```
CuDevice(0): NVIDIA GeForce GTX 1080 Ti
```

```
1 CUDA.device()
```

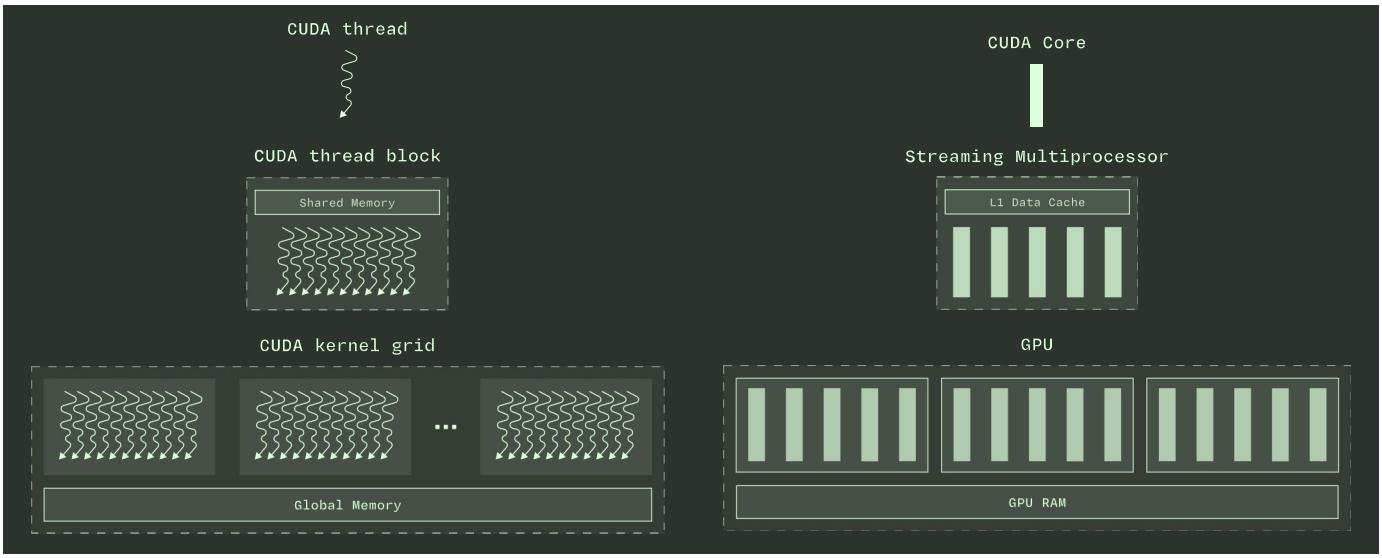
Hello! :)

In Julia, GPU usage is already optimized for many processes through CUDA.jl. Simply by applying a function to a CuArray, operations (e.g. broadcasting and map-reducing) are executed on GPU-specialized code. Additionally, more complex tasks, such as operations in machine learning algorithms like self-attention, have optimized code through cuDNN.jl.

These are done through GPU kernels, which implement functions that exploit the CUDA architecture of GPUs.

Today, we will show how to write such kernels, for when already-optimized kernels do not already exist.

GPU CUDA Architecture



https://modal.com/gpu-glossary/device-software/thread

Threads

Threads are the lowest level of the hierarchy (like in CPUs)! With GPUs, however, all threads within a **warp** should share the same task. While every thread that's called in the kernel executes the same code, they can process different parts of the data.

Thus, threads can coordinate/sync with one another within a block using shared memory. Can be indexed (x, y, z).

Blocks

Blocks are the smallest unit of thread coordination, wherein each block must execute independently and without order (in parallel, given enough resources). A single CUDA kernel launch produces one or more thread blocks that run asynchronously.

Blocks are arbitrarily sized (up to 1028), but typically multiples of warp size (up to 32). Can also be indexed (x, y, z).

Grids

Grids are made up of a collection of thread blocks, and this spans the entire GPU (basically global context of the kernel). Can be 1D, 2D, or 3D.

Warps

A warp is a group of threads that are scheduled together and execute in parallel. Since blocks don't necessarily have to share the same task, warps are the unit of execution on the GPU. Thus, all threads within a warp will have the same task.

This isn't actually part of the GPU architectural hierarchy in CUDA, but more an implementation detail that's useful to keep in mind for optimization.

