Event generation in Julia

HSF Physics Generator WG meeting

July 7, 2022 // Uwe Hernandez Acosta

CASUS
CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

www.casus.science

HZDR HELMHOLTZ ZENTRUM DRESDEN ROSSENDORF

TECHNISCHE UNIVERSITÄT DRESDEN

CBG Max Planck Institute of Molecular Cell Biology and Genetics

HELMHOLTZ CENTRE FOR ENVIRONMENTAL RESEARCH – UFZ

Uniwersytet Wrocławski

SPONSORED BY THE

Federal Ministry of Education and Research

STAATSMINISTERIUM FÜR WISSENSCHAFT UND KUNST

Freistaat SACHSEN

Who am I?

- Uwe Hernandez Acosta
- particle physicist by training
- Phd in physics 2021 from TU Dresden/HZDR
  - topic: strong-field QED
- affiliation: CASUS - Center for Advanced Systems Understanding @ HZDR
- interested in:
  - theoretical particle physics/quantum field theory
  - strong-field physics
  - event generation
  - mathematical modelling
  - coding challenges

Contents

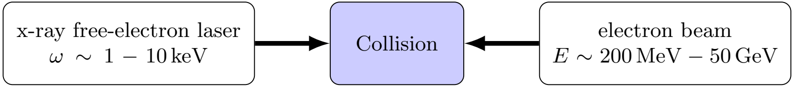
Motivation: electromagnetic cascades

`QED.jl` - current status

Used technologies

Part I: Motivation

Physics use-case



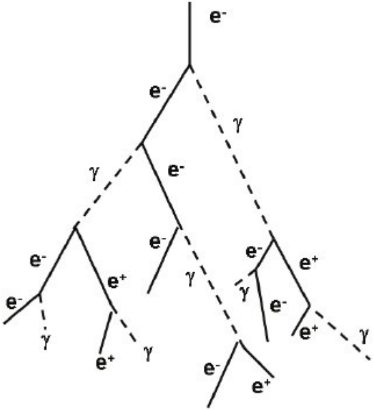
Phenomena

- multi-photon scattering
- non-perturbative effects
- vacuum polarisation effects
- electromagnatic cascades

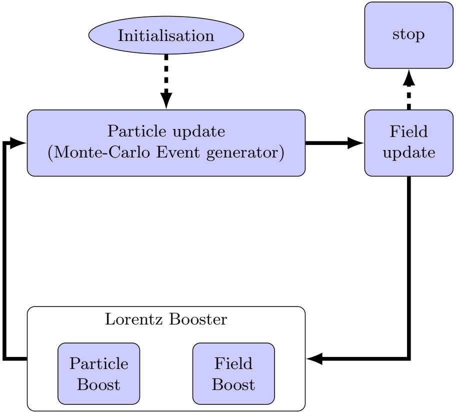
Applications

- Magnetars
- high-luminosity $e^- e^+$ collider
- Dirac/Weyl semi-metals
- relativistic plasma physics

Electromagnetic cascades

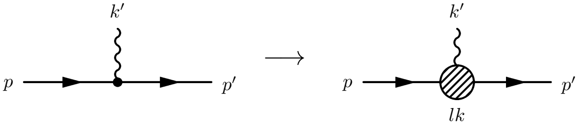


[La Rocca - Advances in Photodiodes. IntechOpen, 2011]



Electromagnetic cascades: Strong-field QED

- Feynman-rule: vertex



$$= -ie\Gamma^{\mu}(l)(2\pi)^4\delta^{(4)}(p + lk - p' - k')$$

- vertex function

$$\Gamma^{\mu}(l, p, p', k) = \Gamma_0^{\mu} B_0(l) + \Gamma_1^{\mu\nu} B_{1\nu}(l) + \Gamma_2^{\mu} B_2(l)$$

- Phase integrals

$$\left.\begin{matrix} B_0(l) \\ B_1^{\mu}(l) \\ B_2(l) \end{matrix}\right\} = \int_{-\infty}^{\infty} \mathrm{d}\phi \, \exp(il\phi + iG(\phi)) \begin{cases} 1 \\ A^{\mu}(\phi) \\ A^{\mu}(\phi)A_{\mu}(\phi) \end{cases}$$

[UHA2021 PhD thesis - TU Dresden]

Where we are at?

1. computing matrix elements depends on the BG field
2. generating single events is expensive
3. cascade events are chains of single events

$\Rightarrow$ distributed computing needed

Part II: `QED.jl` - current status

program structure: `QED.jl` [https://github.com/QEDjl-project]

CASUS
CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

Base Packages

- **QEDmc.jl** — (D)RNG, Vegas, MCMC,...
- **QEDbase.jl** — Lorentz vectors, spinors,...
- **QEDio.jl** — hepMC3, openPMD, LHA,...

Physics Packages

- **QEDmodels.jl** — pQED, sfQED,...
- **QEDphasespaces.jl** — RAMBO/MAMBO, SARGE,...
- **QEDfields.jl** — em. fields, spectra, ...

