9g. Static Vectors for Small Collections

Martin Alfaro
PhD in Economics

INTRODUCTION

The creation of vectors can rapidly become a performance bottleneck, due to the memory-allocation overhead it entails. The issue has far-reaching implications, as vector creation doesn't just occur when we explicitly define a variable holding a new vector. It also happens internally in various scenarios, such as when referencing a slice like `x[1:2]` or computing intermediate results on the fly, like `x .* y` in `sum(x .* y)`.

This section introduces a strategy to address this limitation, while retaining the convenience of vectors for collections. The solution leverages the so-called **static vectors**, provided by the `StaticArrays` package. Unlike built-in vectors, which are allocated on the heap, static vectors are stack-allocated.

Under the hood, static vectors are built on top of tuples. This feature determines that **static vectors are only suitable for collections comprising a few elements**. As a rule of thumb, **you should use static vectors for collections with up to 75 elements**. Exceeding this threshold can lead to increased overhead during creation and access, potentially offsetting any performance benefits or even resulting in a fatal error. [1]

Static vectors offer additional benefits relative to tuples. Firstly, they maintain their performance benefits even at sizes where tuples typically degrade. Secondly, they're manipulated like regular vectors, making them more convenient to work with.

The `StaticArrays` package also supports other array types, including matrices, and provides mutable variants. The latter makes static vectors more flexible than tuples, which are only available in an immutable form. It's worth indicating though that, while the mutable version provides performance benefits relative to regular vectors, the immutable option still offers the best performance.

> **Warning!**
> To avoid repetition, **the entire section assumes all collections are small**. Taking this into account, all the benefits highlighted are contingent upon this assumption. We also suppose that the `StaticArrays` package is already available in the workspace, so the command `using StaticArrays` is omitted from each code snippet.

CREATING STATIC VECTORS

The package `StaticArrays` includes several variants of static arrays. Our focus in particular we'll be on the type `SVector`, whose objects will be referred to as SVectors for simplicity.

There are two approaches to creating an SVector, each serving a distinct purpose: the first one creates an SVector through literal values, while the other option converts a standard vector into an SVector. The implementation of each approach is illustrated below.

```
# all 'sx' define the same static vector '[3,4,5]'


sx = SVector(3,4,5)
sx = SVector{3, Int64}(3,4,5)
sx = SA[3,4,5]
sx = @SVector [i for i in 3:5]
```

```
julia> sx
3-element SVector{3, Int64} with indices SOneTo(3):
 3
 4
 5
```

```
# all 'sx' define a static vector with same elements as 'x'
x = collect(1:10)

sx = SVector(x...)
sx = SVector{length(x), eltype(x)}(x)
sx = SA[x...]
sx = @SVector [a for a in x]
```

```
julia> sx
10-element SVector{10, Int64} with indices SOneTo(10):
  1
  2
  3
  ⋮
  9
 10
```

Of these approaches, we'll primarily rely on the function `SVector`, occasionally employing `SA` for indexing purposes. [2] Note the use of the splatting operator `...` to turn a regular vector into an SVector. This operator is necessary for the function `SVector` to transform a collection into a sequence of arguments. [3]

Regarding slices of SVectors, the approach used for their creation could result in either a regular vector or an SVector. The resulting object depends on the indexing employed: a slice remains an SVector when indices are provided as SVectors, whereas the slice becomes a regular vector when indices are ranges or regular vectors. The sole exception to this rule is when the slice references the whole object (i.e., `sx[:]`), in which case an SVector is returned.

```
x = collect(3:10) ; sx = SVector(x...)

# both define static vectors
slice1 = sx[:]
slice2 = sx[SA[1,2]]        # or slice2 = sx[SVector(1,2)]
```

```
julia> slice1
8-element SVector{8, Int64} with indices SOneTo(8):
  3
  4
  ⋮
  9
 10
julia> slice2
2-element SVector{2, Int64} with indices SOneTo(2):
 3
 4
```

```
x = collect(3:10) ; sx = SVector(x...)

# both define and ordinary vector
slice2 = sx[1:2]
slice2 = sx[[1,2]]
```

```
julia> slice
2-element Vector{Int64}:
 3
 4
```

SVECTORS DON'T ALLOCATE MEMORY AND ARE FASTER

One of the key advantages of SVectors is that they're internally implemented on top of tuples. Consequently, SVectors don't allocate memory.

```
x = rand(10)

function foo(x)
    a = x[1:2]              # 1 allocation (copy of slice)
    b = [3,4]               # 1 allocation (vector creation)

    sum(a) * sum(b)         # 0 allocation (scalars don't allocate)
end
```

```
julia> @btime foo($x)
  29.819 ns (2 allocations: 160 bytes)
```

```
x = rand(10)

@views function foo(x)
    a = x[1:2]              # 0 allocation (view of slice)
    b = [3,4]               # 1 allocation (vector creation)

    sum(a) * sum(b)         # 0 allocation (scalars don't allocate)
end
```

```
julia> @btime foo($x)
  15.015 ns (1 allocation: 80 bytes)
```

```
x = rand(10);   tup = Tuple(x)

function foo(x)
    a = x[1:2]              # 0 allocation (slice of tuple)
    b = (3,4)               # 0 allocation (tuple creation)

    sum(a) * sum(b)         # 0 allocation (scalars don't allocate)
end
```

```
julia> @btime foo($tup)
  1.400 ns (0 allocations: 0 bytes)
```

```
x = rand(10);   sx = SA[x...]

function foo(x)
    a = x[SA[1,2]]          # 0 allocation (slice of static array)
    b = SA[3,4]             # 0 allocation (static array creation)

    sum(a) * sum(b)         # 0 allocation (scalars don't allocate)
end
```

```
julia> @btime foo($sx)
  1.600 ns (0 allocations: 0 bytes)
```