Kernels

A kernel is essentially a function that launches/returns only **once** but is executed many times; once each by x number of threads. These occur in random order and simulataneously (these are what we assign to the warps!).

Launching a kernel

First, we can define our kernel function.

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

```
function kernel()
    # do stuff here
    return nothing
end
```

Then, we can launch it using `@cuda`. This launches a single thread.

```
CUDA.HostKernel for kernel()
```

```
@cuda kernel()
```

We can also get an object out of the compiled kernel for additional info!

```
k = CUDA.HostKernel for kernel()
```

```
k = @cuda launch=false kernel()
```

2

```
CUDA.registers(k)
```

The `launch=false` compiles the function without actually executing it. The result is a HostKernel object.

Looking at the `.registers()` essentially shows the complexity of the kernel (fewer registers = more active threads at once).

Basic kernel operations

Inputs/outputs

GPU kernels cannot return values to the CPU like a regular function, so it must always be set to `return` or `return nothing`. So, to work with values, we can pass a `CuArray` aka writing our results to

GPU arrays!

Note: Though a CuArray is given to the function when it is launched, the input is converted to a CuDeviceArray before execution.

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

```
function log_kernel(input)
    data = input[1]
    input[1] = log(data)
    return nothing
end
```

```
a = 3-element CuArray{Float32, 1, CUDA.DeviceMemory}:
 1.0
 1.0
 1.0
```

```
a = CUDA.ones(Float32, 3)
```

```
@cuda log_kernel(a);
```

```
3-element CuArray{Float32, 1, CUDA.DeviceMemory}:
 0.0
 1.0
 1.0
```

```
a
```

FYI: if you need random numbers, you must use a GPU-compatiable RNG!

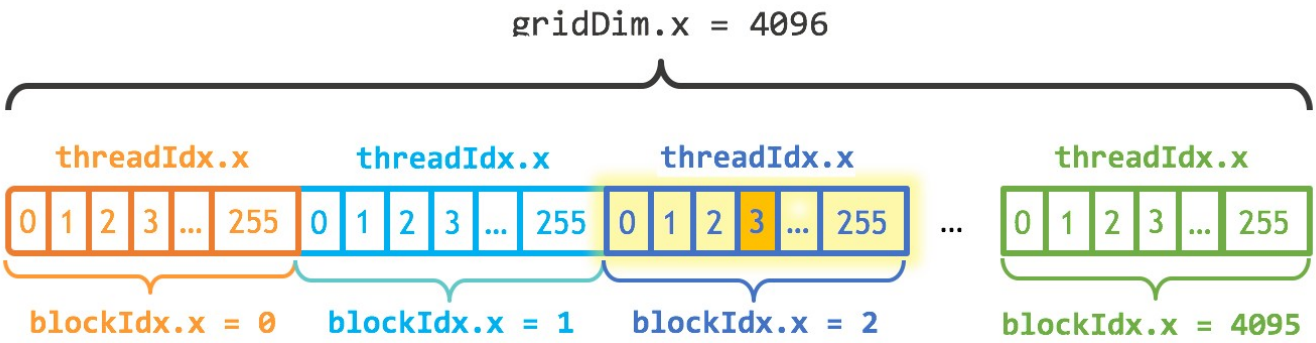
Via: `@cushow rand()`

With the example above, we took the log of only the first value in the array.

Note: you usually want to only access/write into your global memory (`input`) once. Which is why we assign the variable `data`.

Distributing across threads

Since we want to use multiple threads, we can use indexing to differentiate computations for each thread and block. Additionally, we can use `threads=` and `blocks=` when we launch `@cuda`.




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

index =    (2)      *   (256)    +     (3)      = 515
```

https://developer.nvidia.com/blog/even-easier-introduction-cuda/

By threads

To process data that fits inside one block only, we can extract just the thread index using `threadIdx()`.

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

```
function thread_kernel(input)
    i = threadIdx().x
    j = threadIdx().y
    k = threadIdx().z

    x, y, z = size(input)
    if i <= x && j <= y && k <= z
        input[i, j, k] = i + j + k
    end
    return nothing
end
```

```
b = 2×2×2 CuArray{Float32, 3, CUDA.DeviceMemory}:
[:, :, 1] =
 1.0  1.0
 1.0  1.0

[:, :, 2] =
 1.0  1.0
 1.0  1.0
```

```
b = CUDA.ones(Float32, (2, 2, 2))
```

```
@cuda thread_kernel(b);
```

```
2×2×2 CuArray{Float32, 3, CUDA.DeviceMemory}:
[:, :, 1] =
 3.0  1.0
 1.0  1.0

[:, :, 2] =
 1.0  1.0
 1.0  1.0
```

```
b
```

This is why we have to set the `threads=` argument!!! Without doing so, we only use default numbers; 1 block and 1 thread. Thus, the singular thread at (1, 1, 1) calculated 3.0.

