

Outline

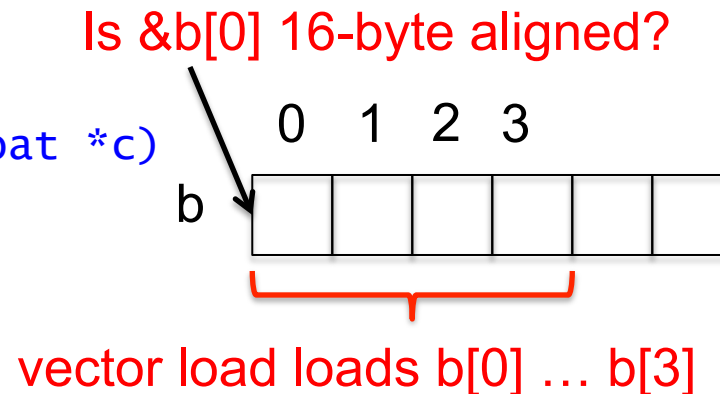
- Overcoming limitations to SIMD-Vectorization
 - Data Dependences
 - Data Alignment
 - Aliasing
 - Non-unit strides
 - Conditional Statements
- Vectorization with intrinsics



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 test1(float *a, float *b, float *c)
{
  for (int 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

```
__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];  
    }  
}
```



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 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);  
    }  
}
```

```
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);  
    }  
}
```

Nanosecond per iteration			
	Core 2 Duo	Intel i7	Power 7
Aligned	0.577	0.580	0.156
Aligned (unaligned ld)	0.689	0.581	0.241
Unaligned	2.176	0.629	0.243



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:

0x7ffe6765f00, 0x7ffe6765f04, 0x7ffe6766004, 0x7ffe6766008

- 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-bytes (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* 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];  
}
```

```
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);  
}
```

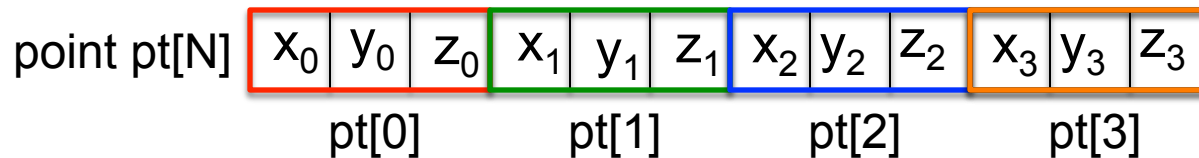


Non-unit Stride – Example I

- 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;  
}
```

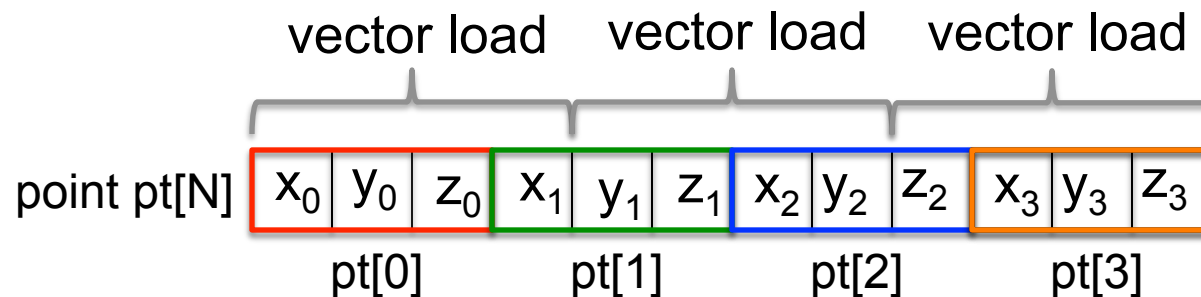


Non-unit Stride – Example I

- 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;  
}
```

