Vectorization in Fortran and C

Eric J. Kostelich

Arizona State University
SCHOOL OF MATHEMATICAL & STATISTICAL SCIENCES

Sept. 19, 2018

Outline of the lecture today

- Vectorization is form of data-parallel computation
- Vector constructs in MATLAB
- Vector constructs in Fortran

Review: The vectorization paradigm

- SIMD: Single Instruction Multiple Data
- To be data parallel, the results of a computation must not depend on the order in which the operands are processed
- Example: Vector assignment is data parallel:

```
y(:) = x(:)
```

Memory hierarchies for the Intel i7 Core processor

RAM (4,096+ MB)
latency: ~250 cycles

L3 cache (6 MB)
latency: 40 cycles

L2 cache
1/4 MB, 12 cycles

L1 cache
1/32 MB, 3 cycles

Example: Cache access considerations, $n \times 2$ arrays

$$\mathsf{y(:,1) = param.a - x(:,1).\hat{}2 + param.b{*}x(:,2);\ y(:,2) = x(:,1);}$$

- The working set is x(1:n,1), x(1:n,2), y(1:n,1), y(1:n,2)
- Caches operate in last-in first-out (LIFO) order
- The L2 cache holds 256 KB $\longrightarrow$ 64 KB per array slice, or a maximum of $\sim 8{,}000$ double-precision numbers each
- If $n > 8{,}000$ then the array slices spill from the L2 cache

Cache access considerations for multicore programming

- On a typical chip, all cores share the same L3 cache (6–12 MB)
- Usually this equals about 1.5 MB of L3 cache per core
- You get good performance in multithreading only if the working set fits in this space
- Otherwise, memory accesses go to main memory, where the stalls get longer as more cores request data
- Memory constraints limit the performance improvement that can be expected with multithreading

Additional comments on data structures

- Example: Computational geometry

Common textbook usage

```
type point
    real x, y, z
end type point
type(point) points(n)
```

Superior data structure:

```
real points(3,n)
```

- Rotation by $\mathbf{R}$: `points=matmul(R, points)`
- Matrix multiplication has highly optimized implementations

Vectorization in C/C++

- Can this loop be vectorized?

```
void sub(float *x, float *y, size_t n) {
  for(size_t j = 0; j < n; j++)
    y[j] = x[j];
}
```

Vectorization in C/C++

- Can this loop be vectorized?

```
void sub(float *x, float *y, size_t n) {
  for(size_t j = 0; j < n; j++)
     y[j] = x[j];
}
```

- In general, no
- The order of operations matters if `x` and `y` point to overlapping sections of memory

Vectorization in C99, 2

- Many other optimizations depend on knowing whether x and y overlap in memory
- The C99 keyword restrict asserts that the pointers aren't aliased

```
void sub(float * restrict x, float * restrict y, size_t n) {
  for(size_t j = 0; j < n; j++)
    y[j] = x[j];
}
```

- This loop vectorizes—but be sure that x and y don't point to overlapping regions of memory!
- C++ does not have restrict (an important incompatibility with C)

Vectorization using the C++ STL

```
void saxpy(double a, vector<double>& x, vector<double>&
y) {
    vector<double>::size_type n = x.size();
    for(vector<double>::size_type j = 0; j < n; j++)
        y[j] += a*x[j];
    }
```

- Be sure to pass the arguments by reference—otherwise C++ passes a copy and the routine has no effect
- Intel's `icpc` vectorizes
- GNU `g++` does not, even with `–O3 –ftree-vectorize`

Vectorization using the C++ STL, 2

```
void
saxpy(double a, valarray<double>& x, valarray<double>& y)
{
    y *= a*x;
}
```

- This code does not work with `vector`
- `valarray` defines helper classes to make such expressions efficient without creating temporaries
- Intel's `icpc` vectorizes this construct but GNU `g++` does not

Vectorization in Fortran

```
subroutine saxpy(a, x, y, n)
integer, intent(in):: n
real, intent(in):: a, x(n)
real, intent(inout):: y(n)

y = y + a*x
```

- Fortran rule: subroutine arguments must not refer to overlapping sections of memory
- Compilers often cannot diagnose violations of the rule
- But if you violate the rule, then you'll get what you deserve

MATLAB and Fortran syntax

- Suppose $\mathbf{A} \in \mathbb{R}^{m \times n}$, $\mathbf{x} \in \mathbb{R}^n$, and $\mathbf{y} = \mathbb{R}^m$
- MATLAB and Fortran: `A(:,k)` refers to the kth column of $\mathbf{A}$ and `A(k,:)` to the kth row
- MATLAB and Fortran: `A(:,k)` is much more efficient (stride 1 memory access)
- MATLAB: `y = A * x` yields $\mathbf{y} = \mathbf{A}\mathbf{x}$
- Fortran: `y = matmul(A,x)` yields $\mathbf{y} = \mathbf{A}\mathbf{x}$

MATLAB and Fortran syntax, 2

- Suppose $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ and $a \in \mathbb{R}$
- MATLAB and Fortran: `y = y + a*x` is a SAXPY
- Alternative syntax: `y(:) = y(:) + a*x(:)`
- Whether you write `x`, `x(:)`, `A`, or `A(:,:)` does not matter in MATLAB or in most Fortran usage