```
@cuda threads=size(b) thread_kernel(b);
```

```
2×2×2 CuArray{Float32, 3, CUDA.DeviceMemory}:
[:, :, 1] =
 3.0  4.0
 4.0  5.0

[:, :, 2] =
 4.0  5.0
 5.0  6.0
```

```
b
```

In specifying the threads, we get a cube of threads of size (2, 2, 2) so we get 8 threads total! However, we don't always set threads to the size of your data. This would be terrible as the maximum number of threads in a block is usually 1024, meaning that the amount of parallelism per block is limited.

Ex. size limitation:

```
big_3y =
2×1025×2 CuArray{Float32, 3, CUDA.DeviceMemory}:
[:, :, 1] =
 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  …  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0     1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0

[:, :, 2] =
 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  …  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0     1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0
```

```
1 big_3y = CUDA.ones(Float32, (2, 1025, 2))
```

Error message from CUDA

> Number of threads in y-dimension exceeds device limit (1025 > 1024).

Show stack trace...

```
1 @cuda threads=size(big_3y) thread_kernel(big_3y);
```

By blocks

To process data where one dimension (but not necessarily the other) fits inside one block only AND process rows and columns independently, we can assign each column to a block via `blockIdx()` and each row to a thread via `threadIdx()` (when looking at 2D inputs where n_rows <= 1024)!

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

```
function block_kernel(input)
    i = threadIdx().x
    j = blockIdx().x

    x, y = size(input)
    if i <= x && j <= y
        input[i, j] = i + j
    end
    return nothing
end
```

```
c = 2×4 CuArray{Float32, 2, CUDA.DeviceMemory}:
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
```

```
c = CUDA.ones(Float32, (2, 4))
```

```
@cuda threads=size(c, 1) blocks=size(c, 2) block_kernel(c);
```

Here, we set blocks to the number of columns because we want to do column-wise calculations (like sample-wise!). Threads within a block can share memory, allowing values within a column (aka block) to work with one another.

```
2×4 CuArray{Float32, 2, CUDA.DeviceMemory}:
 2.0  3.0  4.0  5.0
 3.0  4.0  5.0  6.0
```

```
c
```

Ex. size limitation:

```
big_2x = 1025×4 CuArray{Float32, 2, CUDA.DeviceMemory}:
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
 ⋮
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
```

```
1 big_2x = CUDA.ones(Float32, (1025, 4))
```

Error message from CUDA

> Number of threads in x-dimension exceeds device limit (1025 > 1024).

Show stack trace...

```
1 @cuda threads=size(big_2x, 1) blocks=size(big_2x, 2) block_kernel(big_2x);
```

```
big_2y =
2×1025 CuArray{Float32, 2, CUDA.DeviceMemory}:
 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  …  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0     1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0
```

```
1 big_2y = CUDA.ones(Float32, (2, 1025))
```

```
1 @cuda threads=size(big_2y, 1) blocks=size(big_2y, 2) block_kernel(big_2y);
```

```
2×1025 CuArray{Float32, 2, CUDA.DeviceMemory}:
 2.0  3.0  4.0  5.0  6.0  7.0  8.0   9.0  …  1022.0  1023.0  1024.0  1025.0  1026.0
 3.0  4.0  5.0  6.0  7.0  8.0  9.0  10.0     1023.0  1024.0  1025.0  1026.0  1027.0
```

```
1 big_2y
```

By index

Most commonly, we map our data into our grid by global indexing using `blockIdx()`, `blockDim()`, and `threadIdx()`. We assign each value a unique index, introducing the block dimensions so it knows to skip everything handled by previous blocks. In doing so, we are not limited by the dimension!

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

```
function index_kernel(input)
    i = (blockIdx().x - 1) * blockDim().x + threadIdx().x
    j = (blockIdx().y - 1) * blockDim().y + threadIdx().y

    x, y = size(input)
    if i <= x && j <= y
        input[i, j] = i + j
    end
    return nothing
end
```

```
d = 2×2 CuArray{Float32, 2, CUDA.DeviceMemory}:
 1.0  1.0
 1.0  1.0
```

```
d = CUDA.ones(Float32, (2, 2))
```

We calculate `cld()` to do ceiling division of the input dimension by the number of threads per block (32). Thus, we will have the minimum number of blocks needed to encapsulate all of the data.

This does mean that some threads will remain unused, i.e. when the total number of threads/block times the number of blocks is larger than the number of data points. Smart parameterization can help avoid this, but it's not always feasible to avoid it entirely.

