$$\nabla \cdot \mathbf{D} = \rho$$

$$\nabla \cdot \mathbf{B} = 0$$

$$\nabla \times \mathbf{E} = -\frac{\partial \mathbf{B}}{\partial t}$$

$$\nabla \times \mathbf{H} = \mathbf{J} + \frac{\partial \mathbf{D}}{\partial t}$$

GPU Programming in Computational Electromagnetics

Yunlong Lian
@algorithmx

Outline

- Computational electromagnetics algorithms
  - Finite-Difference Time-Domain (FDTD)
  - Rigorous Coupled-Wave Analysis (RCWA)
  - Finite-Element Method (FEM)
- Optimization of package **RigorousCoupledWaveAnalysis.jl**

Computational electromagnetics algorithms

Computational electromagnetics algorithms

- Classification
    - Space
        - *differential* equations
        - *integral* equations
    - Time
        - time domain discretization
        - frequency domain

Computational electromagnetics algorithms

- Numerical solution of the Maxwell *differential* equations
  - Time domain
    - Uniform spatial discretization
      - Finite-Difference Time-Domain (FDTD)
    - Adaptive spatial discretization
      - Time-Domain Finite-Element Method (TDFEM)

Computational electromagnetics algorithms

- Numerical solution of the Maxwell *differential* equations
  - Frequency domain
    - Static (zero frequency)
      - Finite-Element Method (FEM)
    - Time-harmonic (single frequency)
      - Rigorous Coupled-Wave Analysis (RCWA)

Computational electromagnetics algorithms

- Numerical solution of the Maxwell *integral* equations
  - Boundary-Element Method (BEM)
  - Method of Moments (MoM)
  - ...

Finite-Element Method (FEM)

- Noticeable open-source projects
  - Netgen/NGSolve : https://ngsolve.org/
  - FEniCSx : https://fenicsproject.org/
  - libMesh : http://libmesh.github.io/
  - FreeFEM : https://freefem.org/
- Many commercial softwares
  - COMSOL
  - ANSYS

Finite-Element Method (FEM)

- Basic idea: approximate field in continium by values on finite elements (via Galerkin method)
    - Field → vector
    - Differential operator → linear operator
    - Boundary condition → constraints
- Special considerations for electromagnetic field and Maxwell equations
    - Vector field

Finite-Element Method (FEM)

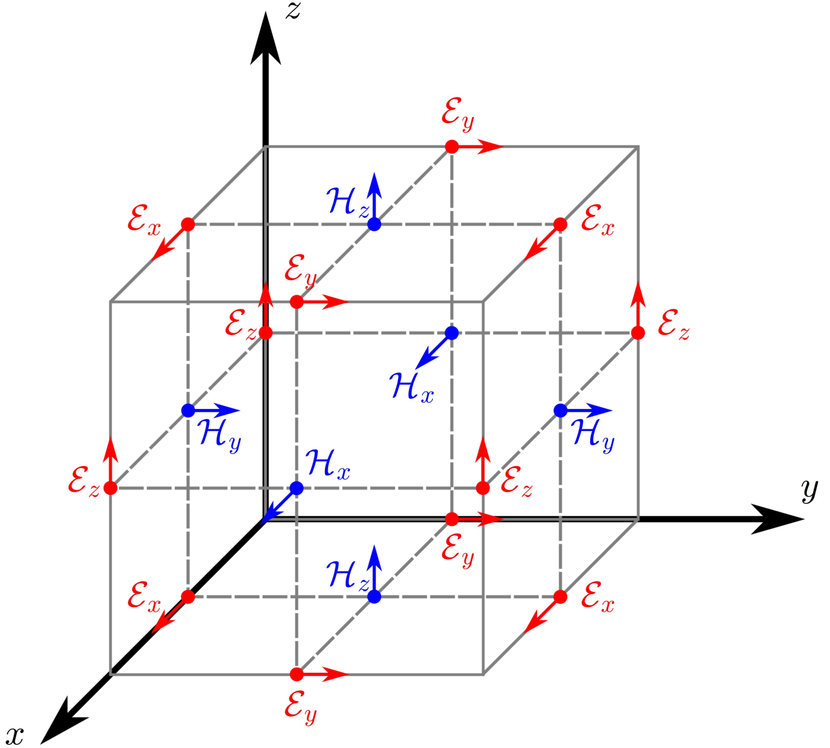
- Technical challenges and potential GPU accelerations
  - Mesh generation
  - Mesh quality control
  - Sparse solver

