Julia in action – An HPC application from the field of particle physics

Anton Reinhard[1,2], Simeon Ehrig[1,2], Dr. Michael Bussmann[1, 2], Uwe Hernandez Acosta[1,2]

[1] Helmholtz-Zentrum Dresden-Rossendorf, [2] Center for Advanced Systems Understanding

09.01.2025

whoami

- Diploma in computer science from TU Dresden in 2024 with Uwe as supervisor

whoami

- Diploma in computer science from TU Dresden in 2024 with Uwe as supervisor
- Previous background in C++, currently mostly Julia

whoami

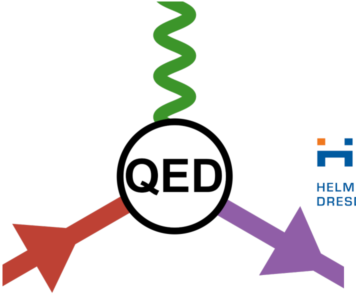
- Diploma in computer science from TU Dresden in 2024 with Uwe as supervisor
- Previous background in C++, currently mostly Julia
- Scientist at CASUS/HZDR

HZDR
HELMHOLTZ ZENTRUM DRESDEN ROSSENDORF

CASUS
CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

whoami

- Diploma in computer science from TU Dresden in 2024 with Uwe as supervisor
- Previous background in C++, currently mostly Julia
- Scientist at CASUS/HZDR
- Currently working on QED in an HPC context in Julia



HZDR
HELMHOLTZ ZENTRUM DRESDEN ROSSENDORF

CASUS
CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

CASUS and Computational Radiation Physics

Center for Advanced Systems Understanding

- Focus on modelling, simulating, visualizing laser-matter interaction

CASUS and Computational Radiation Physics

Center for Advanced Systems Understanding

- Focus on modelling, simulating, visualizing laser-matter interaction
- Open-source project PIConGPU

CASUS and Computational Radiation Physics

Center for Advanced Systems Understanding

- Focus on modelling, simulating, visualizing laser-matter interaction
- Open-source project PIConGPU
- Aim at laser particle acceleration and models for compact radiation sources

CASUS and Computational Radiation Physics

Center for Advanced Systems Understanding

- Focus on modelling, simulating, visualizing laser-matter interaction
- Open-source project PIConGPU
- Aim at laser particle acceleration and models for compact radiation sources
- Further development into laser-driven fusion

PIConGPU Simulation Example

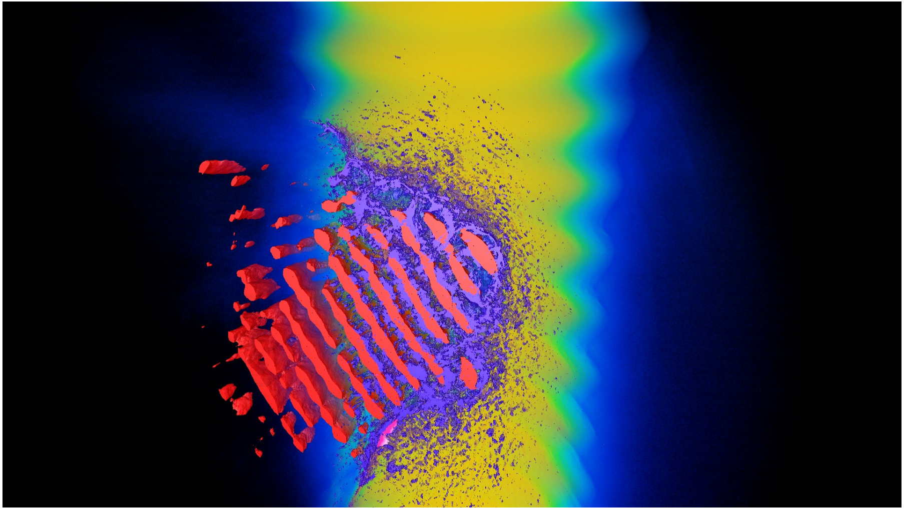


Figure: Laser (red) hitting cryogenic hydrogen (blue-yellow-pink for increasing electron energy density)

Multiple Dispatch

Multiple Dispatch

```
abstract type Particle end
```

Multiple Dispatch

```
abstract type Particle end


struct Electron <: Particle end
struct Positron <: Particle end
struct Photon <: Particle end
```

Multiple Dispatch

```julia
abstract type Particle end


struct Electron <: Particle end
struct Positron <: Particle end
struct Photon <: Particle end

can_interact(p1::T, p2::T) where {T<:Particle} = false
can_interact(p1::T1, p2::T2) where {T1<:Particle, T2<:Particle} = true
```

Multiple Dispatch

```julia
abstract type Particle end


struct Electron <: Particle end
struct Positron <: Particle end
struct Photon <: Particle end

can_interact(p1::T, p2::T) where {T<:Particle} = false
can_interact(p1::T1, p2::T2) where {T1<:Particle, T2<:Particle} = true
```

Multiple Dispatch

```julia
abstract type Particle end


struct Electron <: Particle end
struct Positron <: Particle end
struct Photon <: Particle end

can_interact(p1::T, p2::T) where {T<:Particle} = false
can_interact(p1::T1, p2::T2) where {T1<:Particle, T2<:Particle} = true

using Test
@test can_interact(Photon(), Electron()) == true
@test can_interact(Electron(), Positron()) == true
@test can_interact(Electron(), Electron()) == false

# Done? What if we want to add more particles?
```

Multiple Dispatch

```julia
abstract type Particle end
abstract type Boson <: Particle end
abstract type Fermion <: Particle end
struct Electron <: Fermion end
struct Positron <: Fermion end
struct Photon <: Boson end

can_interact(p1::T, p2::T) where {T<:Fermion} = false
can_interact(p1::Photon, p2::Photon) = false
can_interact(p1::Photon, p2::Fermion) = true
can_interact(p1::Fermion, p2::Photon) = true
can_interact(p1::Electron, p2::Positron)  = true
can_interact(p1::Positron, p2::Electron)  = true
```

Multiple Dispatch - Reusability

```
struct Muon <: Fermion end
struct AntiMuon <: Fermion end
```

Multiple Dispatch - Reusability

```julia
struct Muon <: Fermion end
struct AntiMuon <: Fermion end

can_interact(p1::Muon, p2::AntiMuon) = true
can_interact(p1::AntiMuon, p2::Muon) = true

@test can_interact(Muon(), Photon()) == true
@test can_interact(Muon(), AntiMuon()) == true
@test can_interact(AntiMuon(), AntiMuon()) == false
```

