Scientific GPU computing with Go

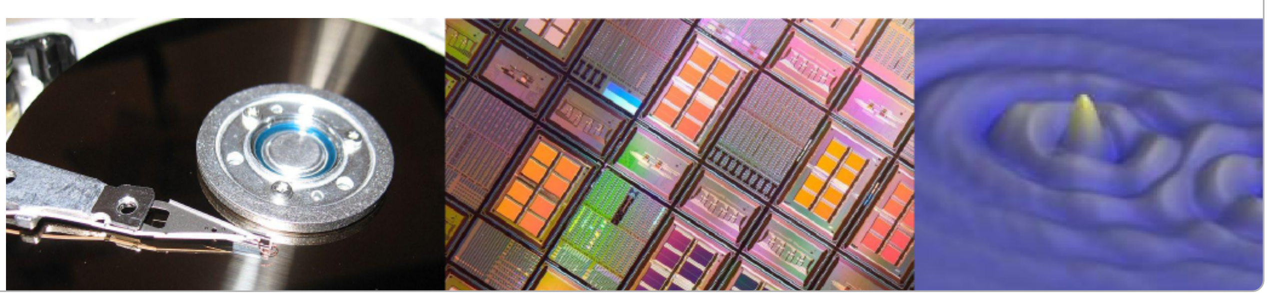
A novel approach to highly reliable CUDA HPC
1 February 2014

Arne Vansteenkiste
Ghent University

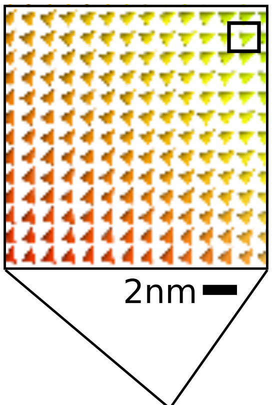
Real-world example (micromagnetism)

DyNaMat LAB @ UGent: Microscale Magnetic Modeling:

- Hard Disks
- Magnetic RAM
- Microwave components
- ...



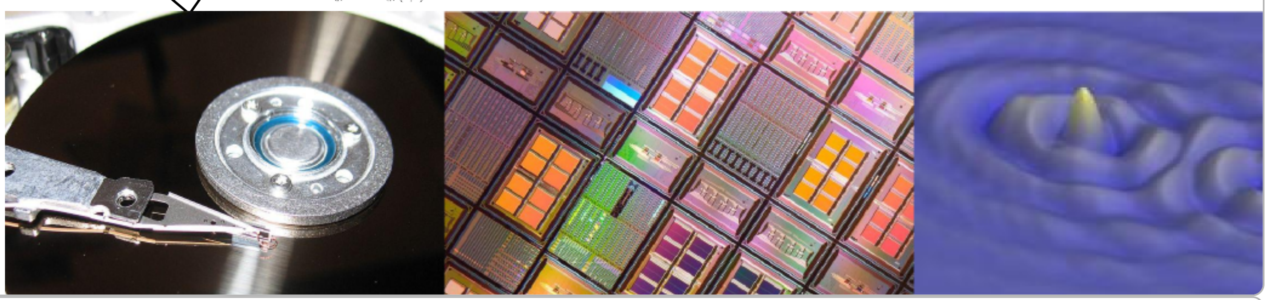
Real-world example (micromagnetism)



