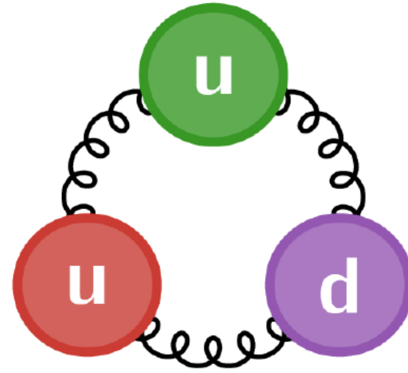


Machine Learning in Julia for Calorimeter Showers

Daniel Regado

JuliaHEP 2024

Graeme Stewart **Pere Mato** **Piyush Raikwar**

Google Summer of Code 2024

Investigate maturity of ML development with Julia and compare ease of use and performance against current popular solutions.

ML development is dominated by Python frameworks with core functionality implemented in C++ and CUDA.

Review of CaloChallenge to find most desired model as main subject for implementation.

Mentored by:

- Graeme Stewart
- Pere Mato
- Piyush Raikwar

Google Summer of Code

Final Report • Repository

CaloChallenge

Aimed to spur the development and benchmarking of fast and high-fidelity calorimeter shower generation using deep learning methods. It also released datasets and evaluation metrics, providing a common benchmark for ML methods.

Traditionally, these simulations are carried out using GEANT4, which represents a major computational bottleneck and is forecast to overwhelm the computing budget of LHC.

Submissions cover 4 architecture types:

- Diffusion (best fidelity, but slower)
- Normalizing Flows
- GANs
- Variational Autoencoders (worse fidelity, but faster)

CaloDiffusion

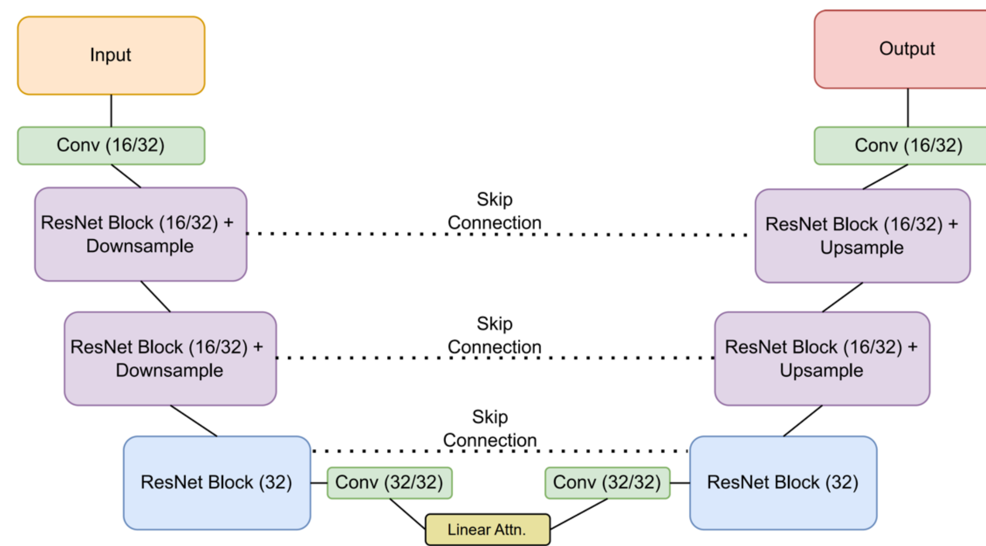
Denoising diffusion model to generate realistic energy showers. It works by gradually adding noise to data over many steps, predicting the noise during training, and then generates new data from random noise.

Cylindrical Convolutions

circular padding is added in the angular dimension, voxels close to the ends of the linear array properly interact with their angular neighbors on the opposite end.

Geometry Latent Mapping (GLaM)

learnable mapping from irregular data geometry to regular geometric structure. Used on datasets 1.



Architecture

Flux.jl

Library for machine learning, provides building blocks for complex models and training them.
CaloDiffusion required the following base implementations:

Layers

- **Dense**
- **Conv**
- **ConvTranspose**
- **GroupNorm**

Activations

- **Swish (SiLU)**
- **GELU**

Custom Layers

Julia

```julia
using Flux


struct ConvBlock
    conv::Conv
    norm::GroupNorm
end

function ConvBlock(dim_in::Int, dim_out::Int, groups::Int=8)
    ConvBlock(
        Conv((3,3,3), dim_in=>dim_out; pad=1),
        GroupNorm(dim_out, groups)
    )
end

Flux.@layer ConvBlock

(cb::ConvBlock)(x::AbstractArray) = cb.conv(x) |> cb.norm |> swish
```