Multiple Dispatch - Reusability

```julia
struct Muon <: Fermion end
struct AntiMuon <: Fermion end

can_interact(p1::Muon, p2::AntiMuon) = true
can_interact(p1::AntiMuon, p2::Muon) = true

@test can_interact(Muon(), Photon()) == true
@test can_interact(Muon(), AntiMuon()) == true
@test can_interact(AntiMuon(), AntiMuon()) == false
```

We can reuse existing functions for new types

Multiple Dispatch - Reusability

```julia
struct Muon <: Fermion end
struct AntiMuon <: Fermion end

can_interact(p1::Muon, p2::AntiMuon) = true
can_interact(p1::AntiMuon, p2::Muon) = true

@test can_interact(Muon(), Photon()) == true
@test can_interact(Muon(), AntiMuon()) == true
@test can_interact(AntiMuon(), AntiMuon()) == false
```

We can reuse existing functions for new types

Without owning the functions

Multiple Dispatch - Reusability

```julia
interaction_result(::T, ::T) where {T<:Fermion}          = nothing
interaction_result(::Photon, ::Photon)                   = nothing
interaction_result(::Photon, ::T) where {T<:Fermion}  = T()
interaction_result(::T, ::Photon) where {T<:Fermion}  = T()
interaction_result(::Electron, ::Positron)               = Photon()
interaction_result(::Positron, ::Electron)               = Photon()
```

Multiple Dispatch - Reusability

```julia
interaction_result(::T, ::T) where {T<:Fermion}           = nothing
interaction_result(::Photon, ::Photon)                    = nothing
interaction_result(::Photon, ::T) where {T<:Fermion}      = T()
interaction_result(::T, ::Photon) where {T<:Fermion}      = T()
interaction_result(::Electron, ::Positron)                = Photon()
interaction_result(::Positron, ::Electron)                = Photon()

@test interaction_result(Positron(), Photon()) == Positron()
@test interaction_result(Photon(), Electron()) == Electron()
@test interaction_result(Positron(), Electron()) == Photon()
@test isnothing(interaction_result(Muon(), Muon()))
```

Multiple Dispatch - Reusability

```julia
interaction_result(::T, ::T) where {T<:Fermion}          = nothing
interaction_result(::Photon, ::Photon)                   = nothing
interaction_result(::Photon, ::T) where {T<:Fermion}     = T()
interaction_result(::T, ::Photon) where {T<:Fermion}     = T()
interaction_result(::Electron, ::Positron)               = Photon()
interaction_result(::Positron, ::Electron)               = Photon()

@test interaction_result(Positron(), Photon()) == Positron()
@test interaction_result(Photon(), Electron()) == Electron()
@test interaction_result(Positron(), Electron()) == Photon()
@test isnothing(interaction_result(Muon(), Muon()))
```

We can reuse existing *types* for new *functions*

Multiple Dispatch - Reusability

```julia
interaction_result(::T, ::T) where {T<:Fermion}          = nothing
interaction_result(::Photon, ::Photon)                   = nothing
interaction_result(::Photon, ::T) where {T<:Fermion}     = T()
interaction_result(::T, ::Photon) where {T<:Fermion}     = T()
interaction_result(::Electron, ::Positron)               = Photon()
interaction_result(::Positron, ::Electron)               = Photon()

@test interaction_result(Positron(), Photon()) == Positron()
@test interaction_result(Photon(), Electron()) == Electron()
@test interaction_result(Positron(), Electron()) == Photon()
@test isnothing(interaction_result(Muon(), Muon()))
```

We can reuse existing *types* for new *functions*

Without owning the types

Multiple Dispatch

But how is this different from function overloading?

Multiple Dispatch

But how is this different from function overloading?

```
particles1 = [rand([Photon(), Electron(), Positron()]) for _ in 1:10]
particles2 = [rand([Photon(), Electron(), Positron()]) for _ in 1:10]

for (p1, p2) in Iterators.zip(particles1, particles2)
      if can_interact(p1, p2)
            println("$p1 and $p2 can interact")
      else
            println("$p1 and $p2 can not interact")
      end
end
```

Multiple Dispatch

But how is this different from function overloading?

```julia
particles1 = [rand([Photon(), Electron(), Positron()]) for _ in 1:10]
particles2 = [rand([Photon(), Electron(), Positron()]) for _ in 1:10]

for (p1, p2) in Iterators.zip(particles1, particles2)
      if can_interact(p1, p2)
            println("$p1 and $p2 can interact")
      else
            println("$p1 and $p2 can not interact")
      end
end
```

- C++ can do static multiple dispatch → function overloading

Multiple Dispatch

But how is this different from function overloading?

```
particles1 = [rand([Photon(), Electron(), Positron()]) for _ in 1:10]
particles2 = [rand([Photon(), Electron(), Positron()]) for _ in 1:10]

for (p1, p2) in Iterators.zip(particles1, particles2)
    if can_interact(p1, p2)
        println("$p1 and $p2 can interact")
    else
        println("$p1 and $p2 can not interact")
    end
end
```

- C++ can do static multiple dispatch → function overloading
- C++ can do single dynamic dispatch → polymorphism

Multiple Dispatch

But how is this different from function overloading?

```
particles1 = [rand([Photon(), Electron(), Positron()]) for _ in 1:10]
particles2 = [rand([Photon(), Electron(), Positron()]) for _ in 1:10]

for (p1, p2) in Iterators.zip(particles1, particles2)
      if can_interact(p1, p2)
            println("$p1 and $p2 can interact")
      else
            println("$p1 and $p2 can not interact")
      end
end
```

- C++ can do static multiple dispatch → function overloading
- C++ can do single dynamic dispatch → polymorphism
- C++ **cannot** do multiple dynamic dispatch

Multiple Dispatch

But how is this different from function overloading?

```julia
particles1 = [rand([Photon(), Electron(), Positron()]) for _ in 1:10]
particles2 = [rand([Photon(), Electron(), Positron()]) for _ in 1:10]

for (p1, p2) in Iterators.zip(particles1, particles2)
      if can_interact(p1, p2)
            println("$p1 and $p2 can interact")
      else
            println("$p1 and $p2 can not interact")
      end
end
```

