1 GPU computing with Julia

- Currently based on CUDA, but structurally other interfaces possible
- No shader code, just use Julia to program everything
- see https://juliagpu.gitlab.io/CUDA.jl/tutorials/introduction/

Necessary packages

```
[1]: using CUDAdrv, CUDAnative, CuArrays
    using LinearAlgebra
    using BenchmarkTools
```

Simple function: add vector on CPU

```
[2]: function sequential_add!(y, x)
    for i in eachindex(y, x)
        @inbounds y[i] += x[i]
    end
    return nothing
end

function parallel_add!(y, x)
    Threads.@threads for i in eachindex(y, x)
        @inbounds y[i] += x[i]
    end
    return nothing
end
```

[2]: parallel_add! (generic function with 1 method)

Create vectors on the CPU:

```
[3]: N=2_000_000
    T=Float32
```
\[ x = \text{rand}(T,N) \]
\[ y = \text{rand}(T,N) \]

[3]: 2000000-element Array{Float32,1}:
  0.408265
  0.32982695
  0.5224091
  0.7111348
  0.6951678
  0.3942306
  0.18355691
  0.2348566
  0.98016524
  0.84530926
  0.7638062
  0.8571855
  0.8986449
  0.23094416
  0.21900618
  0.05888343
  0.46976137
  0.1587727
  0.5450232
  0.78803205
  0.5879576
  0.0341686
  0.96046007
  0.12998486
  0.67565215

Create vectors on the GPU:

[4]:  
  \[ x\_gpu = \text{CuArray}\{T\}(\text{undef}, N); \]
  \[ y\_gpu = \text{CuArray}\{T\}(\text{undef}, N); \]

  \texttt{@btime} \texttt{begin}
    \texttt{copyto!}(x\_gpu,x);
    \texttt{copyto!}(y\_gpu,y);
  \texttt{end}

  2.423 ms (4 allocations: 15.26 MiB)

[4]: 2000000-element CuArray{Float32,1,Nothing}:
  0.408265
  0.32982695
  0.5224091
  0.7111348

CuArrays essentially provide a full vector library for linear algebra which can be controlled from the CPU
- Index access only reasonable on the CPU
- It seems to be like “numpy on steroids”
- Here we just write array broadcast code

```
[5]: function gpu_add_bcast!(y,x)
    CuArrays.@sync y .+= x
end
```

```
[5]: gpu_add_bcast! (generic function with 1 method)
```

Compare timings

```
[6]: @btime sequential_add!($x,$y);
   @btime parallel_add!($x,$y);
   @btime gpu_add_bcast!($x_gpu, $y_gpu);
```

419.601  s (0 allocations: 0 bytes)
308.353  s (30 allocations: 3.02 KiB)
255.614  s (57 allocations: 2.23 KiB)

We know here that the CPU has memory access issues, so 4 threads don’t give too much of speedup
- If high level abstraction is not sufficient, we can write kernels:
function gpu_add1!(y,x)

    function _kernel!(y, x)
        for i = 1:length(y)
            @inbounds y[i] += x[i]
        end
        return nothing
    end

    CuArrays.@sync begin
        @cuda _kernel!(y,x)
    end
end

gpu_add1! (generic function with 1 method)

@btime gpu_add1!(x_gpu, y_gpu)
110.328 ms (27 allocations: 912 bytes)
Linear indexing is incredibly slow here
Try a more thorough adaptation to CUDA data model provides better performance

function gpu_add2!(y, x;nthreads=256)

    function _kernel!(y, x)
        index = threadIdx().x
        stride = blockDim().x
        for i = index:stride:length(y)
            @inbounds y[i] += x[i]
        end
        return nothing
    end

    CuArrays.@sync begin
        @cuda threads=nthreads _kernel!(y,x)
    end
end

gpu_add2! (generic function with 1 method)

@btime gpu_add2!(x_gpu, y_gpu)
2.601 ms (37 allocations: 1.05 KiB)
Go even further and do proper blocking
gridDim.x = 4096

threadIdx.x = 0 1 2 3 ... 255
blockIdx.x = 0

threadIdx.x = 1 2 3 ... 255
blockIdx.x = 1

threadIdx.x = 2 3 ... 255
blockIdx.x = 2

threadIdx.x = 0 1 2 3 ... 255
blockIdx.x = 4095

index = blockIdx.x * blockDim.x + threadIdx.x

\[
\begin{align*}
\text{index} &= (2) \times (256) + (3) \\
&= 515
\end{align*}
\]

(see CUDA.jl tutorial)

[11]: function gpu_add3!(y, x; nthreads=256)
  function _kernel!(y, x)
    index = blockIdx().x - 1 * blockDim().x + threadIdx().x
    stride = blockDim().x * gridDim().x
    for i = index:stride:length(y)
      @inbounds y[i] += x[i]
    end
    return
  end
  numblocks = ceil(Int, length(x)/nthreads)
  @sync begin
    @cuda threads=nthreads blocks=numblocks _kernel!(y, x)
  end
end

[11]: gpu_add3! (generic function with 1 method)

[12]: @btime gpu_add3!($x_gpu, $y_gpu)

258.242  s (42 allocations: 1.13 KiB)

1.1 Working on the abstraction level of whole arrays

- Try iterative solution on the GPU
- Ignore all the complicated stuff, just use CuArrays

[13]: using ExtendableSparse
    using SparseArrays
    using Printf
    using IterativeSolvers

Implement two Jacobi preconditioners based just on a diagonal vector
function LinearAlgebra.ldiv!(b,D::CuVector,a)
    b .= a ./ D
end
function LinearAlgebra.ldiv!(b,D::Vector,a)
    b .= a ./ D
end

Create random matrix and problem data on a 3D rectangular grid with 512000 unknowns (we could have done FE assembly here...)

n = 80
N = n^3
t = @elapsed begin
    M = ExtendableSparse.fdrand(n,n,n,matrixtype=ExtendableSparseMatrix).cscmatrix
    u_exact = rand(N)
    D = Vector(diag(M))
    f = M * u_exact
end
println("Creating matrix: $(t) s")

loading GPU: 0.176134744 s

Run direct solver

# first run for compiling
u = M \ f
t = @elapsed begin
    u = M \ f
end
println("Direct solution on CPU: $(t) s")

Direct solution on CPU: 28.906624061 s

Use cg from IterativeSolvers to solve system with Jacobi preconditioner

# first run for compiling
u, hist = cg(M, f, Pl = D, tol = 1.0e-10, log = true)
t = @elapsed begin
end
u, hist = cg(M, f, Pl = D, tol = 1.0e-10, log = true)
end
println("CG solution on CPU: \(t\) s \((hist.\text{iters}) \text{ iterations})\,\epsilon \quad \text{error} = \$\text{(norm(u_exact - u))}\)"
CG solution on CPU: 1.594508333 s (295 iterations), error=8.681260320231236e-6
Do the same on the GPU ... yes, it is the same cg

[19]: # first run for compiling
u_gpu, hist = cg(M_gpu, f_gpu, Pl = D_gpu, tol = Float32(1.0e-10), log = true)
t = Base.@elapsed begin
    u_gpu, hist = cg(M_gpu, f_gpu, Pl = D_gpu, tol = Float32(1.0e-10), log = true)
end
println("CG solution on GPU: \(t\) s \((hist.\text{iters}) \text{ iterations})\,\epsilon \quad \text{error} = \$\text{(norm(u_exact_gpu - u_gpu))}\)"

CG solution on GPU: 0.60873829 s (295 iterations), error=3.060062695055421e-5

This notebook was generated using Literate.jl.