HPC.NRW

SEVERAL WAYS TO SAXPY

Julia + CUDA.jl

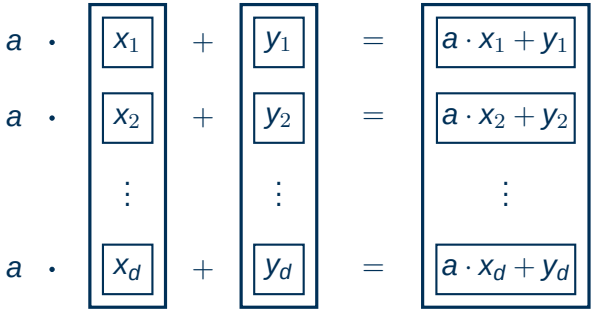
Carsten Bauer, Marius Neumann

AGENDA

HPC.NRW

- The SAXPY problem
- CUDA.jl
  - CPU and GPU memory allocations
  - memory transfer
- SAXPY with CUDA.jl
  - GPU broadcasting (higher-order abstraction)
  - GPU kernel (the manual way)
  - CUBLAS (the library way)
- Summary

- linear combination of two float arrays
- results written to third array



A typical Julia code:

```
# define constants
const dim = 100_000_000
const a = 3.1415

# allocate vectors
x = ones(Float32, dim)
y = ones(Float32, dim)
z = zeros(Float32, dim)


# perform SAXPY
z .= a .* x .+ y
```

CUDA.JL

https://github.com/JuliaGPU/CUDA.jl

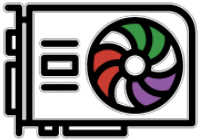
HPC.NRW

Programming interface for working
with NVIDIA CUDA GPUs in Julia.

- high-level array abstractions
- tools for writing CUDA kernels
- wrappers for various CUDA libraries (e.g. cuFFT)

```
julia> ] add CUDA
```

```
using CUDA
CUDA.versioninfo()
```



HPC.NRW

– CPU memory (`Array`)

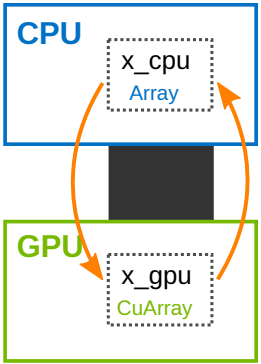
```
x_cpu = ones(Float32, dim)
```

– GPU memory (`CuArray`)

```
x_gpu = CUDA.ones(Float32, dim)
```

– type conversion triggers memory transfer

```
x_gpu = CuArray(x_cpu) # CPU -> GPU
x_cpu = Array(x_gpu)   # GPU -> CPU
```



SAXPY WITH CUDA.JL

GPU broadcasting

HPC.NRW

```
using CUDA

# define constants
const dim = 100_000_000
const a = 3.1415

# allocate vectors on the GPU
x = CUDA.ones(Float32, dim)
y = CUDA.ones(Float32, dim)
z = CUDA.zeros(Float32, dim)

# perform SAXPY
CUDA.@sync z .= a .* x .+ y
```

- almost the same code as for the CPU
- vectors `x, y, z` are initialized as `CuArrays` directly on the GPU
- `CUDA.@sync` make CPU wait until the GPU finishes SAXPY (blocking)

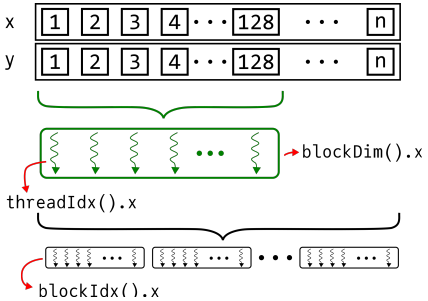
SAXPY WITH CUDA.JL

GPU kernel

HPC.NRW

```
# define GPU kernel
function saxpy_gpu_kernel!(z,a,x,y)
    i = (blockIdx().x - 1) * blockDim().x +
      threadIdx().x
    if i <= length(z)
        @inbounds z[i] = a * x[i] + y[i]
    end
    return nothing
end
```

- `z[i] = a * x[i] + y[i]`
  as if surrounded by implicit for-loop
- `threadIdx().x, blockDim().x, blockIdx().x`
  built-in variables that identify the thread global index

