Vectorization

- Example in the “Vectorization” directory

- “Advanced” topic, but critical for numeric programming these days
Flynn’s Taxonomy

- Up until now, we have basically assumed we have a single instruction, single data (SISD) model:
  - 1 CPU processes data from 1 location
  - Parallelization is achieved by just repeating this:
    - Single program, multiple data (SPMD)

https://en.wikipedia.org/wiki/Flynn%27s_taxonomy
Flynn’s Taxonomy

Single instruction

SISD

Single data

MISD

Multiple instruction

SIMD

Multiple data

MIMD
Flynn’s Taxonomy
SIMD

• SIMD has become more popular because of vectorized processing units
  – GPUs
  – New CPUs

• Vector units do N identical operations in the same amount of time that it takes to do one of them, on different addresses in memory

• Speed comes from data access
  – “Get me N ints” versus “Get me an int”, N times
Architecture of processors

- Processing units now contain caches of memory with fast access.
- Can have different levels of cache (small but fast versus large but slow)
- Then memory
- Then disk

- Accessing cache is much faster than accessing memory
- Accessing memory is much faster than accessing disk

Architecture of processors

• Cooking analogy:
Vector processing

- Same operation done many times over a vector of data (SIMD!)
- Traditionally: done with loops
- Now: done with vector unit, if possible

```plaintext
for (i = 0; i < 1024; i++)
    C[i] = A[i] * B[i];
```

C = A * B;

Get next bread.
Slice.
Get next bread.
Slice.
Get next bread.
Slice.
Slice all the bread.

https://en.wikipedia.org/wiki/Vector_processor
Unrolling loops

- There are also advantages to unrolling loops:

```plaintext
for (i = 0; i < 1024; i++)
    C[i] = A[i]*B[i];

for (i = 0; i < 1024; i+=4){
    C[i+0] = A[i+0]*B[i+0];
    C[i+1] = A[i+1]*B[i+1];
    C[i+2] = A[i+2]*B[i+2];
    C[i+3] = A[i+3]*B[i+3];
}
```

Get next bread.
Slice.
Get next bread.
Slice.
Get next bread.
Slice.
Get four slices of bread.
Slice.

https://en.wikipedia.org/wiki/Loop_unrolling
Vectorization in Practice

- Scalar:

```cpp
for (i = 0; i < 1024; i++)
    C[i] = A[i]*B[i];
```

- Vectorized:

```cpp
for (i = 0; i < 1024; i+=4)
    C[i:i+3] = A[i:i+3]*B[i:i+3];
```

Implicitly or explicitly unrolling loops can allow the compiler to appropriately vectorize the operation

https://en.wikipedia.org/wiki/Automatic_vectorization
Test case

- We will use the simple addition of two large vectors (100000 elements) as a demonstration in various scenarios:
  - C++ without optimization
  - C++ with non-vectorization optimizations
  - C++ adding vectorization optimizations
  - Python itself
  - Numpy within python
## Comparisons

<table>
<thead>
<tr>
<th></th>
<th>C++, -O0</th>
<th>C++, -O1</th>
<th>C++, -O2</th>
<th>python</th>
<th>python+numpy</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Vector, unknown size</strong></td>
<td>825</td>
<td>585</td>
<td>150</td>
<td>31862</td>
<td>460</td>
</tr>
<tr>
<td><strong>Vector, known size</strong></td>
<td>736</td>
<td>564</td>
<td>93</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Static array</strong></td>
<td>576</td>
<td>564</td>
<td>525</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Static array, manual unroll</strong></td>
<td>132</td>
<td>97</td>
<td>93</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>