Remarks on vectorization and memory

Eric J. Kostelich

ARIZONA STATE UNIVERSITY
SCHOOL OF MATHEMATICAL & STATISTICAL SCIENCES

Sept. 17, 2018

Outline of the lecture today

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

The vectorization paradigm

- SIMD: Single Instruction Multiple Data
- Conceptually the simplest form of parallelism
- The elementwise addition $\mathsf{C} = \mathsf{A} + \mathsf{B}$ can be performed for all elements "at the same time"
- Vectorization is especially important for good performance in MATLAB
- Programming philosophy: Use language constructs to concentrate on what to calculate, instead of how to calculate it

Independent order of execution

- 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:

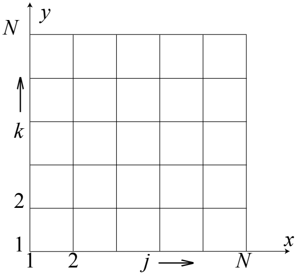
  $$\mathsf{y(:)} = \mathsf{x(:)}$$

- The result is the same whether we start from the beginning, the end, or somewhere in the middle of x

Filtering points for Algorithm B, the Hénon basin

- Lockout property: Once a point is sufficiently far from the origin, its orbit is destined for infinity
- For the parameters in the assignment, distance 100 suffices
- Any norm will do
- Example: in MATLAB and Fortran, if `x` is a scalar, vector, or array, `abs(x)` returns the elementwise absolute value
- `max(abs(x(:,1)), abs(x:,2))` is a vector of distances (L^1 norm)

Schematic outline of the algorithm: Naïve approach



for k=1:N

    for j=1:N

        for i=1:maxiter

            apply Algorithm B to (x_j, y_k)

Schematic outline of the algorithm: Vectorized approach

- The key to parallelization is to observe that Algorithm B may be applied independently to every grid point
- Thus it suffices to interchange the order of the loops:

```
for i=1:maxiter
    for k=1:N
        for j=1:N
            apply Algorithm B to all (x_j, y_k)
```

Arrays as data structures

- Suppose we want to apply the Hénon map to a set of n points
- One way to arrange the data is as a $2 \times n$ array:

$$\begin{pmatrix} x_1 & x_2 & \cdots & x_n \\ y_1 & y_2 & \cdots & y_n \end{pmatrix}$$

- Equivalently, we can define (in C)

```
struct point { double x, y; }
struct point list[n];
```

Arrays as data structures, 2

- Alternatively, we can arrange the data as an $n \times 2$ array:

$$\begin{pmatrix} x_1 & y_1 \\ x_2 & y_2 \\ \vdots & \vdots \\ x_n & y_n \end{pmatrix}$$

- The answer to "Which is better?" depends on how you plan to use and access the data

The main considerations

- Memory stride: How will you access the data in the innermost loop? Stride 1 is the most efficient by far
- Array contiguity: It is most efficient in MATLAB (and Fortran) to access arrays that form one contiguous segment of memory
- Example: If A is $2 \times n$, then `A(:,1:k)` is a contiguous slice (i.e. first k columns/points) but `A(1,:)` is not contiguous (stride is 2)
- Example: If A is $n \times 2$, then `A(:,1)` is contiguous but `A(1:k,:)` (first k points) is not contiguous (unless $k = n$)

Example: The Hénon map, 2

- Goal: Efficiently compute
$$\begin{pmatrix} x_{i+1} \\ y_{i+1} \end{pmatrix} = \begin{pmatrix} a - x_i^2 + b y_i \\ x_i \end{pmatrix}$$
for $i = 1, \ldots, n$ without writing an explicit loop
- Suppose we represent the input data as one array. Which is better: $2 \times n$ or $n \times 2$?

