    a[i] = b[i]+(float)1.0;
    c[i] = b[i]+(float)2.0
}
```

```
for (ii=0; ii<LEN; ii+=strip_size)
  for (i=ii; i<ii+ strip_size; i++) {
    a[i] = b[i]+(float)1.0;
    c[i] = b[i]+(float)2.0
  }
```

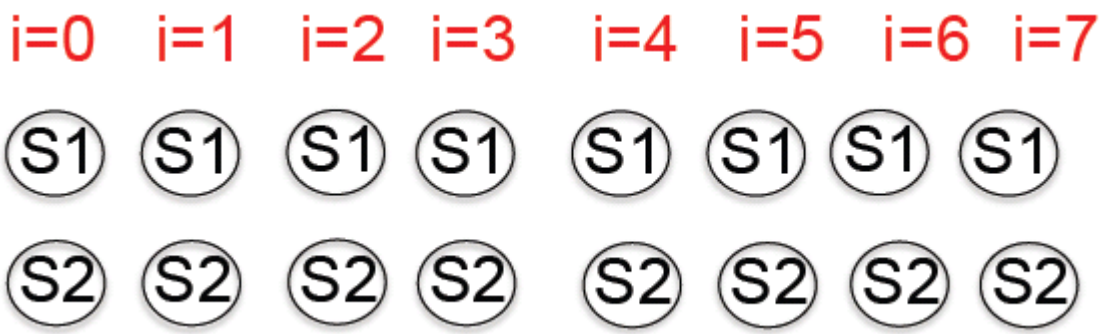
```
for (ii=0; ii<LEN; ii+=strip_size) {
  for (i=ii; i<ii+ strip_size; i++)
    a[i] = b[i]+(float)1.0;
  for (i=ii; i<ii+ strip_size; i++)
    c[i] = b[i]+(float)2.0
}
```

Loop vectorization

- When vectorizing a loop with several statements the compiler need to strip-mine the loop and then apply loop distribution

```
for (i=0; i<LEN; i++) {
  S1: a[i] = b[i]+(float)1.0;
  S2: c[i] = b[i]+(float)2.0
}
```

```
for (ii=0; ii<LEN; ii+=strip_size) {
  for (i=ii; i<ii+ strip_size; i++)
    a[i] = b[i]+(float)1.0;
  for (i=ii; i<ii+ strip_size; i++)
    c[i] = b[i]+(float)2.0
}
```

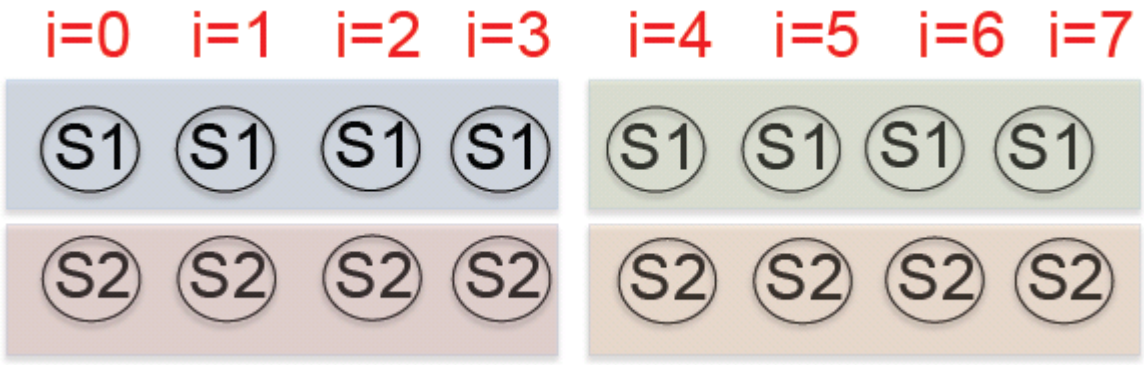


Loop vectorization

- When vectorizing a loop with several statements the compiler need to strip-mine the loop and then apply loop distribution

```
for (i=0; i<LEN; i++) {
  S1: a[i] = b[i]+(float)1.0;
  S2: c[i] = b[i]+(float)2.0
}
```

```
for (ii=0; ii<LEN; ii+=strip_size) {
  for (i=ii; i<ii+ strip_size; i++)
    a[i] = b[i]+(float)1.0;
  for (i=ii; i<ii+ strip_size; i++)
    c[i] = b[i]+(float)2.0
}
```



Examples

- Acyclic dependence graphs: forward dependences

