Event generation in Julia and the path to GPUs

Compute Accelerator Forum

March 8th, 2022 // Uwe Hernandez Acosta

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?

CASUS
CENTER FOR ADVANCED
SYSTEMS UNDERSTANDING

- 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
  - scientific software development

Contents

CASUS
CENTER FOR ADVANCED
SYSTEMS UNDERSTANDING

Motivation

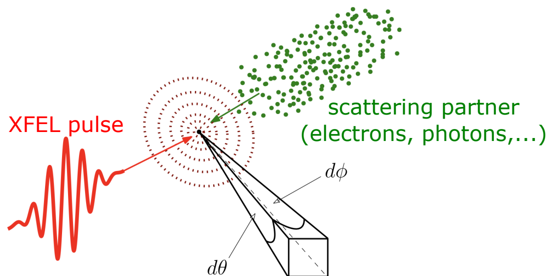
`QED.jl` - current status

Used technologies

Part I: Motivation

Motivation: our physics use-case

CASUS
CENTER FOR ADVANCED
SYSTEMS UNDERSTANDING



Phenomena

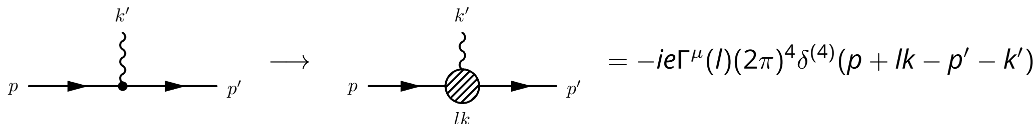
- multi-photon scattering
- non-perturbative effects
- unstable vacuum effects
- electromagnetic cascades

Applications

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

Strong-field quantum electrodynamics

- Feynman-rule: vertex



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

- vertex function

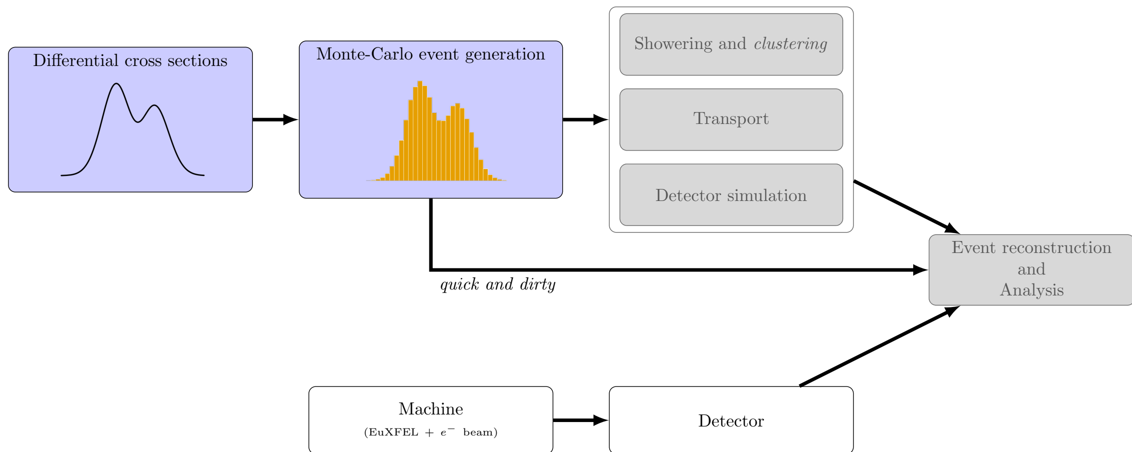
$$\Gamma^{\mu}(l, p, p', k) = \Gamma_0^{\mu} B_0(l) + \Gamma_1^{\mu\nu} B_{1\nu}(l) + \Gamma_2^{\mu} B_2(l)$$

- 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]

Particle-physics-like simulation workflow

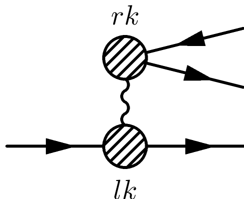


[Stefan Gieseke - MCnet Vietnam summer school (2019)]