Finite-Difference Time-Domain (FDTD)

- Noticeable open-source projects
  - gprMax : https://github.com/gprMax/gprMax
  - mumax3 : http://mumax.github.io/
  - Python 3D FDTD Simulator : https://github.com/flaport/fdtd
  - Meep : https://github.com/NanoComp/meep
  - Tidy3D (commercial, FlexCompute Inc.) : https://github.com/flexcompute/tidy3d

Finite-Difference Time-Domain (FDTD)

- Straightforward discretization of time dependent Maxwell equation
- Yee lattice



http://opticaltweezers.org/chapter-6-computational-methods/667-2/

Finite-Difference Time-Domain (FDTD)

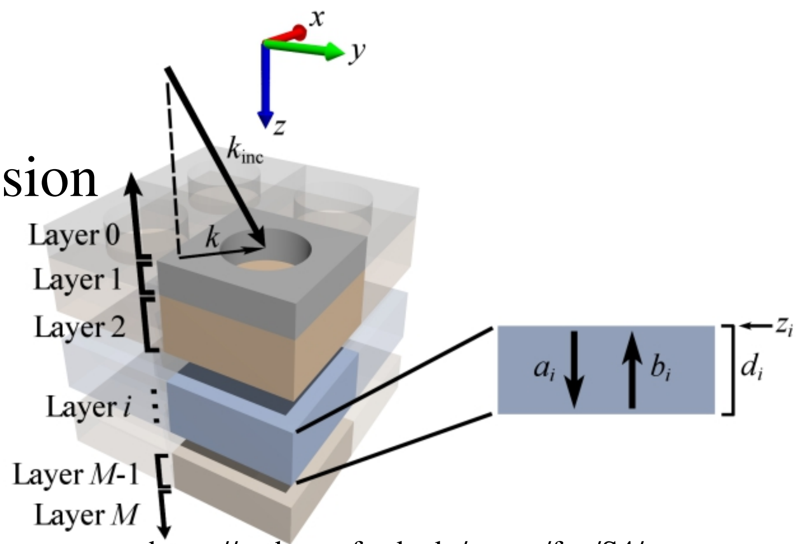
- Technical challenges and potential GPU accelerations
  - Memory cost cannot be reduced
  - Memory access patterns in updating the Yee lattice
  - Grid communications for parallelized algorithm

Rigorous Coupled-Wave Analysis (RCWA)

- Noticeable open-source projects
    - S4, or Stanford Stratified Structure Solver : https://github.com/victorliu/S4
    - GRCWA : https://github.com/weiliangjinca/grcwa
    - EMPossible course : https://github.com/zhaonat/Rigorous-Coupled-Wave-Analysis
    - Jordan Edmunds : https://github.com/edmundsj/RCWA
    - RigorousCoupledWaveAnalysis.jl : https://github.com/jonschlipf/RigorousCoupledWaveAnalysis.jl

Rigorous Coupled-Wave Analysis (RCWA)

- Enhanced Transmission Matrix algorithm by Moharam
  - major steps
    - partition the film into layers
    - perform layerwise 2D Fourier transform
    - calculate the wave amplitudes
      - along the propagation, for transmission
      - backwards, for reflection
    - collect results



https://web.stanford.edu/group/fan/S4/

Rigorous Coupled-Wave Analysis (RCWA)

- Technical challenges and potential GPU accelerations
  - Accuracy vs efficiency: N = FT orders, O(N^6) complexity, eigen solver O(M^3) matrix dimension O(N^2) by O(N^2)
  - Conformity in description of shapes
  - Multiple films on curved surface

Experiment: RCWA

Experiment: RCWA

- Motivation
  - Wide range of applications
    - Nanophotonics (academic)
    - Optical Critical Dimension (OCD) for process control in semiconductor manufacturing (industrial)
  - RCWA has to be fast
- **Use Julia and CUDA.jl**
- Starting point
  - https://github.com/jonschlipf/RigorousCoupledWaveAnalysis.jl
  - Rigorous coupled-wave analysis of a multi-layered plasmonic integrated refractive index sensor, Opt. Express 29, 36201-36210 (2021)
- Project GitHub repository

Why coding with Julia and CUDA.jl?