```
for (i=0; i<LEN; i++) {
  S1: a[i] = b[i]+ c[i];
  S2: d[i] = a[i]+(float)1.0
}
```

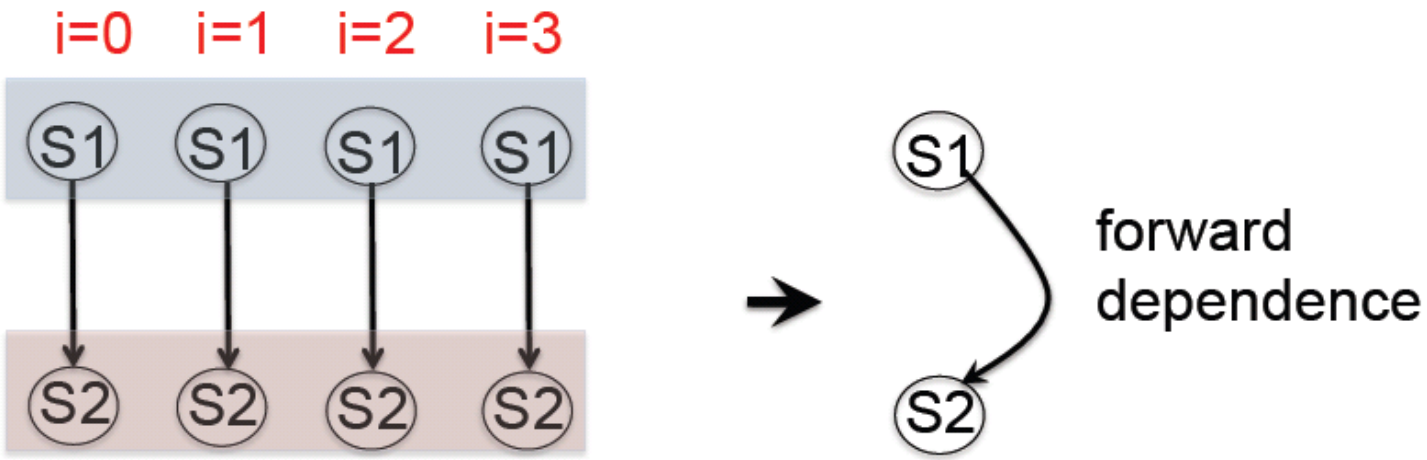
Intel Nehalem

Compiler report: Loop was vectorized

Exec. Time scalar code: 10.2

Exec. Time vector code: 6.3

Speedup: 1.6

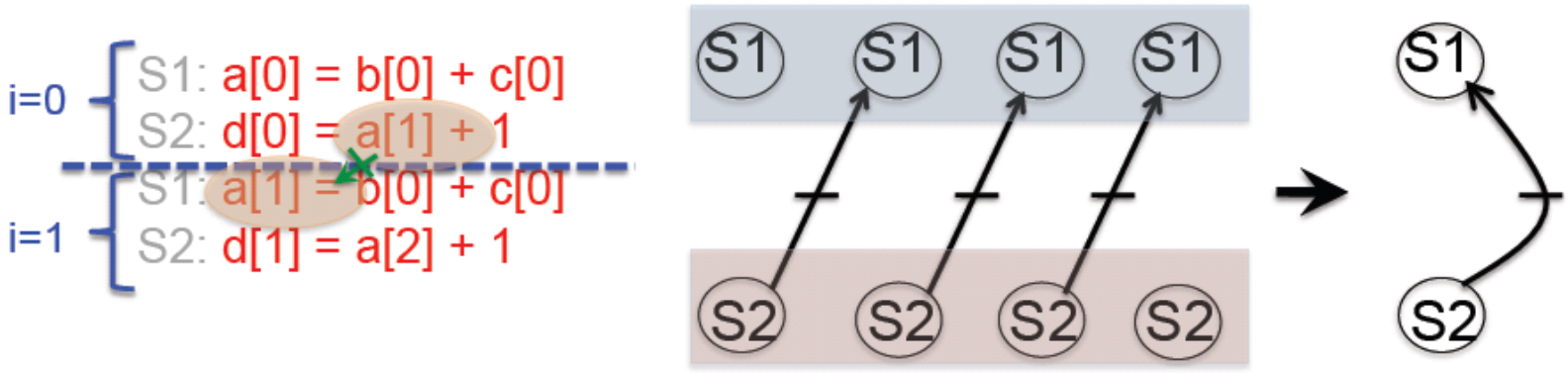


Examples

- Acyclic dependence graphs: backward dependences