Example: The Hénon map, 2

- Goal: Efficiently compute

$$\begin{pmatrix} x_{i+1} \\ y_{i+1} \end{pmatrix} = \begin{pmatrix} a - x_i^2 + by_i \\ x_i \end{pmatrix}$$

  for $i = 1, \ldots, n$ without writing an explicit loop

- Suppose we represent the input data as one array. Which is better: $2 \times n$ or $n \times 2$?
- Perhaps $n \times 2$—but factors elsewhere in the program may dictate $2 \times n$

Example: The Hénon map, 3

- $(x_{n+1}, y_{n+1}) = (a - x_n^2 + by_n,\ x_n)$
- MATLAB code skeleton assuming an $n \times 2$ arrangement:

```
function y = henon(x, param)
    y = zeros(size(x));
    y(:,1) = param.a - x(:,1).^2 + ...
        param.b*x(:,2);    note .^ operator
    y(:,2) = x(:,1);
```

- Initialize with `param.a = 2.12; param.b = -0.3;`

The rest of the algorithm

- Step 1. Apply the Hénon map
- Step 2. Filter the points
- Step 3. Mark the basin for those that diverge
- Step 4. Repack the survivors into new contiguous arrays
- Step 5. Go to Step 1 (up to K times)
- Data structure: $n \times n$ starting grid

Arrays as data structures, 3

- Option #1: Store the grid as an $n^2 \times 2$ array
- Option #2: Store the grid as an $n \times n \times 2$ array
- Option #3: Process the grid by columns and store the points in each column as an $n \times 2$ array
- Which method is "best" depends on the machine and the size of n
- Note: If x is $n \times n \times 2$, then x(:,:,1) and x(:,k,1) are contiguous slices but x(:,k,1:2) is not

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

Cache access considerations

```
y(:,1) = param.a - x(:,1).^2 + param.b*x(:,2);
y(:,2) = x(:,1);
```

- Caches operate in last-in first-out (LIFO) order
- Suppose we store the grid as an $n^2 \times 2$ array
- If n is sufficiently large, then `x(:,1)` won't all fit into the cache
- By the time we finish computing `x(:,1).^2`, the first part of `x` has been spilled from the cache
- We may have to re-load all of `x(:,1)` in the second statement

Tradeoffs in vectorization

- Vector instructions maximize processor throughput
- Memory access is the limiting factor in performance
- Vectors that are too large will spill from the cache
- General approach: Try to do as much as possible on suitably small chunks of data
- $\frac{1}{32}$ Mb is 32 KB or $\sim$ 4,000 double-precision numbers

General programming strategies

- The working set is the data on which the innermost loops of the program are currently operating
- If the cache is small, then try to keep the working set small
- Complete as many operations as possible on the working set moving on to the next chunk of data
- Use stride-1 memory accesses whenever possible
- Break up long loops into multiple smaller loops that shrink the working set
- Minimize the number of temporary arrays

Additional comments on data structures

- The numerical method is the most important consideration
- The OO "is-a" and "has-a" paradigms are secondary
- Example: A spectral method requires Fourier transforms of arrays of data

Example: Global Forecast System (GFS) operational NWP model

- Main dynamical variables: pressure, potential temperature, divergence and vorticity of the wind field
- It is a spectral model—linear terms are computed in Fourier space

Example: Global Forecast System model, 2

- What (if anything) is wrong with the following data structure for an $M \times N \times H$ grid?

```
type grid_point
   real pressure, temperature, divergence
   real vorticity(3)
end type grid_point
type(grid_point) temperature(M,N,H)
```

Example: Global Forecast System model, 2

- What (if anything) is wrong with the following data structure for an $M \times N \times H$ grid?

```
type grid_point
   real pressure, temperature, divergence
   real vorticity(3)
end type grid_point
type(grid_point) temperature(M,N,H)
```

- Computing FFTs involves array strides that are not 1

Additional comments on data structures, 2

- If you want a separate datatype, consider instead (assuming fixed resolution):

```
integer, parameter:: M=192, N=94, H=28
type atmosphere
    real pressure(M,N,H)
    real temperature(M,N,H)
    real uwind(M,N,H)
    real vwind(M,N,H)
    real humidity(M,N,H)
end type atmosphere
```