Event Generation

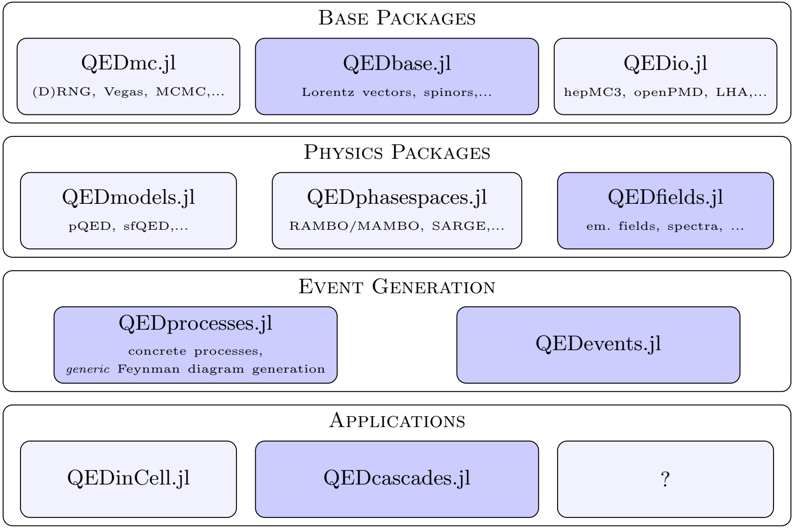
- **QEDprocesses.jl** — concrete processes, *generic* Feynman diagram generation
- **QEDevents.jl**

Applications

- **QEDinCell.jl**
- **QEDcascades.jl**
- ?

program structure: `QED.jl` - proof-of-concept

CASUS
CENTER FOR ADVANCED SYSTEMS UNDERSTANDING



Part III: Used technologies

Why Julia in general?

- modern language
  - powered by LLVM
- easy-to-write
  - garbage collector
  - rich type-system
  - extensive standard library (written mostly in Julia)
  - transparent compilation
- easy-to-use
  - syntax similar to Python/Matlab
  - jit-compiled
  - generic programming + type inference
- blazingly fast number crunching
  - as fast as C/C++ or Fortran
  - specialisation
  - state-of-the-art compiler optimizations

[https://julialang.org]

[J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah - SIAM Rev. 59.1 (2017)]

multiple dispatch - introduction

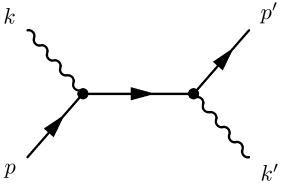
```
abstract type Particle end
struct Electron <: Particle
    name::String
end
struct Positron <: Particle
    name::String
end

function encounter(a::Particle,b::Particle)
    verb = meets(a,b)
    println("$(a.name) meets $(b.name) and they $verb.")
end

meets(a::Electron, b::Electron) = "repel one other"
meets(a::Positron, b::Positron) = "repel one other"
meets(a::Electron, b::Positron) = "attract one other"
meets(a::Positron, b::Electron) = "attract one other"
```

```
>>> encounter(Electron("LittleElectron"),Positron("GrumpyPositron"))
LittleElectron meets GrumpyPositron and they attract one other.
```

[S Karpinski - JuMP-dev2019], [J Bezanson et al. - ARRAY'14]

[S Gowda, et al. - ACM Commun. Comput. Algebra 55.3 (2022)]

multiple dispatch - compiling away configuration



1. External particles

```
init_electron = ParticleState(Electron , Incoming, mom_p)         # u(p)
init_photon   = ParticleState(Photon   , Incoming, mom_k)         # eps(k)
out_electron  = ParticleState(Electron , Outgoing, mom_p_prime)   # ubar(p')
out_photon    = ParticleState(Photon   , Outgoing, mom_k_prime)   # eps'(k')
```

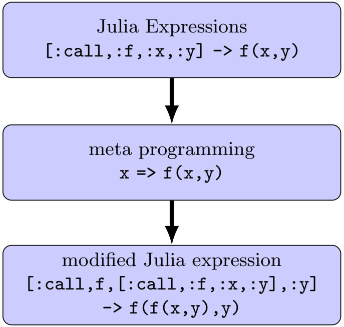
2. Interaction vertices

```
vps1 = VertexProductState(init_electron, init_photon) #  (gamma*eps)*u
vps2 = VertexProductState(out_electron , out_photon ) # ubar*(gamma*eps')
```

3. Propagator and the whole diagram

```
diagram = DiagramProductState(vps1,vps2) # ubar*(gamma*eps')*prop*(gamma*eps)*u
```

meta programming - introduction



- Julia code is a data type in Julia itself
- one can use Julia to manipulate Julia code before compilation
- Julia provides three types of meta programming
  - code generation
  - macros
  - generated functions

meta programming - macros

- the `@unsafe` macro

```
>>> assert_onshell(mom,mass) # checked if mom is on-shell
>>> @unsafe assert_onshell(mom,mass) # nothing is checked -> compiled away
```

```
function assert_onshell(mom::FourMomentum, mass::Real)
    if VALIDITY_CHECK[] && !isonshell(mom,mass)
        throw(MomentumError("The given momentum is assumed to be on-shell."))
    end
end
```