```
for (i=0; i<LEN; i++) {
  S1: a[i] = b[i]+ c[i];
  S2: d[i] = a[i+1]+(float)1.0
}
```

backward dependence



- This loop cannot be vectorized as it is

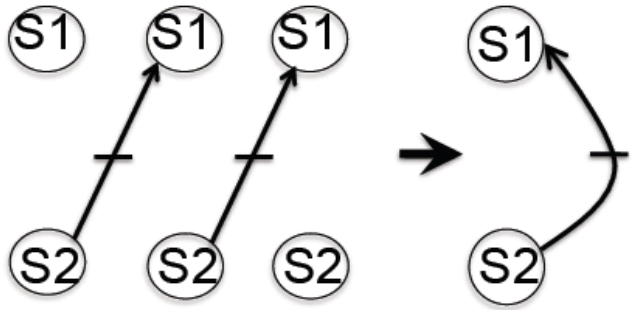
Examples

- Acyclic dependence graphs: backward dependences

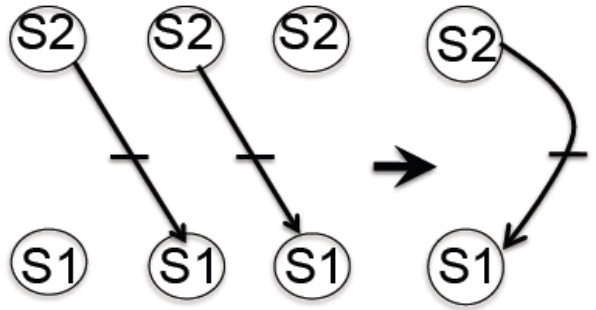
Reorder of statements

```
for (i=0; i<LEN; i++) {
  S1: a[i] = b[i]+ c[i];
  S2: d[i] = a[i+1]+(float)1.0
}
```

```
for (i=0; i<LEN; i++) {
  S2: d[i] = a[i+1]+(float)1.0
  S1: a[i] = b[i]+ c[i];
}
```



backward
depedence



forward
depedence

Examples

- Acyclic dependence graphs: backward dependences

```
for (i=0; i<LEN; i++) {
  S1: a[i] = b[i]+ c[i];
  S2: d[i] = a[i+1]+(float)1.0
}
```

Intel Nehalem

Compiler report: Loop was not vectorized. Existence of vector dependence

Exec. Time scalar code: 12.6

Exec. Time vector code: --

Speedup: --

```
for (i=0; i<LEN; i++) {
  S2: d[i] = a[i+1]+(float)1.0
  S1: a[i] = b[i]+ c[i];
}
```

Intel Nehalem

Compiler report: Loop was vectorized

Exec. Time scalar code: 10.7

Exec. Time vector code: 6.2

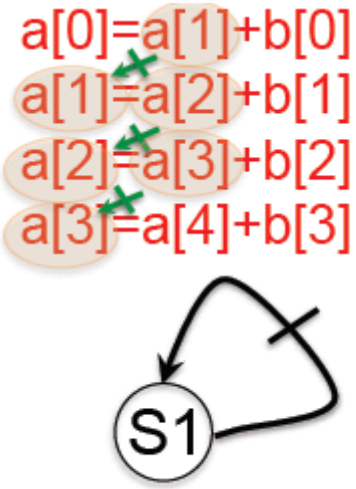
Speedup: 1.72

Speedup vs non-reordered code: 2.03

Examples

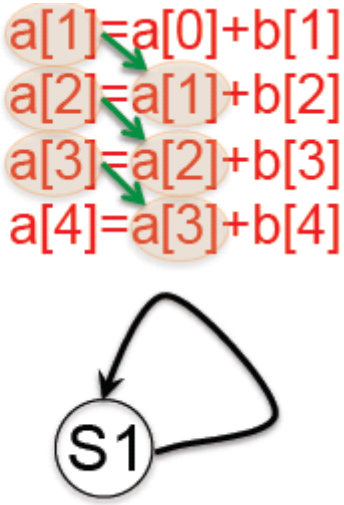
- Cycles in the dependence graphs

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



Self-antidependence
can be vectorized

```
for (i=1; i<LEN; i++) {
  S1: a[i] = a[i-1]+ b[i];
}
```

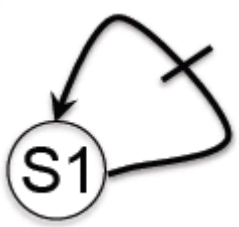


Self true-dependence
can not be vectorized
(as it is)

Examples

- Cycles in the dependence graphs

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



Intel Nehalem

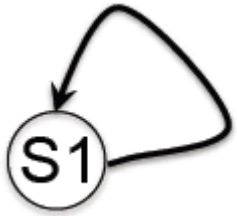
Compiler report: Loop was vectorized.

Exec. Time scalar code: 6.0

Exec. Time vector code: 2.7

Speedup: 2.2