- C++ can do static multiple dispatch → function overloading
- C++ can do single dynamic dispatch → polymorphism
- C++ **cannot** do multiple dynamic dispatch[1]

[1] C++ can of course *functionally* do this, as can every language that is turing-complete

Code Generation

Julia understands its own code

```
julia> Meta.parse("1 + 2")
:(1 + 2)

julia> dump(ans)
Expr
  head: Symbol call
  args: Array{Any}((3,))
    1: Symbol +
    2: Int64 1
    3: Int64 2
```

Code Generation

Julia understands its own code

```
julia> expr = Meta.parse("x + 3")
:(x + 3)

julia> x = 5
5

julia> eval(expr)
8
```


DRESDEN concept HZDR

Code Generation

Julia understands its own code

```
julia> expr = Meta.parse("x + 3")
:(x + 3)

julia> eval(Meta.parse("add3(x) = $expr"))
add3 (generic function with 1 method)

julia> add3(5)
8
```

Code Generation

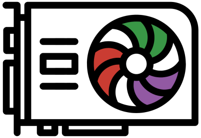
Julia understands its own code

```
julia> expr = Meta.parse("x + 3")
:(x + 3)

julia> eval(Meta.parse("add3(x) = $expr"))
add3 (generic function with 1 method)

julia> add3(5)
8
```

→ We can generate functions at runtime

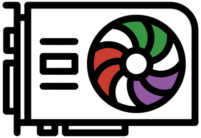
Many more capabilities, see the Julia Docs on Metaprogramming

GPU Usage



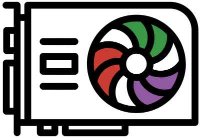
- JuliaGPU with
  - CUDA.jl (NVIDIA GPUs)
  - AMDGPU.jl (AMD GPUs)
  - oneAPI.jl (Intel GPUs)
  - Metal.jl (Apple GPUs)
  - KernelAbstractions.jl for writing kernels that run on any of the above

GPU Usage



- JuliaGPU with
  - CUDA.jl (NVIDIA GPUs)
  - AMDGPU.jl (AMD GPUs)
  - oneAPI.jl (Intel GPUs)
  - Metal.jl (Apple GPUs)
  - KernelAbstractions.jl for writing kernels that run on any of the above
- Varying degrees of support

GPU Usage



- JuliaGPU with
  - CUDA.jl (NVIDIA GPUs)
  - AMDGPU.jl (AMD GPUs)
  - oneAPI.jl (Intel GPUs)
  - Metal.jl (Apple GPUs)
  - KernelAbstractions.jl for writing kernels that run on any of the above
- Varying degrees of support
- Array-driven programming via broadcast

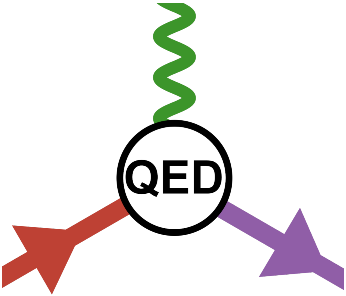
GPU Usage - Example

```
julia> using oneAPI

julia> heavy_compute(x) = begin for _ in 1:100 x = sin(sqrt(x))^2 end; return x end
heavy_compute (generic function with 1 method)

julia> gpu_vec = oneVector([rand(Float32) for _ in 1:10_000])
10000-element oneArray{Float32, 1, oneAPI.oneL0.DeviceBuffer}:
 0.50362855
 ...
 0.042337418

julia> heavy_compute.(gpu_vec)
10000-element oneArray{Float32, 1, oneAPI.oneL0.DeviceBuffer}:
 0.027846681
 ...
 0.01750382
```


DRESDEN concept HZDR

Goal

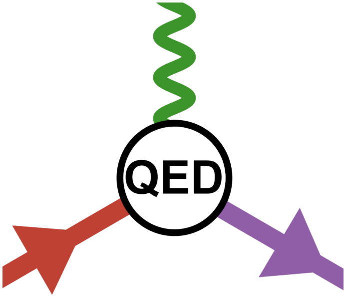
Event Generation



Goal

Event Generation

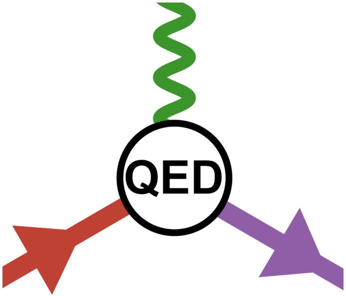
- Subgoal: Computation of matrix elements



Goal

Event Generation

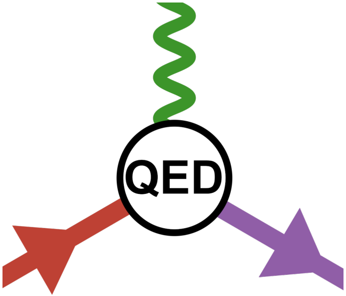
- Subgoal: Computation of matrix elements (for arbitrary scattering processes, in QED, at tree-level, for any/all spin and polarization combinations)



Goal

Event Generation

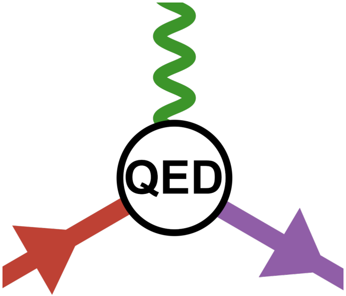
- Subgoal: Computation of matrix elements
- Where to start?



Goal

Event Generation

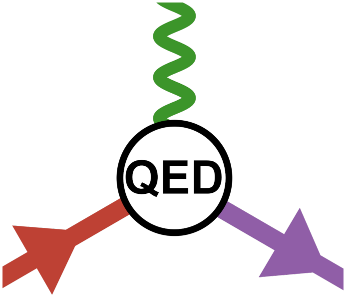
- Subgoal: Computation of matrix elements
- Where to start?
- Simpler model than QED → ABC toy model



Goal

Event Generation

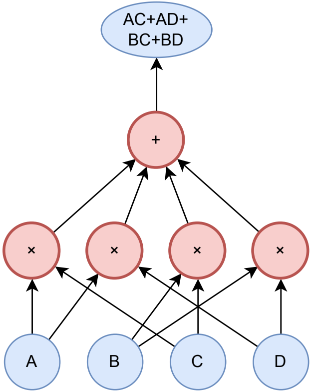
- Subgoal: Computation of matrix elements
- Where to start?
- Simpler model than QED $\rightarrow$ ABC toy model
- Generate code per particle process