Motivation: Computing tasks

CASUS CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

matrix elements



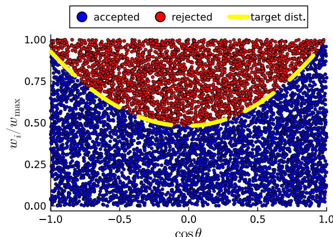
- matrix multiplications
- oscillatory integrals
- diagram structures

total cross sections

$$\sigma = \int_\Pi \frac{d\sigma}{d\Phi_n} d\Phi_n$$

- high-dim. integrals
- adaptive Monte-Carlo
- math. optimization

event generation

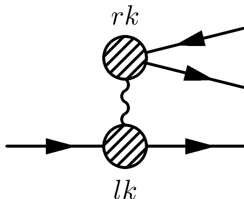


- multi-variate distributions
- fast sample drawing
- no false-positives

Motivation: Computing tasks

CASUS
CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

matrix elements



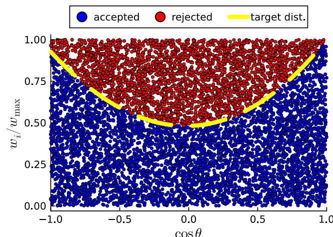
- matrix multiplications
- oscillatory integrals
- diagram structures

total cross sections

$$\sigma = \int_\Pi \frac{d\sigma}{d\Phi_n} d\Phi_n$$

- high-dim. integrals
- adaptive Monte-Carlo
- math. optimization

event generation



- multi-variate distributions
- fast sample drawing
- no false-positives

we need all of that – but parallelized!

Part II: `QED.jl` - current status

`QED.jl` - Strong-field particle physics code

[https://github.com/QEDjl-project]

Requirements

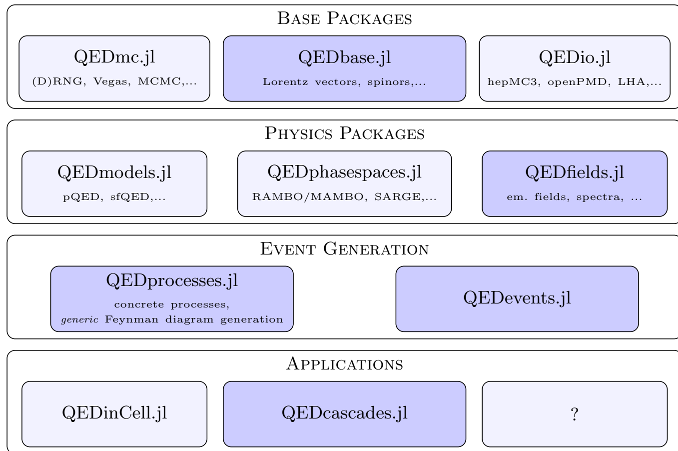
- open source
- written in Julia
- user-friendly
- modularised
- extensible
- performant
- CPU + GPU

QED.jl - Strong-field particle physics code

[https://github.com/QEDjl-project]

Requirements

- open source
- written in Julia
- user-friendly
- modularised
- extensible
- performant
- CPU + GPU



QED.jl - Strong-field particle physics code

[https://github.com/QEDjl-project]

Requirements

- open source
- written in Julia
- user-friendly
- modularised
- extensible
- performant
- CPU + GPU

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
- ?

