DRESDEN concept

HZDR HELMHOLTZ ZENTRUM DRESDEN ROSSENDORF

DAG Optimizations for Feynman Diagrams of High-Multiplicity Scattering Processes in Julia

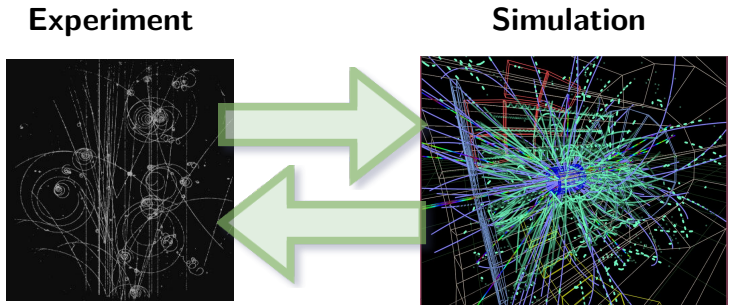
Anton Reinhard[1,2], Simeon Ehrig[1,2], Uwe Hernandez Acosta[1,2], René Widera[1]

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

09.11.2023

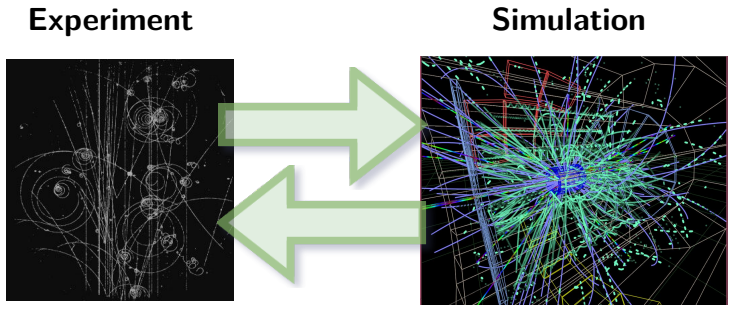
Goal

- Evaluate large QED-processes (superfactorial scaling)



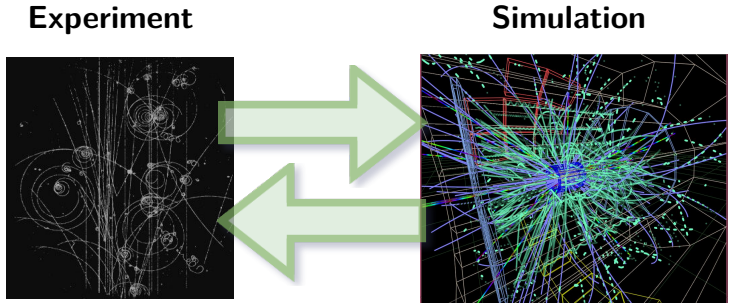
Goal

- Evaluate large QED-processes (superfactorial scaling)
- For this, generate optimized code for a specific machine

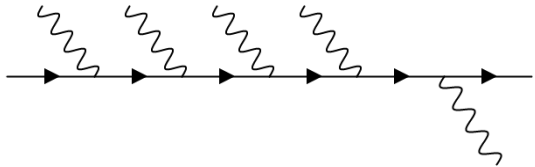


Goal

- Evaluate large QED-processes (superfactorial scaling)
- For this, generate optimized code for a specific machine
- Use high-level knowledge about the problem



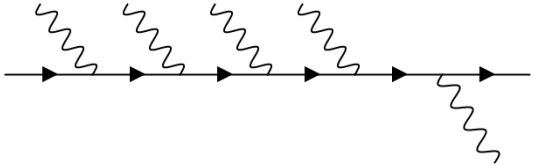
Motivation



State of the art implementations fail to

- employ strategies from computer science

Motivation



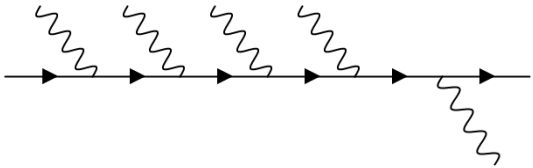
State of the art implementations fail to

- employ strategies from computer science

We try to instead

$\Rightarrow$ use graph representation for high-level optimizations

Motivation



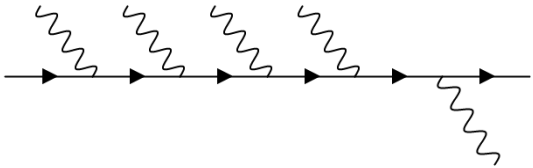
State of the art implementations fail to