DRESDEN concept HZDR

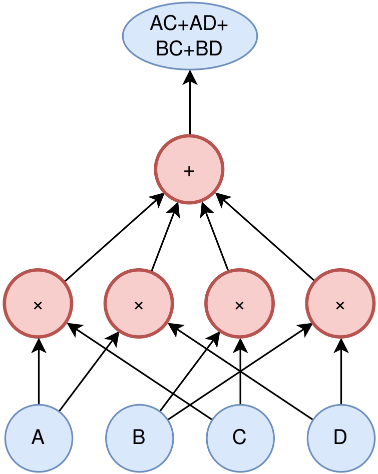
ComputableDAGs.jl

- Represent computation as a directed acyclic graph (DAG) of function calls
- Allows dynamic construction, analysis, scheduling, and execution (threaded, GPU, etc.)
- ~ 2000 lines of code, ~ 700 lines of comments

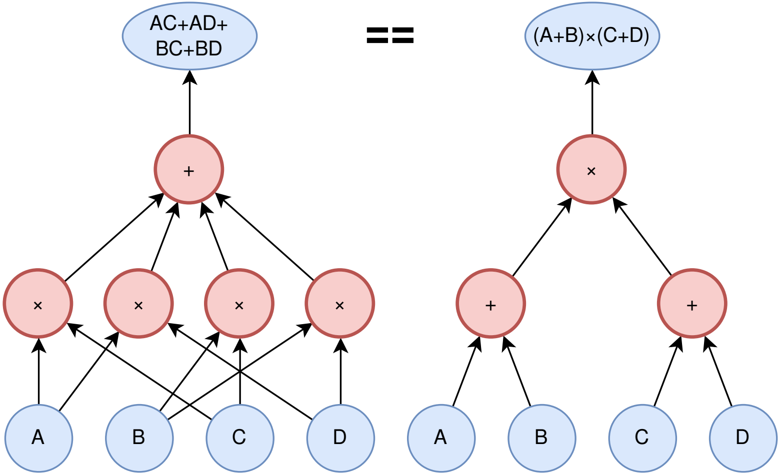


ComputableDAGs.jl

ComputableDAGs.jl - Distributivity



ComputableDAGs.jl - Distributivity



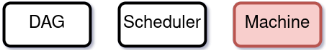
ComputableDAGs.jl - Outline

- Get graph, a scheduler, and machine information

ComputableDAGs.jl - Outline

- Get graph, a scheduler, and machine information

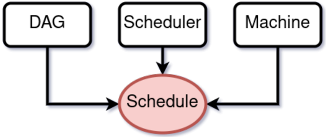
ComputableDAGs.jl - Outline



- Get graph, a scheduler, and machine information

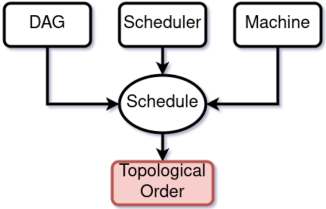
ComputableDAGs.jl - Outline

- Get graph, a scheduler, and machine information
- Use scheduler interface



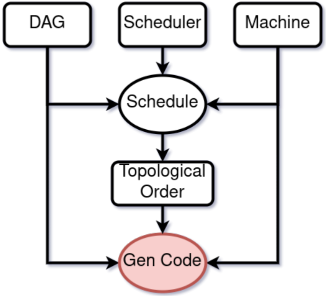
ComputableDAGs.jl - Outline

- Get graph, a scheduler, and machine information
- Use scheduler interface to create a topological ordering of tasks for each device



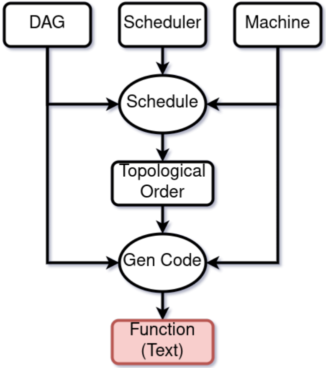
ComputableDAGs.jl - Outline

- Get graph, a scheduler, and machine information
- Use scheduler interface to create a topological ordering of tasks for each device
- For each task in the ordering, generate code using the scheduled device



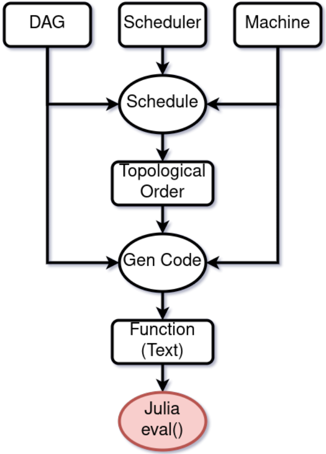
ComputableDAGs.jl - Outline

- Get graph, a scheduler, and machine information
- Use scheduler interface to create a topological ordering of tasks for each device
- For each task in the ordering, generate code using the scheduled device
- Evaluate the function code



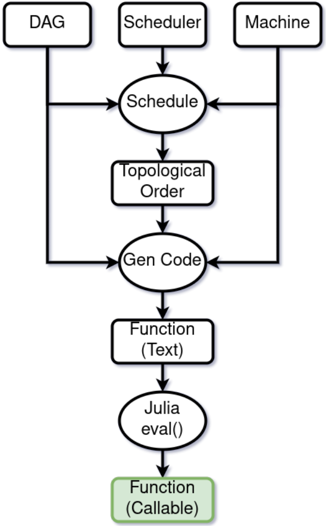
ComputableDAGs.jl - Outline

- Get graph, a scheduler, and machine information
- Use scheduler interface to create a topological ordering of tasks for each device
- For each task in the ordering, generate code using the scheduled device
- Evaluate the function code into a function



ComputableDAGs.jl - Outline

- Get graph, a scheduler, and machine information
- Use scheduler interface to create a topological ordering of tasks for each device
- For each task in the ordering, generate code using the scheduled device
- Evaluate the function code into a function



Code Generation in Detail (Simplified)

```
struct FunctionCall
    func::Function
    arguments::Vector{Symbol}
    return_symbol::Symbol
    return_types::Type
end
```

Code Generation in Detail (Simplified)

```julia
struct FunctionCall
    func::Function
    arguments::Vector{Symbol}
    return_symbol::Symbol
    return_types::Type
end

function expr_from_fc(fc::FunctionCall)
    func_call = Expr(:call, fc.func, fc.arguments...)
    return Expr(:(=), fc.return_symbol, func_call)
end
```