Python

```python
import torch
import torch.nn as nn


class ConvBlock(nn.Module):

    def __init__(self, dim: int, dim_out: int, groups: int = 8,
                 cylindrical: bool = False):
        super().__init__()

        self.conv = nn.Conv3d(dim, dim_out, kernel_size=3, padding=1)
        self.norm = nn.GroupNorm(groups, dim_out)
        self.act = nn.SiLU()

    def forward(self, x: torch.Tensor):
        return self.act(self.norm(self.conv(x)))
```

U-Net Forward Pass

Julia

```julia
function (m::CondUnet)(x::AbstractArray, cond::AbstractArray,
         time::AbstractArray)

    conds = cat(m.timenet(time), m.condnet(cond); dims=1)
    reduceblocks = (out, block) -> block(out, conds)

    @_ x |> m.inconv
         |> reduce(reduceblocks, m.layers; init=__)
         |> m.mid(__, conds)
         |> reduce(reduceblocks, reverse(m.layers); init=__)
         |> m.outconv
end
```

Python

```python
def forward(self, x, cond, time):
    t = self.time_mlp(time)
    c = self.cond_mlp(cond)
    conditions = torch.cat([t,c], axis = -1)

    h = []
    x = self.init_conv(x)

    # Downsample
    for i, (block1, block2, downsample) in enumerate(self.downs):
        x = block1(x, conditions)
        x = block2(x, conditions)
        x = self.downs_attn[i](x)
        h.append(x)
        x = downsample(x)

    # Bottleneck
    x = self.mid_block1(x, conditions)
    x = self.mid_attn(x)
    x = self.mid_block2(x, conditions)

    # Upsample
    for i, (block1, block2, upsample) in enumerate(self.ups):
        s = h.pop()
        x = torch.cat((x, s), dim=1)
        x = block1(x, conditions)
        x = block2(x, conditions)
        x = self.ups_attn[i](x)
        x = upsample(x)

    return self.final_conv(x)
```

Tensor Operations – TensorCast.jl

Python

```
import torch
from einops import rearrange

q, k, v = map(lambda t: rearrange(t, "b (h c) x y z -> b h c (x y z)", h=self.n_heads), qkv)

c = torch.einsum("b h d n, b h e n -> b h d e", k, v)
```

LinearAttention layer used einops to rearrange tensor shape and torch for matrix multiplication.

Julia

```
using Flux, TensorCast


q, k, v = map((t -> @cast _[z⊗y⊗x, c, h, b] := t[z, y, x, c⊗h, b] h in 1:la.nheads), [q, k, v])

@reduce c[e, d, h, b] := sum(n) k[n, d, h, b] * v[n, e, h, b]
```

TensorCast.jl handles both tensor manipulation for reshaping and operations such as multiplication.

Validating Implementation with PyCall

Julia test for ConvBlock

```julia
using PyCall, Flux
torch = pyimport("torch")
pymodels = pyimport("scripts.models")


reversedims(x::AbstractArray) = permutedims(x, ndims(x):-1:1)
fromtorchtensor(t::PyObject) = t.detach().numpy() |> reversedims

@testset "ConvBlock" begin
    data = rand32(9, 16, 45, 16, 128)
    torchdata = torch.Tensor(data |> reversedims)

    torchcb = pymodels.ConvBlock(dim=16, dim_out=16, groups=8,
                                 cylindrical=true)
    cb = ConvBlock(torchcb)

    @test cb(data) ≈ torchcb(torchdata) |> fromtorchtensor
end
```

PyCall allows us to import Python modules and files directly from Julia.

Some objects are converted automatically to Julia corresponding representation, such as NumPy Array to Julia Array.

Once implemented mapping for weights and biases from PyTorch to Flux.jl structs, their output can be compared with the same input.

Training Loop

Python

```python
For (E, data) in loader_train:
    model.zero_grad()
    optimizer.zero_grad()

    data = data.to(device = device)
    E = E.to(device = device)

    t = torch.randint(0, nsteps, (data.size()[0],), device=device)
    noise = torch.randn_like(data)

    batch_loss = model.compute_loss(data, E, noise, t)
    batch_loss.backward()

    optimizer.step()

    del data, E, noise, batch_loss
```

Julia

```julia
for (x, e) in trainloader
    t = rand(1:c.nsteps, size(x)[end]) |> device
    noise = device == gpu ? CUDA.rand(size(x)...) :
            rand32(size(x)...)
    loss, grads = Flux.withgradient(batchloss, model, c, x, e, t,
                                    noise)
    Flux.update!(optim, model, grads[1])
```

Training Benchmarks – CPU

Julia

Batch Size	Step Time	Memory Allocated	GC Time (%)
4	3.09 s	3.46 GiB	44.85%
16	11.54 s	12.54 GiB	40.30%
32	22.91 s	24.65 GiB	39.78%

Python

Batch Size	Step Time	Memory Allocated
4	0.31 s	-
16	1.30 s	-
32	2.75 s	-

Used BenchmarkTools.jl to obtain measurements over 20 samples. Easy to setup and provides total memory allocation and GC time.