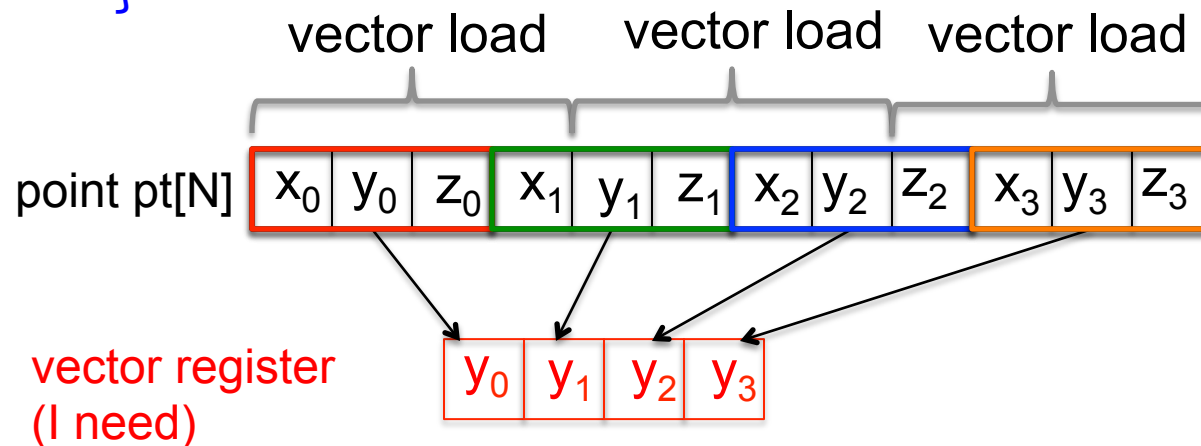


Non-unit Stride – Example I

- 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;  
}
```

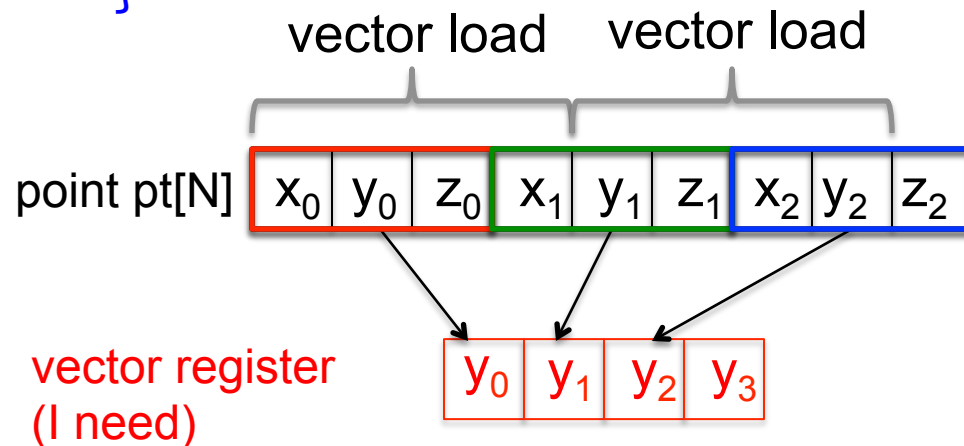


Non-unit Stride – Example I

- Array of a struct

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

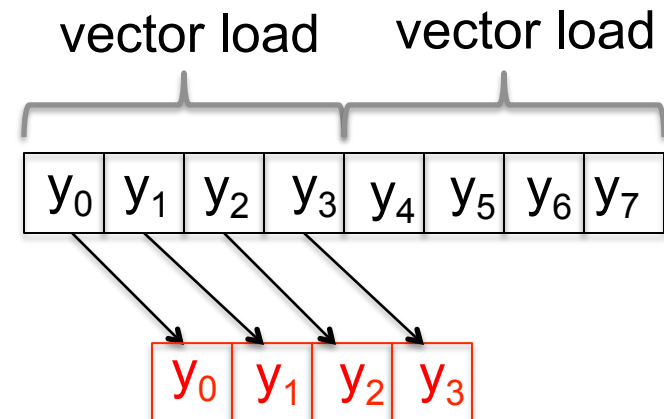
```
for (int i=0; i<LEN; i++) {  
    pt[i].y *= scale;  
}
```



- Arrays

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

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



Conditional Statements – I

- Loops with conditions need `#pragma vector always`
 - Since the compiler does not know if vectorization will be profitable
 - The condition may prevent from an exception

```
#pragma vector always
for (int i = 0; i < LEN; i++){
    if (c[i] < (float) 0.0)
        a[i] = a[i] * b[i] + d[i];
}
```



Conditional Statements – I

<pre>for (int i = 0; i < LEN; i++){ if (c[i] < (float) 0.0) a[i] = a[i] * b[i] + d[i]; }</pre>	<pre>#pragma vector always for (int i = 0; i < LEN; i++){ if (c[i] < (float) 0.0) a[i] = a[i] * b[i] + d[i]; }</pre>
--	--

Intel Nehalem

Compiler report: Loop was not vectorized. Condition may protect exception

Exec. Time scalar code: 10.4

Exec. Time vector code: --

Speedup: --

Intel Nehalem

Compiler report: Loop was vectorized.

Exec. Time scalar code: 10.4

Exec. Time vector code: 5.0

Speedup: 2.0



Conditional Statements

- Compiler removes *if conditions* when generating vector code

