Accelerated Array Expressions in Julia using Expression Templates

Tyler Olsen
Department of Mechanical Engineering
tjolsen@mit.edu

December 13, 2015

1 Introduction

Note: all code used for this project is available publicly on github.

`https://github.com/tjolsen/ExpressionTemplates.jl`

1.1 Vectorization

Traditionally, high-level "slow" scripting languages like Matlab, R, and Python, require the programmer to write code in a vectorized form if there is to be any hope of achieving close to the performance of compiled languages such as C. When this happens, the high-level language invariably calls out to a pre-compiled library, usually written in C, to perform the desired operation on a large amount of data at once. In some cases, such as mathematical programming where it is natural to think in terms of vectors and matrices, this results in concise, expressive, and efficient code. The alternative to this approach is to explicitly hand-write loops as one would in C. In most scripting languages, this is a bad approach since the interpreted loops tend to be orders of magnitude slower than compiled library functions.

Julia is an interesting exception to this rule, however. The JIT compiler gives its loops C-like performance, so it is not immediately clear whether vectorized code will produce a speedup. As it turns out, the developers recommend *against* writing vectorized code if high performance is sought. The reason for this is that vectorized code generally, but not necessarily, introduces memory allocations for sub-expressions, which can be large if the arrays under consideration are large.

As a concrete example, consider the expression

$$R = \alpha A + \beta B + \gamma C + \delta D$$

where α,β, γ, δ are real scalars and A, B, C, D are length N vectors. As humans, we know how to compute this expression optimally. We would allocate an output array, then loop over the range $[1, N] \in \mathbb{Z}$ and compute $R[i] = \alpha A[i] + \beta B[i] + \gamma C[i] + \delta D[i]$. However, a parser is not able to take this big-picture view of the operation as a whole. Instead, it breaks the expression down into a tree of function calls that compute sub-expressions according to the order of operations. These sub-expressions each must be allocated, and this is the major source of inefficiency in native Julia array expression evaluation. For this example, a total of 7 arrays must be allocated, even though we know from our "human" implementation that only 1 is required. The temporaries are

$$\begin{aligned}
tmp_1 &= \alpha * A \\
tmp_2 &= \beta * B \\
tmp_3 &= \gamma * C \\
tmp_4 &= \delta * D \\
tmp_5 &= tmp_1 + tmp_2 \\
tmp_6 &= tmp_5 + tmp_3 \\
tmp_7 &= tmp_6 + tmp_4
\end{aligned}$$

The final temporary, tmp_7, is assigned by reference to the output R, so no additional copy is required here. Memory allocation of an array is slow compared to the speed of iterating over the same array, so the time to create these temporary objects actually dominates the total time to evaluate this expression This finding suggests that vectorization should not be used at all in Julia, since the "preallocate and iterate" approach seems to be a far superior method. Unfortunately, this puts us back to effectively writing C code rather than taking advantage of the fact that we are using a high-level language.

1.2 Expression Templates

This probem has been addressed in the past, most notably in C++ where classes and operator overloading make it easy to create Vector and Matrix classes that replicate the native behavior of vectorized Julia. In C++, it was motivated by a desire to add "syntactic sugar" to their classes, since prevailing wisdom suggests that code with a clean API and friendly syntax is easier to read, and therefore, maintain. Unfortunately, the memory allocation issue was encountered again (not just in the area of linear algebra data structures, but more broadly where functions return objects). This motivated a series of compiler optimizations (eg: copy elision and RVO) that helped with, but did not eliminate, the problem.

The final piece of the puzzle was the creation of an idiom called "Expression Templates," aptly named due to their heavy use of C++ template metaprogramming. The key idea is the use of the C++ type system to communicate information to the compiler what operations are being performed at *compile time*, rather than runtime. The way this is done is to create an abstract base type called a "VectorExpression," from which all types, including the Vector type itself, derives. Other types representing vector operations, such as addition, subtraction, scaling, etc., also derive from the base class. The abstract type acts as a public interface to the rest of the program, guaranteeing that all VectorExpression-derived objects will, at a minimum, be indexable via the "[ ]" operator, and it allows for the use of polymorphism. The last necessary part is a constructor to create a Vector object from an arbitrary VectorExpression. Once these requirements are met, you are able to write $R = \alpha A + \beta B + \gamma C + \delta D$ without incurring the performance hit of several memory allocations. Therein, you will see a few ideas that have not been mentioned here that are necessary to achieve the best efficiency possible.

