Why Vectorization?

- What is vectorization?
  - It is the expression of a computation in terms of identical operations on vectors of data

- Initial motivation:
  - Enhanced performance with pipelined functional units in early supercomputers: independent operations on components of vector to feed the units

- Today: Vector-SIMD ISAs (SSE, AVX, AVX-512,..) and SIMD functional units in GPUs enable both energy efficiency and high performance with vectorizable computations
First effective vector supercomputer: Cray-1
- 64-element vector registers
- Pipelined execution of operations
- Vectorizing FORTRAN compiler
# Vector Machines

<table>
<thead>
<tr>
<th>Machine</th>
<th>Year</th>
<th>Clock</th>
<th>Regs</th>
<th>Elements</th>
<th>FUs</th>
<th>LSUs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cray 1</td>
<td>1976</td>
<td>80 MHz</td>
<td>8</td>
<td>64</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>Cray XMP</td>
<td>1983</td>
<td>120 MHz</td>
<td>8</td>
<td>64</td>
<td>8</td>
<td>2L, 1S</td>
</tr>
<tr>
<td>Cray YMP</td>
<td>1988</td>
<td>166 MHz</td>
<td>8</td>
<td>64</td>
<td>8</td>
<td>2L, 1S</td>
</tr>
<tr>
<td>Cray C-90</td>
<td>1991</td>
<td>240 MHz</td>
<td>8</td>
<td>128</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>Cray T-90</td>
<td>1996</td>
<td>455 MHz</td>
<td>8</td>
<td>128</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>Conv. C-1</td>
<td>1984</td>
<td>10 MHz</td>
<td>8</td>
<td>128</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Conv. C-4</td>
<td>1994</td>
<td>133 MHz</td>
<td>16</td>
<td>128</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Fuj. VP200</td>
<td>1982</td>
<td>133 MHz</td>
<td>8-256</td>
<td>32-1024</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Fuj. VP300</td>
<td>1996</td>
<td>100 MHz</td>
<td>8-256</td>
<td>32-1024</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>NEC SX/2</td>
<td>1984</td>
<td>160 MHz</td>
<td>8+8K</td>
<td>256+var</td>
<td>16</td>
<td>8</td>
</tr>
<tr>
<td>NEC SX/3</td>
<td>1995</td>
<td>400 MHz</td>
<td>8+8K</td>
<td>256+var</td>
<td>16</td>
<td>8</td>
</tr>
</tbody>
</table>

Source: F. Chong
Vectors in Commodity Systems

- Initial use of vectors was only in high-end supercomputers
- By mid 90’s, desktops became powerful enough for real-time gaming and video
- Performance demands led to vector-SIMD instruction set architectures (ISAs) on commodity processors

<table>
<thead>
<tr>
<th>Year</th>
<th>Vendor</th>
<th>ISA</th>
</tr>
</thead>
<tbody>
<tr>
<td>1995</td>
<td>Sun</td>
<td>VIS</td>
</tr>
<tr>
<td>1996</td>
<td>Intel</td>
<td>MMX</td>
</tr>
<tr>
<td>1997</td>
<td>IBM/Motorola</td>
<td>Altivec</td>
</tr>
<tr>
<td>1998</td>
<td>AMD</td>
<td>3DNow!</td>
</tr>
<tr>
<td>1999</td>
<td>Intel</td>
<td>SSE</td>
</tr>
<tr>
<td>2001</td>
<td>Intel</td>
<td>SSE2</td>
</tr>
<tr>
<td>2004</td>
<td>Intel</td>
<td>SSE3</td>
</tr>
<tr>
<td>2006</td>
<td>Intel</td>
<td>SSSE3</td>
</tr>
<tr>
<td>2006</td>
<td>Intel</td>
<td>SSE4</td>
</tr>
<tr>
<td>2008</td>
<td>Intel</td>
<td>AVX</td>
</tr>
<tr>
<td>2011</td>
<td>ARM</td>
<td>NEON</td>
</tr>
<tr>
<td>2015</td>
<td>Intel</td>
<td>AVX-512</td>
</tr>
</tbody>
</table>
Vector-SIMD Model