The decrease in memory allocations from SVectors is especially relevant for operations that result in temporary vectors, such as broadcasting.

```
x = rand(10)

foo(x) = sum(2 .* x)
```

```
julia> @btime foo($x)
  17.936 ns (1 allocation: 144 bytes)
```

```
x = rand(10);   sx = SVector(x...)

foo(x) = sum(2 .* x)
```

```
julia> @btime foo($sx)
  1.800 ns (0 allocations: 0 bytes)
```

Interestingly, the performance benefits of SVectors extend beyond memory allocation. This means that, even when operations on regular vectors don't allocate memory, SVectors could still deliver significant speedups. Below, we demonstrate this through the function `sum(f, <vector>)`, which sums the elements of `<vector>` after they're transformed via `f`. The example shows that the implementation with SVectors yields faster execution times, despite regular vectors already not incurring memory allocations.

```
x  = rand(10)

foo(x) = sum(a -> 10 + 2a +  3a^2, x)
```

```
julia> @btime foo($x)
  4.400 ns (0 allocations: 0 bytes)
```

```
x  = rand(10);  sx = SVector(x...);

foo(x) = sum(a -> 10 + 2a +  3a^2, x)
```

```
julia> @btime foo($sx)
  2.900 ns (0 allocations: 0 bytes)
```

SVECTOR TYPE AND ITS MUTABLE VARIANT

Similar to tuples, **SVectors are immutable**, meaning that their elements can't be added, removed, or modified. To accommodate mutable collections, the package `StaticArrays` additionally provides a variant given by the type `MVector`. The creation of MVectors and their slices follow the same syntax as SVectors, where the function `SVector` is replaced with `MVector`. This is illustrated below.

```
x  = [1,2,3]
sx = SVector(x...)

sx[1] = 0
```

```
ERROR: setindex!(::SVector{3, Int64}, value, ::Int) is not defined
```

```
x  = [1,2,3]
mx = MVector(x...)

mx[1] = 0
```

```
julia> mx
3-element MVector{3, Int64} with indices SOneTo(3):
 0
 2
 3
```

The mutability of MVectors makes them ideal for initializing a vector that will eventually be filled via a for-loop. In fact, executing `similar(sx)` when `sx` is an SVector automatically returns an MVector.

▼ 'similar' for SVectors

```
sx = SVector(1,2,3)

mx = similar(sx)        # it defines an MVector with undef elements
```

```
3-element MVector{3, Int64} with indices SOneTo(3):
        2073873450800
 -1152921504606846976
                    0
```

TYPE STABILITY: SIZE IS PART OF THE STATIC VECTORS' TYPE

SVectors are formally defined as objects with type `SVector{N,T}`, where `N` specifies the number of elements and `T` denotes the element's type. For instance, `SVector(4,5,6)` has type `SVector{3,Int64}`, indicating that it comprises 3 elements with type `Int64`. Importantly, this implies that **the number of elements is part of the `SVector` type**. This feature, which is shared with MVectors and inherited from tuples, can readily introduce type instabilities if not handled carefully.

To ensure type stability, you can employ approaches similar to those employed for tuples. Essentially, this entails that we should either pass SVectors and MVectors as function arguments, or dispatch by the number of elements through the `Val` technique.

```
x = rand(50)

function foo(x)
    output = SVector{length(x), eltype(x)}(undef)
    output = MVector{length(x), eltype(x)}(undef)

    for i in eachindex(x)
        temp      = x .> x[i]
        output[i] = sum(temp)
    end

    return output
end

@code_warntype foo(x)                       # type unstable
```

```
x = rand(50);   sx = SVector(x...)

function foo(x)

    output = MVector{length(x), eltype(x)}(undef)

    for i in eachindex(x)
        temp      = x .> x[i]
        output[i] = sum(temp)
    end

    return output
end

@code_warntype foo(sx)                      # type stable
```

```
x = rand(50)

function foo(x, ::Val{N}) where N
    sx     = SVector{N, eltype(x)}(x)
    output = MVector{N, eltype(x)}(undef)

    for i in eachindex(x)
        temp      = x .> x[i]
        output[i] = sum(temp)
    end

    return output
end

@code_warntype foo(x, Val(length(x)))      # type stable
```

PERFORMANCE COMPARISON

MVectors offer performance benefits over regular vectors. However, bear in mind that they're never more performant than SVectors. In fact, certain operations on MVectors may still trigger memory allocations. For this reason, the general guideline is to prefer SVectors when the collection doesn't need to be mutated, restricting MVectors when in-place mutation is necessary.