Code Generation in Detail (Simplified)

```julia
struct FunctionCall
    func::Function
    arguments::Vector{Symbol}
    return_symbol::Symbol
    return_types::Type
end

function expr_from_fc(fc::FunctionCall)
    func_call = Expr(:call, fc.func, fc.arguments...)
    return Expr(:(=), fc.return_symbol, func_call)
end

function function_body(function_calls::Vector{FunctionCall})
    return Expr(:block, expr_from_fc.(function_calls)...)
end
```

Code Generation in Detail (Simplified)

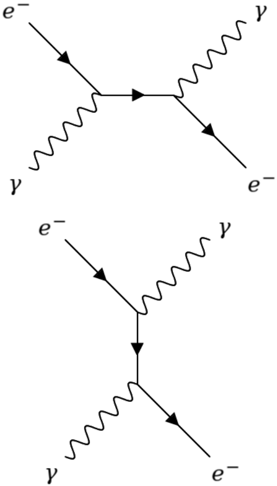
```julia
struct FunctionCall
    func::Function
    arguments::Vector{Symbol}
    return_symbol::Symbol
    return_types::Type
end

function expr_from_fc(fc::FunctionCall)
    func_call = Expr(:call, fc.func, fc.arguments...)
    return Expr(:(=), fc.return_symbol, func_call)
end

function function_body(function_calls::Vector{FunctionCall})
    return Expr(:block, expr_from_fc.(function_calls)...)
end
```

Not a lot of Code!

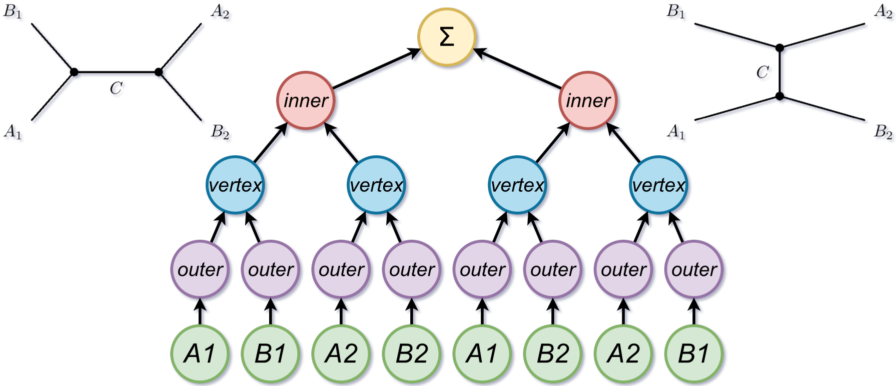
QEDFeynmanDiagrams.jl

- From a given particle process, e.g., $e^- + \gamma \to e^- + \gamma$, generate a function for its squared matrix element, given the particle momenta
- Built on top of ComputableDAGs.jl
- ~ 1700 lines of code, ~ 270 lines of comments

QEDFeynmanDiagrams.jl



From Process to Function



Code Adventure: Fun with Multiple Dispatch

Problem: We want a process object that contains external particles and their respective spins or polarizations.

```
struct ScatteringProcess{INT,OUTT,INSP,OUTSP}
        where {INT<:Tuple,OUTT<:Tuple,INSP<:Tuple,OUTSP<:Tuple}
    incoming_particles::INT
    outgoing_particles::OUTT

    incoming_spin_pols::INSP
    outgoing_spin_pols::OUTSP
end
```

Code Adventure: Fun with Multiple Dispatch

Problem: We want a process object that contains external particles and their respective spins or polarizations.

```
struct ScatteringProcess{INT,OUTT,INSP,OUTSP}
        where {INT<:Tuple,OUTT<:Tuple,INSP<:Tuple,OUTSP<:Tuple}
    incoming_particles::INT
    outgoing_particles::OUTT

    incoming_spin_pols::INSP
    outgoing_spin_pols::OUTSP
end
```

How do we make sure that fermions have a spin and bosons have a polarization?

Code Adventure: Fun with Multiple Dispatch

Problem: We want a process object that contains external particles and their respective spins or polarizations.

```
struct ScatteringProcess{INT,OUTT,INSP,OUTSP}
        where {INT<:Tuple,OUTT<:Tuple,INSP<:Tuple,OUTSP<:Tuple}
    incoming_particles::INT
    outgoing_particles::OUTT

    incoming_spin_pols::INSP
    outgoing_spin_pols::OUTSP
end
```

How do we make sure that fermions have a spin and bosons have a polarization?
→ Input validation!

Code Adventure: Input Validation

```
function ScatteringProcess(
    in_particles::NTuple{I,AbstractParticleType},
    out_particles::NTuple{O,AbstractParticleType},
    in_spin_pols::NTuple{I,AbstractSpinOrPolarization},
    out_spin_pols::NTuple{O,AbstractSpinOrPolarization},
) where {I,O}
    _assert_spin_pol_particle_compatibility(in_particles, in_spin_pols)
    _assert_spin_pol_particle_compatibility(out_particles, out_spin_pols)

    return new{
        typeof(in_particles),
        typeof(out_particles),
        typeof(in_spin_pols),
        typeof(out_spin_pols),
    }(
        in_particles, out_particles, in_spin_pols, out_spin_pols
    )
end
```

Code Adventure: Input Validation

```
_assert_spin_pol_particle_compatibility(::Tuple{}, ::Tuple{}) = nothing
```

Code Adventure: Input Validation

```
_assert_spin_pol_particle_compatibility(::Tuple{}, ::Tuple{}) = nothing

function _assert_spin_pol_particle_compatibility(
    particles::Tuple{AbstractParticleType,Vararg},
    spin_pols::Tuple{AbstractSpinOrPolarization,Vararg},
)
    if is_fermion(particles[1]) && !(spin_pols[1] isa AbstractSpin)
        throw("invalid combination")
    end
    if is_boson(particles[1]) && !(spin_pols[1] isa AbstractPolarization)
        throw("invalid combination")
    end
    return _assert_spin_pol_particle_compatibility(
        particles[2:end],
        spin_pols[2:end],
    )
end
```