2 Julia Implementation

2.1 Type Creation

The expression template system can be implemented in Julia as well. As before, it begins with the definition of an abstract base class VectorizedExpression (the slight nomenclature difference is because my Julia implementation allows for vectors, matrices, or arbitrary-order N-arrays without any modification). The VectorizedExpression type is parameterized on the array order and the datatype.

```
abstract VectorizedExpression{T,R}
```

Next, a set of operations that operate in an index-by-index manner are selected, and specialized types and function overloads are created. They can be broadly grouped into three categories: vector-vector operations, vector-scalar operations, and functions-acting-on-a-vector operations. For the vector-vector operations, I selected addition, subtraction, elementwise multiplication, and elementwise division. The specialized types and overloads are generated using Julia's metaprogramming utliities, since there would be a significant amount of duplicated code if written by hand. To demonstrate this concretely, the vector-vector operation code is included below. Vector-scalar operations and functions-on-vectors are implemented in much the same way, so they are omitted from this report for brevity.

```
vvops = [:+, :-, :.*, :./]
vvop_names = [:ET_vvplus, :ET_vvminus, :ET_vvdtimes, :ET_vvddivide]

macro gen_all_vvops()
    for i = 1:length(vvops)
```

```
        local op = vvops[i]
        local opname = vvop_names[i]
        eval(quote
             #define type, parameterized on LHS and RHS types,
             #   which must inherit from VectorizedExpression{T,R}
             immutable $opname{T,R, ET1<:VectorizedExpression,
             ET2<:VectorizedExpression} <: VectorizedExpression{T,R}
             lhs::ET1
             rhs::ET2
             end

             # Overload getindex operator. This is where result of
             # a vector operation is computed at index i...
             @inline function getindex(A::$opname, i...)
             return $op(A.lhs[i...], A.rhs[i...])
             end

             @inline function length(A::$opname)
                 return length(A.lhs)
             end

            @inline function size(A::$opname)
                return size(A.lhs)
            end

            # overload operator. Take lhs and rhs and return instance of type $opname
            @inline function ($op){T,R}(lhs::VectorizedExpression{T,R},
                                        rhs::VectorizedExpression{T,R})
                return $opname{T,R,typeof(lhs), typeof(rhs)}(lhs,rhs)
            end
            export $op, $opname
        end)
    end
end

# call macro to generate vv-op types
@gen_all_vvops
```

The last component of the Expression Template framework is the definition of the "ETContainer" type. This acts as a practically non-existant wrapper around Julia's native Array type, but we are able to define constructors of this type that allow us to control exactly when expression templates are finally evaluated. Its definition is

```
immutable ETContainer{T,R} <: VectorizedExpression{T,R}
    data::Array{T,R}
end

function ETContainer{T,R}(A::VectorizedExpression{T,R})
    L = length(A)
    tmp = Array{T,R}(size(A)...)
    for i = 1:L
        @inbounds tmp[i] = A[i]
    end
```

```
    return ETContainer(tmp)
end
```

The constructor shown is functionally identical to the one used in the real implementation, but the implementation has some manual loop unrolling to let the JIT go to work on some instruction-level parallelism more easily.

2.2 @et Macro

The type system described above is able to achieve the desired result of delaying array expression evaluation until a constructor call to ETContainer, but it is undeniably tedious to write out. The "idiomatic Julia" way to address this is to devise some sort of macro that makes everything automagically work. For this reason, we again turn to the metaprogramming utilities in Julia to do some of the work for us.

I developed the "@et" macro to act as a on/off switch for expression tepmlates. Using this, the user need not ever know that the custom types or the ETContainer type even exist The basic idea of the @et macro is to traverse the parse tree of an expression, wrap all of the arrays in a call to ETContainer (which will then cause the multiple-dispatch system of Julia to call appropriate +, -, ... functions), wrap the entire expression in a final call to ETContainer (this triggers evaluation of the whole expression), and return the "data" field of the ETContainer type.

