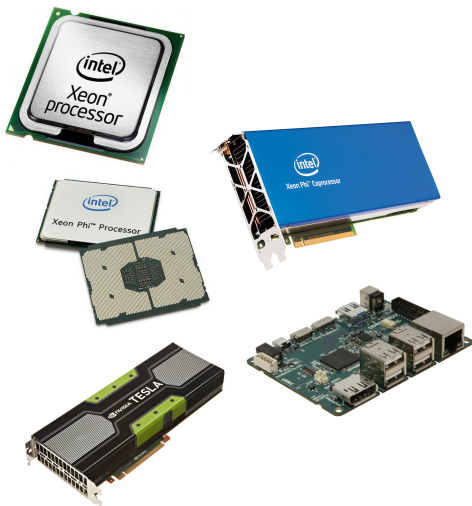


OpenMP: Vectorization and `#pragma omp simd`

Where does it come from?

$$c_i = a_i + b_i \quad \forall i$$

	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8
+	b_1	b_2	b_3	b_4	b_5	b_6	b_7	b_8
=	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8



Why would I care?



Why would I care?



Everywhere...

x86	SSE	128 bit
	AVX(2)	256 bit
	AVX-512 (IMCI)	512 bit
ARM	NEON	128 bit
POWER	Altivec/VMX/VSX	128 bit
	QPX	256 bit
SPARC	HPC-ACE	128 bit
	HPC-ACE2	256 bit

Vectorization on Intel CPUs

`[v]movap/s reg1, reg2/mem`

`reg: xmm0-xmm15 (128bit)`

`ymm0-ymm15 (256bit)`

`zmm0-zmm15 (512bit)`

`vaddps: Vectorized`

`vaddss: Scalar`

How to see assembly: add `-S` to command line.

Vectorization: Indication

Profiling! Worth it on the hot path!

- ▶ Increases available memory bandwidth to cache
- ▶ **Increases throughput of compute operations**
- ▶ More power efficient
- ▶ Reduce frontend pressure (fewer instructions to decode)

Keep in mind Ahmdahl's law!

Cool! How can I use that?

- ▶ Libraries
(MKL, OpenBLAS, BLIS, fftw, numpy, OpenCV)
- ▶ Hoping for a good compiler: Autovectorization
- ▶ Assisting compiler through annotations: OpenMP SIMD pragma
- ▶ Writing intrinsics/assembly code (not covered here)

Autovectorization

- ▶ The compiler needs to prove that the optimization is legal
- ▶ And the compiler needs to prove that the optimization is beneficial (under almost all circumstances)
- ▶ What could possibly go wrong?
- ▶ Conditionals (different vector lanes executing different code)
- ▶ Inner loops (might have different trip counts)
- ▶ Function calls (the functions might not be vectorized)
- ▶ Cross-iteration dependencies
- ▶ OpenMP addresses the last two points in particular

The OpenMP simd pragma

- ▶ Unifies the enforcement of vectorization for `for` loop
- ▶ Introduced in OpenMP 4.0
- ▶ Explicit vectorization of `for` loops
- ▶ Same restrictions as `omp for`, and then some
- ▶ Executions in chunks of `simdlength`, concurrently executed
- ▶ Only directive allowed inside: `omp ordered simd` (OpenMP 4.5)
- ▶ Can be combined with `omp for`
- ▶ No exceptions

Clauses

- ▶ `safelen(len)`: Maximum number of iterations per chunk
- ▶ `simdlen(len)`: Recommended number of iterations per chunk
- ▶ `linear(stride: var, ...)`: with respect to iteration variable
- ▶ `aligned(alignment: var)`: alignment of variable
- ▶ `private`, `lastprivate`, `reduction`, `collapse`: As with `omp for`

Issues with your code

- ▶ Aliasing
- ▶ Alignment
- ▶ Floating point issues
- ▶ Correctness
- ▶ Function calls
- ▶ Ordering

Aliasing

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

Compiler does not know that a and b do not overlap.
Has to be conservative.

Aliasing

```
float * a = ...;
float * b = ...;
float aa[N];
memcpy(aa, a, N * sizeof(float));
float bb[N];
memcpy(bb, b, N * sizeof(float));
float s;
...
for (int i = 0; i < N; i++) {
    aa[i] += s * bb[i];
}
memcpy(a, aa, N * sizeof(float));
```

Aliasing

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

Aliasing

```
float * a = ...;  
float * b = ...;  
float s;  
...  
#pragma omp simd  
for (int i = 0; i < N; i++) {  
    a[i] += s * b[i];  
}
```

Alignment

- ▶ Loading a chunk of data is cheaper if the address is aligned.
- ▶ Allows for faster hardware instructions to load a vector.
- ▶ Avoid cache line splits.
- ▶ Ex: Recent Intel CPUs have 64 byte cache lines, and 32 byte vectors, best alignment is 32 bytes.

Cache lines A,B,...:

AAAABBBBCCCCDDDDDEEEEEEFFFFGGGGHHHHIIII

<- Want to load this data? Unaligned.

<- Want to load this data? Aligned.

Alignment

```
float * a;  
posix_memalign(&a, ...);  
float * b;  
posix_memalign(&b, ...);  
float s;  
...  
__assume_aligned(a, 32); // Intel  
__assume_aligned(b, 32); // Intel  
#pragma omp simd  
for (int i = 0; i < N; i++) {  
    a[i] += s * b[i];  
}
```

Alignment

```
float * a;  
posix_memalign(&a, ...);  
float * b;  
posix_memalign(&b, ...);  
float s;  
...  
#pragma omp simd aligned(a, b: 32)  
for (int i = 0; i < N; i++) {  
    a[i] += s * b[i];  
}
```

Floating Point Models

```
for (int i = 0; i < n; i++) {  
    sum += a[i];  
}
```

```
-ffast-math /fp:fast -fp-model fast=2
```

```
#pragma omp simd reduction(+: sum)  
for (int i = 0; i < n; i++) {  
    sum += a[i];  
}
```

<https://msdn.microsoft.com/en-us/library/aa289157.aspx>

Correctness

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

Vectorization is only legal if the elements in `d` are distinct.
This case occurs in applications!

```
#pragma omp simd  
for (int i = 0; i < N; i++) {  
    int j = d[i];  
    a[j] += s * b[i];  
}
```

Functions

```
// This won't vectorize unless foo inlined.  
foo(float a, float * b, float c);  
  
float s;  
#pragma omp simd  
for (int i = 0; i < N; i++) {  
    int j = d[i];  
    a[j] += foo(s, &b[i], a[j]);  
}
```

The OpenMP declare simd directive

- ▶ asks compiler to generate vectorized version of a function
- ▶ allows vectorization of loops with function calls
- ▶ notinbranch, inbranch: Generate masking code, non-masking code
- ▶ everything from the simd pragma + uniform
- ▶ uniform: does not change
- ▶ linear: increases with index

Functions

```
#pragma omp declare simd uniform(a) linear(1: b)
foo(float a, float * b, float c);

float s;
#pragma omp simd
for (int i = 0; i < N; i++) {
    int j = d[i];
    a[j] += foo(s, &b[i], a[j]);
}
```

Masks

```
#pragma omp simd
for (int i = 0; i < n; i++) {
    if (a[i] < 1.0) continue;
    // ..
    int j = d[i];
    a[j] = ...;
}
```

Ordering

```
#pragma omp simd
for (int i = 0; i < n; i++) {
    if (a[i] < 1.0) continue;
    // ..
    int j = d[i];
    #pragma omp ordered simd
    a[j] = ...;
}
```

If d is not containing distinct elements.