⟶ **not** restricted to quantum electrodynamics

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

CASUS CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

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

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

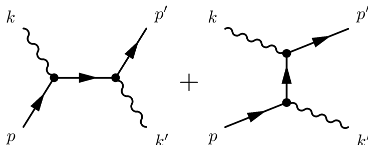
```
meets(a::Electron, b::Positron) = "attract one other"
meets(a::Positron, b::Electron) = "attract one other"
meets(a::T, b::T) where {T<:AbstractParticle} = "repel one other"
```

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

2. Interaction vertices

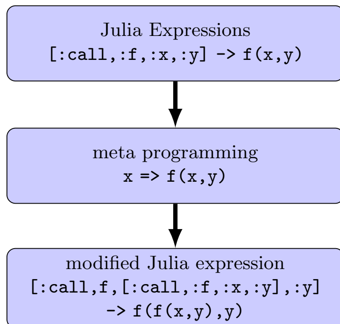
```
vertex1 = interact(pertQED(), init_electron, init_photon)   # (gamma*eps)*u
vertex2 = interact(pertQED(), init_electron, out_photon)    # (gamma*eps')*u
```

3. Propagator and the whole diagrams

```
diagram1 = interact(pertQED(), out_electron, out_photon,  vertex1)
diagram2 = interact(pertQED(), out_electron, init_photon, vertex2)
diagram  = diagram1 + diagram2
```

meta programming - introduction

CASUS
CENTER FOR ADVANCED SYSTEMS UNDERSTANDING



- Julia code is a data type in Julia itself
- one can use Julia to manipulate/generate Julia code before compilation
- Julia provides three types of metaprogramming
  - 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

Example: sample drawing – CPU version

Neural importance sampling (using `Flux.jl`)

[Müller et al. ACM ToG 38.5 (2019), Bothmann et al. SciPost Phys. 8 (2020) 4]

CASUS CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

```julia
using Flux, Random, QuasiMonteCarlo, BSON

target_dist(x,y) = exp(-x^2 - y^2)
N_train, N_perbatch, N_epochs, N_binsperaxis = 1024, 32, 10, 100

cl1 = CouplingLayer(2, 1, N_binsperaxis, false)
cl2 = CouplingLayer(2, 1, N_binsperaxis, true)

model = Chain(cl1, cl2)
```

```julia
x_train = rand(Float32, 2, N_train)
loader = Flux.DataLoader(x_train, batchsize=N_perbatch, shuffle=true)
opt_state = Flux.setup(Adam(0.005), model)

for data in loader
    val, grads = Flux.withgradient(m-> pearsonχ2divergence(m, transform,
          target_dist, data), model)
    Flux.update!(opt_state, model, grads[1])
end
BSON.@save "trained_model_cpu.bson" model
```

```
$ export JULIA_NUM_THREADS=6
$ julia nis_cpu.jl
```

[WIP: Tom Jungnickel]

Example: sample drawing – GPU version

Neural importance sampling (using `Flux.jl`)

[Müller et al. ACM ToG 38.5 (2019), Bothmann et al. SciPost Phys. 8 (2020) 4]

CASUS
CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

```julia
using Flux, Random, QuasiMonteCarlo, BSON, CUDA

target_dist(x,y) = exp(-x^2 - y^2)
N_train, N_perbatch, N_epochs, N_binsperaxis = 1024, 32, 10, 100

cl1 = CouplingLayer(2, 1, N_binsperaxis, false)
cl2 = CouplingLayer(2, 1, N_binsperaxis, true)

model = Chain(cl1, cl2) |> gpu
```

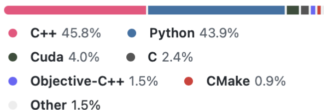
```julia
x_train = rand(Float32, 2, N_train) |> gpu
loader = Flux.DataLoader(x_train, batchsize=N_perbatch, shuffle=true)
opt_state = Flux.setup(Adam(0.005), model)

for data in loader
    val, grads = Flux.withgradient(m-> pearsonχ2divergence(m, transform,
          target_dist, data), model)
    Flux.update!(opt_state, model, grads[1])
end
BSON.@save "trained_model_gpu.bson" model |> cpu
```

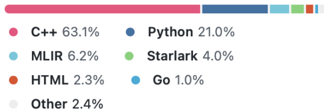
```
$ julia nis_gpu.jl
```

[WIP: Tom Jungnickel]