```
@cuda threads=(32, 32) blocks=(cld(size(d, 2), 32), cld(size(d, 2), 32))
index_kernel(d);
```

```
2×2 CuArray{Float32, 2, CUDA.DeviceMemory}:
 2.0  3.0
 3.0  4.0
```

```
d
```

Ex. no size limitation:

```
big_2xy =
1025×1025 CuArray{Float32, 2, CUDA.DeviceMemory}:
 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  …  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0     1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0     1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0     1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0     1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  …  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0     1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0
 ⋮                        ⋮              ⋱                    ⋮
 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0     1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  …  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0     1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0     1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0     1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0     1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0
```

```
big_2xy = CUDA.ones(Float32, (1025, 1025))
```

```
@cuda threads=(32, 32) blocks=(cld(size(big_2xy, 2), 32), cld(size(big_2xy, 2), 32))
index_kernel(big_2xy);
```

```
1025×1025 CuArray{Float32, 2, CUDA.DeviceMemory}:
    2.0     3.0     4.0     5.0     6.0  …  1022.0  1023.0  1024.0  1025.0  1026.0
    3.0     4.0     5.0     6.0     7.0     1023.0  1024.0  1025.0  1026.0  1027.0
    4.0     5.0     6.0     7.0     8.0     1024.0  1025.0  1026.0  1027.0  1028.0
    5.0     6.0     7.0     8.0     9.0     1025.0  1026.0  1027.0  1028.0  1029.0
    6.0     7.0     8.0     9.0    10.0     1026.0  1027.0  1028.0  1029.0  1030.0
    7.0     8.0     9.0    10.0    11.0  …  1027.0  1028.0  1029.0  1030.0  1031.0
    8.0     9.0    10.0    11.0    12.0     1028.0  1029.0  1030.0  1031.0  1032.0
    ⋮                                    ⋱     ⋮
 1021.0  1022.0  1023.0  1024.0  1025.0     2041.0  2042.0  2043.0  2044.0  2045.0
 1022.0  1023.0  1024.0  1025.0  1026.0  …  2042.0  2043.0  2044.0  2045.0  2046.0
 1023.0  1024.0  1025.0  1026.0  1027.0     2043.0  2044.0  2045.0  2046.0  2047.0
 1024.0  1025.0  1026.0  1027.0  1028.0     2044.0  2045.0  2046.0  2047.0  2048.0
 1025.0  1026.0  1027.0  1028.0  1029.0     2045.0  2046.0  2047.0  2048.0  2049.0
 1026.0  1027.0  1028.0  1029.0  1030.0     2046.0  2047.0  2048.0  2049.0  2050.0
```

```
big_2xy
```

Calling functions

A thread can jump into helper functions as well! However, we must ensure that the helper function is specialized (ie. type stable) at compile time. This ensures that the GPU doesn't crash from having to figure out the types at runtime.

```
md"""
## Calling functions
A thread can jump into helper functions as well! However, we must ensure that the helper function is specialized (ie. type stable) at compile time. This ensures that the GPU doesn't crash from having to figure out the types at runtime.
"""
```

```
# do more testing on this; i'm not sure why but the type instability error only results from when we use @view (or broadcast??).
```

```
calculations_unstable! (generic function with 1 method)
```

```
function calculations_unstable!(x, f)
    x .= f.(x)
end
```

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

```
function main_kernel_unstable(input, fxn)
    i = threadIdx().x
    calculations_unstable!(@view(input[i, :]), fxn)
    return nothing
end
```

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