```
for (i=1; i<LEN; i++) {
  S1: a[i] = a[i-1]+ b[i];
}
```



Intel Nehalem

Compiler report: Loop was not vectorized. Existence of vector dependence

Exec. Time scalar code: 7.2

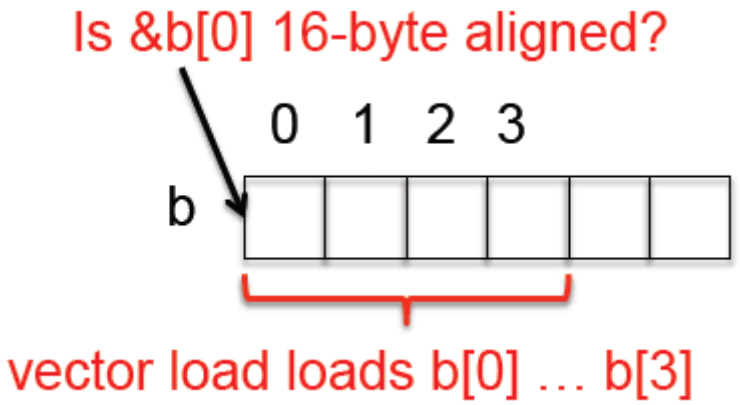
Exec. Time vector code: --

Speedup: --

Data Alignment

- Vector loads/stores load/store 128 consecutive bits to a vector register
- Data addresses need to be 16-byte (128 bits) aligned to be loaded/stored
  - Intel platforms support aligned and unaligned load/stores
  - IBM platforms do not support unaligned load/stores

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



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
- 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 multiple of 16

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

Note: `memalign` is considered obsolete --> `posix_memalign`

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

Aliasing

- Can the compiler vectorize this loop?

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

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

- a and b are aliasing
- There is a self-true dependence
- Vectorizing this loop would be illegal
- To vectorize, the compiler needs to guarantee that the pointers are not aliased

Aliasing

- Two solutions can be used to avoid run-time checks
  1. Static and global arrays

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

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

Aliasing

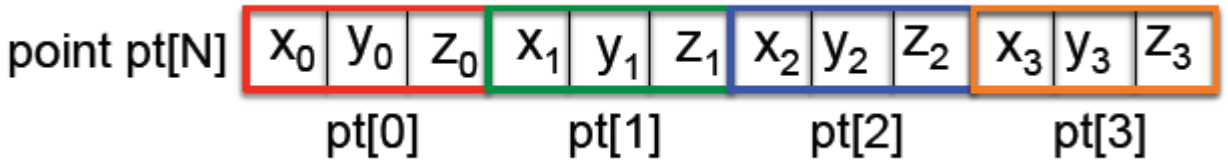
- Two solutions can be used to avoid run-time checks
  2. __restrict__ keyword

```
void func1 (float* __restrict__ a, float* __restrict__ b,
float* __restrict__ c) {
  __assume_aligned(a, 16);
  __assume_aligned(b, 16);
  __assume_aligned(c, 16);
  for (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);
}
```

Non-unit Stride - Example

- Array of a struct

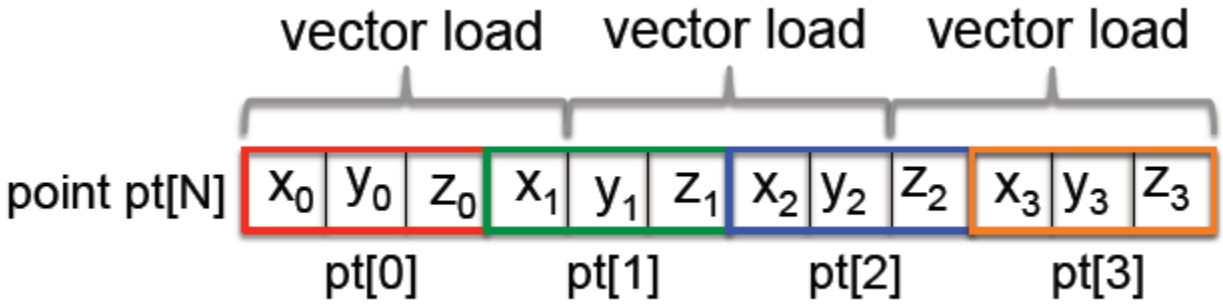
```
typedef struct {int x, y, z} point;
point pt[LEN];

