Processor-level SIMD

- SIMD instructions can perform an operation on multiple words simultaneously
- This is a form of data parallelism
- SIMD: single-instruction, multiple data

Recent SIMD versions

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Nehalem</td>
<td>SSE 4.2 (128 bit)</td>
</tr>
<tr>
<td>Sandy Bridge</td>
<td>AVX (256 bit)</td>
</tr>
<tr>
<td>Haswell</td>
<td>AVX2 (256 bit with FMA)</td>
</tr>
<tr>
<td>MIC</td>
<td>AVX-512 (512 bit)</td>
</tr>
</tbody>
</table>

Versions are not backward compatible, i.e., cannot use AVX instructions on Nehalem.
Loops such as the following could exploit SIMD

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

With AVX, 8 floats or 4 doubles are computed at the same time.

Historical note: CRAY computers had vector units (often 64 words long) that operated in pipelined fashion.

- processor-level SIMD instructions are often called short vector instructions, and using these instructions is called vectorization
- exploiting SIMD is essential in HPC codes, especially on processors with wide SIMD units, e.g., Intel Xeon Phi
Many ways to exploit SIMD

- use the auto-vectorizer in the compiler (i.e., do nothing except help make sure that the coding style will not prevent vectorization)
- use SIMD intrinsic functions in your code (when the auto-vectorizer does not seem to do what you want)
Example use of intrinsic functions (AVX)

```c
__mm256 qa, qb;

for (i=0; i<n/4; i++) {
    qa = _mm256_load_ps(a);
    qb = _mm256_load_ps(b);
    qa = _mm256_add_ps(qa, qb);
    _mm256_store_ps(a, qa);
    a += 4;
    b += 4;
}
```
Data alignment

- Many things can prevent automatic vectorization or produce suboptimal vectorized code.
- In the previous example, the loads into the vector $q_a$ must come from an address $a$ that is 256-bit (32 bytes) aligned.

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

- If $a$ and/or $b$ can be unaligned, then the compiler will not vectorize the code or will generate extra code at the beginning and/or end of the loop to handle the misaligned elements.
  - Need to allocate memory that is aligned.
  - Need to tell the compiler that the memory is aligned.
Allocating aligned memory

- In these examples, assume AVX-512, or 64-byte alignment needed

```c
#include <stdlib.h>
...
buffer = _mm_malloc(num_bytes, 64);
...
_mm_free(buffer);
```

- Other functions for allocating aligned memory also available

```c
#include <stdlib.h>
...
ret = posix_memalign(&buffer, 64, num_bytes);
buffer = aligned_alloc(64, num_bytes); // C11
...
free(buffer);
```
Compiler hints

- Aligned memory on the stack

```c
__declspec(align(64)) double a[4000];
```

- Telling the compiler that memory is aligned

```c
__assume_aligned(ptr, 64);
```
Auto vectorization

Tell the compiler what architecture you are using. Examples:

- **-mmic** (Intel)
- **-msse4.2** (Gnu)
Auto vectorization

- Vectorization is enabled with `-O1` and above (otherwise vector instructions not used?)
- Default is `-O2` which is the same as `-O` or not specifying the optimization level flag
Disabling vectorization

- `no-vec` disables auto-vectorization
- `no-simd` disables vectorization of loops with Intel SIMD pragmas
  (see also `-qno-openmp-simd`)
Vectorization reports

The compiler can tell you how well your code was vectorized. Compile with

-`-qopt-report=1 -qopt-report-phase=vec`

The compiler will output an * .optrpt file.
Will these loops vectorize?

```
for (i=0; i<n; i+=2)
    b[i] += a[i]*x[i];
```
Will these loops vectorize?

```c
for (i=0; i<n; i+=2)
    b[i] += a[i]*x[i];

for (i=0; i<n; i++)
    b[i] += a[i]*x[index[i]];
```
Obstacles to vectorization

- non-contiguous memory access
Will these loops vectorize?

```c
for (i=1; i<=n; i++)
    a[i] = a[i-1] + 1.;

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

No and Yes.
Will these loops vectorize?

```
for (i=1; i<=n; i++)
    a[i] = a[i-1] + 1.;

a[1] = a[0] + 1;

for (i=1; i<=n; i++)
    a[i-1] = a[i] + 1.;

a[0] = a[1] + 1;
```
Will these loops vectorize?

```c
for (i=1; i<=n; i++)
    a[i] = a[i-1] + 1.;

a[1] = a[0] + 1;

for (i=1; i<=n; i++)
    a[i-1] = a[i] + 1.;

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

No and Yes.
Obstacles to vectorization

- non-contiguous memory access
- data dependencies
Will this loop vectorize?

double sum = 0.;
for (j=0; j<n; j++)
    sum += a[j]*b[j];
Will this loop vectorize?

```
double sum = 0.;
for (j=0; j<n; j++)
    sum += a[j]*b[j];
```

Yes, the compiler recognizes this as a reduction.
Will this loop vectorize?

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

No, array c might overlap with array a or b.

However, the compiler might generate code that tests for overlap and use different code depending on whether or not there is overlap.
Use the `restrict` keyword

Tell the compiler that a pointer is the only way to access an array

```c
void add(int n, double * restrict a,
         double * restrict b, double * restrict c)
{
    for (int i=0; i<n; i++)
        c[i] = a[i] * b[i];
}
```

- `restrict` is not available on all compilers
- on Intel compilers, use `-restrict` on the compile line
Use `#pragma ivdep`

Tell the compiler to ignore any potential data dependencies

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

Another example (`restrict` cannot help):

```c
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;  // cannot vectorize if k<0
}
```
ivdep and other pragmas

- #pragma ivdep - ignore potential data dependencies in loop
- #pragma novector - do not vectorize loop
- #pragma loop count(n) - typical trip count tells compiler whether vectorization is worthwhile
- #pragma vector always - always vectorize
- #pragma vector align - asserts data in loop is aligned
- #pragma vector nontemporal - hints to compiler that data will not be reused, and therefore to use streaming stores that bypass cache
Which is better?

```c
struct Particle {
    double pos[3];
    double radius;
    int    type;
};

// array of structures
struct Particle allparticles[10000];

struct AllParticles {
    double pos[3][10000];
    double radius[10000];
    int    type[10000];
};

// structure of arrays
struct AllParticles allparticles;
```
Data layout may affect vectorization e.g., how will positions be updated in AOS case?

SOA is often better for vectorization

AOS may also be bad for cache line alignment (but padding can be used here)

If particles are accessed in a “random” order and particle radius and type are accessed with positions, then SOA may underutilize cache lines
Can this pseudocode be vectorized?

```plaintext
for i = 1 to npos
    for j = 1 to npos
        if (i == j)
            continue
        vec = p(i) - p(j)
        dist = norm(vec);
        force(i) += k * vec / (dist^2);
    end
end
```
Trick to remove the if test

```plaintext
for i = 1 to npos
    for j = 1 to npos
        vec = p(i) - p(j)
        dist = norm(vec) + eps;
        force(i) += k*vec/(dist^2);
    end
end
```
Next time: #pragma omp simd