Below, we compare the performance of SVectors and MVectors. The examples demonstrate that they may exhibit similar performance, although SVectors are more performant in certain cases. Likewise, SVectors and MVectors consistently outperform regular vectors for small collections.

```
x  = rand(10)
sx = SVector(x...);  mx = MVector(x...)

foo(x) = sum(a -> 10 + 2a +  3a^2, x)
```

```
julia> @btime foo($x)
  4.400 ns (0 allocations: 0 bytes)
julia> @btime foo($sx)
  2.800 ns (0 allocations: 0 bytes)
julia> @btime foo($mx)
  2.900 ns (0 allocations: 0 bytes)
```

```
x  = rand(10)
sx = SVector(x...);  mx = MVector(x...)

foo(x) = 10 + 2x +  3x^2
```

```
julia> @btime foo.($x)
  19.739 ns (1 allocation: 144 bytes)
julia> @btime foo.($sx)
  1.600 ns (0 allocations: 0 bytes)
julia> @btime foo.($mx)
  6.600 ns (1 allocation: 96 bytes)
```

STATIC VECTORS VS PRE-ALLOCATIONS

Considering the advantages of static vectors over regular vectors, let's compare their performance to other strategies that reduce memory allocations. In particular, we'll examine how they stack up against pre-allocating memory for intermediate outputs. Our examples demonstrate that static vectors can efficiently store intermediate results, making pre-allocation techniques unnecessary. Moreover, the examples reveal that storing the final output in an MVector can lead to performance gains over using a regular vector.

For the illustration, consider a for-loop that requires an intermediate result during each iteration `i`. This involves counting the number of elements in `x` that are greater than `x[i]`, which can be formally implemented as `sum(x .> x[i])`. To make the comparison stark, we'll isolate the computation of the intermediate step `x .> x[i]`. Note that every implementation below requires

pre-allocating the vector `output`, leaving us with only one decision to make: whether to pre-allocate the temporary vector `temp`. This also explains why all the implementations below involve at least one memory allocation.

```
x = rand(50)

function foo(x; output = similar(x))
    for i in eachindex(x)
        temp      = x .> x[i]
        output[i] = sum(temp)
    end

    return output
end
```

```
julia> @btime foo($x)
  3.188 μs (101 allocations: 5.17 KiB)
```

```
x = rand(50)

function foo(x; output = similar(x), temp = similar(x))
    for i in eachindex(x)
        @. temp      = x > x[i]
           output[i] = sum(temp)
    end

    return output
end
```

```
julia> @btime foo($x)
  695.745 ns (2 allocations: 992 bytes)
```

```
x = rand(50);   sx = SVector(x...)

function foo(x; output = Vector{Float64}(undef, length(x)))
    for i in eachindex(x)
        temp      = x .> x[i]
        output[i] = sum(temp)
    end

    return output
end
```

```
julia> @btime foo($sx)
  183.661 ns (1 allocation: 496 bytes)
```

```
x = rand(50);   sx = SVector(x...)

function foo(x; output = similar(x))
    for i in eachindex(x)
        temp      = x .> x[i]
        output[i] = sum(temp)
    end

    return output
end
```

```
julia> @btime foo($sx)
  148.817 ns (1 allocation: 448 bytes)
```

```
x = rand(50);   sx = SVector(x...)

function foo(x; output = MVector{length(x),eltype(x)}(undef))
    for i in eachindex(x)
        temp      = x .> x[i]
        output[i] = sum(temp)
    end

    return output
end
```

```
julia> @btime foo($sx)
  148.975 ns (1 allocation: 448 bytes)
```

The "No-Preallocation" tab serves as our baseline, providing a reference point for comparison with the other methods. As for the "Pre-allocating" tab, it reuses a regular vector to compute `temp`. In contrast, the "SVector" tab converts `x` to an SVector `sx` without pre-allocating `temp`. The benchmarks reveal that the latter approach is more performant, as it avoids memory allocation and benefits from additional optimizations provided by SVectors.

As for the last two tabs, they continue to define `x` as an SVector, but additionally treat `output` as an MVector. The last tab in particular does this by using `similar(sx)` to initialize `output`, whereas the other tab explicitly specifies an MVector. Comparing these cases, we observe that MVectors deliver additional performance gains.

FOOTNOTES

1. The recommended number of elements provided is actually lower than the documentation's suggested (100 elements). The reason for this discrepancy is that, as you approach the upper limit, the performance benefits of static vectors compared to regular vectors decrease significantly. As a result, the time spent benchmarking with collections of 100 elements will likely offset any potential advantage.

2. The approach based on the macro `@SVector` requires some caveats. For instance, it doesn't support definitions based on local variables. Because of this, it precludes the use of `eachindex(x)` in an array comprehension, unless `x` is a global variable.

3. For instance, `foo(x...)` is equivalent to `foo(x[1], x[2], x[3])` given a vector or tuple `x` with 3 elements.