$$
\begin{aligned}
\frac{\partial \vec{\mathbf{m}}}{\partial t} &= \gamma_0 \left(\vec{\tau}_{\mathrm{LL}} + \vec{\tau}_{\mathrm{STT}}\right) \\
\vec{\tau}_{\mathrm{LL}} &= \frac{1}{1+\alpha^2}\left(\vec{\mathbf{m}} \times \vec{\mathbf{B}}_{\mathrm{eff}} + \alpha\left(\vec{\mathbf{m}} \times \left(\vec{\mathbf{m}} \times \vec{\mathbf{B}}_{\mathrm{eff}}\right)\right)\right) \\
\alpha &= \alpha\left(\vec{\mathbf{r}}, t\right) \\
\vec{\mathbf{m}} &= \frac{\vec{\mathbf{M}}\left(\vec{\mathbf{r}}, t\right)}{M_s} \\
M_s &= \left|\vec{\mathbf{M}}\left(\vec{\mathbf{r}}, t\right)\right| \\
\vec{\mathbf{B}}_{\mathrm{eff}} &= \vec{\mathbf{B}}_{\mathrm{d}} + \vec{\mathbf{B}}_{\mathrm{ex}} + \vec{\mathbf{B}}_{\mathrm{z}} + \vec{\mathbf{B}}_{\mathrm{a}} + \vec{\mathbf{B}}_{\mathrm{th}} \\
\vec{\mathbf{B}}_{\mathrm{d}}\left(\vec{\mathbf{r}}, t\right) &= \iiint_V \vec{\vec{\mathbf{K}}}(\vec{\mathbf{r}} - \vec{\mathbf{r}}') \cdot \mu_0 \vec{\mathbf{M}}(\vec{\mathbf{r}}', t) \mathrm{d}^3\vec{\mathbf{r}}' \\
&= \mathcal{F}^{-1}\left(\mathcal{F}\left(\vec{\vec{\mathbf{K}}}(\vec{\mathbf{r}})\right) \cdot \mathcal{F}\left(\mu_0 \vec{\mathbf{M}}\left(\vec{\mathbf{r}}, t\right)\right)\right) \\
\vec{\mathbf{K}}_i(\vec{\mathbf{r}}) &= \frac{1}{4\pi}\left(\frac{3(\vec{\mathbf{e}}_i \cdot \hat{\mathbf{r}})\hat{\mathbf{r}} - \vec{\mathbf{e}}_i}{r^3}\right) + \frac{2}{3}\vec{\mathbf{e}}_i \delta^3(\vec{\mathbf{r}}) \\
\vec{\mathbf{B}}_{\mathrm{ex}} &= \frac{2A}{M_s}\Delta\vec{\mathbf{m}} + \frac{2D}{M_s}\left(\frac{\partial m_z}{\partial x}, \frac{\partial m_z}{\partial y}, -\frac{\partial m_x}{\partial x} - \frac{\partial m_y}{\partial y}\right) \\
A &= A\left(\vec{\mathbf{r}}, t\right) \\
\vec{\mathbf{B}}_{\mathrm{z}} &= \vec{\mathbf{B}}_{\mathrm{z}}\left(\vec{\mathbf{r}}, t\right) \\
\vec{\mathbf{B}}_{\mathrm{a}} &= \vec{\mathbf{B}}_{\mathrm{u}} + \vec{\mathbf{B}}_{\mathrm{c}} \\
\vec{\mathbf{B}}_{\mathrm{u}} &= 2K_{u1}\left(\vec{\mathbf{m}} \cdot \vec{\mathbf{u}}\right)\vec{\mathbf{u}} \\
K_{u1} &= K_{u1}\left(\vec{\mathbf{r}}, t\right)
\end{aligned}
$$

$$
\begin{aligned}
\vec{\mathbf{u}} &= \vec{\mathbf{u}}\left(\vec{\mathbf{r}}, t\right) \\
\vec{\mathbf{B}}_{ci} &= \left(A_{cx}c_{1i} + A_{cy}c_{2i} + A_{cz}c_{3i}\right) \\
\vec{\mathbf{A}}_c &= K_{c1}\left(a_1(a_2^2 + a_3^2),\ a_2(a_1^2 + a_3^2),\ a_3(a_1^2 + a_2^2)\right) \\
a_i &= \vec{\mathbf{c}}_i \cdot \vec{\mathbf{m}} \\
\vec{\mathbf{c}}_1 &= \vec{\mathbf{c}}_1\left(\vec{\mathbf{r}}, t\right) \\
\vec{\mathbf{c}}_2 &= \vec{\mathbf{c}}_2\left(\vec{\mathbf{r}}, t\right) \\
\vec{\mathbf{c}}_3 &= \vec{\mathbf{c}}_1 \times \vec{\mathbf{c}}_2 \\
\vec{\mathbf{B}}_{\mathrm{th}}\left(\vec{\mathbf{r}}, t\right) &= \eta\left(\vec{\mathbf{r}}, t\right)\sqrt{\frac{k_B^2 \alpha T}{\mu_0 \gamma_0 M_s \Delta V \Delta t}} \\
\vec{\tau}_{\mathrm{STT}} &= \vec{\tau}_{\mathrm{ZL}} + \vec{\tau}_{\mathrm{SL}} \\
\vec{\tau}_{\mathrm{ZL}} &= \frac{1}{1+\alpha^2}\left((1+\xi\alpha)\,\vec{\mathbf{m}} \times (\vec{\mathbf{m}} \times (\vec{\mathbf{u}} \cdot \nabla)\vec{\mathbf{m}}) + (\xi - \alpha)\,\vec{\mathbf{m}} \times (\vec{\mathbf{u}} \cdot \nabla)\vec{\mathbf{m}}\right) \\
\vec{\mathbf{u}} &= \frac{\mu_B \mu_0}{2e\gamma_0 B_s(1+\xi^2)}\vec{\mathbf{j}} \\
\vec{\mathbf{j}} &= \vec{\mathbf{j}}\left(\vec{\mathbf{r}}, t\right) \\
\vec{\tau}_{\mathrm{SL}} &= \beta\epsilon(\vec{\mathbf{m}} \times \vec{\mathbf{m}}_P \times \vec{\mathbf{m}}) - \beta\epsilon'\vec{\mathbf{m}} \times \vec{\mathbf{m}}_P \\
\beta &= \frac{j_z \hbar}{M_s e d} \\
\epsilon &= \frac{P\left(\vec{\mathbf{r}}, t\right)\Lambda^2}{(\Lambda^2 + 1) + (\Lambda^2 - 1)(\vec{\mathbf{m}} \cdot \vec{\mathbf{m}}_P)} \\
\epsilon' &= \epsilon'\left(\vec{\mathbf{r}}, t\right)
\end{aligned}
$$