Code Adventure: Input Validation

Okay, but why so complicated? Why not just loop it?

```julia
function _assert_spin_pol_particle_compatibility_loop(
    particles::Tuple{AbstractParticleType,Vararg},
    spin_pols::Tuple{AbstractSpinOrPolarization,Vararg},
)
    for (p, s) in Iterators.zip(particles, spin_pols)
        if is_fermion(p) && !(s isa AbstractSpin)
            throw("invalid combination")
        end
        if is_boson(p) && !(s isa AbstractPolarization)
            throw("invalid combination")
        end
    end
    return nothing
end
```

Code Adventure: Comparison

```
julia> using BenchmarkTools, QEDcore

julia> # define functions from above...

julia> @btime _assert_spin_pol_particle_compatibility(
           (Electron(), Photon()),
           (SpinUp(), PolX()),
       )
  1.021 ns (0 allocations: 0 bytes)
```

Code Adventure: Comparison

```
julia> using BenchmarkTools, QEDcore

julia> # define functions from above...

julia> @btime _assert_spin_pol_particle_compatibility(
           (Electron(), Photon()),
           (SpinUp(), PolX()),
       )
  1.021 ns (0 allocations: 0 bytes)

julia> @btime _assert_spin_pol_particle_compatibility_loop(
           (Electron(), Photon()),
           (SpinUp(), PolX()),
       )
  12.256 ns (0 allocations: 0 bytes)
```

What gives?

DRESDEN concept HZDR

Code Adventure: Code Inspection

- Generate with `@code_native <call>`
- The blue lines are assembly instructions
- It compiles the loop into code

Native code of the looped version

Code Adventure: Code Inspection

Native code of the recursive version

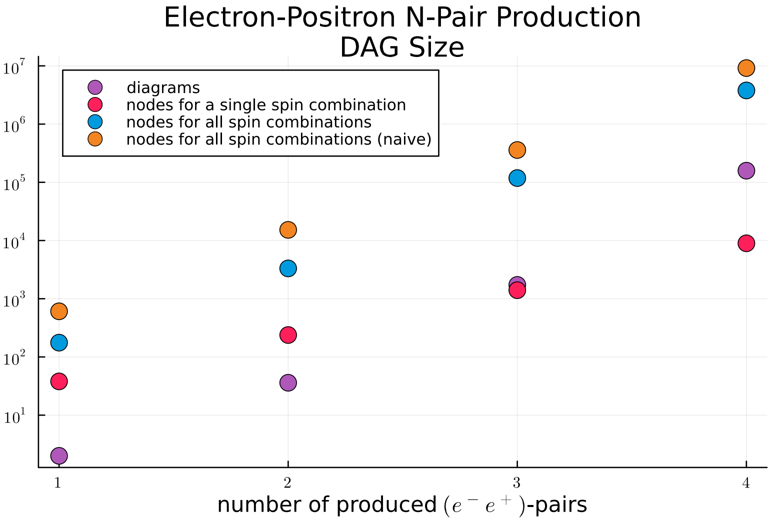
```
julia> @code_native _assert_spin_pol_particle_compatibility((Electron(), Photon()), (SpinUp(), PolX()))
        .text
        .file   "_assert_spin_pol_particle_compatibility"
        .globl  julia__assert_spin_pol_particle_compatibility_2920 # -- Begin function julia__assert_spin_pol_particle_compatibility_2920
        .p2align        4, 0x90
        .type   julia__assert_spin_pol_particle_compatibility_2920,@function
julia__assert_spin_pol_particle_compatibility_2920: # @julia__assert_spin_pol_particle_compatibility_2920
; Function Signature: _assert_spin_pol_particle_compatibility(Tuple{QEDcore.Electron, QEDcore.Photon}, Tuple{QEDbase.SpinUp, QEDbase.PolarizationX})
; ┌ @ REPL[3]:1 within `_assert_spin_pol_particle_compatibility`
# %bb.0:                                # %top
        push    rbp
        mov     rbp, rsp
        pop     rbp
        ret
.Lfunc_end0:
        .size   julia__assert_spin_pol_particle_compatibility_2920, .Lfunc_end0-julia__assert_spin_pol_particle_compatibility_2920
; └
                                        # -- End function
        .section        ".note.GNU-stack","",@progbits
```

Results - Reproducibility

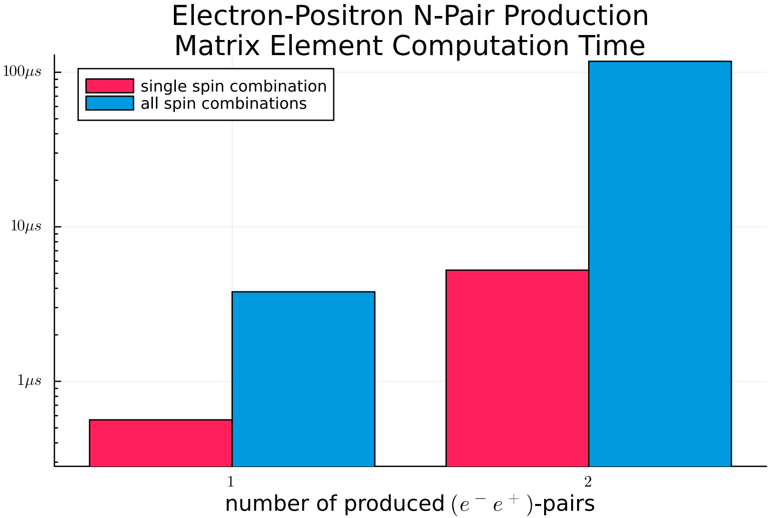
The following results can be reproduced using the Jupyter notebooks at this URL:

