```julia
using PlutoUI, Test
```

Table of Contents


```julia
TableOfContents()
```

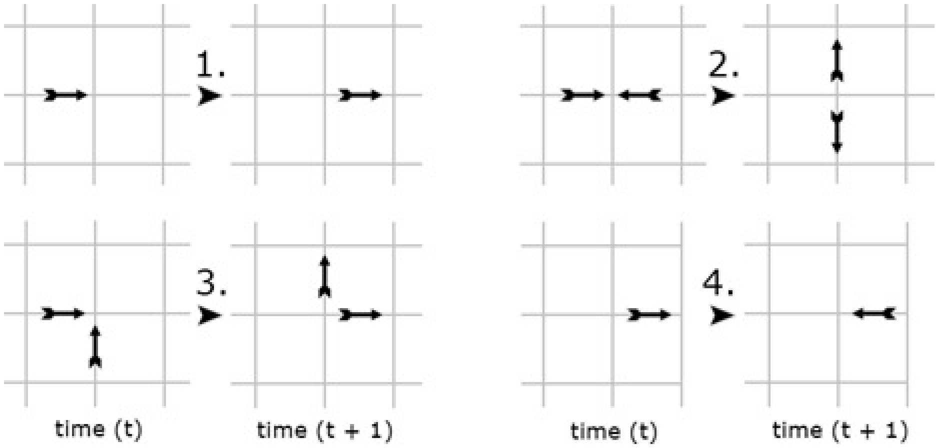
Simulting lattice gas cellular automata

Cellular automata

- A descretized space and time,
- A state defined on the space,
- A simple set of rules (local & finite) to describe the evolution of the state.

Reference:

- Hardy J, Pomeau Y, De Pazzis O. Time evolution of a two-dimensional model system. I. Invariant states and time correlation functions[J]. Journal of Mathematical Physics, 1973, 14(12): 1746-1759.



- Particles exist only on the grid points, never on the edges or surface of the lattice.
- Each particle has an associated direction (from one grid point to another immediately adjacent grid point).
- Each lattice grid cell can only contain a maximum of one particle for each direction, i.e., contain a total of between zero and four particles.

The following rules also govern the model:

- A single particle moves in a fixed direction until it experiences a collision.
- Two particles experiencing a head-on collision are deflected perpendicularly.
- Two particles experience a collision which isn't head-on simply pass through each other and continue in the same direction.
- Optionally, when a particles collides with the edges of a lattice it can rebound.

CUDA programming with Julia

CUDA programming is a parallel computing platform and programming model developed by NVIDIA for performing general-purpose computations on its GPUs (Graphics Processing Units). CUDA stands for Compute Unified Device Architecture.

References:

1. JuliaComputing/Training
2. arXiv: 1712.03112

Goal

1. Run a CUDA program
2. Write your own CUDA kernel
3. Create a CUDA project

Run a CUDA program

1. Make sure you have a NVIDIA GPU device and its driver is properly installed.

```
Process(`nvidia-smi`, ProcessExited(0))
```

```
run(`nvidia-smi`)
```

```
Wed May 10 15:34:25 2023
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 525.105.17   Driver Version: 525.105.17   CUDA Version: 12.0     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  NVIDIA RTX A4500     Off  | 00000000:B3:00.0 Off |                  Off |
| 33%   55C    P8    26W / 200W |   4947MiB / 20470MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
```

2. Install the CUDA.jl package, and disable scalar indexing of CUDA arrays.

CUDA.jl provides wrappers for several CUDA libraries that are part of the CUDA toolkit:

- Driver library: manage the device, launch kernels, etc.
- CUBLAS: linear algebra
- CURAND: random number generation
- CUFFT: fast fourier transform
- CUSPARSE: sparse arrays
- CUSOLVER: decompositions & linear systems

There's also support for a couple of libraries that aren't part of the CUDA toolkit, but are commonly used:

- CUDNN: deep neural networks
- CUTENSOR: linear algebra with tensors

```julia
using CUDA; CUDA.allowscalar(false)
```

```julia
CUDA.versioninfo()
```

```
CUDA runtime 12.1, artifact installation
CUDA driver 12.1
NVIDIA driver 525.105.17, originally for CUDA 12.0

Libraries:
- CUBLAS: 12.1.0
- CURAND: 10.3.2
- CUFFT: 11.0.2
- CUSOLVER: 11.4.4
- CUSPARSE: 12.0.2
- CUPTI: 18.0.0
- NVML: 12.0.0+525.105.17

Toolchain:
- Julia: 1.9.0-rc3
- LLVM: 14.0.6
- PTX ISA support: 3.2, 4.0, 4.1, 4.2, 4.3, 5.0, 6.0, 6.1, 6.3, 6.4, 6.5, 7.0, 7.1, 7.2, 7.3, 7.4, 7.5
- Device capability support: sm_37, sm_50, sm_52, sm_53, sm_60, sm_61, sm_62, sm_70, sm_72, sm_75, sm_80, sm_86

1 device:
  0: NVIDIA RTX A4500 (sm_86, 14.871 GiB / 19.990 GiB available)
```

3. Choose a device (if multiple devices are available).

```
CUDA.DeviceIterator() for 1 devices:
0. NVIDIA RTX A4500
```

```
devices()
```

```
dev = CuDevice(0): NVIDIA RTX A4500
```

```
dev = CuDevice(0)
```

grid > block > thread

```
1024
```

```
attribute(dev, CUDA.DEVICE_ATTRIBUTE_MAX_THREADS_PER_BLOCK)
```

```
1024
```

```
attribute(dev, CUDA.CU_DEVICE_ATTRIBUTE_MAX_BLOCK_DIM_X)
```

```
2147483647
```

```
attribute(dev, CUDA.CU_DEVICE_ATTRIBUTE_MAX_GRID_DIM_X)
```

4. Create a CUDA Array

```
10-element CuArray{Float32, 1, CUDA.Mem.DeviceBuffer}:
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
```

```
CUDA.zeros(10)
```

```
cuarray1 = 10-element CuArray{Float32, 1, CUDA.Mem.DeviceBuffer}:
             1.0669513
            -1.0011338
            -0.45724186
             0.6212741
            -0.90845925
             0.36932236
            -0.33143434
             2.1754394
             0.45788017
             0.97185755
```

```
cuarray1 = CUDA.randn(10)
```

```
Test Passed
      Thrown: ErrorException
```

```
@test_throws ErrorException cuarray1[3]
```

```
9.542758f0
```

```julia
CUDA.@allowscalar cuarray1[3] += 10
```

Upload a CPU Array to GPU

```
10-element CuArray{Float64, 1, CUDA.Mem.DeviceBuffer}:
 -0.5106224197359117
  0.08435781951985412
  1.3145343984144213
  0.7198074551780037
  1.9659466115028255
  0.31945629591804797
 -0.5025034746872823
 -1.7102368720734844
  0.3221473609865356
 -1.2079721286819451
```

```julia
CuArray(randn(10))
```

5. Compute

Computing a function on GPU Arrays

1. Launch a CUDA job - a few micro seconds
2. Launch more CUDA jobs...
3. Synchronize threads - a few micro seconds

Computing matrix multiplication.

```
0.202039044
```

```julia
@elapsed rand(2000,2000) * rand(2000,2000)
```

InterruptException:

```julia
@elapsed CUDA.@sync CUDA.rand(2000,2000) * CUDA.rand(2000,2000)
```

```
WARNING: Force throwing a SIGINT
```

Broadcasting a native Julia function Julia -> LLVM (optimized for CUDA) -> CUDA

```
poor_besselj (generic function with 1 method)
```

```julia
# this function is copied from lecture 9
function poor_besselj(ν::Int, z::T; atol=eps(T)) where T
    k = 0
    s = (z/2)^ν / factorial(ν)
    out = s::T
    while abs(s) > atol
        k += 1
        s *= -(k+ν) * (z/2)^2 / k
        out += s
    end
    out
end
```

```
factorial (generic function with 1 method)
```

```julia
factorial(n) = n == 1 ? 1 : factorial(n-1)*n
```

```
x = 1001-element CuArray{Float64, 1, CUDA.Mem.DeviceBuffer}:
  0.0
  0.01
  0.02
  0.03
  0.04
  0.05
  0.06
  ⋮
  9.95
  9.96
  9.97
  9.98
  9.99
 10.0
```

```julia
x = CUDA.CuArray(0.0:0.01:10)
```

```
1001-element CuArray{Float64, 1, CUDA.Mem.DeviceBuffer}:
 0.0
 0.0049997500093746875
 0.009998000299960006
 0.014993252277441754
 0.01998400959488256
 0.02496877927248
 0.029946072812618307
 ⋮
 1.6337875313577208e-5
 0.005418409574627547
 0.00675994621802067
 4.02498425561632e-6
 2.043448513104033e-9
 9.927485697704232e-6
```

```julia
poor_besselj.(1, x)
```

```julia
using BenchmarkTools
```