```
for (int i = 0; i < LEN; i++){  
    if (c[i] < (float) 0.0)  
        a[i] = a[i] * b[i] + d[i];  
}
```



Conditional Statements

```
for (int i=0;i<1024;i++){
    if (c[i] < (float) 0.0)
        a[i]=a[i]*b[i]+d[i];
}
```

rC	2	-1	1	-2
rCmp	False	True	False	True
rThen	0	3.2	0	3.2
rElse	1.	0	1.	0
rS	1.	3.2	1.	3.2

```
vector bool char = rCmp
vector float r0={0.,0.,0.,0.};
vector float rA,rB,rC,rD,rS, rT,
rThen,rElse;
for (int i=0;i<1024;i+=4){
    // load rA, rB, and rD;
    rCmp = vec_cmplt(rC, r0);
    rT= rA*rB+rD;
    rThen = vec_and(rT.rCmp);
    rElse = vec_andc(rA.rCmp);
    rS = vec_or(rthen, relse);
    //store rS
}
```



Conditional Statements

```
for (int i=0;i<1024;i++){  
    if (c[i] < (float) 0.0)  
        a[i]=a[i]*b[i]+d[i];  
}
```

Speedups will depend on the values on c[i]

Compiler tends to be conservative, as the condition may prevent from segmentation faults

```
vector bool char = rCmp  
vector float r0={0.,0.,0.,0.};  
vector float rA,rB,rC,rD,rS, rT,  
rThen,rElse;  
for (int i=0;i<1024;i+=4){  
    // load rA, rB, and rD;  
    rCmp = vec_cmplt(rC, r0);  
    rT= rA*rB+rD;  
    rThen = vec_and(rT.rCmp);  
    rElse = vec_andc(rA.rCmp);  
    rS = vec_or(rthen, relse);  
    //store rS  
}
```



Compiler Directives

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

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



Compiler Directives

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

Compiler Hints for IBM XLC	Semantics
<code>#pragma ibm independent_loop</code>	Ignore assumed data dependences
<code>#pragma nosimd</code>	disable vectorization
<code>__restrict__</code>	assert exclusive access through pointer
<code>__attribute__((aligned(int-val)))</code>	request memory alignment
<code>memalign(int-val,size);</code>	malloc aligned memory
<code>__alignx (int-val, exp)</code>	assert alignment property



Outline

- Overcoming limitations to SIMD-Vectorization
 - Data Dependences
 - Data Alignment
 - Aliasing
 - Non-unit strides
 - Conditional Statements
- Vectorization with intrinsics



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



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 <pmmmintrin.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) single 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);
    }
}
```



Intel SSE intrinsics

A complete example

```
#define n 1024
```

```
int main() {  
    float a[n], b[n], c[n];  
    for (i = 0; i < n; i+=4) {  
        c[i:i+3]=a[i:i+3]+b[i:i+3];  
    }  
}
```

Header file

```
#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);  
    }  
}
```



Intel SSE intrinsics

A complete example

```
#define n 1024
```

```
int main() {  
    float a[n], b[n], c[n];  
    for (i = 0; i < n; i+=4) {  
        c[i:i+3]=a[i:i+3]  
    }  
}
```

Declare 3 vector registers



```
#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);  
    }  
}
```



Intel SSE intrinsics

A complete example

```
#define n 1000
```

```
int main() {  
    float a[n], b[n], c[n];  
    for (i = 0; i < n; i+=4) {  
        c[i:i+3]=a[i:i+3]+b[i:i+3];  
    }  
}
```



Execute vector
statements

```
#include <xmmintrin.h>
```

```
#define n 1024
```

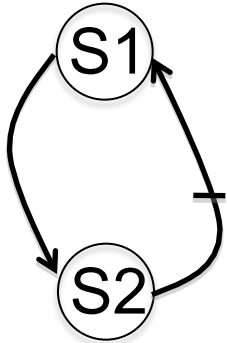
```
__attribute__((aligned(16))) float  
a[n], b[n], c[n];
```

```
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);  
    }  
}
```

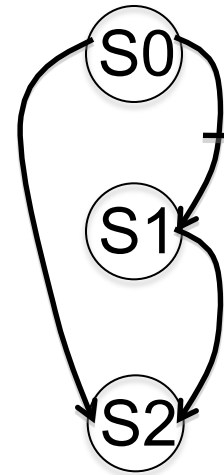


Node Splitting

```
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;  
}
```



Node Splitting with intrinsics

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

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

Which code runs faster ?