$$
\begin{aligned}
\left.\frac{\partial m_x}{\partial x}\right|_{\partial V} &= -\frac{D}{2A}m_z \\
\left.\frac{\partial m_y}{\partial x}\right|_{\partial V} &= 0 \\
\left.\frac{\partial m_z}{\partial x}\right|_{\partial V} &= \frac{D}{2A}m_x \\
\left.\frac{\partial m_x}{\partial y}\right|_{\partial V} &= 0 \\
\left.\frac{\partial m_y}{\partial y}\right|_{\partial V} &= -\frac{D}{2A}m_z \\
\left.\frac{\partial m_z}{\partial y}\right|_{\partial V} &= \frac{D}{2A}m_y \\
\left.\frac{\partial m_x}{\partial z}\right|_{\partial V} &= 0 \\
\left.\frac{\partial m_y}{\partial z}\right|_{\partial V} &= 0 \\
\left.\frac{\partial m_z}{\partial z}\right|_{\partial V} &= 0
\end{aligned}
$$

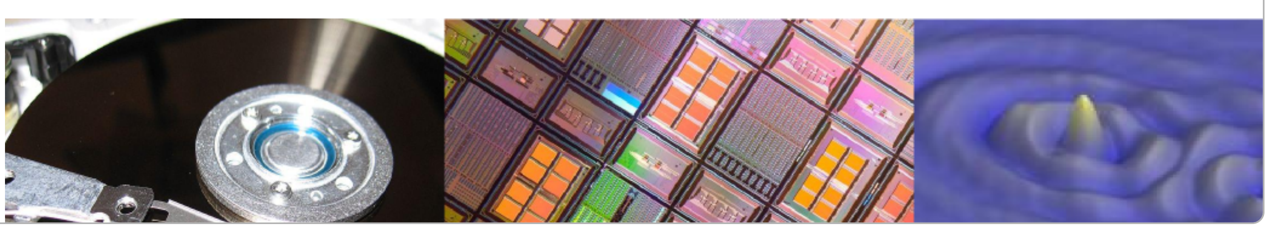


Real-world example (micromagnetism)

MuMax3 (GPU, script + GUI): ~ 11,000 lines CUDA, Go (http://mumax.github.io)

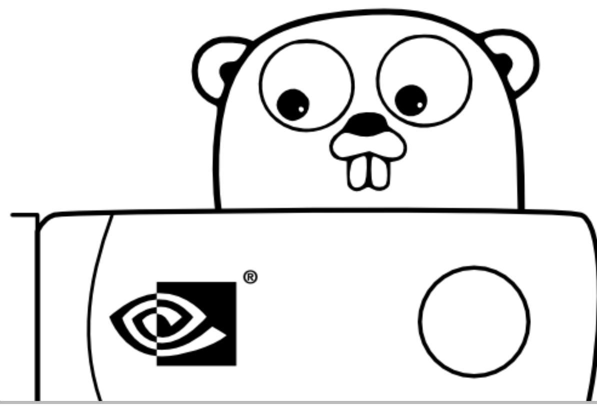
Compare to:

- **OOMMF** (script + GUI): ~100,000 lines C++, tcl
- **Magnum** (GPU, script only): ~ 30,000 lines CUDA, C++, Python



How suitable is Go for HPC?

- **Pure Go number crunching**
- Go plus {C, C++, CUDA} number crunching
- Concurrency



Go is

- compiled
- statically typed

but also

- garbage collected
- memory safe
- dynamic

Hello, math!

```
func main() {
    fmt.Println("(1+1e-100)-1 =", (1+1e-100)-1)
    fmt.Println("√-1          =", cmplx.Sqrt(-1))
    fmt.Println("J₁(0.3)      =", math.J1(0.3))
    fmt.Println("Bi(666, 333) =", big.NewInt(0).Binomial(666, 333))
}
```

Run

Go math features:

- precise compile-time constants
- complex numbers
- special functions
- big numbers.

```
(1+1e-100)-1 = 1e-100
√-1          = (0+1i)
J₁(0.3)      = 0.148318816273104
Bi(666, 333) = 946274279373497391369043379702061302
```

```
Program exited.
```

But missing:

- matrices
- matrix libraries (BLAS, FFT, ...)