- employ strategies from computer science
- use hardware for larger processes

We try to instead

$\Rightarrow$ use graph representation for high-level optimizations

Motivation



State of the art implementations fail to

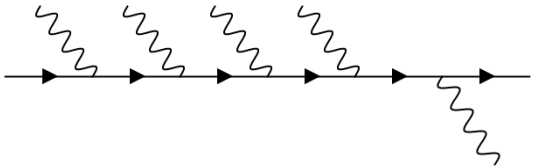
- employ strategies from computer science
- use hardware for larger processes

We try to instead

$\Rightarrow$ use graph representation for high-level optimizations

$\Rightarrow$ scale the code with the process

Motivation



State of the art implementations fail to

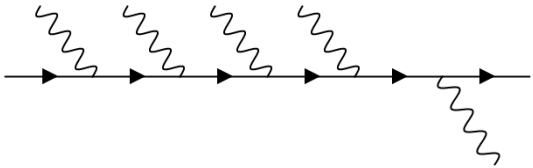
- employ strategies from computer science
- use hardware for larger processes
- be platform independent and portable

We try to instead

$\Rightarrow$ use graph representation for high-level optimizations

$\Rightarrow$ scale the code with the process

Motivation



State of the art implementations fail to

- employ strategies from computer science
- use hardware for larger processes
- be platform independent and portable

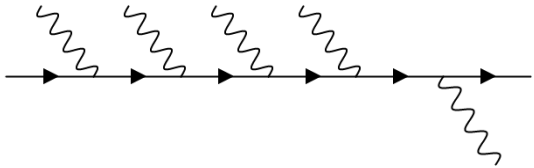
We try to instead

⇒ use graph representation for high-level optimizations

⇒ scale the code with the process

⇒ support multiple platforms (CPU, GPU) with generic code

Motivation



State of the art implementations fail to

- employ strategies from computer science
- use hardware for larger processes
- be platform independent and portable
- use heterogeneous architectures

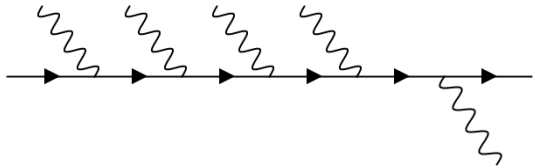
We try to instead

⇒ use graph representation for high-level optimizations

⇒ scale the code with the process

⇒ support multiple platforms (CPU, GPU) with generic code

Motivation



State of the art implementations fail to

- employ strategies from computer science
- use hardware for larger processes
- be platform independent and portable
- use heterogeneous architectures

We try to instead

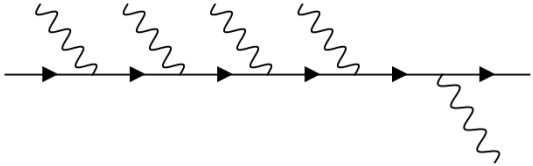
⇒ use graph representation for high-level optimizations

⇒ scale the code with the process

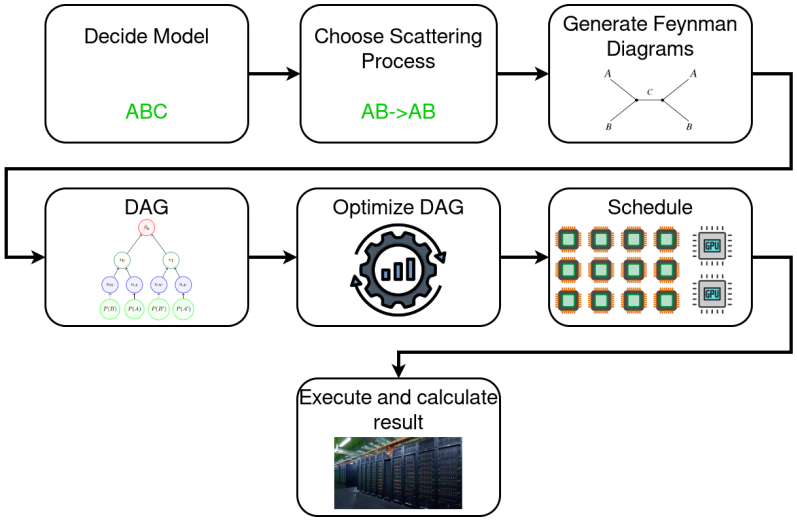
⇒ support multiple platforms (CPU, GPU) with generic code