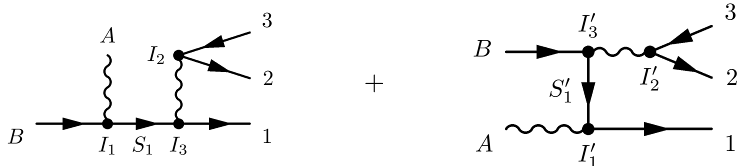
Library-level parallelism

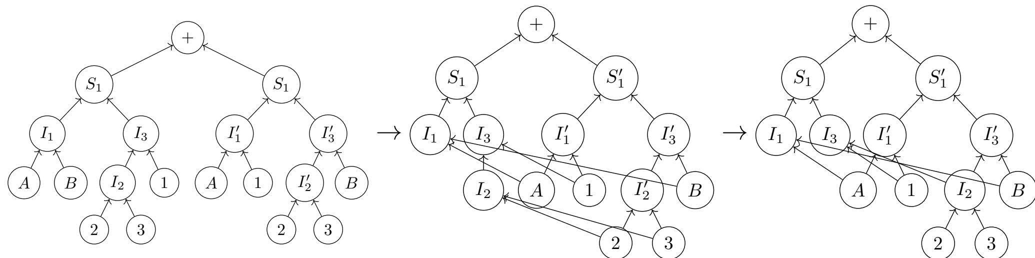
Library-level parallelism





High-level parallelism - directed acyclic graphs



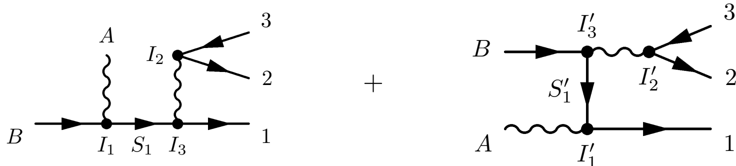


[A Kanaki, C G Papadopoulos - Comput.Phys.Commun. 132 (2000)]

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

High-level parallelism - DAG evalulation [Dagger.jl, DaggerGPU.jl]

CASUS
CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

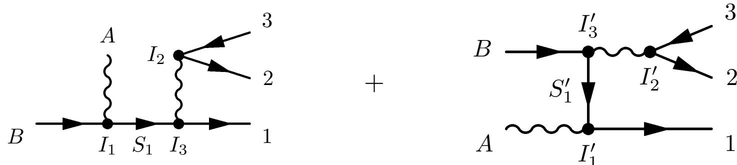



```
pA      = @spawn Particle(Photon()   , Incoming(), mom_A)
pB      = @spawn Particle(Electron(), Incoming(), mom_B)
p1      = @spawn Particle(Electron(), Outgoing(), mom_1)
p2      = @spawn Particle(Electron(), Outgoing(), mom_2)
p3      = @spawn Particle(Positron(), Outgoing(), mom_3)
I1      = @spawn interact(pertQED(), pA, pB)
I1prime = @spawn interact(pertQED(), pA, p1)
I2      = @spawn interact(pertQED(), p2, p3)
D1      = @spawn interact(pertQED(), I2, I1, p1)
D2      = @spawn interact(pertQED(), I2, I1prime, pB)
res     = @spawn D1 + D2
fetch(res)
```

```
$ julia --threads 8 dag.jl
```

High-level parallelism - DAG evaluation [Dagger.jl, DaggerGPU.jl]

CASUS CENTER FOR ADVANCED SYSTEMS UNDERSTANDING




```
pA      = @spawn Particle(Photon()  , Incoming(), mom_A)
pB      = @spawn Particle(Electron(), Incoming(), mom_B)
p1      = @spawn Particle(Electron(), Outgoing(), mom_1)
p2      = @spawn Particle(Electron(), Outgoing(), mom_2)
p3      = @spawn Particle(Positron(), Outgoing(), mom_3)
I1      = @spawn interact(pertQED(), pA, pB)
I1prime = @spawn interact(pertQED(), pA, p1)
I2      = @spawn interact(pertQED(), p2, p3)
D1      = @spawn interact(pertQED(), I2, I1, p1)
D2      = @spawn interact(pertQED(), I2, I1prime, pB)
res     = @spawn D1 + D2
fetch(res)
```

```
$ julia --threads 8 dag.jl
```