This approach is one glaring flaw, though: the types of variables have not yet been determined when macros are evaluated. Thus, there is no way to know which parts of the parsed expression to wrap in a call to ETContainer. The solution to this problem is to use generated functions. Julia's generated functions are special functions that are executed only once for a given input type, and they must return an expression that can then be evaluated. Thus, I define a generated function called `__symbol_wrapper`. This generated function has two methods, shown below.

```
@generated function __symbol_wrapper{T,R}(x::Array{T,R})
    return :(ETContainer{T,R}(x))
end

@generated function __symbol_wrapper(x)
    return :x
end
```

As you can see, any time the function is called with an array as its argument, it is wrapped in the ETContainer type. For anything else, the argument itself is returned.

Given that this mechanism is in place, the @et macro can now simply traverse the parse tree of a given expression, wrap *every* symbol with the `__symbol_wrapper` function, and know that everything will work out correctly once the types of variables have been deduced. This extra step is not necessary in C++ since the compiler has all of the type information up front, rather than getting it at runtime, as is the case in Julia. The full @et macro and associated helper functions are shown below.

```
macro et(expr)
    if (typeof(expr) == Expr)
        if (expr.head == :call)
            local e = handle_expr(expr)
            return :(ETContainer($e).data)
        elseif (expr.head == :(=))
            return esc(:($(expr.args[1]) = ETContainer($(handle_expr(expr.args[2]))).data ))
        end
    end
end

# private recursive function to convert arrays into ETContainers
```

```
# and let Julia + multiple dispatch run from there
function handle_expr(expr)

    if (typeof(expr) == Symbol)
        return :(__symbol_wrapper($expr))

    elseif (typeof(expr) == Expr)

        for i = 2:length(expr.args)
            expr.args[i] = handle_expr(expr.args[i])
        end
        return expr

    else
        return expr
    end
end

@generated function __symbol_wrapper{T,R}(x::Array{T,R})
    return :(ETContainer{T,R}(x))
end

@generated function __symbol_wrapper(x)
    return :x
end
```

3 Performance Results

The Expression Template framework was tested against the native Julia array implementation and simple hand-rolled loops in order to determine the best way to evaluate a vectorized expression. The expression evaluated was the one shown above

$$R = \alpha A + \beta B + \gamma C + \delta D$$

where the array lengths varied from 10^1 to $10^{7.25}$, where the exponent was increased in increments of 0.0625. At each array length the expression was evaluated 1000 times with each evaluation method, and the average of these execution times was recorded. The primary metric of interest here is the speedup over the native array implementation. The definition of this is $S = T_{native}/T$, where T_{native} is the execution time of the native array implementation, and T is the execution time for the method in question. The speedup factor at different array lengths N is plotted in figure 1. Note that the x-axis of the plot is on a logarithmic scale.