⇒ benefit from all available hardware

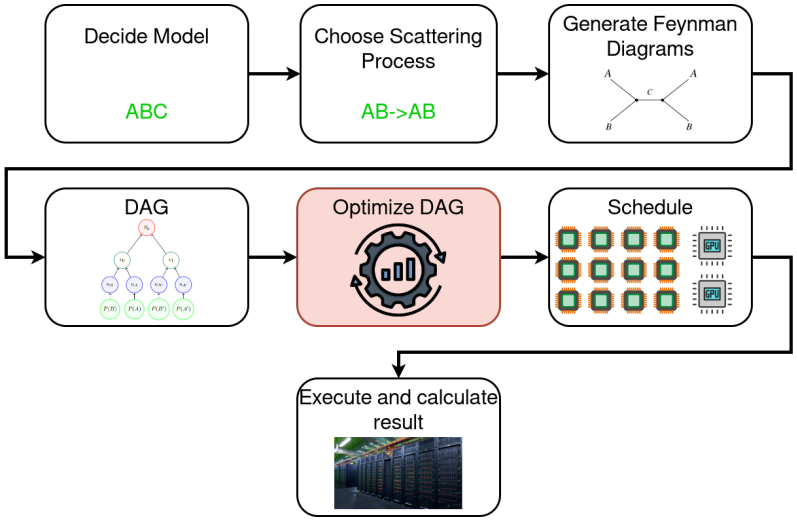
Motivation



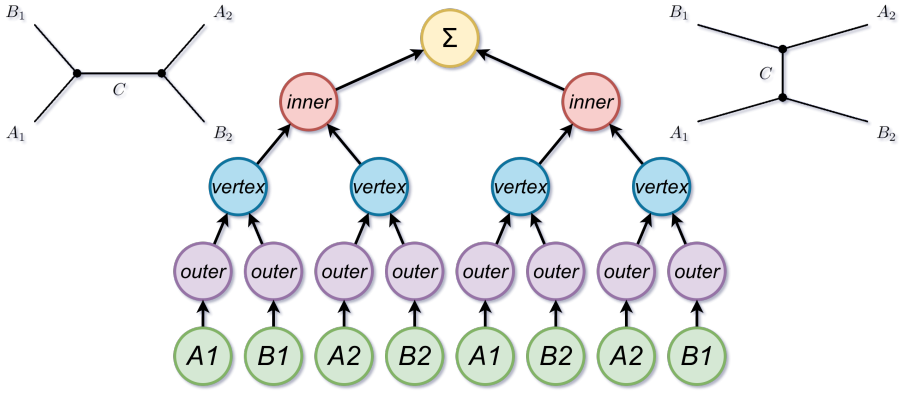
The Pipeline - Top Down



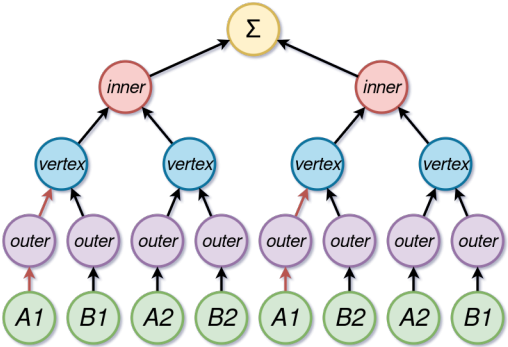
The Pipeline - Top Down

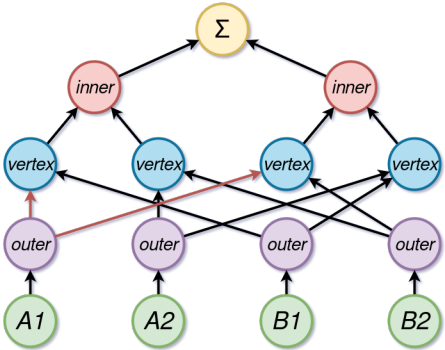


The Pipeline - The (Naive) Directed Acyclic Graph (DAG)