https://github.com/AntonReinhard/QEDFeynmanDiagrams.jl/tree/profiling/profiling

Results - $e^- + e^+ \to n(e^- + e^+)$ - DAG Sizes



Results - $e^- + e^+ \rightarrow n(e^- + e^+)$ - DAG Computation Time



Results - Profiling Flamegraph

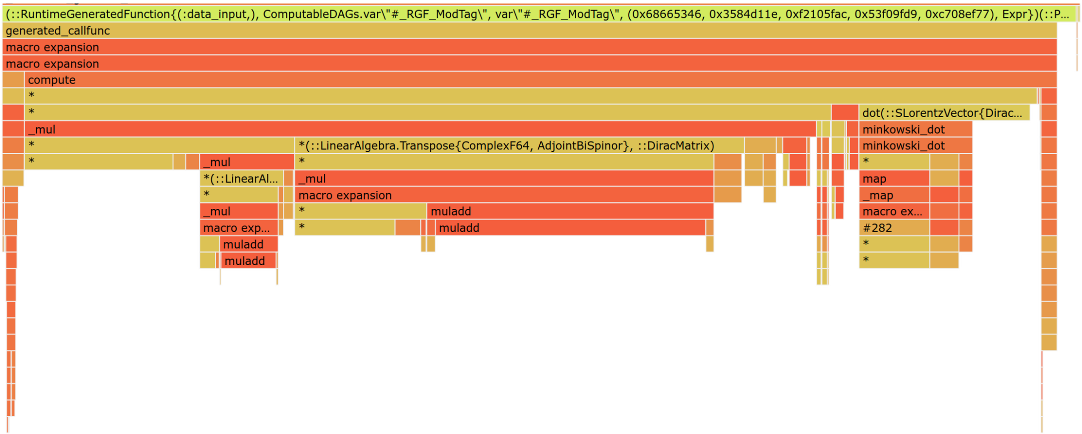


Figure: Created with PProf.jl

Results - $e^- + n\gamma \to e^- + \gamma$ - 4 GPU HPC Node heterogeneous

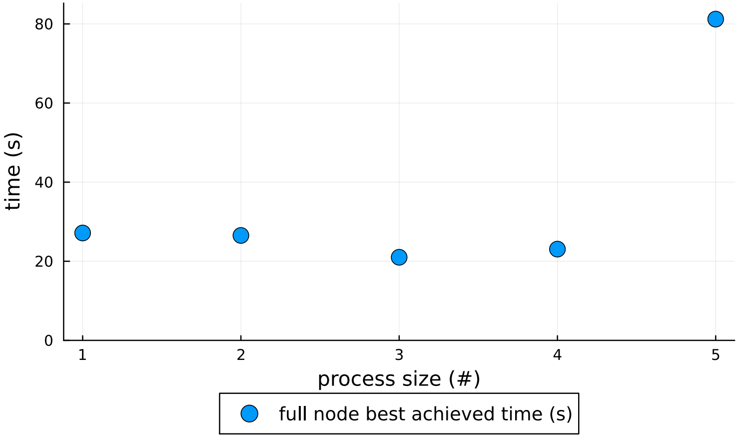


Figure: Time taken for $2^{30} \approx 10^9$ squared matrix elements for n-photon Compton events

Outlook

- Ongoing development
- Compare to existing solutions (MadGraph5 [1], O'Mega [2], SHERPA [3])
- Extension for other quantum field theories through generalized diagram generation
- Consider vectorization inside the graph

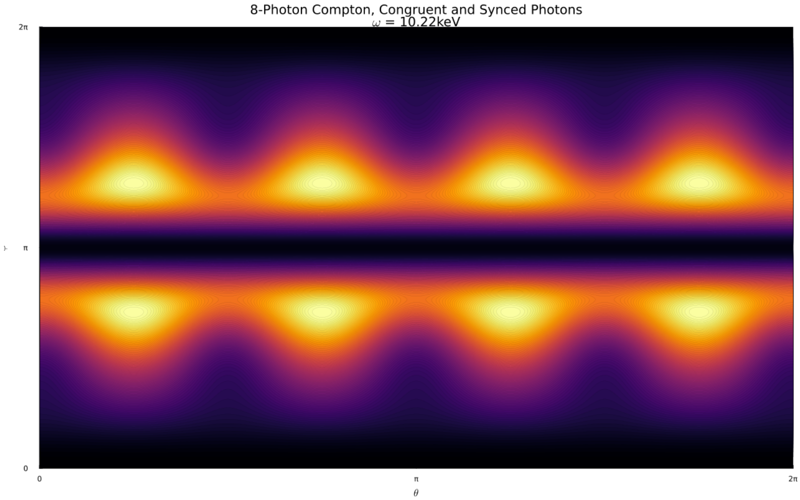
Acknowledgements

Collaborators:

- **Dr. Uwe Hernandez Acosta**[1,2]
- **Simeon Ehrig**[1,2]
- **Dr. Klaus Steiniger**[1,2]
- **Dr. Michael Bussmann**[1,2]

[1] Center for Advanced Systems Understanding (CASUS)
[2] Helmholtz-Zentrum Dresden-Rossendorf (HZDR)

Discussion


8-Photon Compton, Congruent and Synced Photons
ω = 10.22keV
2π
π
0
0
π
2π
θ

References

[1] Johan Alwall et al. "The automated computation of tree-level and next-to-leading order differential cross sections, and their matching to parton shower simulations". In: *Journal of High Energy Physics* 2014.7 (2014), pp. 1–157.

[2] Mauro Moretti, Thorsten Ohl, and Jürgen Reuter. *O'Mega: An Optimizing Matrix Element Generator*. 2001. arXiv: hep-ph/0102195 [hep-ph]. URL: https://arxiv.org/abs/hep-ph/0102195.

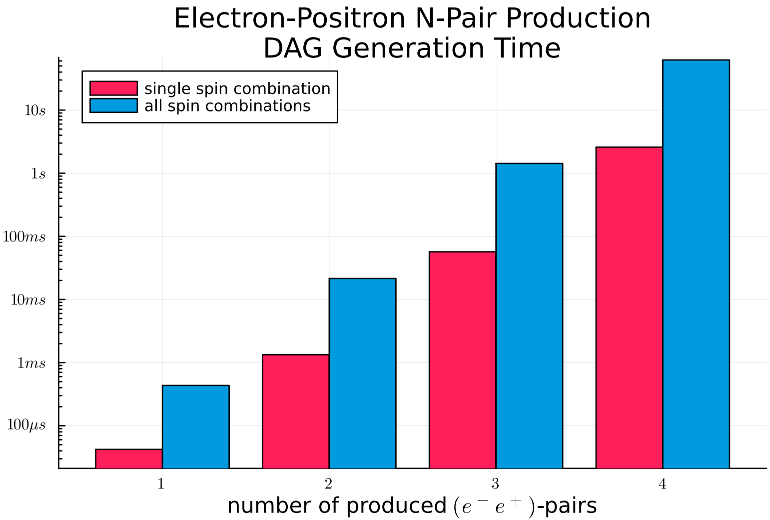
[3] Tanju Gleisberg et al. "Event generation with SHERPA 1.1". In: *Journal of High Energy Physics* 2009.02 (2009), p. 007.