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

for (i=0; i<n; i+=4)
    c[i:i+3]=a[i:i+3]+b[i:i+3];

Source: Maria Garzaran and David Padua
Vector-SIMD Characteristics

- SIMD: Single Instruction Multiple Data
  - Synchronized lock-step execution by multiple functional units
  - Data generally contiguous in memory
- SIMD functional units in most CPUs and GPUs
- High performance and energy efficiency
- But, greater burden on programmer, unless automatically vectorized by compiler
Utilizing Vector-SIMD Units

Three choices

1. C code and a vectorizing compiler

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

2. Macros or Vector Intrinsics

   ```c
   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. Assembly Language

   ```assembly
   .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
   ```

Source: Maria Garzaran and David Padua
How well do compilers vectorize?

<table>
<thead>
<tr>
<th>Loops</th>
<th>Compiler</th>
<th>XLC</th>
<th>ICC</th>
<th>GCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td></td>
<td>159</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vectorized</td>
<td></td>
<td>74</td>
<td>75</td>
<td>32</td>
</tr>
<tr>
<td>Not vectorized</td>
<td></td>
<td>85</td>
<td>84</td>
<td>127</td>
</tr>
<tr>
<td>Average Speed Up</td>
<td></td>
<td>1.73</td>
<td>1.85</td>
<td>1.30</td>
</tr>
</tbody>
</table>

By adding manual vectorization the average speedup was 3.78 (versus 1.73 obtained by the XLC compiler)

Source: Maria Garzaran and David Padua
Example: Stride-1, Independent Ops

for (i=0; i<N; i++) A[i] = A[i]+1;

- Stride-1 accesses to Array A
- Loop has independent operations (no loop carried dependences)
- Data is resident in L1 cache
- SSE: 128-bit vector = 2 doubles = 4 floats/int = 16 chars
- AVX: 256-bit vector = 4 doubles = 8 floats/int = 32 chars
- Intel icc v.13.1.3; compiled –fast; enable/disable vectorization

<table>
<thead>
<tr>
<th></th>
<th>char</th>
<th>int</th>
<th>float</th>
<th>dble</th>
</tr>
</thead>
<tbody>
<tr>
<td>No-Vector</td>
<td>0.5</td>
<td>0.5</td>
<td>0.5</td>
<td>0.67</td>
</tr>
<tr>
<td>Vector</td>
<td>9.7</td>
<td>2.5</td>
<td>2.5</td>
<td>1.36</td>
</tr>
<tr>
<td>Speedup</td>
<td>19.4</td>
<td>5.0</td>
<td>5.0</td>
<td>2.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>char</th>
<th>int</th>
<th>float</th>
<th>dble</th>
</tr>
</thead>
<tbody>
<tr>
<td>No-Vector</td>
<td>0.88</td>
<td>0.9</td>
<td>0.95</td>
<td>0.87</td>
</tr>
<tr>
<td>Vector</td>
<td>9.5</td>
<td>3.8</td>
<td>7.2</td>
<td>3.8</td>
</tr>
<tr>
<td>Speedup</td>
<td>10.8</td>
<td>4.2</td>
<td>7.6</td>
<td>4.4</td>
</tr>
</tbody>
</table>

**Ops/cycle**

Nehalem (Core i7-920 @2.67 Ghz)

<table>
<thead>
<tr>
<th></th>
<th>char</th>
<th>int</th>
<th>float</th>
<th>dble</th>
</tr>
</thead>
<tbody>
<tr>
<td>No-Vector</td>
<td>0.88</td>
<td>0.9</td>
<td>0.95</td>
<td>0.87</td>
</tr>
<tr>
<td>Vector</td>
<td>9.5</td>
<td>3.8</td>
<td>7.2</td>
<td>3.8</td>
</tr>
<tr>
<td>Speedup</td>
<td>10.8</td>
<td>4.2</td>
<td>7.6</td>
<td>4.4</td>
</tr>
</tbody>
</table>

**Ops/cycle**

Haswell (Core i7-4770K @3.49 Ghz)
Example: Stride-16, Independent Ops

for (i=0; i<N; i+=16) A[i] = A[i]+1;

- Stride-16 accesses to Array A
- No performance gain; often loss of performance from vectorization
- Unlike past vector machines like the Cray-1, current vector-SIMD ISAs do not support efficient strided data access from memory