Why?

```
#include <xmmintrin.h>  
#define n 1000  
  
int main() {  
    __m128 rA1, rA2, rB, rC, rD;  
    __m128 r5=_mm_set1_ps((float)0.5)  
    for (i = 0; i < LEN-4; i+=4) {  
        rA2= _mm_loadu_ps(&a[i+1]);  
        rB= _mm_load_ps(&b[i]);  
        rC= _mm_load_ps(&c[i]);  
        rA1= _mm_add_ps(rB, rC);  
        rD= _mm_mul_ps(_mm_add_ps(rA1, rA2), r5);  
        _mm_store_ps(&a[i], rA1);  
        _mm_store_ps(&d[i], rD);  
    }  
}
```



Node Splitting with intrinsics

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

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

```
#include <xmmintrin.h>  
#define n 1000  
  
int main() {  
    __m128 rA1, rA2, rB, rC, rD;  
    __m128 r5=_mm_set1_ps((float)0.5)  
    for (i = 0; i < LEN-4; i+=4) {  
        rA2= _mm_loadu_ps(&a[i+1]);  
        rB= _mm_load_ps(&b[i]);  
        rC= _mm_load_ps(&c[i]);  
        rA1= _mm_add_ps(rB, rC);  
        rD= _mm_mul_ps(_mm_add_ps(rA1, rA2), r5);  
        _mm_store_ps(&a[i], rA1);  
        _mm_store_ps(&d[i], rD);  
    }  
}
```



Node Splitting with intrinsics

Intel Nehalem

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

Exec. Time scalar code: 12.6

Exec. Time vector code: --

Speedup: --

Intel Nehalem

Compiler report: Loop was vectorized.

Exec. Time scalar code: 13.2

Exec. Time vector code: 9.7

Speedup: 1.3

Intel Nehalem

Exec. Time intrinsics: 6.1

Speedup (versus vector code): 1.6



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 is to program the vector extensions directly using intrinsics or assembly language.



Data Dependences

- The correctness of many many loop transformations including vectorization can be decided using dependences.
- A good introduction to the notion of dependence and its applications can be found in D. Kuck, R. Kuhn, D. Padua, B. Leasure, M. Wolfe: Dependence Graphs and Compiler Optimizations. POPL 1981.



Compiler Optimizations

- For a longer discussion see:
 - Kennedy, K. and Allen, J. R. 2002 Optimizing Compilers for Modern Architectures: a Dependence-Based Approach. Morgan Kaufmann Publishers Inc.
 - U. Banerjee. Dependence Analysis for Supercomputing. Kluwer Academic Publishers, Norwell, Mass., 1988.
 - Advanced Compiler Optimizations for Supercomputers, by David Padua and Michael Wolfe in Communications of the ACM, December 1986, Volume 29, Number 12.
 - Compiler Transformations for High-Performance Computing, by David Bacon, Susan Graham and Oliver Sharp, in ACM Computing Surveys, Vol. 26, No. 4, December 1994.



Algorithms

- W. Daniel Hillis and Guy L. Steele, Jr.. 1986.
Data parallel algorithms. *Commun. ACM* 29, 12
(December 1986), 1170-1183.
- Shyh-Ching Chen, D.J. Kuck, "Time and Parallel
Processor Bounds for Linear Recurrence
Systems," *IEEE Transactions on Computers*, pp.
701-717, July, 1975



Measuring execution time

```
time1 = time();  
  
for (i=0; i<32000; i++)  
    c[i] = a[i] + b[i];  
  
time2 = time();
```



Measuring execution time

- Added an outer loop that runs (serially)
 - to increase the running time of the loop

```
time1 = time();  
for (j=0; j<200000; j++){  
    for (i=0; i<32000; i++)  
        c[i] = a[i] + b[i];  
}  
time2 = time();
```



Measuring execution times

- Added an outer loop that runs (serially)
 - to increase the running time of the loop
- **Call a dummy () function that is compiled separately**
 - to avoid loop interchange or dead code elimination

```
time1 = time();  
for (j=0; j<200000; j++){  
    for (i=0; i<32000; i++)  
        c[i] = a[i] + b[i];  
    dummy();  
}  
time2 = time();
```