Run Kill Close

Performance

Example: dot product

```
func Dot(A, B []float64) float64{
    dot := 0.0
    for i  := range A{
        dot += A[i] * B[i]
    }
    return dot
}
```

Performance

```
func Dot(A, B []float64) float64{
    dot := 0.0
    for i  := range A{
        dot += A[i] * B[i]
    }
    return dot
}

func BenchmarkDot(b *testing.B) {
    A, B := make([]float64, 1024), make([]float64, 1024)

    sum := 0.0
    for i:=0; i<b.N; i++{
        sum += Dot(A, B)
    }
    fmt.Fprintln(DevNull, sum)  // use result
}
```

```
PASS
BenchmarkDot        1000000        1997 ns/op

Program exited.
```

```
go test -bench .
```

times all BenchmarkXXX functions

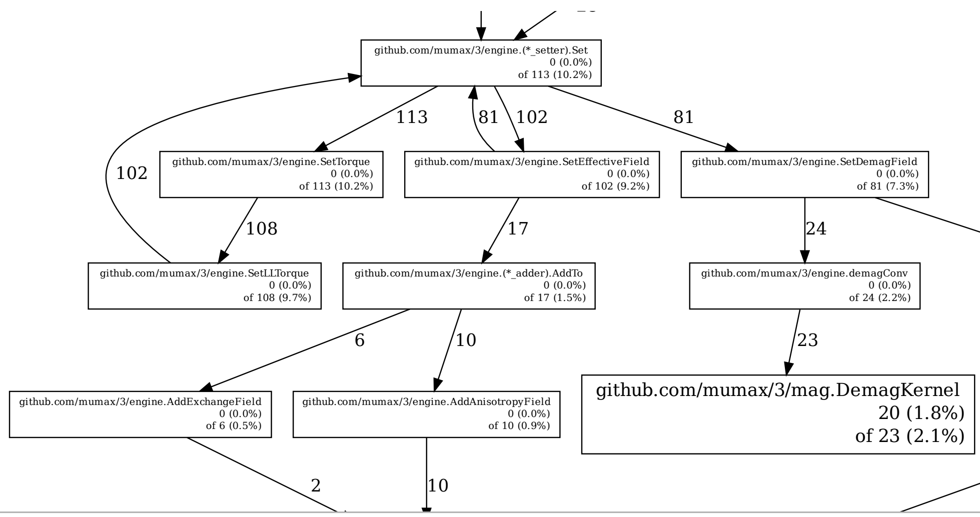
Run Kill Close

Profiling

Go has built-in profiling

```
go tool pprof
```

outputs your program's call graph with time spent per function



Performance

Dot product example

```
Go (gc)               1 980 ns/op
Go (gcc -O3)          1 570 ns/op

C  (gcc -O3)          1 460 ns/op
C  (gcc -march=native)  760 ns/op

Java                  2 030 ns/op
Python              200 180 ns/op
```

- Typically, Go is ~10% slower than optimized, portable C
- But can be 2x - 3x slower than machine-tuned C

Pure Go number crunching

On the up side

- Good standard math library
- Built-in testing, benchmarking & profiling
- Managed memory

On the down side

- Still slower than machine-tuned C
- No matrix libraries etc.

How suitable is Go for HPC?

- Pure Go number crunching
- **Go plus {C, C++, CUDA} number crunching**
- Concurrency

Hello, GPU!

Go can call C/C++ libs

```
//#include <cuda.h>
//#cgo LDFLAGS: -lcuda
import "C"
import "fmt"

func main() {
    buf := C.CString(string(make([]byte, 256)))
    C.cuDeviceGetName(buf, 256, C.CUdevice(0))
    fmt.Println("Hello, your GPU is:", C.GoString
}
```

```
Hello, your GPU is: GeForce GT 650M

Program exited.
```

Building:

```
go build
```

All build information is in the source

Run Kill Close

Hello, GPU! (wrappers)

```
import(
    "github.com/barnex/cuda5/cu"
    "fmt"
)

func main(){
    fmt.Println("Hello, your GPU is:", cu.Device(0).Name())
}
```

Run

```
Hello, your GPU is: GeForce GT 650M

Program exited.
```

Installing 3rd party code:

```
go get github.com/user/repo
```

(dependencies are compiled-in)

Run Kill Close

Calling CUDA kernels (the C way)

GPU (code for one element)

```
__global__ void add(float *a, float *b, float *c, N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < N)
       c[i] = a[i] + b[i];
}
```

CPU wrapper (divide and launch)

```
void gpu_add(float *a, float *b, float *c, int N){
    dim3 block = ...
    add<<<N/BLOCK, BLOCK>>>(a, b, c);
}
```

Go wrapper wrapper

```
func Add(a, b, c []float32){
    C.gpu_add(unsafe.Pointer(&a[0]), unsafe.Pointer(&b[0]),
              unsafe.Pointer(&c[0]), C.int(len(a)))
}
```

Calling CUDA kernels (cuda2go)

- CUDA kernel to Go wrapper (calling **nvcc** once).
- Further deployment without **nvcc** or **CUDA** libs.