6. manage your GPU devices

nvml_dev = NVML.Device(0): NVIDIA RTX A4500

```
nvml_dev = NVML.Device(parent_uuid(device()))
```

62.568

```
NVML.power_usage(nvml_dev)
```

(compute = 0.99, memory = 0.0)

```
NVML.utilization_rates(nvml_dev)
```

Dict(372567 ⇒ (used_gpu_memory = 673185792), 312821 ⇒ (used_gpu_memory = 811597824))

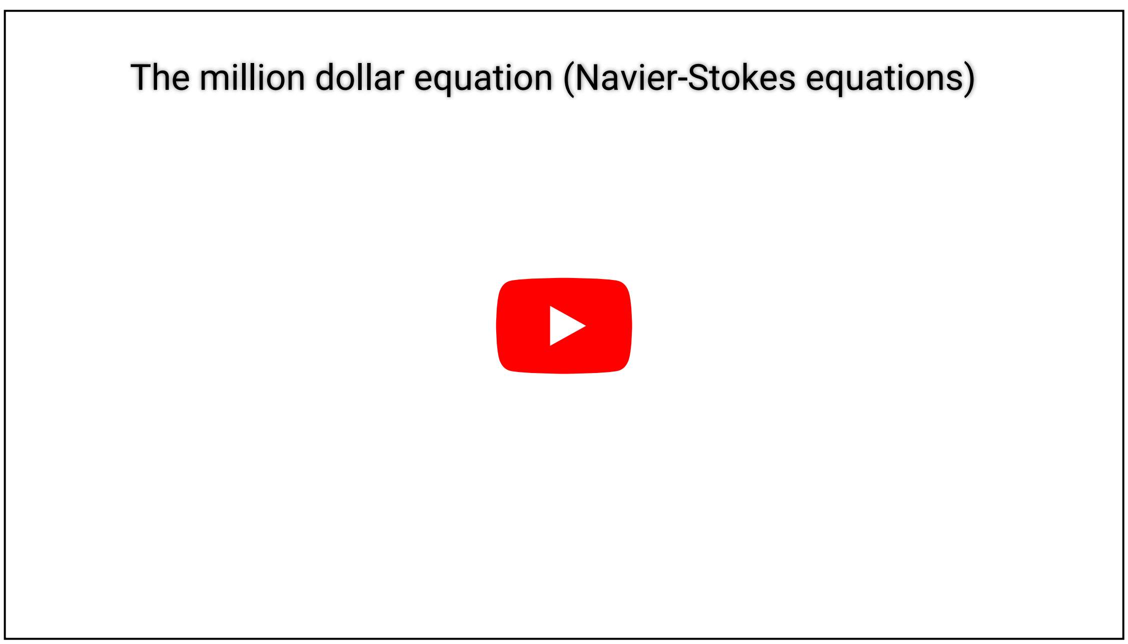
```
NVML.compute_processes(nvml_dev)
```

CUDA libraries and Kernel Programming

Please check lib/CUDATutorial

Appendix: The Navier-Stokes equation

Reference: https://youtu.be/Ra7aQlenTb8



The navier stokes equation describes the fluid dynamics, which contains the following two parts.

The first one describes the conservation of volume

$$\nabla \underbrace{u}_{\text{velocity } u \in \mathbb{R}^d} = 0$$

The second one describes the dynamics

$$\underbrace{\rho}_{\text{density}} \frac{du}{dt} = \underbrace{-\nabla p}_{\text{pressure}} + \underbrace{\mu \nabla^2 u}_{\text{viscosity (or friction)}} + \underbrace{f}_{\text{external force}} .$$