```
math_stuff(x) = x + 1
```

```
e = 2×2 CuArray{Float32, 2, CUDA.DeviceMemory}:
 1.0  1.0
 1.0  1.0
```

```
e = CUDA.ones(Float32, (2,2))
```

Error message from GPUCompiler

InvalidIRError: compiling MethodInstance for Main.var"workspace#5".main_kernel_unstable(::CuDeviceMatrix{Float32, 1}, ::typeof(Main.var"workspace#5".math_stuff)) resulted in invalid LLVM IR

Reason: unsupported dynamic function invocation (call to calculations_unstable!(x, f) @ Main.var"workspace#5" /home/golem/scratch/munozc/GPU_workshop/kernels-workshop/notebook.jl#==#3e25a74b-f32c-4896-9c88-0b038cccb157:1)

Stacktrace:

[1] **main_kernel_unstable**

@ /home/golem/scratch/munozc/GPU_workshop/kernels-workshop/notebook.jl#==#04541ad0-120a-488d-b31a-4381e43f5e89:3

Hint: catch this exception as `err` and call `code_typed(err; interactive = true)` to introspect the erroneous code with Cthulhu.jl

Show stack trace...

```
@cuda threads=size(e, 1) main_kernel_unstable(e, math_stuff);
```

But if we redefine the `calculations()` function:

```
calculations_stable! (generic function with 1 method)
```

```
function calculations_stable!(x::X, f::F) where {X, F}
    x .= f.(x)
end
```

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

```
function main_kernel_stable(input, fxn)
    i = threadIdx().x
    calculations_stable!(@view(input[i, :]), fxn)
    return nothing
end
```

```
@cuda threads=size(e, 1) main_kernel_stable(e, math_stuff);
```

```
2×2 CuArray{Float32, 2, CUDA.DeviceMemory}:
 2.0  2.0
 2.0  2.0
```

```
e
```

Synchronization

Sometimes, we need to sync threads within a block to ensure we don't overwrite data that another thread has worked on!

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

```
function sync_kernel(input)
    i = threadIdx().x
    j = blockDim().x
    data = input[i]
    sync_threads()
    input[j - i + 1] = data
    return nothing
end
```

```
f = 5×3 CuArray{Int64, 2, CUDA.DeviceMemory}:
 1  1  1
 2  2  2
 3  3  3
 4  4  4
 5  5  5
```

```
f = CuArray([Vector(1:5) Vector(1:5) Vector(1:5)])
```

```
@cuda threads=length(f) sync_kernel(f);
```

```
5×3 CuArray{Int64, 2, CUDA.DeviceMemory}:
 5  5  5
 4  4  4
 3  3  3
 2  2  2
 1  1  1
```

```
1 f
```

Some additional uses of synchronization:

To verify or count conditions across threads (our predicates **pred**):

- `sync_threads_count(pred)` : returns the number of threads for which pred was true
- `sync_threads_and(pred)` : returns true if pred was true for all threads
- `sync_threads_or(pred)` : returns true if pred was true for any thread

To maintain multiple thread synchronizations via execution (i.e. have different sync_threads in different situations), we can use:

- `barrier_sync()`

To maintain multiple thread synchronizations via memory (e.g. to make sure parts of the memory are visible to other threads at the correct moment), we can use:

- `threadfence_block` : ensure memory ordering for all threads in the block
- `threadfence` : the same, but for all threads on the device
- `threadfence_system` : the same, but including host threads and threads on peer devices

Shared memory

To communicate between threads, we can utilize static and dynamic shared arrays.

Static

For when we know the amount of shared memory beforehand.

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

```
function static_kernel(input::CuDeviceArray{T}) where T
    i = threadIdx().x # row
    j = blockIdx().x # col
    index = (j - 1) * 5 + i

    data = CuStaticSharedArray(T, 5)
    @inbounds begin
        data[5 - i + 1] = input[index]
        sync_threads()
        input[index] = data[i]
    end
    return nothing
end
```

Note: we can do `@inbounds` here to indicate when we know 100% that the index is in bounds. Otherwise, indexing a CuArray will do bounds checking by defualt and throwing the error can be very costly!