Others to fetch your CUDA project the usual way:

```
go get github.com/user/my-go-cuda-project
```

```
// THIS FILE IS GENERATED BY CUDA2GO, EDITING IS FUTILE
func Add(a, b, c unsafe.Pointer, N int, cfg *config) {
    args := add_args_t{a, b, c, N}
    cu.LaunchKernel(add_code, cfg.Grid.X, cfg.Grid.Y, cfg.Grid.Z, cfg.Block.X, cfg.Block.Y, cfg.Bl
}

// PTX assembly
const add_ptx_20 = `
.version 3.1
.target sm_20
.address_size 64

.visible .entry add(
```

A note on memory (CPU)

Go is memory-safe, garbage collected.
Your typical C library is not.

Fortunately:

- Go is aware of C memory (no accidental garbage collection).
- Go properly aligns memory (needed by some HPC libraries)

Allocate in Go, pass to C, let Go garbage collect

A note on memory (GPU)

GPU memory still needs to be managed manually.
But a **GPU memory pool** is trivial to implement in Go.

```
var pool = make(chan cu.DevicePtr, 16)

func initPool(){
    for i:=0; i<16; i++{
        pool <- cu.MemAlloc(BUFSIZE)
    }
}

func recycle(buf cu.DevicePtr){
    pool <- buf
}

func main(){
    initPool()

    GPU_data := <- pool
    defer recycle(GPU_data)
    // ...
}
```

Run Run

Vector add example

Adding two vectors on GPU (example from nvidia)

```
#include "../common/book.h"
#define N 10

int main( void ) {
    int a[N], b[N], c[N];
    int *dev_a, *dev_b, *dev_c;
    // allocate the memory on the GPU
    HANDLE_ERROR( cudaMalloc( (void**)&dev_a, N * sizeof(int) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_b, N * sizeof(int) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_c, N * sizeof(int) ) );

    // fill the arrays 'a' and 'b' on the CPU
    for (int i=0; i<N; i++) {
        a[i] = -i;
        b[i] = i * i;
    }
    // copy the arrays 'a' and 'b' to the GPU
    HANDLE_ERROR( cudaMemcpy( dev_a, a, N * sizeof(int), cudaMemcpyHostToDevice ) );
    HANDLE_ERROR( cudaMemcpy( dev_b, b, N * sizeof(int), cudaMemcpyHostToDevice ) );
    add<<<N,1>>>( dev_a, dev_b, dev_c ); // copy the array 'c' back from the GPU to the CPU
    HANDLE_ERROR( cudaMemcpy( c, dev_c, N * sizeof(int), cudaMemcpyDeviceToHost ) );
    // display the results
    for (int i=0; i<N; i++) {
```

Vector add example

Adding two vectors on GPU (Go)

```
package main

import "github.com/mumax/3/cuda"

func main(){

    N := 3
    a := cuda.NewSlice(N)
    b := cuda.NewSlice(N)
    c := cuda.NewSlice(N)
    defer a.Free()
    defer b.Free()
    defer c.Free()

    a.CopyHtoD([]float32{0, -1, -2})
    b.CopyHtoD([]float32{0,  1,  4})

    cfg := Make1DConfig(N)
    add_kernel(a.Ptr(), b.Ptr(), c.Ptr(), cfg)

    fmt.Println("result:", a.HostCopy())
}
```

Go plus {C, C++, CUDA} number crunching

On the downside

- Have to write C wrappers

On the upside

- You *can* call C
- Have Go manage your C memory

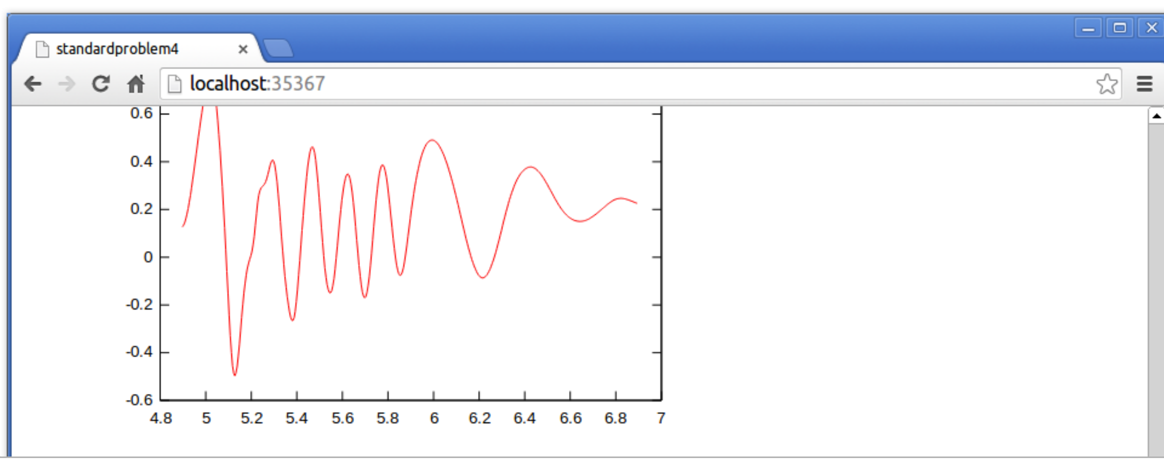
How suitable is Go for HPC?