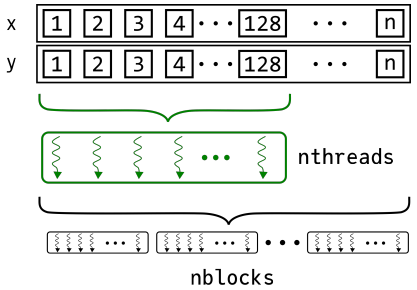


SAXPY WITH CUDA.JL

launching the kernel

HPC.NRW

```julia
using CUDA
# [define GPU kernel, constants, and
# allocate vectors on the GPU ...]

# define GPU execution parameters
nthreads = CUDA.attribute(
    device(),
    CUDA.DEVICE_ATTRIBUTE_MAX_THREADS_PER_BLOCK
)
nblocks = cld(dim, nthreads)

CUDA.@sync @cuda(
  threads=nthreads,
  blocks=nblocks,
  saxpy_gpu_kernel!(z,a,x,y)
)
```



– `@cuda` macro launches kernel on the GPU with the given launch configuration.

SAXPY WITH CUDA.JL

CUBLAS

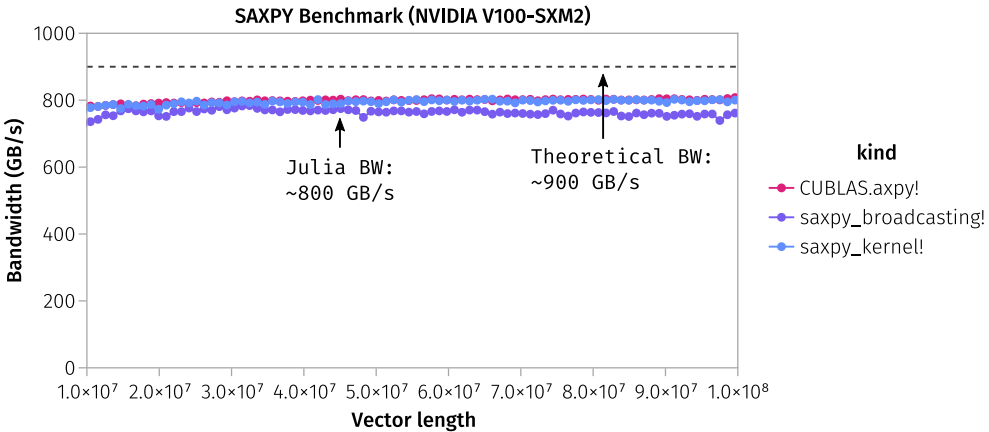
HPC.NRW

```julia
using CUDA
using CUDA.CUBLAS

# define constants
const dim = 100_000_000
const a = 3.1415

# allocate vectors on the GPU
x = CUDA.ones(Float32, dim)
y = CUDA.ones(Float32, dim)

# perform SAXPY
# (y is overwritten with the result)
CUDA.@sync CUBLAS.axpy!(dim, a, x, y)
```

- CUDA.jl provides low-level wrappers of the vendor library CUBLAS.
- `CUBLAS.axpy!` for `Float32` input directly calls the cuBLAS function `cublasSaxpy_v2`.
- `CUDA.@sync` make CPU wait until the GPU finishes SAXPY (blocking)

SAXPY PERFORMANCE

HPC.NRW



SUMMARY

- Julia + CUDA.jl
  - NVIDIA CUDA interface: higher-order abstractions, library wrappers
- SAXPY with CUDA.jl
  - GPU broadcasting: `CuArray` moves computation to the GPU
    - `zeros(Float32, N)` vs. `CUDA.zeros(Float32, N)`
    - SAXPY: essentially the same code as for the CPU
  - GPU kernel
    - definition: `blockIdx(), blockDim(), threadIdx()`
    - launching: `CUDA.@sync @cuda threads=T blocks=B kernel()`
  - CUBLAS: `CUDA.@sync CUBLAS.axpy!()`
  - transfer GPU result to CPU: `result_cpu = Array(result_gpu)`
  - memory release: `result_gpu = nothing; GC.gc(true)`
- SAXPY performance on NVIDIA V100: 800 GB/s