Julia Expression Templates for Vector Arithmetic

Tyler Olsen

Department of Mechanical Engineering
Massachusetts Institute of Technology

December 7, 2015

Outline

Outline

1. Introduction & Motivation

2. Implementation
   - High Level Overview
   - Julia: Parametric Types + Multiple Dispatch

3. Performance

4. Conclusions

Vectorization

Advantages:

- Compact, expressive code
- Less bug-prone than explicitly written loops
- *Significant* speedup in “slow” languages

Vectorization

Advantages:

- Compact, expressive code
- Less bug-prone than explicitly written loops
- *Significant* speedup in "slow" languages

Disadvantages:

- Straightforward implementation $\implies$ more memory requirements

Vectorization

Advantages:

- Compact, expressive code
- Less bug-prone than explicitly written loops
- *Significant* speedup in "slow" languages

Disadvantages:

- Straightforward implementation $\implies$ more memory requirements

Conclusion:

- Like to write vectorized code when it makes sense
- Need to be smart about implementation

Vectorization causes extra allocation!

Consider the following expression from a computer's point of view:

$$Res = \alpha * a + \beta * b + \gamma * c + \delta * d$$

Vectorization causes extra allocation!

Consider the following expression from a computer's point of view:

$$Res = \underbrace{\alpha * a}_{tmp_1} + \underbrace{\beta * b}_{tmp_2} + \underbrace{\gamma * c}_{tmp_3} + \underbrace{\delta * d}_{tmp_4}$$

Vectorization causes extra allocation!

Consider the following expression from a computer's point of view:

$$Res = \underbrace{\underbrace{\alpha * a}_{tmp_1} + \underbrace{\beta * b}_{tmp_2}}_{tmp_5} + \underbrace{\gamma * c}_{tmp_3} + \underbrace{\delta * d}_{tmp_4}$$

Vectorization causes extra allocation!

Consider the following expression from a computer's point of view:

$$Res = \underbrace{\underbrace{\underbrace{\alpha * a}_{tmp_1} + \underbrace{\beta * b}_{tmp_2}}_{tmp_5} + \underbrace{\gamma * c}_{tmp_3}}_{tmp_6} + \underbrace{\delta * d}_{tmp_4}$$

Vectorization causes extra allocation!

Consider the following expression from a computer's point of view:

$$Res = \underbrace{\underbrace{\underbrace{\underbrace{\alpha * a}_{tmp_1} + \underbrace{\beta * b}_{tmp_2}}_{tmp_5} + \underbrace{\gamma * c}_{tmp_3}}_{tmp_6} + \underbrace{\delta * d}_{tmp_4}}_{tmp_7}$$

Vectorization causes extra allocation!

Consider the following expression from a computer's point of view:

$$Res = \underbrace{\underbrace{\underbrace{\underbrace{\alpha * a}_{tmp_1} + \underbrace{\beta * b}_{tmp_2}}_{tmp_5} + \underbrace{\gamma * c}_{tmp_3}}_{tmp_6} + \underbrace{\delta * d}_{tmp_4}}_{tmp_7}$$

- Each $tmp_i \in \mathbb{R}^n$ requires a memory allocation
- Total memory allocation for this expression is 7x arrays
- If arrays are 1,000,000-element Float64 arrays, allocates 56 MB
- Theoretical memory allocation requirement is a single 8 MB array
- Memory allocation is *slow*, can degrade performance

What should happen

As humans, we can see that

$$Res = \alpha * a + \beta * b + \gamma * c + \delta * d$$

should be evaluated as

$$Res[i] = \alpha * a[i] + \beta * b[i] + \gamma * c[i] + \delta * d[i] \quad \text{for } i \in \{1..N\}$$

What should happen

As humans, we can see that

$$Res = \alpha * a + \beta * b + \gamma * c + \delta * d$$

should be evaluated as

$$Res[i] = \alpha * a[i] + \beta * b[i] + \gamma * c[i] + \delta * d[i] \quad \text{for } i \in \{1..N\}$$

- Need to teach computer to do this without manually writing loops

Idea: Expression Templates

- Main Idea: Exploit type system to represent result of vectorized operations ($+$, $-$, $.*$, $./$, ...)
    - VectorAddition
    - VectorDifference
    - VectorScaled
    - ...