Measuring execution times

- Added an outer loop that runs (serially)
 - to increase the running time of the loop
- Call a dummy () function that is compiled separately
 - to avoid loop interchange or dead code elimination
- **Access the elements of one output array and print the result**
 - to avoid dead code elimination

```
time1 = time();
for (j=0; j<200000; j++){
    for (i=0; i<32000; i++)
        c[i] = a[i] + b[i];
    dummy();
}
time2 = time();
for (j=0; j<32000; j++)
    ret+= a[j];
printf ("Time %f, result %f", (time2 -time1), ret);
```



Compiling

- Intel icc scalar code

icc -O3 –no-vec dummy.o tsc.o –o runnovec

- Intel icc vector code

icc -O3 –vec-report[n] –xSSE4.2 dummy.o tsc.o –o runvec

[n] can be 0,1,2,3,4,5

- **vec-report0**, no report is generated
- **vec-report1**, indicates the line number of the loops that were vectorized
- **vec-report2 .. 5**, gives a more detailed report that includes the loops that were not vectorized and the reason for that.



Compiling

```
flags = -O3 -qaltivec -qhot -qarch=pwr7 -qtune=pwr7  
-qipa=malloc16 -qdebug=NSIMDCOST  
-qdebug=always spec -qreport
```

- IBM xlc scalar code
xlc -qnoenablevmx dummy.o tsc.o -o runnovec
- IBM vector code
xlc -qenablevmx dummy.o tsc.o -o runvec







Compiler Directives (I)

- When the compiler does not vectorize automatically due to dependences the programmer can inform the compiler that it is safe to vectorize:

`#pragma ivdep (ICC compiler)`

`#pragma ibm independent_loop (XLC compiler)`

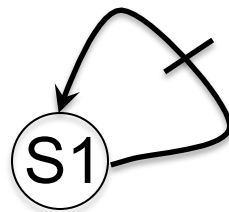


Compiler Directives (I)

- This loop can be vectorized when $k < -3$ and $k \geq 0$.
- Programmer knows that $k \geq 0$

```
for (int i=val; i<LEN-k; i++)  
    a[i]=a[i+k]+b[i];
```

If ($k \geq 0$) \rightarrow no dependence
or self-anti-dependence

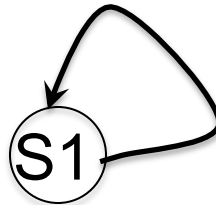


$k = 1$

$a[0] = a[1] + b[0]$
 $a[1] = a[2] + b[1]$
 $a[2] = a[3] + b[2]$

Can
be vectorized

If ($k < 0$) \rightarrow self-true dependence



$k = -1$

$a[1] = a[0] + b[0]$
 $a[2] = a[1] + b[1]$
 $a[3] = a[2] + b[2]$

Cannot
be vectorized



Compiler Directives (I)

- This loop can be vectorized when $k < -3$ and $k \geq 0$.
- Programmer knows that $k \geq 0$

How can the programmer tell the compiler that $k \geq 0$

```
for (int i=val; i<LEN-k; i++)  
    a[i]=a[i+k]+b[i];
```



Compiler Directives (I)

- This loop can be vectorized when $k < -3$ and $k \geq 0$.
- Programmer knows that $k \geq 0$

Intel ICC provides the `#pragma ivdep` to tell the compiler that it is safe to ignore unknown dependences

```
#pragma ivdep
for (int i=val; i<LEN-k; i++)
    a[i]=a[i+k]+b[i];
```

wrong results will be obtained if loop is vectorized when $-3 < k < 0$



Compiler Directives (I)

S124

```
for (int i=0;i<LEN-k;i++)  
    a[i]=a[i+k]+b[i];
```

S124_1

```
if (k>=0)  
    for (int i=0;i<LEN-k;i++)  
        a[i]=a[i+k]+b[i];  
if (k<0)  
    for (int i=0;i<LEN-k;i++)  
        a[i]=a[i+k]+b[i];
```

S124_2

```
if (k>=0)  
    #pragma ivdep  
    for (int i=0;i<LEN-k;i++)  
        a[i]=a[i+k]+b[i];  
if (k<0)  
    for (int i=0;i<LEN-k;i++)  
        a[i]=a[i+k]+b[i];
```

S124 and S124_1

Intel Nehalem

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

Exec. Time scalar code: 6.0

Exec. Time vector code: --

Speedup: --

S124_2

Intel Nehalem

Compiler report: Loop was vectorized

Exec. Time scalar code: 6.0

Exec. Time vector code: 2.4

Speedup: 2.5



Compiler Directives (I)