- Pure Go number crunching
- Go plus {C, C++, CUDA} number crunching
- **Concurrency**

Real-world concurrency (MuMax3)

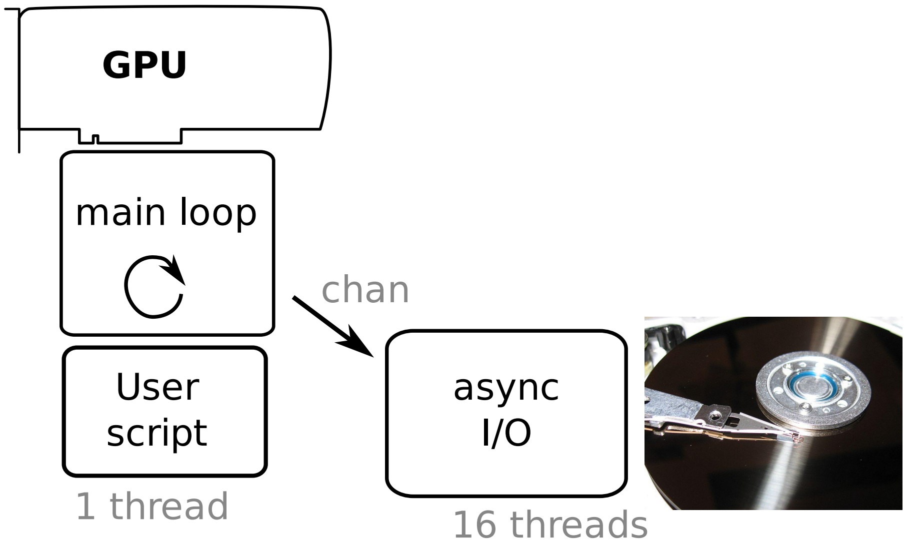
There's more to HPC then number crunching and memory management

- I/O
- Interactive supercomputing
- ...



Real-world concurrency (MuMax3)

Output: GPU does not wait for hard disk



Real-world concurrency (MuMax3)

Go **channels** are like type-safe UNIX pipes between threads.

```
var pipe = make(chan []float64, BUFSIZE)

func runIO(){
    for{
        data := <- pipe  // receive data from main
        save(data)
    }
}

func main() {
    go runIO()          // start I/O worker
    pipe <- data        // send data to worker
}
```

Run Run

Real example: 60 lines Go, ~2x I/O speed-up

Real-world concurrency (MuMax3)

You can send function **closures** over channels.

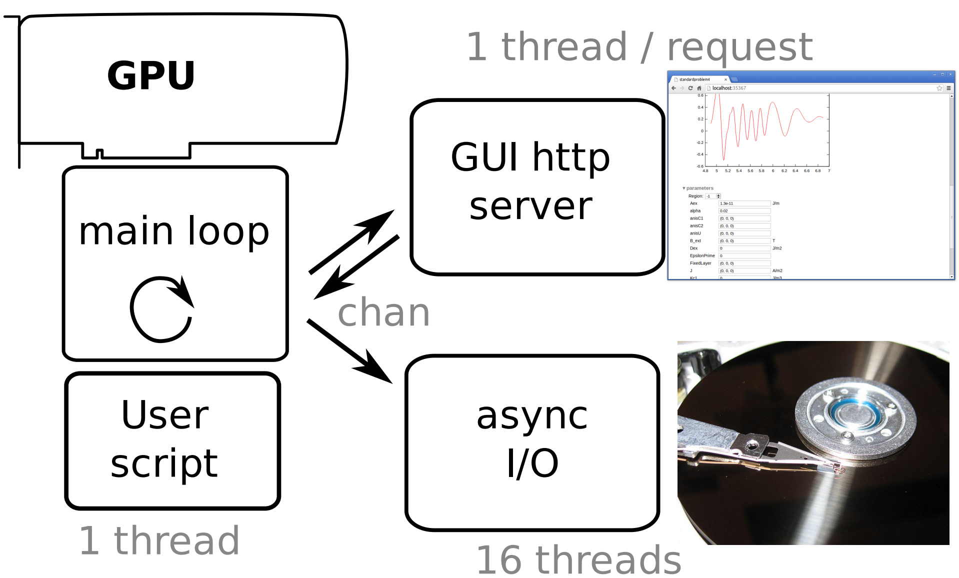
```
var pipe = make(chan func())         // channel of functions

func main() {
    for {
        select{
            case f := <- pipe:       // execute function if in pipe
                f()
            default: doCalculation() // nothing in pipe, crunch on
        }
    }
}

func serveHttp(){
    pipe <- func(){ value = 2 }      // send function to main loop
    ...
}
```

Run Run

Concurrency without mutex locking/unlocking.