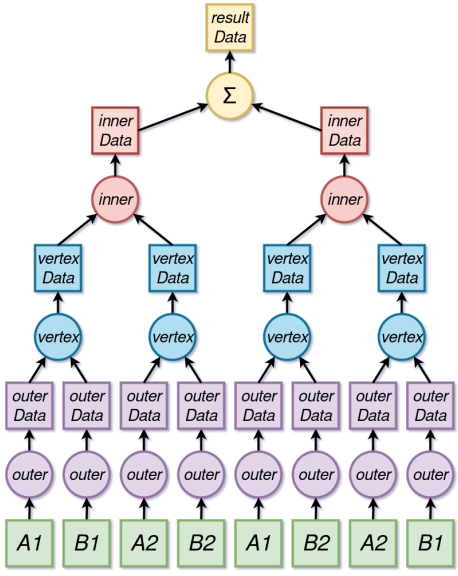
The Pipeline - The (Naive) DAG, Reduced





Optimize the DAG

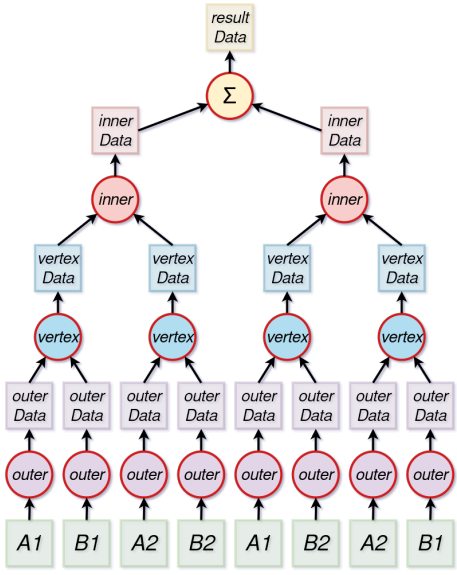
What are we optimizing?



Optimize the DAG

What are we optimizing?

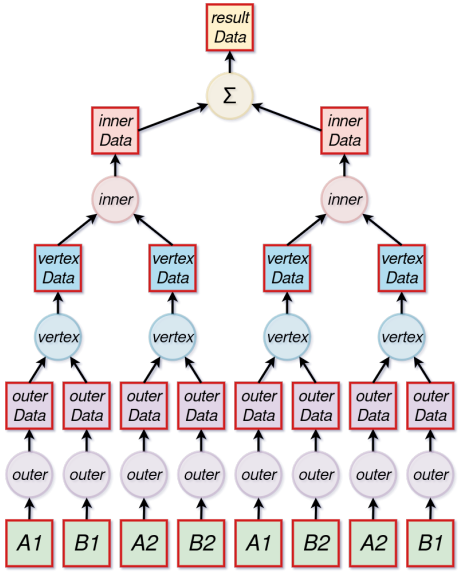
- Compute Effort



Optimize the DAG

What are we optimizing?

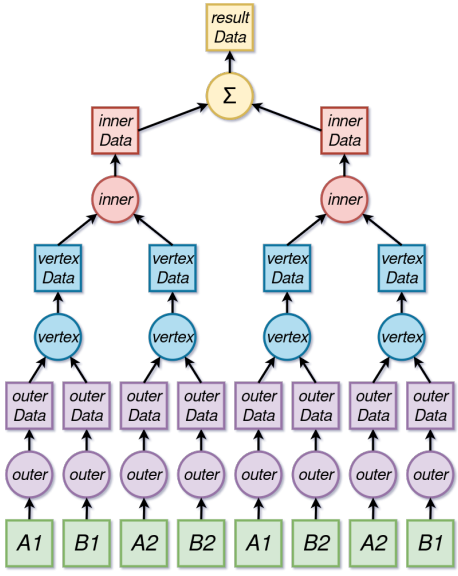
- Compute Effort
- Data Transfer



Optimize the DAG

What are we optimizing?

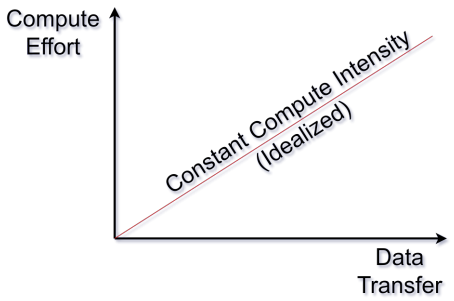
- Compute Effort
- Data Transfer
- Compute Intensity $= \frac{\text{Compute Effort}}{\text{Data Transfer}}$

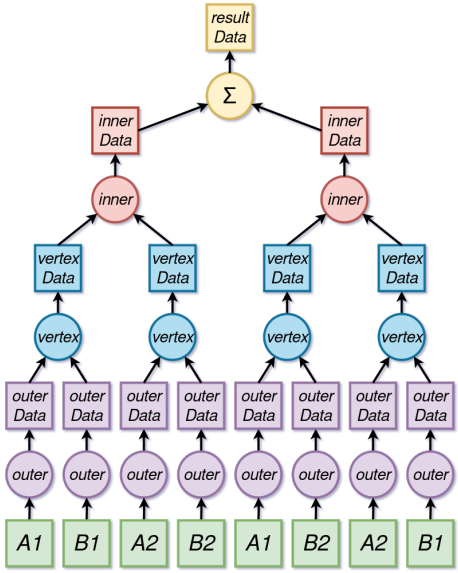


Optimize the DAG

What are we optimizing?