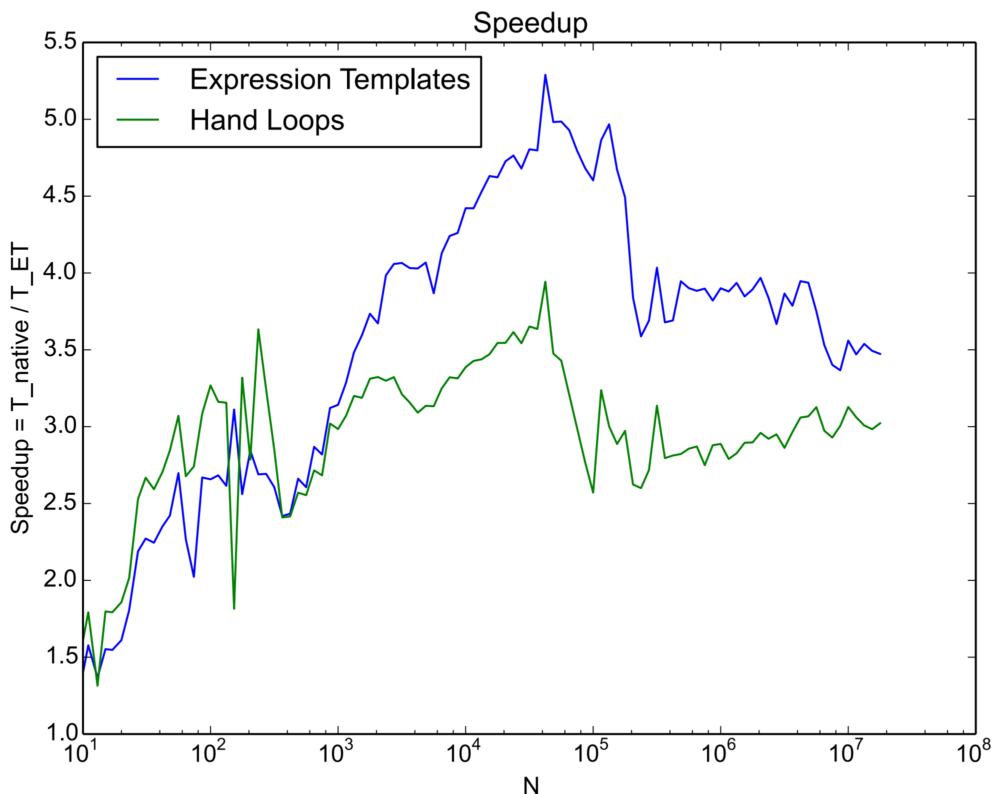

Figure 1: Plot of speedup over native arrays vs vector size on a semi-log scale

It can be seen from the plot that both the expression templates and the hand-written loops enjoy a significant speed advantage over the native arrays. Up to array lengths of about 1000, they have nearly identical speedups. Past that length, though, the expression templates actually enjoy a significant speed advantage over eventhe hand-rolled loops! This was a surprising result, and can most likely be attributed to the more highly-optimized constructor of the ETContainer object. In the constructor, I unrolled the main loop by a factor of 8, allowing the compiler to pipeline more instructions at once (and checking the loop condition less frequently). This causes the expression template method to hit its peak value of approximately 5.25, while the hand-loops hit its peak speedup value of 4. For very large arrays, ie $N > 300000$, the speedups level off between 3.5 and 4 for the expression templates, and between 2.75 and 3.25 for the hand-loops.

The final metric that can and should be considered is the memory allocation reuquirements for each method. There were no surprises here, with the native array method allocating memory equal to 7 times the storage requirement of the output array. The hand-loops only allocated a single output array, which is entirely unsurprising, given that it was manually allocated. The nice result from the experiment is that the Expression Template method also only allocated a single output array, plus approximately 350 bytes of data to hold data type information. I count this as a resounding success, since the 350 bytes of overhead is entirely negligible when the array sizes are in the tens of MB range.

4 Conclusions

The expression template framework presented in this report was shown to be an effective method for delaying the evaluation of vectorized expressions. It not only entirely eliminates the extraneous memory allocation that plagues the naive approach, but in doing so, it confers a significant speedup to the overall process across the whole range of reasonable array sizes. Equally important as the performance improvement, the method

requires almost zero effort on the part of the programmer. Due to Julia's macro system, the entire framework is brought to bear with the addition of three characters (@et) prepended to a line containing an expression.

4.1 Future Work

The framework is not quite ready for publishing as a public Julia package. In particular, I need to write tests to ensure correctness and efficiency for arbitrary N-array order, arbitrary data type, and to test the vector function types. In addition, I would like to set up additional benchmarks, since I knew ahead of time that the expression that I tested here was a good candidate for demonstrating the efficacy of the method. In addition, the @et macro needs to be tested for robustness. Currently, it will probably fail if the expression it acts on does not contain any arrays. While this might not seem like a bad thing, it assumes more user competence than I am comfortable with. In addition, special cases need to be included in order to do reasonable things for operations that don't make sense for expression templates (matmul, sum, max, to name a few). This shouldn't be difficult, but it will require careful design so that the system doesn't end up as 99% special cases.

In the meantime, I invite you to clone/fork the git repository given above. Any feedback from you or others would be greatly appreciated.