Real-world concurrency (MuMax3)

GUI: change parameters while running,
without race conditions



And we can prove it's thread-safe

Go has built-in testing for race conditions

```
go build -race
```

enables race testing. Output if things go wrong:

```
==================
WARNING: DATA RACE
Write by goroutine 3:
  main.func·001()
      /home/billgates/buggycode/race.go:10 +0x38

Previous read by main goroutine:
  main.main()
      /home/billgates/buggycode/race.go:21 +0x9c

Goroutine 3 (running) created at:
  main.main()
      /home/billgates/buggycode/race.go:12 +0x33
==================
```

Go concurrency

On the up side

- Easy, safe, built-in concurrency

On the down side

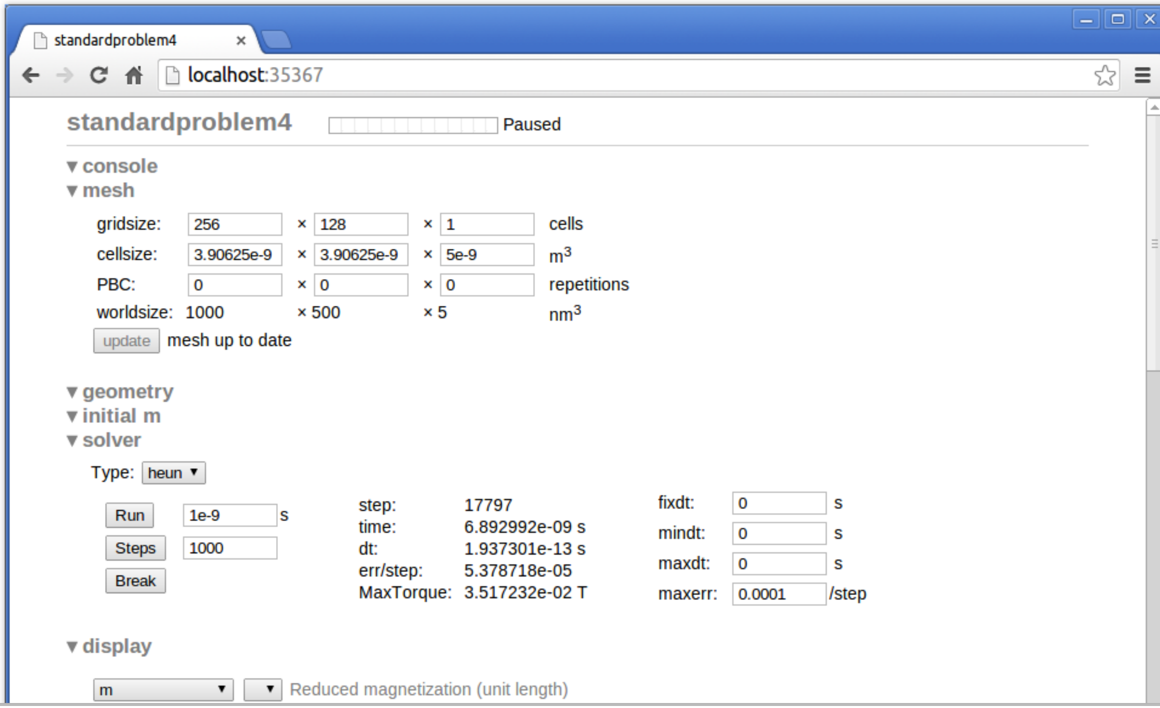
- There is no downside

Demonstration

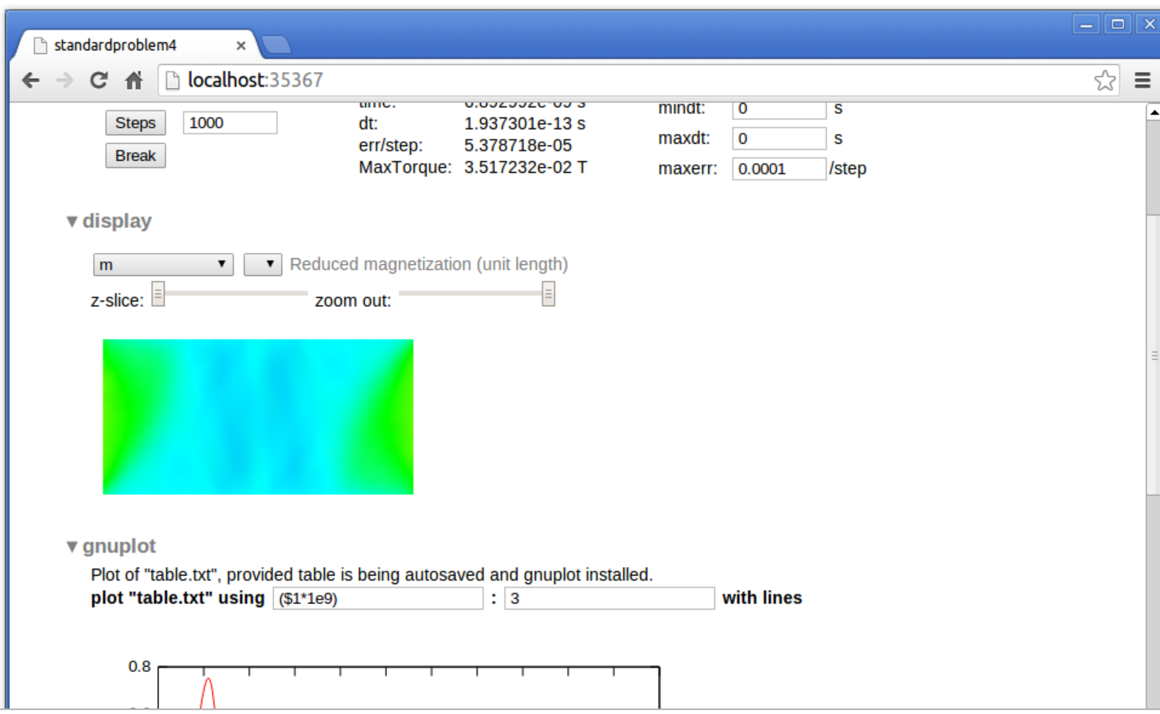
