Parallel & Distibuted Computing: Lecture 10

Alberto Paoluzzi

April 5, 2017

Code vectorization

1. Vectorization is code "Transposition"
2. In bounds checking
3. Reductions
4. Rules: (1) type-stable code
5. references

Vectorization is code “Transposition”

Start here

Fast Numeric Computation in Julia

- First, make it correct
- Devectorize expressions
- Merge computations into a single loop
- Write cache-friendly codes
- Avoid creating arrays in loops
- Identify opportunities to use BLAS
- Explore available packages

What Is Vectorization?

"Vectorization" has two different meanings in Julia, both related to operating on chunks of data:

- Writing style in terms of operations that operate on whole arrays.

For example, writing d=a+b-c where the variables all indicate array objects.

See the Julia (???) package for more information about this style of code.

- Compiler transformations that improve performance by using SIMD (Single Instruction Multiple Data) instructions

...

For example, hardware with Intel® Advanced Vector Extensions (Intel® AVX) can do eight 32-bit floating-point additions at once.

Stride

- Stride of an array (also referred to as increment, pitch or step size) is the number of locations in memory between beginnings of successive array elements, measured in bytes or in units of the size of the array's elements.
- The stride cannot be smaller than the element size but can be larger, indicating extra space between elements.
- An array with stride of exactly the same size as the size of each of its elements is contiguous in memory.
- Such arrays are sometimes said to have unit stride

Stride

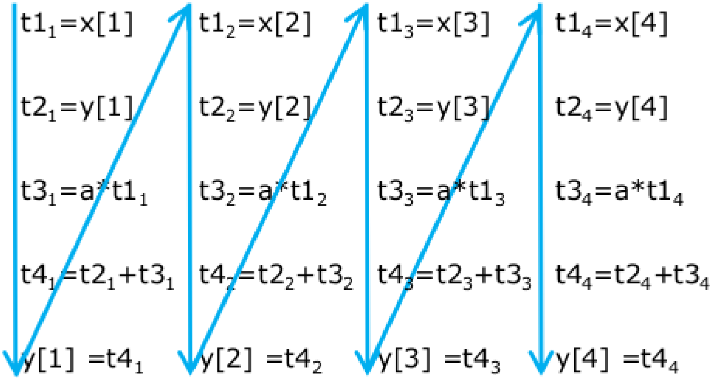

Figure 1: stride

Remark

non-unit stride arrays can be more efficient for 2D or multi-dimensional

Stride

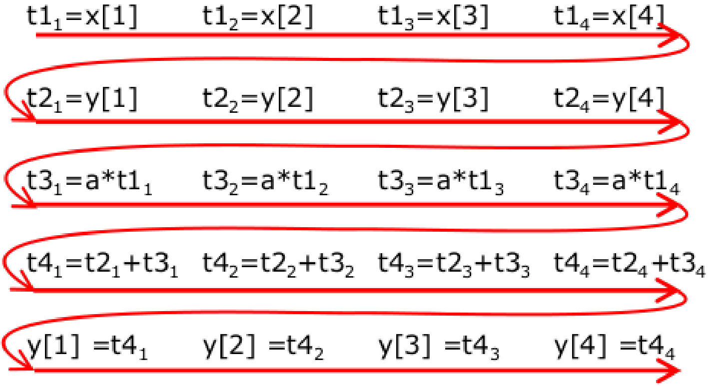

Figure 2: transposizion

The best stride depends on hardware . . .

BLAS level 1

Level 1[edit] This level consists of all the routines described in the original presentation of BLAS (1979), which defined only vector operations on strided arrays :

dot products, vector norms, a generalized vector addition of the form

$$\boldsymbol{y} \leftarrow \alpha \boldsymbol{x} + \boldsymbol{y}$$

(called “axpy”)

and several other operations.

Vectorization Examples

```
function axpy(a,x,y)
    @simd for i=1:length(x)
        @inbounds y[i] += a*x[i]
    end
end

n = 1003
x = rand(Float32,n)
y = rand(Float32,n)
axpy(1.414f0, x, y)
```

Examples

Examples

Examples

In bounds checking

Slide_1

Slide_2

The Loop Body Should Be Straight-Line Code

- The current vectorizer for Julia generally requires that the loop body not contain branches or function calls.
- Code with constructs that might throw exceptions also contains branches, and so will not vectorize.
- This is why the `@inbounds` notation is currently necessary.
- It turns off subscript checking that might throw an exception.

Reductions

Slide_1

Slide_2

Slide_3

Rules: (1) type-stable code

Type-stable definition

```julia
function sumofsins1(n::Integer)
    r = 0
    for i in 1:n
        r += sin(3.4)
    end
    return r
end

function sumofsins2(n::Integer)
    r = 0.0
    for i in 1:n
        r += sin(3.4)
    end
    return r
end
```

Definition

Code is said to be type-stable if the type of every variable does not vary over time

Times of stable vs non-stable code

Compare the timing of generated code:

```
julia> sumofsins1(100_000)
-25554.110202663698

julia> sumofsins2(100_000)
-25554.110202663698

julia> @time [sumofsins1(100_000) for i in 1:100];
  0.476308 seconds (30.11 M allocations: 462.815 MB, 6.94% gc

julia> @time [sumofsins2(100_000) for i in 1:100];
  0.143783 seconds (12.10 k allocations: 542.811 KB)
```

Complexity of code

Compare the complexity of generated code:

```
julia> code_llvm(sumofsins1, (Int, ))

julia> code_llvm(sumofsins2, (Int, ))
```

Subscripts Should Be Unit Stride

The amount that an array subscript changes between iterations is called its stride.

for a `@simd` loop with index `i`, you want array subscripts to either be:

```
loop_invariant_value
i
i + loop_invariant_value
i - loop_invariant_value
```

nested loops on two-dimensional arrays

use `@simd` on the inner loop and make that loop index the leftmost subscript of arrays.

```
function updateV(irange, jrange, U, Vx, Vy, A)
    for j in jrange
        @simd for i in irange
            @inbounds begin
                Vx[i,j] += (A[i,j+1]+A[i,j])*(U[i,j+1]-U[i,j])
                Vy[i,j] += (A[i+1,j]+A[i,j])*(U[i+1,j]-U[i,j])
            end
        end
    end
end

# Shows that code vectorizes for Float32
R = 1:8
A = Float32[]
@code_llvm updateV(R,R,A,A,A,A)
```

32-Bit Floating-Point Arithmetic Is Often Faster

However, if 32-bit precision suffices for your purpose and speed is your goal, let use 32-bit arithmetic

- Each SIMD instruction can process twice as many 32-bit operands as 64-bit operands.
- Loading/storing 32-bit operands requires half the memory bandwidth, which is often the limiting resource.

Summary Recommendations for Effective Vectorization in Julia

For implicit or explicit vectorization:

- No cross-iteration dependencies
- Straight-line loop body only. Use **ifelse** for conditionals.
- Use **@inbounds**
- Make sure that all calls are inlined. Write type-stable code.
- Use unit-stride subscripts
- Reduction variables should be local variables.
- 32-bit arithmetic is often faster

For *implicit* vectorization, the additional constraints are:

- Access no more than about 4 arrays inside the loop.
- Do not use floating-point reductions.

Otherwise, use *explicit* vectorization by marking your loop with **@simd**.

references

Sources of slides

- https://software.intel.com/en-us/articles/vectorization-in-julia
- Writing Type-Stable Code in Julia
- Bounds checking
- Basic Linear Algebra Subprograms for Fortran Usage