MATLAB and Fortran syntax, 3

- To add $\mathbf{x}$ to each column of the $n \times n$ matrix $\mathbf{A}$:
- MATLAB #1:

```
for k=1:n
    A(:,k) = A(:,k) + x;
end
```

- MATLAB #2: Add a $1 \times n$ tiling of $\mathbf{x}$ to $\mathbf{A}$:

```
A = A + repmat(x,1,n);
```

- Either alternative is faster than

```
for k=1:n; for j=1:n;
    A(j,k) = A(j,k) + x(j);
end; end
```

MATLAB and Fortran syntax, 4

- Although vectorized, this code is inefficient in both MATLAB and Fortran:

```
for k=1:n
    A(k,:) = A(k,:) + x(k);
end
```

- If n is greater than the cache line size, then successive memory accesses must wait for main memory
- The performance loss can be up to a factor of 80 for Intel Core i7 processors

MATLAB and Fortran syntax, 5

- MATLAB and Fortran syntax is identical for the elementwise operations $+$ and $-$
- $*$, $/$, and $**$ are elementwise in Fortran:

```
do j=1,n
    c(j) = a(j) * b(j)
enddo
```

Equivalent assignment:

`c = a * b`

- MATLAB requires `c = a .* b;` (and analogously `./` and `.^`)
- To solve $\mathbf{A}\mathbf{x} = \mathbf{b}$ in MATLAB: `x = A \ b;` (no Fortran equivalent)
- Remember: `A*x` is matrix multiplication in MATLAB

Comments on matmul

- `c = matmul(a,b)` works for matrix-vector and matrix-matrix multiplication
- Remember: $\mathbf{A}_{m\times n}\mathbf{B}_{n\times k} = \mathbf{C}_{m\times k}$
- The result of `matmul` is undefined if the dimensions of the operands aren't compatible
- Assuming compatible dimensions, $\mathbf{C} = \mathbf{A}^{\mathrm{T}}\mathbf{B}$ is `C=matmul(transpose(A),B)` in Fortran (and is `C = A' * B` in MATLAB)

Common matrix-vector functions with optimized implementations

- Dot product: MATLAB: `dp = dot(x,y)`; Fortran: `dp = dot_product(x,y)`
- Arithmetic mean: MATLAB and Fortran: `avg = sum(x)/n`
- Maximum value in MATLAB: 1-d vector: `big = max(x)`; 2-d vector: `big = max(max(x))`, etc
- Maximum value in Fortran: All ranks: `big = maxval(x)`
- Fortran: `z = max(x,y)` returns $\max(x_i, y_i)$ elementwise
- Analogously for minimum values

Vectorized loops in Fortran

- A big advance in compiler technology in the 1970's was the advent of vectorizing Fortran compilers
- Loop constructs like

```
do j=1,n
    y(j) = y(j) + a*x(j)
enddo
```

  are converted automatically to the appropriate sequence of vector instructions
- There is no performance difference with `y = y + a*x`, except that the latter is more concise

Requirements for vectorizability in Fortran

- Must be a counted `do` loop
- The result must not depend on the order of operations
- Any `if` statements must be simple (no `else if` clauses)
- No other branches within or out of the loop are permitted

Not vectorizable:

```
do j=1,n
    a(j) = a(j-1) + b(j)
enddo
```

Vectorizable:

```
do j=1,n
  if(x(j) > 0.0) x(j)=log(x(j))
enddo
```

Requirements for vectorizability, 2

- The result must not depend on the order of operations
- Consequence: The destination must not overlap with the source!

Not vectorizable:

```
do j=1,n
    x(j)=x(n-j+1)
enddo
```

Vectorizable, if `x` and `y` don't overlap:

```
do j=1,n
  y(j)=x(n-j+1)
enddo
y=x
```

Use array constructs with care

Legal Fortran construct:

```
x=x(n:1:-1)
```

Equivalent to:

```
do j=1,n
    y(j)=x(n-j+1)
enddo
y=x
```

- The array-valued assignment requires the compiler to generate a temporary array
- The assignment is as if the right-hand side were evaluated at once, then assigned to the left-hand side

Masked assignment in Fortran

- Given `real:: x(n)`, the following two constructs are equivalent:

Vectorizable loop:

```
do j=1,n
  if(x(j) > 0) x(j)=log(x(j))
enddo
```

Masked assignment:

```
where(x > 0) x=log(x)
```

- Masked assignment with alternative:

```
where(x > 0)
   x=log(x)
elsewhere
  x= -huge(x)
endwhere
```

most negative possible value

Masked assignment in Fortran, 2

- The where statement is a block construct:

```
where(x > 0)
    x=log(x)
    y=x*log(x)
endwhere
```

- and it can be nested:

```
where(x > 0)
    y=log(x)
    where(y > 0) z=x*log(y)
endwhere
```

Masked assignment in MATLAB

- The constructs apply to arrays of any rank
- Elegant: $x(x > 0) = 0$ is equivalent to $x = \min(x, 0)$
- But nesting is clunky:

Fortran:

```
where(x > 0)
      y=log(x)
      where(y > 0) z=x*log(y)
endwhere
```

MATLAB:

```
ix = find(x > 0);
y(ix) = log(x(ix));
iy = find(y(ix) > 0);
z(ix(iy)) = ...
   x(ix(iy))*log(y(ix(iy))
```