- Compute Effort
- Data Transfer
- Compute Intensity $= \frac{\text{Compute Effort}}{\text{Data Transfer}}$

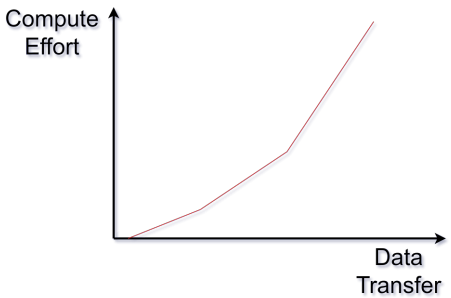


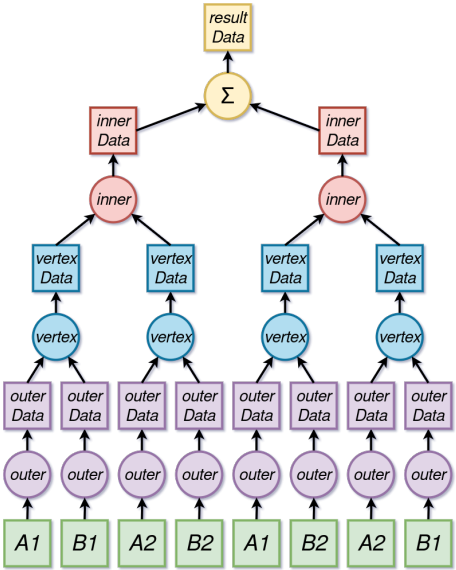


Optimize the DAG

What are we optimizing?

- Compute Effort
- Data Transfer
- Compute Intensity $= \frac{\text{Compute Effort}}{\text{Data Transfer}}$





Implementation - Structure

Several Components:

Graph and Operations

Implementation - Structure

Several Components:

Graph and Operations

Optimizer

Implementation - Structure

Several Components:

Graph and Operations

Optimizer

Devices

Implementation - Structure

Several Components:

Graph and Operations

Optimizer

Devices

Cost Estimators

Implementation - Structure

Several Components:

Graph and Operations

☑ DAG

Optimizer

Devices

Cost Estimators

Implementation - Structure

Several Components:

Graph and Operations

- ☑ DAG
- ☑ Node Fusion

Optimizer

Devices

Cost Estimators

Implementation - Structure

Several Components:

Graph and Operations

- ☑ DAG
- ☑ Node Fusion
- ☑ Node Reduction

Optimizer

Devices

Cost Estimators

Implementation - Structure

Several Components:

Graph and Operations

- ☑ DAG
- ☑ Node Fusion
- ☑ Node Reduction
- ☑ Node Split

Devices

Optimizer

Cost Estimators

Implementation - Structure

Several Components:

Graph and Operations

- ☑ DAG
- ☑ Node Fusion
- ☑ Node Reduction
- ☑ Node Split

Devices

Optimizer

- ☐ Greedy optimizer

Cost Estimators

Implementation - Structure

Several Components:

Graph and Operations

- ☑ DAG
- ☑ Node Fusion
- ☑ Node Reduction
- ☑ Node Split

Devices

Optimizer

- ☐ Greedy optimizer
- ☐ Simulated Annealing

Cost Estimators

Implementation - Structure

Several Components:

Graph and Operations

- ☑ DAG
- ☑ Node Fusion
- ☑ Node Reduction
- ☑ Node Split

Devices

- ☑ NUMA Nodes

Optimizer

- ☐ Greedy optimizer
- ☐ Simulated Annealing

Cost Estimators

Implementation - Structure

Several Components:

Graph and Operations
- ☑ DAG
- ☑ Node Fusion
- ☑ Node Reduction
- ☑ Node Split

Devices
- ☑ NUMA Nodes
- ☐ Nvidia GPUs

Optimizer
- ☐ Greedy optimizer
- ☐ Simulated Annealing

Cost Estimators

Implementation - Structure

Several Components:

Graph and Operations

- ☑ DAG
- ☑ Node Fusion
- ☑ Node Reduction
- ☑ Node Split

Devices

- ☑ NUMA Nodes
- ☐ Nvidia GPUs
- ☐ AMD GPUs

Optimizer

- ☐ Greedy optimizer
- ☐ Simulated Annealing

Cost Estimators

Implementation - Structure

Several Components:

Graph and Operations
- ☑ DAG
- ☑ Node Fusion
- ☑ Node Reduction
- ☑ Node Split

Devices
- ☑ NUMA Nodes
- ☐ Nvidia GPUs
- ☐ AMD GPUs

Optimizer
- ☐ Greedy optimizer
- ☐ Simulated Annealing

Cost Estimators
- ☑ Global Compute and Data

Implementation - Structure

Several Components:

Graph and Operations
- ☑ DAG
- ☑ Node Fusion
- ☑ Node Reduction
- ☑ Node Split

Devices
- ☑ NUMA Nodes
- ☐ Nvidia GPUs
- ☐ AMD GPUs

Optimizer
- ☐ Greedy optimizer
- ☐ Simulated Annealing

Cost Estimators
- ☑ Global Compute and Data
- ☐ Scheduled time estimation

Implementation - Structure

Several Components:

Graph and Operations
- ☑ DAG
- ☑ Node Fusion
- ☑ Node Reduction
- ☑ Node Split

Devices
- ☑ NUMA Nodes
- ☐ Nvidia GPUs
- ☐ AMD GPUs

Optimizer
- ☐ Greedy optimizer
- ☐ Simulated Annealing

Cost Estimators
- ☑ Global Compute and Data
- ☐ Scheduled time estimation
- ☐ Microbenchmark-based

Implementation - Structure

Several Components:

Graph and Operations
- ☑ DAG
- ☑ Node Fusion
- ☑ Node Reduction
- ☑ Node Split

Devices
- ☑ NUMA Nodes
- ☐ Nvidia GPUs
- ☐ AMD GPUs
- ☐ etc.

Optimizer
- ☐ Greedy optimizer
- ☐ Simulated Annealing
- ☐ etc.

Cost Estimators
- ☑ Global Compute and Data
- ☐ Scheduled time estimation
- ☐ Microbenchmark-based
- ☐ etc.