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



Non-unit Stride - Example

- Array of a struct

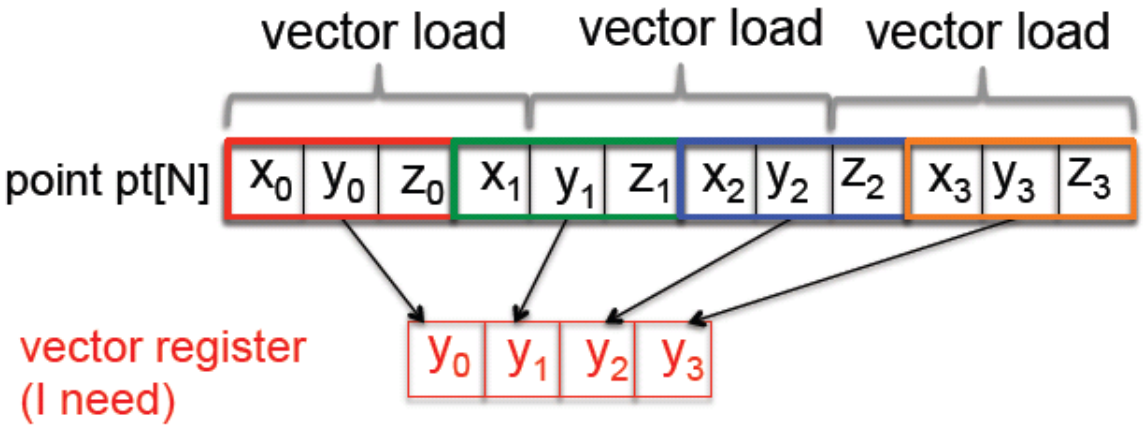
```
typedef struct {int x, y, z} point;
point pt[LEN];

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



Non-unit Stride - Example

- Array of a struct

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

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



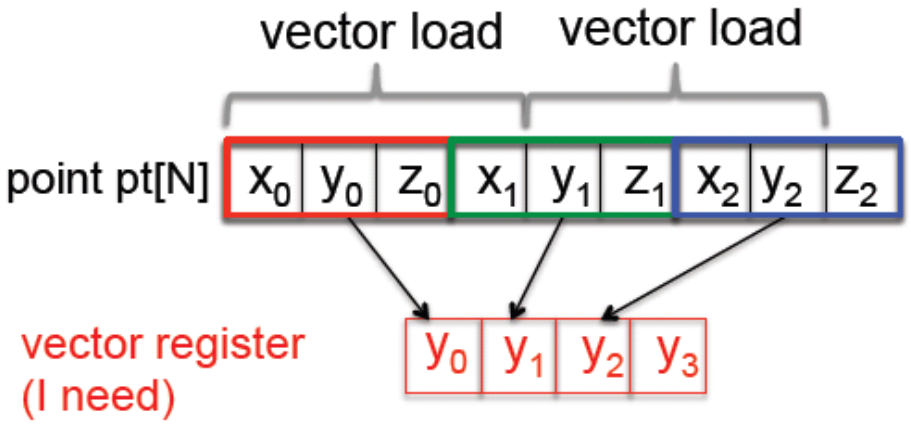
Non-unit Stride - Example

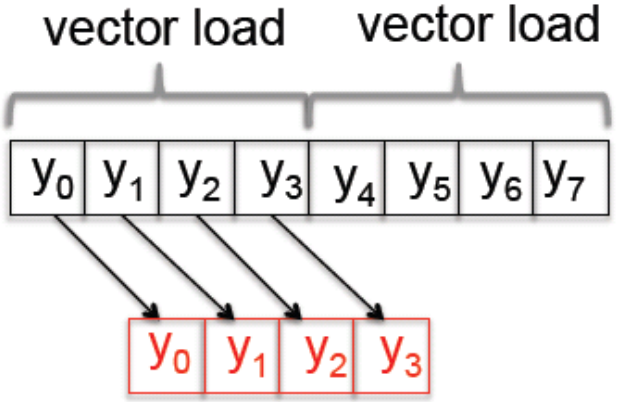
- Array of a struct

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

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

- Arrays

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

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





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

Summary

- Microprocessor vector extensions can contribute to improve program performance and the amount of this contribution is likely to increase in the future as vector lengths grow
- Compilers are only partially successful at vectorizing
- When the compiler fails, programmers can
  - Add compiler directives
  - Apply loop transformations
- If after transforming the code, the compiler still fails to vectorize (or the performance of the generated code is poor), the only option to program the vector extensions directly using assembly language (or intrinsics)