Speeding up Quantum Mechanics - Matrix Types

Carsten Bauer

August 5, 2019

0.1 Free Fermions on a Chain

$$\mathcal{H} = -t \sum_{\langle i,j \rangle} c_i^\dagger c_j + \mu \sum_i n_i$$

Here, t is the hopping amplitude, μ is the chemical potential, and $c, c^\dagger$ are creation and annihilation operators. For the sake of the argument of this tutorial, we'll consider **open boundary conditions**. Since the fermions are *not* interacting, we can work in the *single particle basis* and do not have to worry about how to construct a basis for the many-body Fock space. We use the canonical cartesian basis in which one uses 0s to indicate empty sites and a 1 for the particle's site, i.e. $|00100\rangle$ represents the basis state which has the particle exclusively on the 3rd site. If you aren't familiar with second quantization just think of $\mathcal{H}$ as any quantum mechanical operator that can be represented as a matrix.

Let's build the Hamiltonian matrix.

```
using LinearAlgebra # makes Julia speak linear algebra fluently

N = 100 # number of sites
t = 1
μ = -0.5

H = diagm(0 => fill(μ, N), 1 => fill(-t, N-1), -1 => fill(-t, N-1))
```

```
100×100 Array{Float64,2}:
 -0.5  -1.0   0.0   0.0   0.0   0.0  …   0.0   0.0   0.0   0.0   0.0   0.0
 -1.0  -0.5  -1.0   0.0   0.0   0.0      0.0   0.0   0.0   0.0   0.0   0.0
  0.0  -1.0  -0.5  -1.0   0.0   0.0      0.0   0.0   0.0   0.0   0.0   0.0
  0.0   0.0  -1.0  -0.5  -1.0   0.0      0.0   0.0   0.0   0.0   0.0   0.0
  0.0   0.0   0.0  -1.0  -0.5  -1.0      0.0   0.0   0.0   0.0   0.0   0.0
  0.0   0.0   0.0   0.0  -1.0  -0.5  …   0.0   0.0   0.0   0.0   0.0   0.0
  0.0   0.0   0.0   0.0   0.0  -1.0      0.0   0.0   0.0   0.0   0.0   0.0
  0.0   0.0   0.0   0.0   0.0   0.0      0.0   0.0   0.0   0.0   0.0   0.0
  0.0   0.0   0.0   0.0   0.0   0.0      0.0   0.0   0.0   0.0   0.0   0.0
  0.0   0.0   0.0   0.0   0.0   0.0      0.0   0.0   0.0   0.0   0.0   0.0
  ⋮                             ⋮    ⋱         ⋮
  0.0   0.0   0.0   0.0   0.0   0.0      0.0   0.0   0.0   0.0   0.0   0.0
  0.0   0.0   0.0   0.0   0.0   0.0      0.0   0.0   0.0   0.0   0.0   0.0
  0.0   0.0   0.0   0.0   0.0   0.0     -1.0   0.0   0.0   0.0   0.0   0.0
  0.0   0.0   0.0   0.0   0.0   0.0     -0.5  -1.0   0.0   0.0   0.0   0.0
  0.0   0.0   0.0   0.0   0.0   0.0  …  -1.0  -0.5  -1.0   0.0   0.0   0.0
  0.0   0.0   0.0   0.0   0.0   0.0      0.0  -1.0  -0.5  -1.0   0.0   0.0
  0.0   0.0   0.0   0.0   0.0   0.0      0.0   0.0  -1.0  -0.5  -1.0   0.0
  0.0   0.0   0.0   0.0   0.0   0.0      0.0   0.0   0.0  -1.0  -0.5  -1.0
  0.0   0.0   0.0   0.0   0.0   0.0      0.0   0.0   0.0   0.0  -1.0  -0.5
```

A typical thing one does in quantum mechanics is calculate the expectation value of an operator, like $\mathcal{H}$, with respect to some quantum mechanical state ψ.

```
ψ = normalize(rand(N)); # some state

"""ev(O, ψ)Calculates the quantum mechanical expectation value `<ψ|O|ψ>`, where `O` is an operator
    and `ψ` is a state."""
ev(O, ψ) = ψ'*O*ψ;

ev(H, ψ) # calculate the energy
```

```
-2.0373841572859384
```

Let's see how long this calculation takes.

```
using BenchmarkTools
@btime ev($H, $ψ);
```

```
8.000 μs (1 allocation: 896 bytes)
```

0.1.1 Utilizing the matrix structure

Since `typeof(H) == Matrix{Float64}`, we have so far dispatched to generic matrix multiplications (and transposition) in the `ev` function. What we should do is exploit the *structure* of our Hamiltonian and use specialized, faster implementations of these operations. After all, most entries in $\mathcal{H}$ are equal to zero and multiplications involving those zeros are clearly unnecessary. To speed up our expectation value computation, we can tell Julia about the *sparsity* of our Hamiltonian. The way we do this, is by changing `H`'s type.

```
using SparseArrays
Hsparse = sparse(H)
```

```
100×100 SparseArrays.SparseMatrixCSC{Float64,Int64} with 298 stored entries
:
  [1  ,   1]  =  -0.5
  [2  ,   1]  =  -1.0
  [1  ,   2]  =  -1.0
  [2  ,   2]  =  -0.5
  [3  ,   2]  =  -1.0
  [2  ,   3]  =  -1.0
  [3  ,   3]  =  -0.5
  [4  ,   3]  =  -1.0
  [3  ,   4]  =  -1.0
  ⋮
  [98 ,  97]  =  -1.0
  [97 ,  98]  =  -1.0
  [98 ,  98]  =  -0.5
  [99 ,  98]  =  -1.0
  [98 ,  99]  =  -1.0
  [99 ,  99]  =  -0.5
  [100,  99]  =  -1.0
  [99 , 100]  =  -1.0
  [100, 100]  =  -0.5
```

```
typeof(Hsparse)
```

```
SparseArrays.SparseMatrixCSC{Float64,Int64}
```

As we can see, `Hsparse` is now a sparse matrix. Let's see how long our computation takes after this simple type change.

```
@btime ev($Hsparse, $ψ);
```

```
366.507 ns (2 allocations: 912 bytes)
```

That's a solid **30x speedup**! Utilizing matrix structure apparently pays off a lot!