Idea: Expression Templates

- Main Idea: Exploit type system to represent result of vectorized operations ($+$, $-$, $.*$, $./$, ...)
  - VectorAddition
  - VectorDifference
  - VectorScaled
  - ...
- The name for this is an "Expression Template"
- Originally developed for C++ linear algebra libraries to improve performance & readability
  - Used by Eigen and Boost.uBlas libraries, among others

Idea: Expression Templates

- Main Idea: Exploit type system to represent result of vectorized operations (+, −, .∗, ./, ...)
  - VectorAddition
  - VectorDifference
  - VectorScaled
  - ...
- The name for this is an “Expression Template”
- Originally developed for C++ linear algebra libraries to improve performance & readability
  - Used by Eigen and Boost.uBlas libraries, among others
- Implemented via parametric types and operator overloading in Julia

Outline

High-Level Implementation

- Base type to represent a vector expression (VectorExpression), from which all types (including Vector) derive
- Define types to represent vectorized operations *without* copying arguments
  - Addition
  - Subtraction
  - Scaling
  - Element-wise multiplication/division
- Define `operator`[] (`getindex`() in Julia) for each type
- Overload appropriate operators & functions to construct VectorExpression subtypes
- Create constructor for Vector type from generic VectorExpressions

Julia Implementation

Definition of base type and main Vector subtype:

```
#define abstract base type
abstract VectorExpression;

# Vector is subtype of base type
immutable ETVector{Float64} <: VectorExpression
    data::Array{Float64,1}
    len::Int64
end

# construct vector from VectorExpression
function ETVector(A::VectorExpression)
    len = A.len
    data = zeros(len)
    for i = 1:len
        data[i] = A[i]
    end
    return ETVector(data, len)
end

# define indexing function
@inline function getindex(A::ETVector, i::Int64)
    return A.data[i]
end
```

Julia Implementation

Example definition of parametric type representing addition of vectors

```
immutable ETVectorAddition{T1<:VectorExpression,
                           T2<:VectorExpression} <: VectorExpression
    lhs::T1
    rhs::T2
    len::Int64
end

# Inline everything!
@inline function getindex(A::ETVectorAddition, i::Integer)
    (A.lhs[i]+A.rhs[i])::Float64
end

@inline function +(lhs::VectorExpression, rhs::VectorExpression)
    return ETVectorAddition(lhs,rhs, lhs.len)
end
```

- Note references to arbitrary VectorExpressions lhs and rhs (not necessarily of same type)
- Note definition of `getindex`() for VectorAddition
- Note overload of "+" function returns VectorAddition type

Outline

Benchmarking

Expression to evaluate:

$$Res = \alpha * a + \beta * b + \gamma * c + \delta * d$$
$$\text{where} \quad a, b, c, d \in \mathbb{R}^N,$$
$$\alpha, \beta, \gamma, \delta \in \mathbb{R}$$

- Good candidate for expression templates due to many sub-expressions
- Will compare:
  - Native arrays
  - Expression Templates using C++-style constructor calls
  - Expression templates using custom `@et` macro
  - Hand-coded loop

What the code looks like

- Native Arrays: $\texttt{Res} = \alpha * \texttt{a} + \beta * \texttt{b} + \gamma * \texttt{c} + \delta * \texttt{d}$
- ET w/ Ctors:

$$\begin{aligned}\texttt{Res} = {} & \alpha * \texttt{ETVector}(\texttt{a}) + \beta * \texttt{ETVector}(\texttt{b}) \\ & + \gamma * \texttt{ETVector}(\texttt{c}) + \delta * \texttt{ETVector}(\texttt{d})\end{aligned}$$

- ET w/ macro: $\texttt{Res} = \texttt{@et}\ \alpha * \texttt{a} + \beta * \texttt{b} + \gamma * \texttt{c} + \delta * \texttt{d}$
- Hand-looped:

$$\begin{aligned}&\texttt{Res} = \texttt{zeros}(\texttt{N}) \\ &\texttt{for i} = 1 : \texttt{N} \\ &\quad \texttt{res}[\texttt{i}] = \alpha * \texttt{a}[\texttt{i}] + \beta * \texttt{b}[\texttt{i}] + \gamma * \texttt{c}[\texttt{i}] + \delta * \texttt{d}[\texttt{i}] \\ &\texttt{end}\end{aligned}$$

Benchmark Results: Timing

- Relative Timing Results (results are runtime/"native" runtime)
- (Lower is better!)

N	Native	ET w/ ctors	ET w/ macro	Hand-loop
10^3	1.0	0.36	0.36	0.41
10^4	1.0	0.31	0.31	0.42
10^5	1.0	0.32	0.32	0.33
10^6	1.0	0.33	0.33	0.33
10^7	1.0	0.32	0.32	0.32

Benchmark Results: Memory

Memory Allocation Results (results in kB):

N	Native	ET w/ ctors	ET w/ macro	Hand-loop
10^3	56.656	8.464	8.464	8.064
10^4	560.544	80.448	80.448	80.064
10^5	5600.544	800.448	800.448	800.064
10^6	56000.544	8000.448	8000.448	8000.064
10^7	560000.544	80000.448	80000.448	80000.064

Outline

Conclusions

- Expression templates yield roughly 60-70% speedup over native arrays!
- For the expression tested, expression templates use 1/7 the memory of native arrays, and less than only 400 B regardless of array size
- Expression templates either meet or beat the performance of hand-rolled loops in all cases

Future Work

- Extend to generic vector functions ($sin(a)$, $exp(a)$,...)
- Template based on container (vector, matrix, distributed array, ...)
- Make data type generic (specialized to Float64 for this prototype)