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)

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!

```go
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))
}
```

Go math features:

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

But missing:

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

Performance

Example: dot product

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

```go
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
}
```

go test -bench .
times all BenchmarkXXX functions

Profiling

Go has built-in profiling

go tool pprof

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

Dot product example

<table>
<thead>
<tr>
<th>Language</th>
<th>Time (ns/op)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Go (gc)</td>
<td>1 980</td>
</tr>
<tr>
<td>Go (gcc -O3)</td>
<td>1 570</td>
</tr>
<tr>
<td>C (gcc -O3)</td>
<td>1 460</td>
</tr>
<tr>
<td>C (gcc -march=native)</td>
<td>760</td>
</tr>
<tr>
<td>Java</td>
<td>2 030</td>
</tr>
<tr>
<td>Python</td>
<td>200 180</td>
</tr>
</tbody>
</table>

- 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

```go
#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(buf))
}
```

Program exited.

Building:

go build

All build information is in the source
Hello, GPU! (wrappers)

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

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

Installing 3rd party code:

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

(dependencies are compiled-in)

Calling CUDA kernels (the C way)

**GPU (code for one element)**

```c
__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)**

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

**Go wrapper wrapper**

```go
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)
}

// 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.

```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)
    // ... 
}
```

Vector add example

Adding two vectors on GPU (example from nvidia)

```c
#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)

```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

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

```go
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
}
```

Real example: 60 lines Go, ~2x I/O speed-up
Real-world concurrency (MuMax3)

You can send function closures over channels.

```go
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
    ...       // execute the function
}
```

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

```plaintext
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

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

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