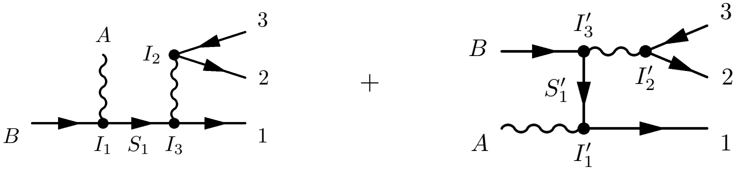
```
const VALIDITY_CHECK = Ref(true)

macro unsafe(ex)
    return quote
        VALIDITY_CHECK.x = false
        local val=$(esc(ex))
        VALIDITY_CHECK.x = true
    end
end
```

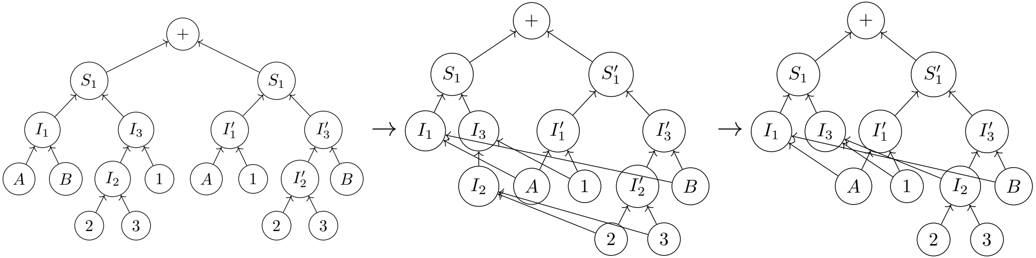
distributed computing - introduction

CASUS CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

- Julia supports OpenMP-like compute models
  - parallelisation of loops
- Julia supports $M \to N$ threading (hybrid threading)
  - M threads from the user are mapped onto N kernel threads
- Julia supports task-migration between native threads
  - task begins execution on a native thread $\to$ gets suspended $\to$ resumes on another thread

distributed computing - directed acyclic graphs

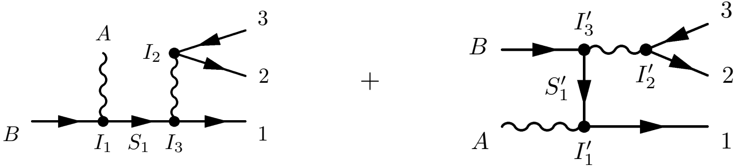




[A Kanaki, C G Papadopoulos - Comput.Phys.Commun. 132 (2000)]

[T Ohl - AIP Conf.Proc. 583 (2002)] [A Valassi et al. - EPJ Web Conf. 251 (2021) 03045]

distributed computing - DAG evaluation [https://github.com/JuliaParallel/Dagger.jl]




```
pA      = @spawn ParticleState(Photon  , Incoming, mom_A)
pB      = @spawn ParticleState(Electron, Incoming, mom_B)
p1      = @spawn ParticleState(Electron, Outgoing, mom_1)
p2      = @spawn ParticleState(Electron, Outgoing, mom_2)
p3      = @spawn ParticleState(Positron, Outgoing, mom_3)
I1      = @spawn interact(pA,pB)
I1prime = @spawn interact(pA,p1)
I2      = @spawn interact(p2,p3)
I3      = @spawn interact(p1,I2)
I3prime = @spawn interact(pB,I2)
S1      = @spawn interact_prop(I3,I1)
S1prime = @spawn interact_prop(I3prime,I1prime)
res     = @spawn add(S1,S1prime)
```

```
>>> fetch(res)
```

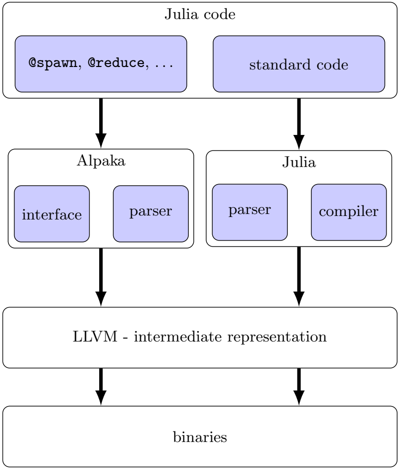
Distributed computing - Julia/Alpaka integration

CASUS
CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

alpaka

Abstraction Library for Parallel Kernel Acceleration

- abstraction layer above parallelisation
- header-only
- zero-overhead
- written in C++
- support of heterogeneous architectures



DISCLAIMER: concept and implementation are work-in-progress

Summary

- Laser-driven cascades are computationally expensive
  - ⇒ Event generation needs to be parallelized
- New library `QED.jl`
  - ⇒ first principal calculations in QED and background-field approximation
  - ⇒ written in Julia
  - ⇒ modularised + highly extensible
- Julia in general
  - ⇒ "as easy as python, as fast as C/C++"
  - ⇒ multiple dispatch
  - ⇒ meta-programming
  - ⇒ native threads
- Dream about the future: compiler extension with Alpaka
  - ⇒ hardware-agnostic parallelisation