S124

```
for (int i=0;i<LEN-k;i++) if (k>=0)
    a[i]=a[i+k]+b[i];
```

S124_1

```
for (int i=0;i<LEN-k;i++)
    a[i]=a[i+k]+b[i];
if (k<0)
    for (int i=0;i<LEN-k;i++)
        a[i]=a[i+k]+b[i];
```

S124_2

```
if (k>=0)
    #pragma ibm independent_loop
    for (int i=0;i<LEN-k;i++)
        a[i]=a[i+k]+b[i];
if (k<0)
    for (int i=0;i<LEN-k;i++)
        a[i]=a[i+k]+b[i];
```

S124 and S124_1

IBM Power 7

Compiler report: Loop was not vectorized because a data dependence prevents SIMD vectorization

Exec. Time scalar code: 2.2

Exec. Time vector code: --

Speedup: --

S124_2

#pragma ibm independent_loop
needs AIX OS (we ran the experiments on Linux)



Strip Mining

This transformation improves locality and is usually combined with vectorization



Strip Mining

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

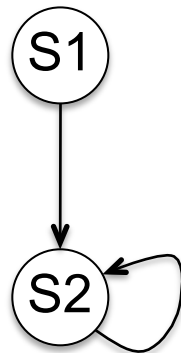


```
for (i=1; i<LEN; i++)  
    a[i]= b[i];  
  
for (i=1; i<LEN; i++)  
    c[i] = c[i-1] + a[i];
```

- first statement can be vectorized
- second statement cannot be vectorized because of self-true dependence

By applying loop distribution the compiler will vectorize the first statement

But, ... loop distribution will increase the cache miss ratio if array a[] is large



Strip Mining

Loop Distribution

```
for (i=1; i<LEN; i++)  
    a[i]= b[i];  
for (i=1; i<LEN; i++)  
    c[i] = c[i-1] + a[i];
```



Strip Mining

```
for (i=1; i<LEN; i  
+=strip_size){  
    for (j=i; j<strip_size; j++)  
        a[j]= b[j];  
    for (j=i; j<strip_size; j++)  
        c[j] = c[j-1] + a[j];  
}
```

strip_size is usually a small value (4, 8, 16 or 32).



Strip Mining

- Another example

```
int v[N];  
...  
for (int i=0;i<N;i++){  
    Transform (v[i]);  
}  
for (int i=0;i<N;i++){  
    Light (v[i]);  
}
```

```
int v[N];  
...  
for (int i=0;i<N;i+=strip_size){  
    for (int j=i;j<strip_size;j++){  
        Transform (v[j]);  
    }  
    for (int j=i;j<strip_size;j++){  
        Light (v[j]);  
    }  
}
```



Compiler Directives (I)

- When the compiler does not vectorize automatically due to dependences the programmer can inform the compiler that it is safe to vectorize:

`#pragma ivdep (ICC compiler)`

`#pragma ibm independent_loop (XLC compiler)`



Compiler Directives (II)

- Programmer can disable vectorization of a loop when the when the vector code runs slower than the scalar code

`#pragma novector` (ICC compiler)

`#pragma nosimd` (XLC compiler)

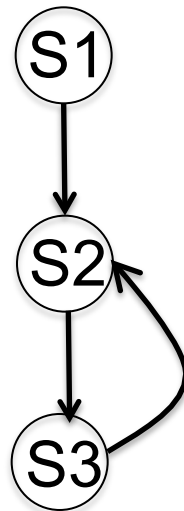


Compiler Directives (II)

Vector code can run slower than scalar code

```
for (int i=1;i<LEN;i++){  
S1  a[i] = b[i] + c[i];  
S2  d[i] = a[i] + e[i-1];  
S3  e[i] = d[i] + c[i];  
}
```

Less locality when
executing in vector mode



S1 can be vectorized
S2 and S3 cannot be vectorized (as they are)



Compiler Directives (II)

S116

`#pragma novector`

```
for (int i=1;i<LEN;i++){  
    a[i] = b[i] + c[i];  
    d[i] = a[i] + e[i-1];  
    e[i] = d[i] + c[i];  
}
```

S116

Intel Nehalem

Compiler report: Loop was
partially vectorized

Exec. Time scalar code: 14.7

Exec. Time vector code: 18.1

Speedup: 0.8