<table>
<thead>
<tr>
<th></th>
<th>char</th>
<th>int</th>
<th>float</th>
<th>dble</th>
</tr>
</thead>
<tbody>
<tr>
<td>No-Vec</td>
<td>0.65</td>
<td>0.66</td>
<td>0.5</td>
<td>0.5</td>
</tr>
<tr>
<td>Vector</td>
<td>0.52</td>
<td>0.56</td>
<td>0.4</td>
<td>0.65</td>
</tr>
<tr>
<td>Speedup</td>
<td>0.80</td>
<td>0.85</td>
<td>0.80</td>
<td>1.30</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>char</th>
<th>int</th>
<th>float</th>
<th>dble</th>
</tr>
</thead>
<tbody>
<tr>
<td>No-Vec</td>
<td>0.76</td>
<td>0.81</td>
<td>0.69</td>
<td>0.69</td>
</tr>
<tr>
<td>Vector</td>
<td>0.54</td>
<td>0.66</td>
<td>0.51</td>
<td>1.0</td>
</tr>
<tr>
<td>Speedup</td>
<td>0.71</td>
<td>0.81</td>
<td>0.74</td>
<td>1.45</td>
</tr>
</tbody>
</table>

**Ops/cycle**
Nehalem (Core i7-920 @2.67 Ghz)

**Ops/cycle**
Haswell (Core i7-4770K @3.49 Ghz)
Example: Stride-1, Dependent Ops

for (i=1; i<N; i++) A[i] = A[i-1]+1;

- Stride-1 accesses to Array A
- Loop has loop-carried dependence
- Cannot use vector instructions since independent operations are not available

|        | char | int  | float | dbles
|--------|------|------|-------|-------
| No-Vec | 0.66 | 0.66 | 0.33  | 0.33  |
| Vector | 0.66 | 0.66 | 0.33  | 0.33  |
| Speedup| 1.0  | 1.0  | 1.0   | 1.0   |

|        | char | int  | float | dbles
|--------|------|------|-------|-------
| No-Vec | 1.0  | 1.0  | 0.33  | 0.33  |
| Vector | 1.0  | 1.0  | 0.33  | 0.33  |
| Speedup| 1.0  | 1.0  | 1.0   | 1.0   |

**Ops/cycle**
Nehalem (Core i7-920 @2.67 Ghz)

**Ops/cycle**
Haswell (Core i7-4770K @3.49 Ghz)
Vectorization of Loop Programs

1. Independent operations (usually in innermost loops) can be vectorized

2. For short-vector SIMD ISAs, the operands must also have unit stride w.r.t. inner loop (or stride 0, i.e., be loop invariant)

- For simple single-statement, single-loop programs, loop carried dependences can inhibit vectorization
- For multi-statement and/or multi-loop code:
  - Data dependence graph analysis reveals vectorizability
  - Loop transformations may enhance vectorizability
- Feedback from compiler shows which loops were/weren’t vectorized and why
  - gcc -O3 -ftree-vectorizer-verbose=n (n=1 or higher)
Examples: Acyclic Dependences

for (i=0; i<N-1; i++) {
    w[i+1] = x[i]+1;
    y[i] = 2*w[i];
}

w[1:N-1] = x[0:N-2]+1;    S1
y[0:N-2] = 2*w[0:N-2];    S2

for (i=0; i<N-1; i++) {
    w[i] = y[i]+1;
    y[i+1] = 2*x[i];
}

y[1:N-1] = 2*x[0:N-2];    S2
w[0:N-2] = y[0:N-2]+1;    S1

• Loop Distribution is needed to vectorize multi-statement loops
Examples: Cyclic Dependences

for (i=0; i<N-1; i++) {
    x[i+1] = y[i]+1;
    y[i+1] = 2*x[i];
}

S1
S2

Not vectorizable

for (i=0; i<N-1; i++) {
    x[i+1] = y[i]+x[i];
    y[i+1] = 2*y[i]+z[i];
}

S1
S2

Not vectorizable

- Cycles in dependence graph prevent vectorization
Example: Vectorizable After Transform

- Can vectorize if dependence is carried at an outer loop
Example: Vectorizable After Transform

for (i=0; i<N; i++) {
    sum = 0.0;
    for (j=0; j<N; j++)
        sum += A[j][i]*A[j][i];
    x[i] = sum;
}

for (i=0; i<N; i++)
    sum[i] = 0;
for (i=0; i<N; i++) {
    for (j=0; j<N; j++)
        sum[j] += A[i][j]*A[i][j];
    x[i] = sum[i];

- Anti-dependences in loops may be removable via scalar expansion
Example: Cyclic Dependence

A loop can be partially vectorized