- Features of Julia programming language
  - garbage collection
  - array 1-based
  - support multiple dispatch and meta-programming
  - convinient profiling (@profview + VSCode)
- Features of CUDA.jl
  - using Julia syntax and grammar to **write, compile and run** CUDA kernels
  - easy memory management
  - **no loss of performance**
  - requires experience in both Julia and CUDA C programming (to understand error messages)

Why coding with Julia and CUDA.jl?

- Julia code for convolution kernel (zero padding, same size)

```julia
using CUDA | ✓

function convolve(
    source, destination,
    filter, filter_norm::Int64,
    dim1_s::Int64, dim2_s::Int64,
    dim1_f::Int64, dim2_f::Int64
    )
    # Julia array is 1-based
    i = (blockIdx().x-1) * blockDim().x + threadIdx().x
    j = (blockIdx().y-1) * blockDim().y + threadIdx().y
    if i <= dim1_s && j <= dim2_s
        x0 = dim1_f ÷ 2 + 1
        y0 = dim2_f ÷ 2 + 1
        p0 = max(1+x0-i,1)
        p1 = min(dim1_s+x0-i,dim1_f)
        q0 = max(1+y0-j,1)
        q1 = min(dim2_s+y0-j,dim2_f)
        s  = 0
        for p = p0:p1
            for q = q0:q1
                @inbounds s += source[i+p-x0,j+q-y0] * filter[p,q]
            end
        end
        destination[i,j] = (s ÷ filter_norm)
    end
    return
end
```

```julia
gaussian_filter(n,σ) = [floor(255*exp(-(i^2+j^2)/(σ^2))) for i=-(n÷2):(n÷2),j=-(n÷2):(n÷2)]

dims  = (8000,4000) | (8000, 4000)
nGF   = 31 | 31
GF    = CuArray( gaussian_filter(nGF, 8.0) .|> Int64 ) | 31×31 CuArray{Int64, 2, CUDA.Mem.De
sumGF = sum(Array(GF)) | 50195

src = rand(0:255, dims) .|> Int64 | 8000×4000 Matrix{Int64}:
dst = zeros(Int64, dims) | 8000×4000 Matrix{Int64}:
cu_src = CuArray(src) | 8000×4000 CuArray{Int64, 2, CUDA.Mem.DeviceBuffer}:
cu_dst = CuArray(dst) | 8000×4000 CuArray{Int64, 2, CUDA.Mem.DeviceBuffer}:

t = 32 | 32
@time @cuda threads=(t,t) blocks=(dims[1]÷t+1,dims[2]÷t+1) convolve(
    cu_src, cu_dst, GF, sumGF, dims[1], dims[2], nGF, nGF) | CUDA.HostKernel{typeof(convolve
```

OUTPUT PROBLEMS DEBUG CONSOLE TERMINAL JUPYTER

```
 0.000066 seconds (45 allocations: 2.234 KiB)
CUDA.HostKernel{typeof(convolve), Tuple{CuDeviceMatrix{Int64, 1}, CuDeviceMatrix{Int64, 1}, CuDevic
Int64, Int64, Int64}}(convolve, CuFunction(Ptr{Nothing} @0x0000000006462890, CuModule(Ptr{Nothing}
```

Why coding with Julia and CUDA.jl?