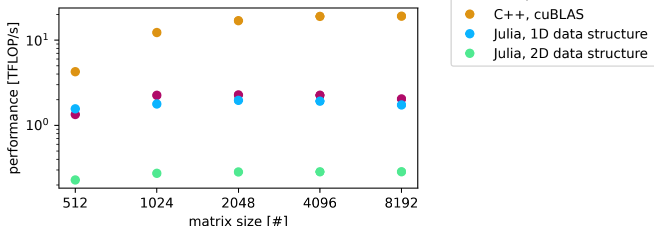
⟶ Under the hood: DAG optimization + scheduling **for the specific hardware architecture**

[WIP: Anton Reinhard]

Low-level parallelism

matmul – `CUDA/C++` vs `CUDA.jl` on A100 [WIP: Anton Reinhard, Prof. W. Nagel]

```julia
using CUDA
function matmul_kernel!(a, b, c, n)
    row = (blockIdx().y - 1) * blockDim().y + threadIdx().y - 1
    col = (blockIdx().x - 1) * blockDim().x + threadIdx().x - 1
    (row >= n || col >= n) ? return : nothing
    sum = zero(eltype(a))
    @inbounds for i = 0:n - 1
        sum += a[row * n + i + 1] * b[i * n + col + 1]
    end
    c[row * n + col + 1] = sum
    return
end
```



Low-level paralellism
Rodinia Benchmarks [Che et al. Proc. IISWC 2009.]

CASUS
CENTER FOR ADVANCED
SYSTEMS UNDERSTANDING

speedup vs CUDA C

1%
0%
−2%
−4%

−0.50%

Rodinia
geomean

nn
backprop
nw
bfs
streamcluster
hotspot
particlefilter
pathfinder
leukocyte
lud

[Besard et al. IEEE Trans. Parallel Distrib. Syst. 30.4 (2018)]

Hardware-agnostic parallelism
KernelAbstractions.jl

CUDA.jl
AMDGPU.jl
OneAPI.jl
} KernelAbstractions.jl

```
using KernelAbstractions
@kernel function matmul_kernel!(a, b, c)
    i, j = @index(Global, NTuple)

    # creating a temporary sum variable for matrix multiplication
    tmp_sum = zero(eltype(c))
    for k = 1:size(a)[2]
        tmp_sum += a[i,k] * b[k, j]
    end

    c[i,j] = tmp_sum
end
```

Dream about the future: zero overhead abstraction

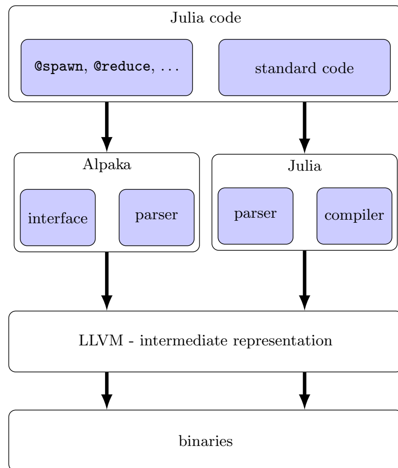
Julia/Alpaka integration [Zenker et al. IPDPSW, IEEE, 2016.]