With Python, used PyTorch's built-in `torch.utils.benchmark`, which only measures time execution.

Training Benchmarks – CUDA

Julia

Batch Size	Step Time	Memory Allocated	GC Time (%)
4	87.95 ms	3.07 GiB	13.95%
16	333.53 ms	12.39 GiB	62.56%
32	651.30 ms	24.69 GiB	66.23%

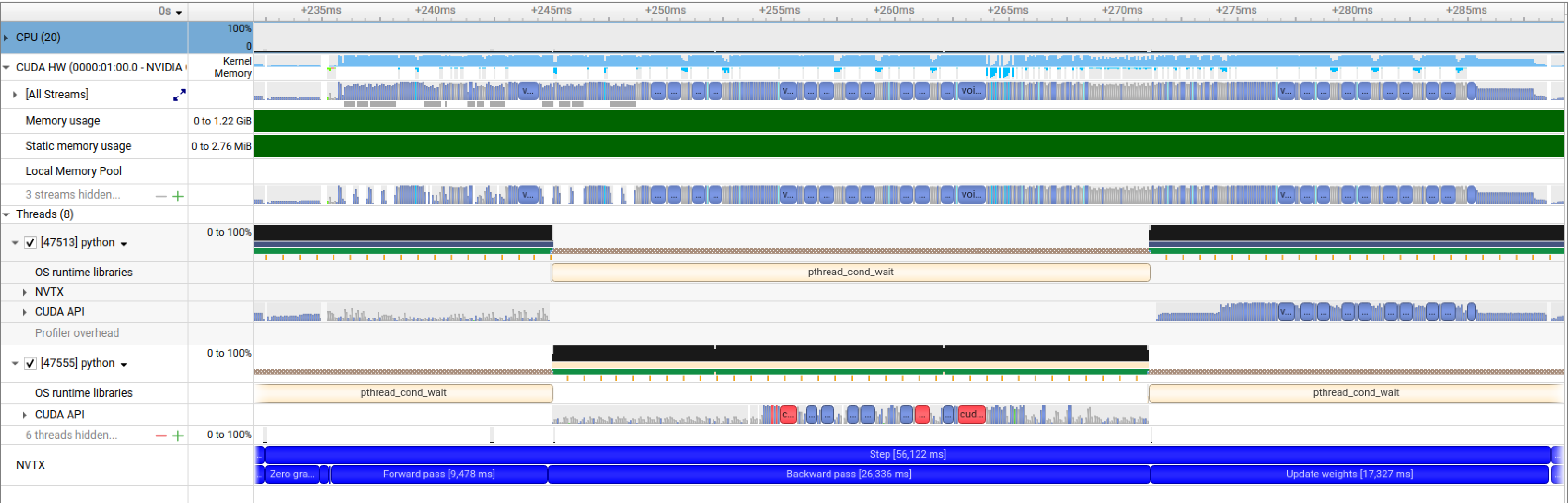
Python

Batch Size	Step Time	Memory Allocated
4	24.02 ms	282.60 MiB
16	37.72 ms	724.60 MiB
32	53.67 ms	1.22 GiB

When using NVIDIA GPUs, NVIDIA Nsight Systems can be used as an external profiler. It allows for a detailed analysis over time in terms of GPU utilization, memory utilization, kernel execution and more.