[4] Tim Besard, Christophe Foket, and Bjorn De Sutter. "Effective extensible programming: unleashing Julia on GPUs". In: *IEEE Transactions on Parallel and Distributed Systems* 30.4 (2018), pp. 827–841.

[5] Stefan Karpinski et al. *Why we created julia*. Feb. 2012. URL: https://julialang.org/blog/2012/02/why-we-created-julia/.

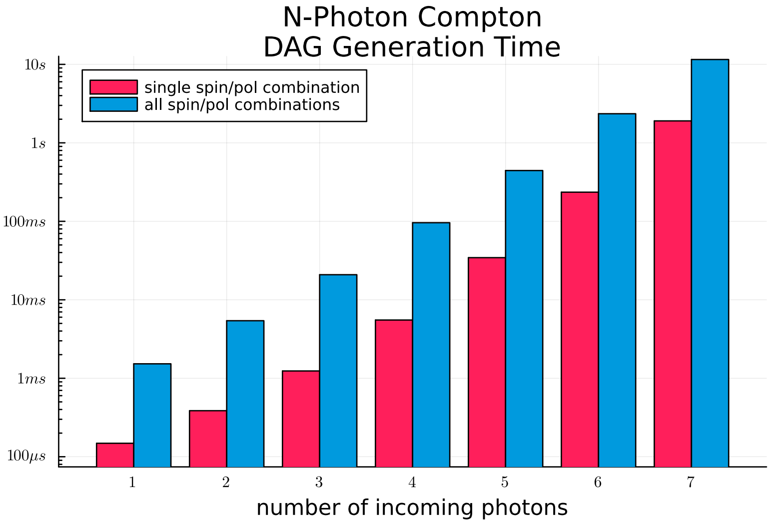
[6] Valentin Churavy et al. "Bridging HPC Communities through the Julia Programming Language". In: *arXiv preprint arXiv:2211.02740* (2022).

Results - $e^- + e^+ \rightarrow n(e^- + e^+)$ - DAG Generation Time

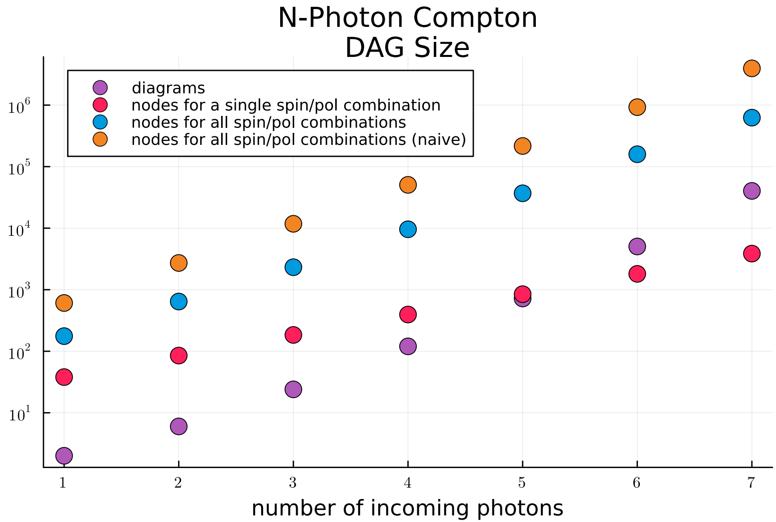


DRESDEN concept HZDR

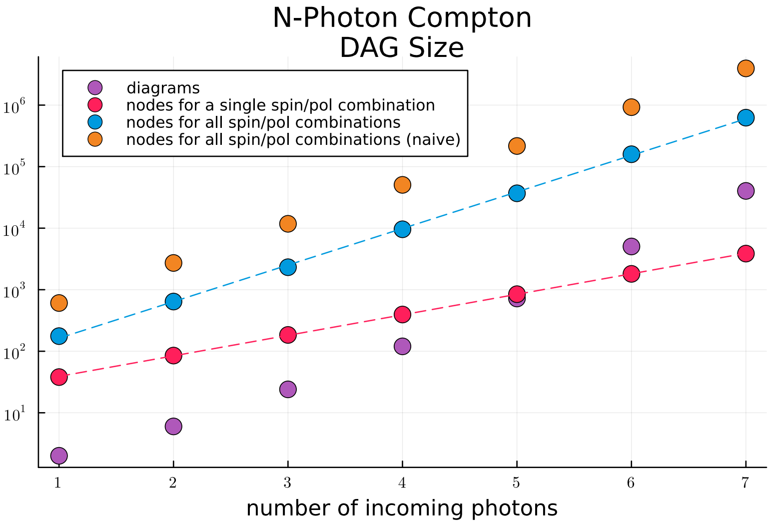
Results - $e^- + k\gamma \to e^- + \gamma$ - DAG Generation Time



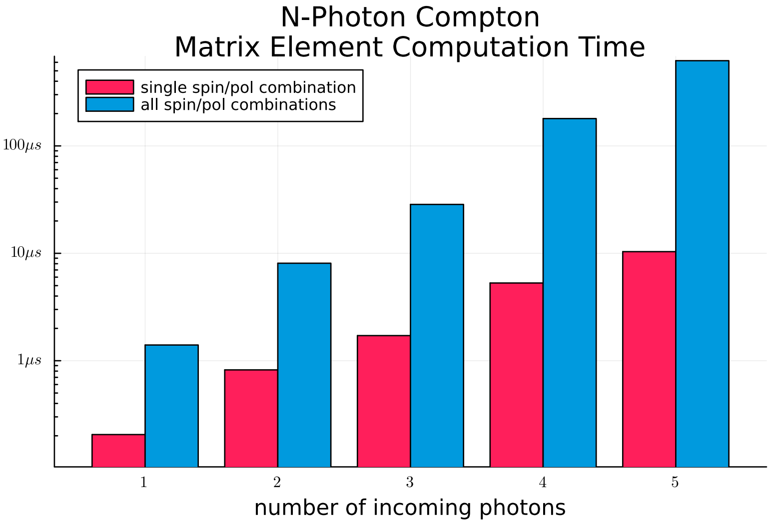
Results - $e^- + k\gamma \to e^- + \gamma$ - DAG Sizes



Results - $e^- + k\gamma \to e^- + \gamma$ - DAG Sizes



Results - $e^- + k\gamma \to e^- + \gamma$ - DAG Computation Time



Benchmarking Machine

Home PC with

- Ryzen 7900X3D
- 2×32GB DDR5 RAM @ 6000MHz
- Julia v1.10