```
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];
}
```

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

Source: Maria Garzaran and David Padua
Example: Cyclic Dependence

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

Self-antidependence can be vectorized

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

Source: Maria Garzaran and David Padua
Example: Cyclic Dependence

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

Can this loop be vectorized?

- i=0, j=0: $a[1][0] = a[0][0] + b$
- j=1: $a[1][1] = a[0][1] + b$
- j=2: $a[1][2] = a[0][2] + b$
- i=1, j=0: $a[2][0] = a[1][0] + b$
- j=1: $a[2][1] = a[1][1] + b$
- j=2: $a[2][2] = a[1][2] + b$

Source: Maria Garzaran and David Padua
Example: Cyclic Dependence

for (int i = 0; i < LEN-1; i++) {
  for (int j = 0; j < LEN; j++)
    a[i+1][j] = a[i][j] + (float) 1.0;
}

Can this loop be vectorized?

Dependences occur in the outermost loop.
- outer loop runs serially
- inner loop can be vectorized

for (int i=0;i<LEN;i++){
  a[i+1][0:LEN-1]=a[i][0:LEN-1]+b;
}
AoS versus SoA

Array-of-Structures often prevents vectorization

typedef struct
{ float x;float y; float z;
  float dsquared;
} coords;

coords nbodies_aos[N];

for(i=0;i<N;i++)
  nbodies_aos[i].dsquared += nbodies_aos[i].x*nbodies_aos[i].x
  + nbodies_aos[i].y*nbodies_aos[i].y
  + nbodies_aos[i].z*nbodies_aos[i].z;
AoS versus SoA

Converting to Structure-of-Arrays can enable vectorization

```c
typedef struct {
    float x[N]; float y[N]; float z[N];
    float dsquared[N];
} coords_arr;
```

```c
coords_arr nbodies_soa;
```

```c
for(i=0;i<N;i++)
    nbodies_soa.dsquared[i] += nbodies_soa.x[i]*nbodies_soa.x[i]
    + nbodies_soa.y[i]*nbodies_soa.y[i]
    + nbodies_soa.z[i]*nbodies_soa.z[i];
```
Summary

- Vectorization requires independent operations with stride 0/1 operands
- Innermost loop vectorization is most common; outerloop vectorization is also possible
- Loop permutation can enable vectorization by 1) making innermost loop parallel, and 2) make stride 0/1
- Anti-dependences due to scalars can inhibit vectorization; scalar expansion can enable vectorization
- Loop distribution enables vectorization of multi-statement loops
- Cycle of flow-dependences in the dependence graph prevents vectorization
- Conditional statements in innermost loop prevents vectorization
- Array-of-Structures typically inhibits vectorization; converting to Structure-of-Arrays can enable vectorization