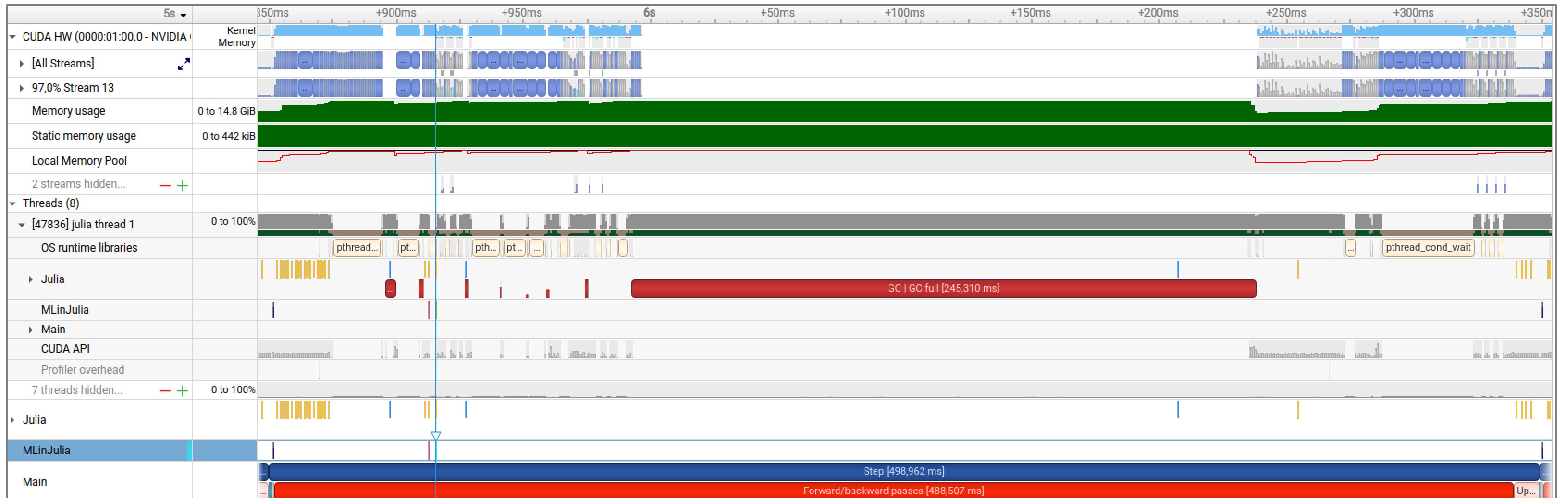
Python Profiling

Batch size 32



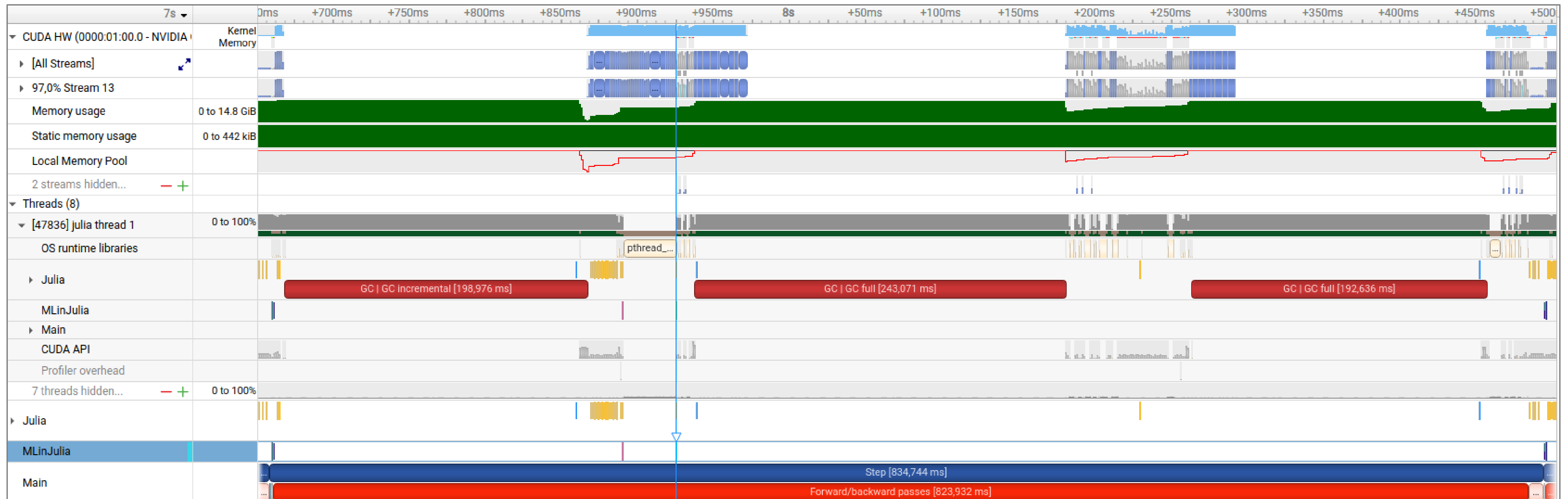
Julia Profiling

Batch size 32 - Best case scenario



Julia Profiling

Batch size 32 - Worst case scenario



Conclusions

- In its current form, it's possible to implement complex diffusion models with Flux.jl that are equivalent to PyTorch's implementation.
- Garbage collection calls results in considerable performance degradation, in worst cases is 10x slower.

Next steps include benchmarking each custom layer individually and look for optimizations on forward pass.