Input script

```
setgridsize(512, 256, 1)
setcellsize(5e-9, 5e-9, 5e-9)
ext_makegrains(40e-9, 256, 0)

Aex    = 10e-12    // J/m
Msat   = 600e3     // A/m
alpha  = 0.1
m      = uniform(0, 0, 1)

// set random parameters per grain
for i:=0; i<256; i++{
    AnisU.SetRegion(i, vector(0.1*(rand()-0.5), 0.1*(rand()-0.5), 1))

    for j:=i+1; j<256; j++{
        ext_scaleExchange(i, j, rand())
    }
}

// Write field
f     := 0.5e9     // Hz
B_ext  = sin(2*pi*f*t)

// spin HD and write
```

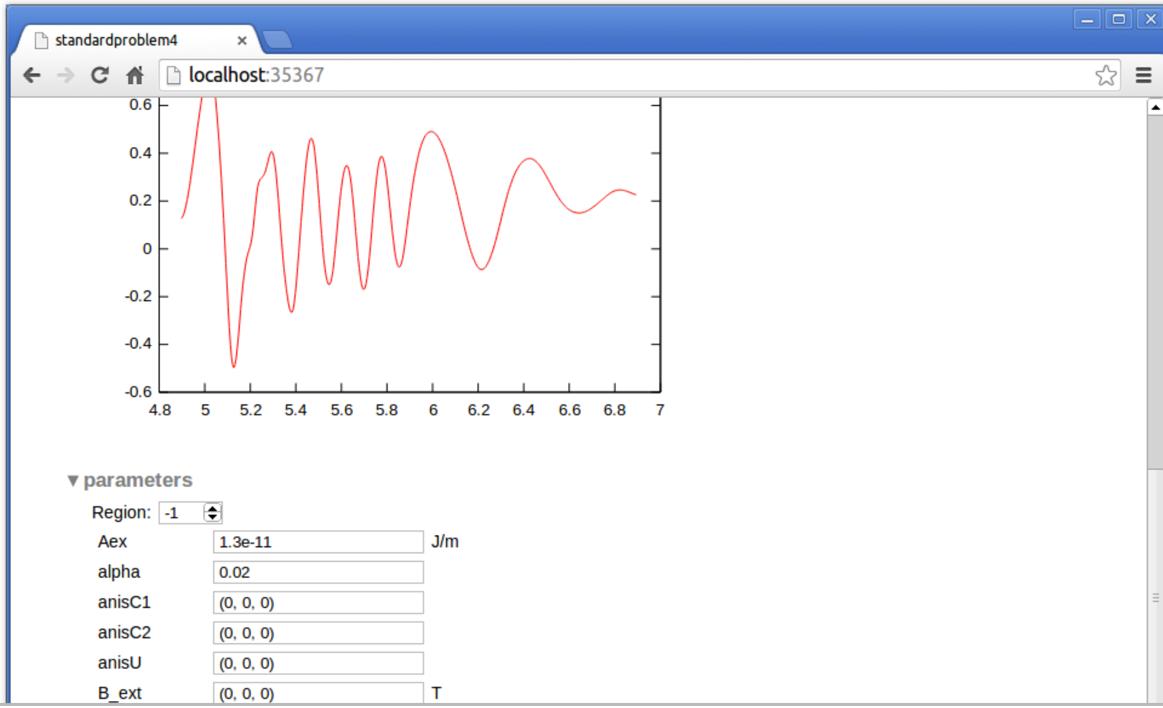
Demonstration



Demonstration



Demonstration



Demonstration

Hard disk magnetization (white = up = 1, black = down = 0)

Thank you

Arne Vansteenkiste
Ghent University
Arne.Vansteenkiste@Ugent.be (mailto:Arne.Vansteenkiste@Ugent.be)
http://mumax.github.io (http://mumax.github.io/)