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

CASUS
CENTER FOR ADVANCED SYSTEMS UNDERSTANDING

- Event generation will be computationally expensive
  - $\Rightarrow$ needs to be parallelized
- New library `QED.jl`
  - $\Rightarrow$ first principal calculations in (strong-field) QED
  - $\Rightarrow$ written in Julia
  - $\Rightarrow$ modularised + extensible
  - $\Rightarrow$ CPU + GPU
- Julia in general
  - $\Rightarrow$ "as easy as python, as fast as C/C++"
  - $\Rightarrow$ multiple dispatch + meta-programming
- Distributed computing on GPU
  - $\Rightarrow$ Library support
  - $\Rightarrow$ High-level optimisations for heterogeneous architectures
  - $\Rightarrow$ Low-level GPU programming (CUDA, ROCm, OneAPI)
  - $\Rightarrow$ hardware-agnostic parallelisation (KernelAbstractions.jl, Alpaka)

- Collaborators

Michael Bussmann
Klaus Steiniger
Simeon Ehrig
Jiri Vyskocil
Tom Jungnickel
Anton Reinhard

Prof. Burkhard Kämpfer
Prof. Thomas Cowan
René Widera

- Acknowledgements:

F. Siegert (TUD)

A. Cangi (CASUS)
Lokamani (CASUS)

Prof. W. Nagel (TUD)
Prof. J. Castrillon (TUD)

Backup

Muliple-dispatch – the inner sum

credits: Stefan Karpinski [S Karpinski - JuMP-dev2019]


```julia
using LinearAlgebra, BenchmarkTools

function inner_sum(A,vs)
    s = zero(eltype(A))
    for v in vs
        s += inner(v,A,v) # dispatch here
    end
    return s
end

inner(v,A,w) = dot(v,A*w) # much generic!
```

```julia
A = rand(10,10)
vs = [rand(10) for _ in 1:4]

@benchmark inner_sum($A,$vs)
```

```
BenchmarkTools.Trial: 10000 samples with 198 evaluations.
 Range (min … max):  454.126 ns … 1.666 μs  ┊ GC (min … max):
0.00% … 70.26%
 Time  (median):     464.015 ns             ┊ GC (median):
0.00%
 Time  (mean ± σ):   474.009 ns ± 57.482 ns ┊ GC (mean ± σ):  0
.56% ±  3.41%
```

Muliple-dispatch – the inner sum

One-hot vector: $v = (0, \ldots, 0, 1, 0, \ldots, 0)$

```
struct OneHotVector <: AbstractVector{Bool}
    hot::Int64
    len::Int64
end
```

```
Base.size(v::OneHotVector) = (v.len,)
Base.getindex(v::OneHotVector,idx::Integer) = (idx==v.hot)
```

```
A = rand(10,10)
one_hots = [OneHotVector(rand(1:10),10) for _ in 1:4]

@benchmark inner_sum($A,$one_hots)
```

```
BenchmarkTools.Trial: 10000 samples with 313 evaluations.
 Range (min … max):  270.367 ns …  1.844 μs  ┊ GC (min … max):
0.00% … 0.00%
 Time  (median):     292.198 ns              ┊ GC (median):
0.00%
 Time  (mean ± σ):   296.943 ns ± 56.244 ns  ┊ GC (mean ± σ):  1
.03% ± 4.47%
```

$\rightarrow$ uses just generic fallbacks!

Muliple-dispatch – the inner sum

Specialize the inner sum

$$v * A * w = \sum_{m=1}^{N} \sum_{n=1}^{N} v_m A_{mn} w_n \overset{\text{one-hot}}{=} A_{m_{\text{hot}} n_{\text{hot}}}$$

```
function inner(v::OneHotVector,A::AbstractMatrix, w::OneHotVector)
    return A[v.hot,w.hot]
end
```

```
A = rand(10,10)
one_hots = [OneHotVector(rand(1:10),10) for _ in 1:4]

@benchmark inner_sum($A,$one_hots)
```

```
BenchmarkTools.Trial: 10000 samples with 1000 evaluations.
 Range (min … max):  4.250 ns … 18.208 ns  ┊ GC (min … max): 0.
00% … 0.00%
 Time  (median):     4.375 ns              ┊ GC (median):    0.
00%
 Time  (mean ± σ):   4.458 ns ±  0.259 ns  ┊ GC (mean ± σ):  0.0
0% ± 0.00%
```

→ can be arbitrary efficient