Inspecting our Hamiltonian more closely, we note that it is not just sparse but actually *tridiagonal*.

Interlude: Visualizing matrix structure A nice visualization of the sparsity pattern of a matrix can be obtained by calling the function `spy` of the UnicodePlots.jl package. (Note that the `canvas=DotCanvas` option is necessary due to some rendering issue in Jupyter.)

```
using UnicodePlots
spy(H, canvas=DotCanvas) # sparsity pattern plot
```

```
Sparsity Pattern

     1 ::.                          > 0
        '::.                        < 0
          '::.
            '::.
              '::.
                '::.
                  '::.
                    '::.
                      '::.
                        '::.
                          '::.
                            '::.
                              '::.
                                '::.
   100                            '::

       1                          100
                 nz = 298
```

Let's exploit this fact and benchmark the expectation value computation once again.

```
Htri = Tridiagonal(H)
```

```
100×100 LinearAlgebra.Tridiagonal{Float64,Array{Float64,1}}:
 -0.5  -1.0    ⋅     ⋅     ⋅     ⋅   …   ⋅     ⋅     ⋅     ⋅     ⋅     ⋅
 -1.0  -0.5  -1.0    ⋅     ⋅     ⋅       ⋅     ⋅     ⋅     ⋅     ⋅     ⋅
   ⋅   -1.0  -0.5  -1.0    ⋅     ⋅       ⋅     ⋅     ⋅     ⋅     ⋅     ⋅
   ⋅     ⋅   -1.0  -0.5  -1.0    ⋅       ⋅     ⋅     ⋅     ⋅     ⋅     ⋅
   ⋅     ⋅     ⋅   -1.0  -0.5  -1.0      ⋅     ⋅     ⋅     ⋅     ⋅     ⋅
   ⋅     ⋅     ⋅     ⋅   -1.0  -0.5  …   ⋅     ⋅     ⋅     ⋅     ⋅     ⋅
   ⋅     ⋅     ⋅     ⋅     ⋅   -1.0      ⋅     ⋅     ⋅     ⋅     ⋅     ⋅
   ⋅     ⋅     ⋅     ⋅     ⋅     ⋅       ⋅     ⋅     ⋅     ⋅     ⋅     ⋅
   ⋅     ⋅     ⋅     ⋅     ⋅     ⋅       ⋅     ⋅     ⋅     ⋅     ⋅     ⋅
   ⋅     ⋅     ⋅     ⋅     ⋅     ⋅       ⋅     ⋅     ⋅     ⋅     ⋅     ⋅
  ⋮                             ⋮    ⋱           ⋮
   ⋅     ⋅     ⋅     ⋅     ⋅     ⋅       ⋅     ⋅     ⋅     ⋅     ⋅     ⋅
   ⋅     ⋅     ⋅     ⋅     ⋅     ⋅       ⋅     ⋅     ⋅     ⋅     ⋅     ⋅
   ⋅     ⋅     ⋅     ⋅     ⋅     ⋅     -1.0    ⋅     ⋅     ⋅     ⋅     ⋅
   ⋅     ⋅     ⋅     ⋅     ⋅     ⋅     -0.5  -1.0    ⋅     ⋅     ⋅     ⋅
   ⋅     ⋅     ⋅     ⋅     ⋅     ⋅   …  -1.0  -0.5  -1.0    ⋅     ⋅     ⋅
   ⋅     ⋅     ⋅     ⋅     ⋅     ⋅       ⋅   -1.0  -0.5  -1.0    ⋅     ⋅
   ⋅     ⋅     ⋅     ⋅     ⋅     ⋅       ⋅     ⋅   -1.0  -0.5  -1.0    ⋅
   ⋅     ⋅     ⋅     ⋅     ⋅     ⋅       ⋅     ⋅     ⋅   -1.0  -0.5  -1.0
   ⋅     ⋅     ⋅     ⋅     ⋅     ⋅       ⋅     ⋅     ⋅     ⋅   -1.0  -0.5
```

```
@btime ev($Htri, $ψ);
```

```
191.770 ns (4 allocations: 976 bytes)
```

That's almost **another 2x speedup**!

What we learn from this is that we can tell Julia about the structure of our Hamiltonian through types. Instead of using generic (and slow) matrix operations, we then dispatch to fast special implementations that have been optimized for this particular structure.

On a final note, choosing the best type (and therewith an algorithm) can be tricky and one has to play around a bit. The good thing is that it's very easy to try out different types! Note that apart from Julia's built-in matrix types there are also great custom types available in the ecosystem. See JuliaMatrices for more information.

1 Core message of this tutorial

- **Indicate the structure of a matrix**, like hermiticity or sparsity, through types. Fallback to generic types only if you run into method errors.

This notebook was generated using Literate.jl.