- Julia code to call cuBLAS / cuSolver routines
  - Example 1
    - CPU code : **A * Y**
    - GPU code : **cuA * cuY |> Array**
  - Example 2
    - CPU code : **A \ b**
    - GPU code : **CuArray(A) \ CuArray(b) |> Array**

RigorousCoupledWaveAnalysis.jl

- Enhanced Transmission Matrix (ETM) algorithm by Moharam [Moharam1995]
  - Backward and forward iteration
  - Numerical stability (key contribution of the [Moharam1995] paper)
- RigorousCoupledWaveAnalysis.jl by Jon Schlipf
  - GitHub repo: https://github.com/jonschlipf/RigorousCoupledWaveAnalysis.jl
  - Code is in good quality
  - Bottlenecks (identified with Julia profiling macro @profview)
    - #1: eigen(M) with **non-Hermitian M**
      - ==> cuSolver cannot help :-([need improvement on the algorithm level]
    - #2: A \ b via Lapack (solution of the linear equation A.x = b)
      - ==> **Array(CuArray(A) \ CuArray(b)) via cuSolver** [verified, 10x~100x speed up]

RigorousCoupledWaveAnalysis.jl

- Enable GPU acceleration with CUDA.jl
  - Install CUDA.jl into the package project : https://docs.julialang.org/en/v1/stdlib/Pkg/
  - Minimal change leads to at least 10x performance gain for bottleneck #2

```julia
# 2. [forward iteration]
# compute the reflected wave and the forward wave in the first layer
# bottleneck
ψref,ψm1 = slicehalf( -cat([I;sup.V],F(em[1])*[em[1].X*(a[1]/b[1])*em[1].X;I],dims=2) \ ([I;-sup.V]*ψin) )
```

```julia
# 2. [forward iteration]
# compute the reflected wave and the forward wave in the first layer
# CUDA version : Array(CuArray(A) \ CuArray(b))
ψref,ψm1 = slicehalf(
    Array(CuArray(-cat([I;sup.V],F(em[1])*[em[1].X*(a[1]/b[1])*em[1].X;I],dims=2)) \ CuArray(([I;-sup.V]*ψin)))
)
```

```
etm_reftra_fast():
  8.315675 seconds (334 allocations: 394.104 MiB, 0.19% gc time)
  2.548487 seconds (30.57 k allocations: 1.263 GiB, 1.41% gc time)
---

etm_reftra():
  8.519581 seconds (345 allocations: 394.104 MiB, 0.33% gc time)
 26.735786 seconds (30.38 k allocations: 1.879 GiB, 0.90% gc time)
---

julia>
```

master* ⊗ 0 ⚠ 0 make clean make build make run make all rm tests/*.o Pyth

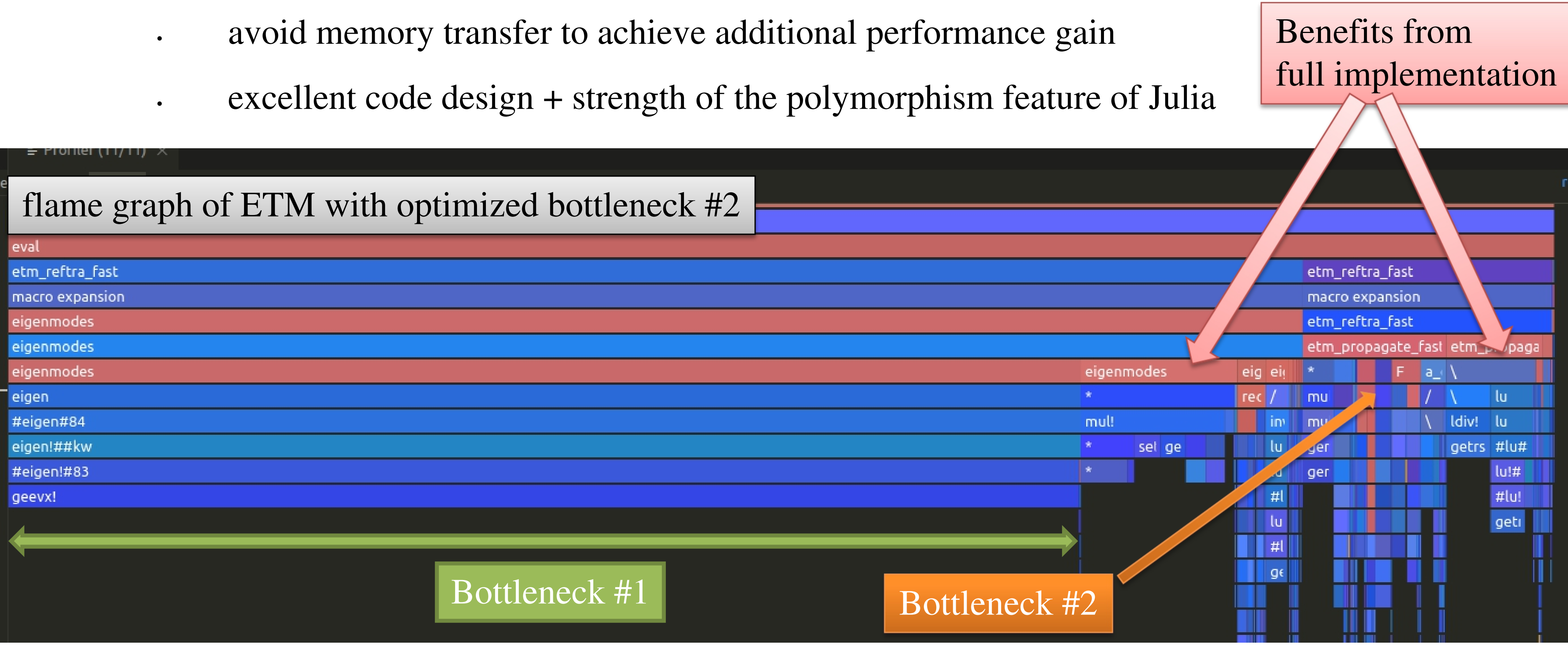
```
etm_reftra_fast():
  1.641657 seconds (323 allocations: 85.186 MiB)
  0.396935 seconds (14.41 k allocations: 277.851 MiB, 9.70% gc time)
---

etm_reftra():
  1.541210 seconds (323 allocations: 85.186 MiB)
  1.585677 seconds (14.24 k allocations: 414.954 MiB, 0.40% gc time)
---

julia>
```