Implementation - Code Generation

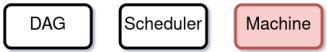
DAG

- Get graph, a scheduler, and machine information

Implementation - Code Generation

- Get graph, a scheduler, and machine information

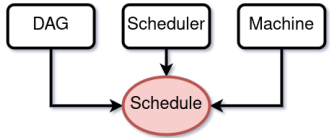
Implementation - Code Generation



- Get graph, a scheduler, and machine information

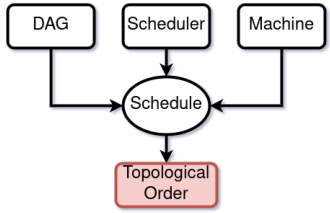
Implementation - Code Generation

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



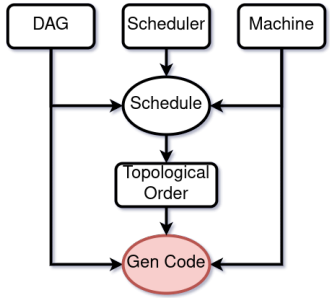
Implementation - Code Generation

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



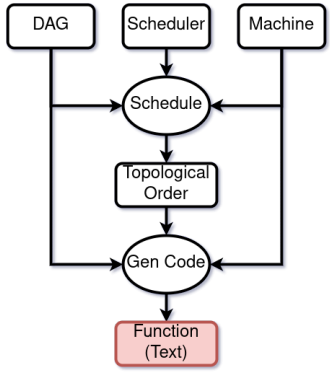
Implementation - Code Generation

- 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



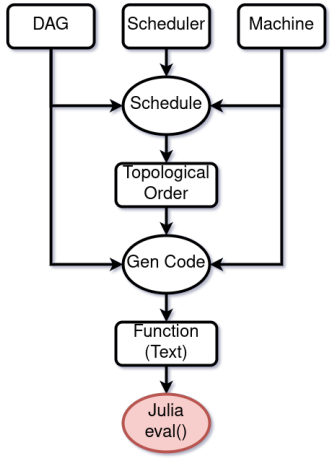
Implementation - Code Generation

- 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



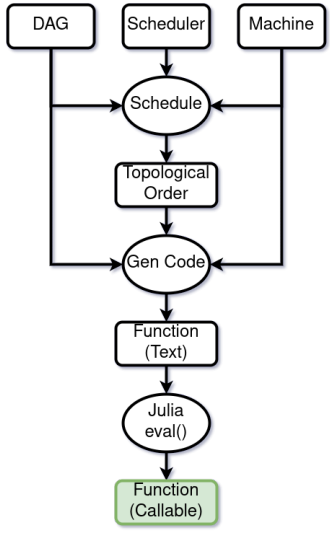
Implementation - Code Generation

- 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



Implementation - Code Generation

- 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



Results - Benchmark AB→ABBB - Proof of Concept

```
julia> bench_compute = @benchmark compute_function.(input_data)
BenchmarkTools.Trial: 960 samples with 1 evaluat
ion.
 Range (min … max):  4.202 ms … 10.953 ms  ┊ GC (min … max):  0.00% … 55.28%
 Time  (median):     4.558 ms              ┊ GC (median):     0.00%
 Time  (mean ± σ):   5.212 ms ±  1.636 ms  ┊ GC (mean ± σ):  12.30% ± 17.24%

  4.2 ms         Histogram: frequency by time        9.52 ms <

 Memory estimate: 6.20 MiB, allocs estimate: 145010.

julia> bench_compute_reduced = @benchmark compute_function_reduced.(input_data)
BenchmarkTools.Trial: 1158 samples with 1 evaluation.
 Range (min … max):  3.466 ms … 9.212 ms  ┊ GC (min … max):  0.00% … 49.98%
 Time  (median):     3.749 ms             ┊ GC (median):     0.00%
 Time  (mean ± σ):   4.317 ms ± 1.510 ms  ┊ GC (mean ± σ):  12.68% ± 17.75%

  3.47 ms         Histogram: frequency by time        8.67 ms <

 Memory estimate: 5.29 MiB, allocs estimate: 125010.

julia> judge(median(bench_compute_reduced), median(bench_compute))
BenchmarkTools.TrialJudgement:
  time:   -17.75% => improvement (5.00% tolerance)
  memory: -14.76% => improvement (1.00% tolerance)
```

Results - Benchmark AB→ABBB - Proof of Concept

```
julia> bench_compute = @benchmark compute_function.(input_data)
BenchmarkTools.Trial: 960 samples with 1 evaluat
ion.
 Range (min … max):  4.202 ms … 10.953 ms  ┊ GC (min … max):  0.00% … 55.28%
 Time  (median):     4.558 ms              ┊ GC (median):     0.00%
 Time  (mean ± σ):   5.212 ms ±  1.636 ms  ┊ GC (mean ± σ):  12.30% ± 17.24%

  4.2 ms          Histogram: frequency by time          9.52 ms <

 Memory estimate: 6.20 MiB, allocs estimate: 145010.

julia> bench_compute_reduced = @benchmark compute_function_reduced.(input_data)
BenchmarkTools.Trial: 1158 samples with 1 evaluation.
 Range (min … max):  3.466 ms … 9.212 ms  ┊ GC (min … max):  0.00% … 49.98%
 Time  (median):     3.749 ms             ┊ GC (median):     0.00%
 Time  (mean ± σ):   4.317 ms ± 1.510 ms  ┊ GC (mean ± σ):  12.68% ± 17.75%

  3.47 ms         Histogram: frequency by time          8.67 ms <

 Memory estimate: 5.29 MiB, allocs estimate: 125010.

julia> judge(median(bench_compute_reduced), median(bench_compute))
BenchmarkTools.TrialJudgement:
  time:   -17.75% => improvement (5.00% tolerance)
  memory: -14.76% => improvement (1.00% tolerance)
```