```
g = 5×3 CuArray{Int64, 2, CUDA.DeviceMemory}:
 1  1  1
 2  2  2
 3  3  3
 4  4  4
 5  5  5
```

```
g = CuArray([Vector(1:5) Vector(1:5) Vector(1:5)])
```

```
@cuda threads=5 blocks=3 static_kernel(g);
```

```
5×3 CuArray{Int64, 2, CUDA.DeviceMemory}:
 5  5  5
 4  4  4
 3  3  3
 2  2  2
 1  1  1
```

```
g
```

Thus, different threads are able to modify the same array without interfering with one another.

Dynamic

For when we don't know the amount of shared memory beforehand.

Here, we pass the size of the shared memory (`shmem=`) **in bytes** as an argument to the kernel.

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

```
function dynamic_kernel(input::CuDeviceArray{T}) where T
    i = threadIdx().x # row
    j = blockIdx().x # col
    n = blockDim().x # n_rows
    index = (j - 1) * n + i

    data = CuDynamicSharedArray(T, n)
    @inbounds begin
        data[n - i + 1] = input[index]
        sync_threads()
        input[index] = data[i]
    end
    return nothing
end
```

```
h = 5×3 CuArray{Int64, 2, CUDA.DeviceMemory}:
 1  1  1
 2  2  2
 3  3  3
 4  4  4
 5  5  5
```

```
h = CuArray([Vector(1:5) Vector(1:5) Vector(1:5)])
```

```
@cuda threads=size(h, 1) blocks=size(h, 2) shmem=sizeof(h[:, 1]) dynamic_kernel(h);
```

Because we set the shared memory to the size of one column in h, we will share one column across all the threads in a block so that the threads do not interfere with each other.

```
5×3 CuArray{Int64, 2, CUDA.DeviceMemory}:
 5  5  5
 4  4  4
 3  3  3
 2  2  2
 1  1  1
```

```
h
```

We can also introduce the parameter `offset`: the offset in bytes from the start of the shared memory.

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

```
function dynamic_kernel_multi(input::CuDeviceArray{T}) where T
    i = threadIdx().x # row
    j = blockIdx().x # col
    n = blockDim().x # n_rows
    index = (j - 1) * n + i

    data = CuDynamicSharedArray(T, n)
    data2 = CuDynamicSharedArray(T, n, sizeof(data))

    @inbounds begin
        data[n - i + 1] = input[index]
        data2[n - i + 1] = input[index]
        sync_threads()
        input[index] = data[i]+data[i]
    end
    return nothing
end
```

```
h2 = 5×3 CuArray{Int64, 2, CUDA.DeviceMemory}:
 1  1  1
 2  2  2
 3  3  3
 4  4  4
 5  5  5
```

```
h2 = CuArray([Vector(1:5) Vector(1:5) Vector(1:5)])
```

```
@cuda threads=size(h2, 1) blocks=size(h2, 2) shmem=sizeof(h2[:, 1])*2
dynamic_kernel_multi(h2);
```

```
5×3 CuArray{Int64, 2, CUDA.DeviceMemory}:
 10  10  10
  8   8   8
  6   6   6
  4   4   4
  2   2   2
```

```
h2
```

Atomic operations

These are operations that execute read/modify/write in one step, such that when working with shared memory, there are no interruptions. Essentially locks a piece of data while it's being operated on so nothing else can touch it!

Low-level

These take pointer inputs (via `pointer(CuArray)`). Some supported operations are:

- binary operations: `add`, `sub`, `or`, `xor`, `min`, `max`, `xchg`, `inc`, `dec`
- compare-and-swap: `cas`

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

```
function low_atomic_kernel(input)
    CUDA.atomic_add!(pointer(input), Float32(1))
    return nothing
end
```

```
o = 2×2 CuArray{Float32, 2, CUDA.DeviceMemory}:
 1.0  1.0
 1.0  1.0
```

```
o = CUDA.ones(Float32, 2, 2)
```

```
@cuda threads=size(o) low_atomic_kernel(o);
```

```
2×2 CuArray{Float32, 2, CUDA.DeviceMemory}:
 5.0  1.0
 1.0  1.0
```

```
o
```

Without using the atomics, all threads would have read the initial value (1.0) then simultaneously overwritten one another, which would have resulted in 2.0. We can see from the result that using `CUDA.atomic_add!()`, the threads are forced to essentially take turns and properly accumulate the sums to 5.0.

Note: Atomics other than CUDA.atomic_add!() are not supported for float values.

High-level

We can also use the `CUDA.@atomic` macro. This will automatically convert inputs to the appropriate type and other fallbacks but may have issues with the `Base.@atomic` macro...

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

```
function high_atomic_kernel(input)
    CUDA.@atomic input[1, 1] += 1
    return nothing
end
```