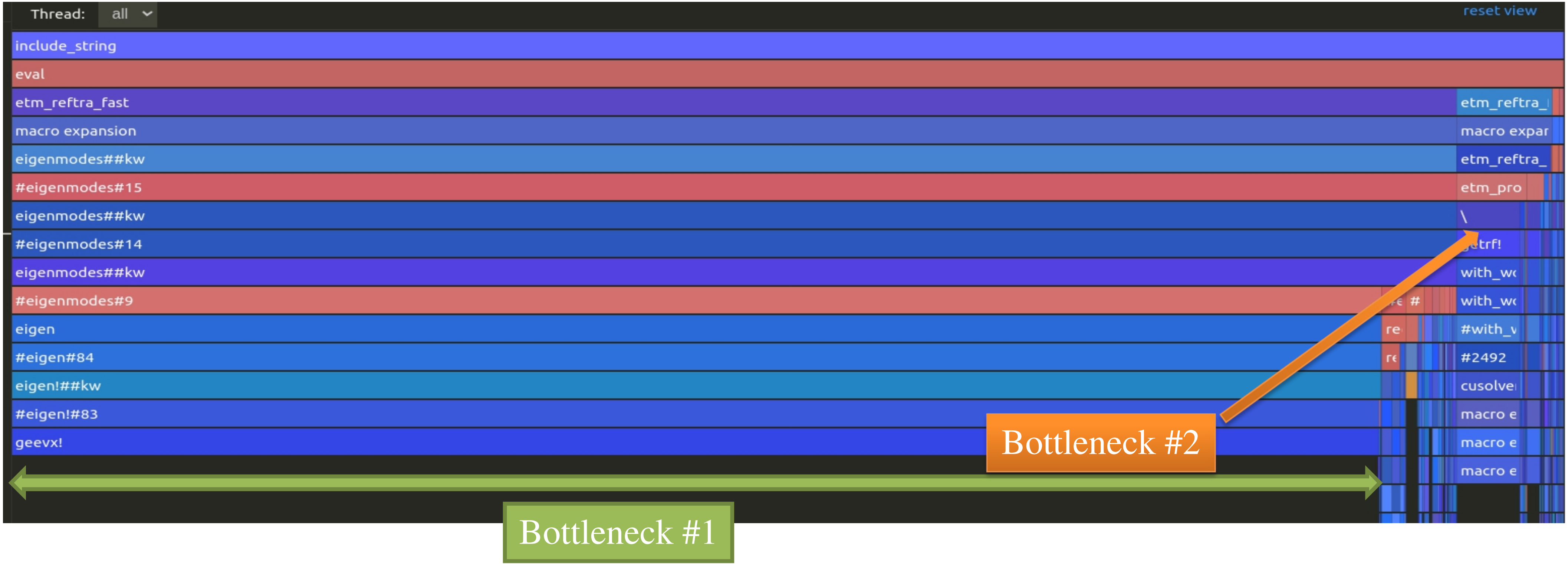
RigorousCoupledWaveAnalysisCUDA.jl

- Full implementation of the Enhanced Transmission Matrix algorithm with CUDA.jl
  - avoid memory transfer to achieve additional performance gain
  - excellent code design + strength of the polymorphism feature of Julia



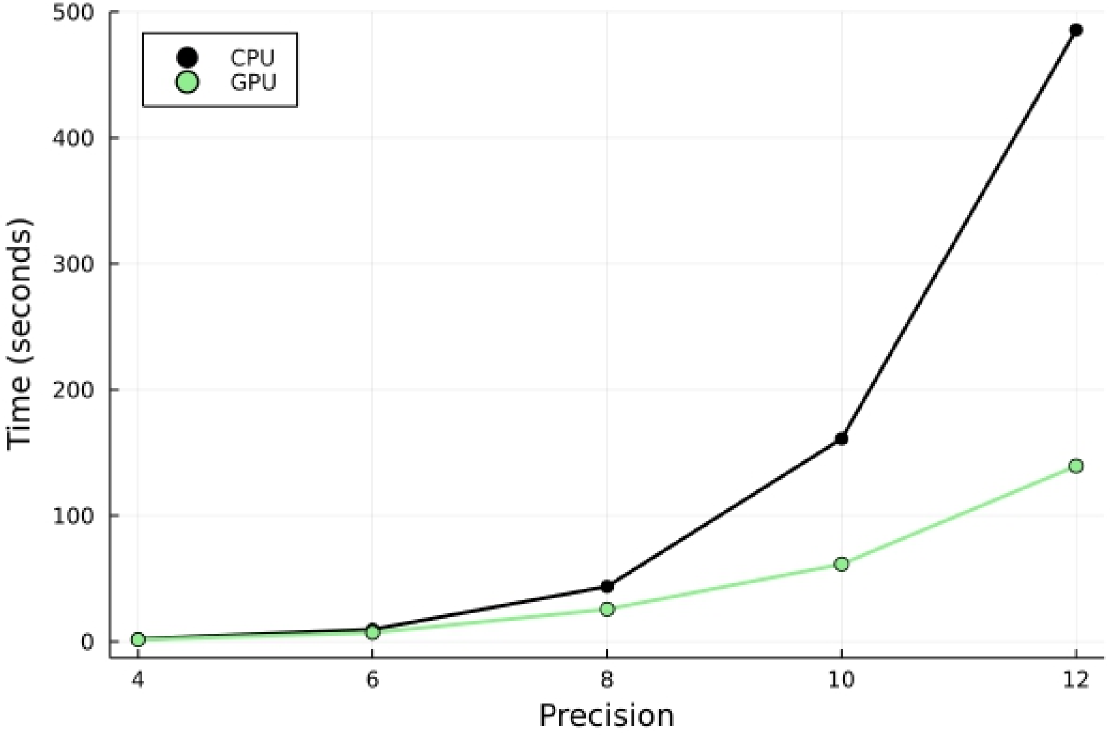
RigorousCoupledWaveAnalysisCUDA.jl

- Full implementation of the Enhanced Transmission Matrix algorithm with CUDA.jl
  - Result of optimization: Bottleneck#1 now takes 85% execution time (60% before)



Test

- Test for correctness: against CPU code
  - integrated in the package test/runtests.jl
- Overall performance gain:
- *nvprof



References / Useful materials

FEM

FDTD

RCWA

Summary

- Common algorithms in computational electromagnetics often faces unique technical challenges
- Paralellization with GPU can improve, but has limitations such as memory size and inter-node communication
- Parallelized RCWA algorithm can easily be implemented in Julia programming language, with the help of CUDA.jl package
- The performance gain is significant despite that only one of the two bottlenecks is resolved.