Results - Benchmark AB→ABBB - Proof of Concept

```
julia> bench_compute = @benchmark compute_function.(input_data)
BenchmarkTools.Trial: 960 samples with 1 evaluat
ion.
 Range (min … max):   4.202 ms … 10.953 ms  ┊ GC (min … max):  0.00% … 55.28%
 Time  (median):      4.558 ms              ┊ GC (median):     0.00%
 Time  (mean ± σ):    5.212 ms ±  1.636 ms  ┊ GC (mean ± σ):  12.30% ± 17.24%

  4.2 ms         Histogram: frequency by time        9.52 ms <

 Memory estimate: 6.20 MiB, allocs estimate: 145010.

julia> bench_compute_reduced = @benchmark compute_function_reduced.(input_data)
BenchmarkTools.Trial: 1158 samples with 1 evaluation.
 Range (min … max):   3.466 ms … 9.212 ms  ┊ GC (min … max):  0.00% … 49.98%
 Time  (median):      3.749 ms              ┊ GC (median):     0.00%
 Time  (mean ± σ):    4.317 ms ± 1.510 ms   ┊ GC (mean ± σ):  12.68% ± 17.75%

  3.47 ms        Histogram: frequency by time        8.67 ms <

 Memory estimate: 5.29 MiB, allocs estimate: 125010.

julia> judge(median(bench_compute_reduced), median(bench_compute))
BenchmarkTools.TrialJudgement:
  time:   -17.75% => improvement (5.00% tolerance)
  memory: -14.76% => improvement (1.00% tolerance)
```

Results - Benchmark AB→ABBB - Proof of Concept

```
julia> bench_compute = @benchmark compute_function.(input_data)
BenchmarkTools.Trial: 960 samples with 1 evaluat
ion.
 Range (min … max):   4.202 ms … 10.953 ms  ┊ GC (min … max):  0.00% … 55.28%
 Time  (median):      4.558 ms              ┊ GC (median):     0.00%
 Time  (mean ± σ):    5.212 ms ±  1.636 ms  ┊ GC (mean ± σ):  12.30% ± 17.24%

  4.2 ms         Histogram: frequency by time        9.52 ms <

 Memory estimate: 6.20 MiB, allocs estimate: 145010.

julia> bench_compute_reduced = @benchmark compute_function_reduced.(input_data)
BenchmarkTools.Trial: 1158 samples with 1 evaluation.
 Range (min … max):   3.466 ms … 9.212 ms  ┊ GC (min … max):  0.00% … 49.98%
 Time  (median):      3.749 ms              ┊ GC (median):     0.00%
 Time  (mean ± σ):    4.317 ms ± 1.510 ms   ┊ GC (mean ± σ):  12.68% ± 17.75%

  3.47 ms         Histogram: frequency by time        8.67 ms <

 Memory estimate: 5.29 MiB, allocs estimate: 125010.

julia> judge(median(bench_compute_reduced), median(bench_compute))
BenchmarkTools.TrialJudgement:
  time:   -17.75% => improvement (5.00% tolerance)
  memory: -14.76% => improvement (1.00% tolerance)
```

Outlook

The basic code structure is done, now:

- Add QED Model
- Compare different optimization algorithms and estimators
- Determine a machine's scaling functions and working point graph using microbenchmarks

Acknowledgements

Collaborators:

- **Dr. Uwe Hernandez Acosta**[1,2]
- **Simeon Ehrig**[1,2]
- **René Widera**[2]

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

References

[1] 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.

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

[3] John Clark and Derek Allan Holton. *A first look at graph theory*. Reprint. Hackensack [u.a.]: World Scientific, 2005. ISBN: 9789810204891. URL: http://slubdd.de/katalog?TN_libero_mab2.

[4] John A Hartigan and Manchek A Wong. "Algorithm AS 136: A k-means clustering algorithm". In: *Journal of the royal statistical society. series c (applied statistics)* 28.1 (1979), pp. 100–108.

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

[6] Jinhong Luo et al. "Learning to optimize dag scheduling in heterogeneous environment". In: *arXiv preprint arXiv:2103.06980* (2021).

[7] Andrea Valassi et al. "Design and engineering of a simplified workflow execution for the MG5aMC event generator on GPUs and vector CPUs". In: *EPJ Web of Conferences*. Vol. 251. EDP Sciences, 2021, p. 03045.

Backup

```julia
using MetagraphOptimization

# get our machine's info
machine = get_machine_info()

# create a model identifier
model = ABCModel()

# create a process in our model
process = parse_process("AB->ABBB", model)

# read the graph (of the same process) from a file
graph = parse_dag("../input/AB->ABBB.txt", model)

# generate some random input data for our process
input_data = gen_process_input(process)

# "compile" a compute function
compute = get_compute_function(graph, process, machine)

# compute a result using the generated function
result = compute(input_data)
```

```
Found 1 NUMA nodes
CUDA is non-functional
-2.5102328686552435e-14
```