```
p = 2×2 CuArray{Float32, 2, CUDA.DeviceMemory}:
 1.0  1.0
 1.0  1.0
```

```
p = CUDA.ones(Float32, 2, 2)
```

```
@cuda threads=(2,2) high_atomic_kernel(p);
```

```
2×2 CuArray{Float32, 2, CUDA.DeviceMemory}:
 5.0  1.0
 1.0  1.0
```

```
p
```

Basically the exact same thing as the low-level but we can just write our code as usual instead of

pointer-ing and type-ing, etc.

Dynamic parallelism

For things like recursive functions, we can utilize dynamic parallelism. This essentially spawns a new grid of threads by launching an additional kernel through `@cuda ...` (kernel from inside a kernel)!

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

```
function update_kernel(input, i, j)
    input[i, j] -= 0.1
    return nothing
end
```

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

```
function check_kernel(input)
    i = threadIdx().x
    j = threadIdx().y
    while input[i, j] > 0.5
        @cuda dynamic=true threads=1 update_kernel(input, i, j)
        device_synchronize()
    end
    return nothing
end
```

```
q = 2×2 CuArray{Float32, 2, CUDA.DeviceMemory}:
 0.262174  0.485098
 0.419075  0.679126
```

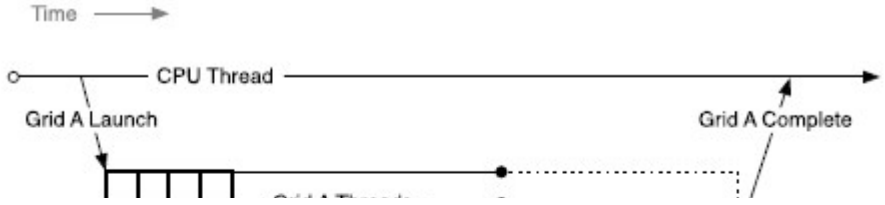
```
q = CUDA.rand(Float32, 2, 2)
```

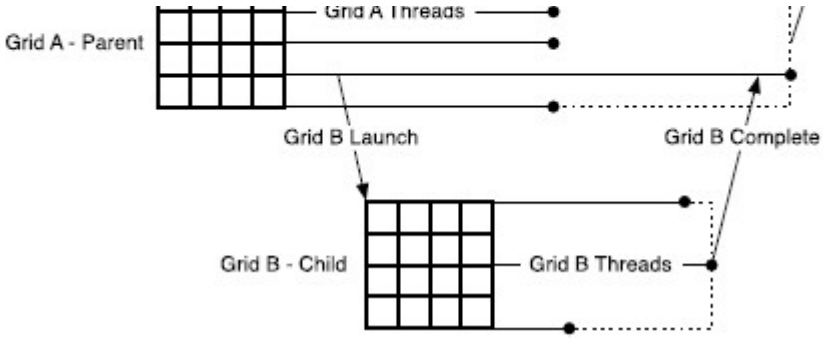
```
@cuda threads=(2, 2) check_kernel(q);
```

```
2×2 CuArray{Float32, 2, CUDA.DeviceMemory}:
 0.262174  0.485098
 0.419075  0.479126
```

```
q
```

Why `device_synchronize()`?


Time
CPU Thread
Grid A Launch
Grid A Complete


Grid A - Parent
Grid A Threads
Grid B Launch
Grid B Complete
Grid B - Child
Grid B Threads

https://docs.nvidia.com/cuda/cuda-programming-guide/04-special-topics/dynamic-parallelism.html

Essentially when we launch our parent grid (`check_kernel`), the second `@cuda` call will launch a child grid (`update_kernel`). Since they are asynchronous, the parent kernel will continue executing without

waiting for the child to finish.

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

```
function check_kernel_bad(input)
    i = threadIdx().x
    j = threadIdx().y
    while input[i, j] > 0.5
        @cuda dynamic=true threads=1 update_kernel(input, i, j)
        # device_synchronize()
    end
    return nothing
end
```

```
r = 2×2 CuArray{Float32, 2, CUDA.DeviceMemory}:
 15.0609  15.9713
 15.0473  15.2099
```

```
r = CUDA.rand(Float32, 2, 2) .+ 15
```

```
@cuda threads=(2, 2) check_kernel_bad(r);
```

```
2×2 CuArray{Float32, 2, CUDA.DeviceMemory}:
 -4.03909  -5.02869
 -4.05271  -4.19012
```

```
r
```

This results in decrementing the value way too far because additional updates have been launched before the first child has finished modifying the value. Adding `device_synchronize` allows the parent to first wait for the child to fully complete its work before continuing with an additional while loop.

Demo time! :D