CUDA.jl

By: George Rauta

What is CUDA

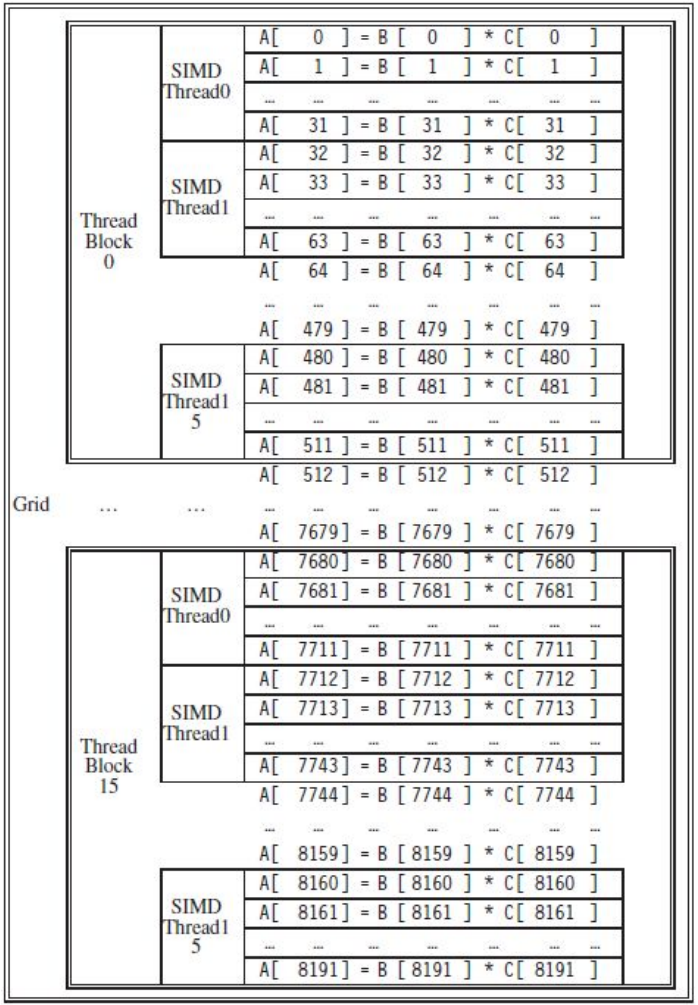
- Compute Unified Device Architecture (CUDA)
- Proprietary API made by Nvidia to allow software to use any Nvidia GPU
- Built as an extension of C language
- Comes with extra libraries like:
  - cuBLAS – CUDA Basic Linear Algebra Subroutines
  - cuSPARSE – CUDA Sparse Matrices
  - cuSOLVER – CUDA Dense/Sparse Linear Solvers

CPU vs GPU

- CPU (Host)
  - Meant as a general purpose processor
  - Has a few fast cores for parallel computing
  - Relies on cache to make up for slow memory accesses
- GPU (Device)
  - Specifically designed for SIMD like operations
  - Has a massive amount of slower cores
  - Relies on heavy computation to make up for slow memory accesses
  - Has its own pool of memory called VRAM as well as access to regular RAM

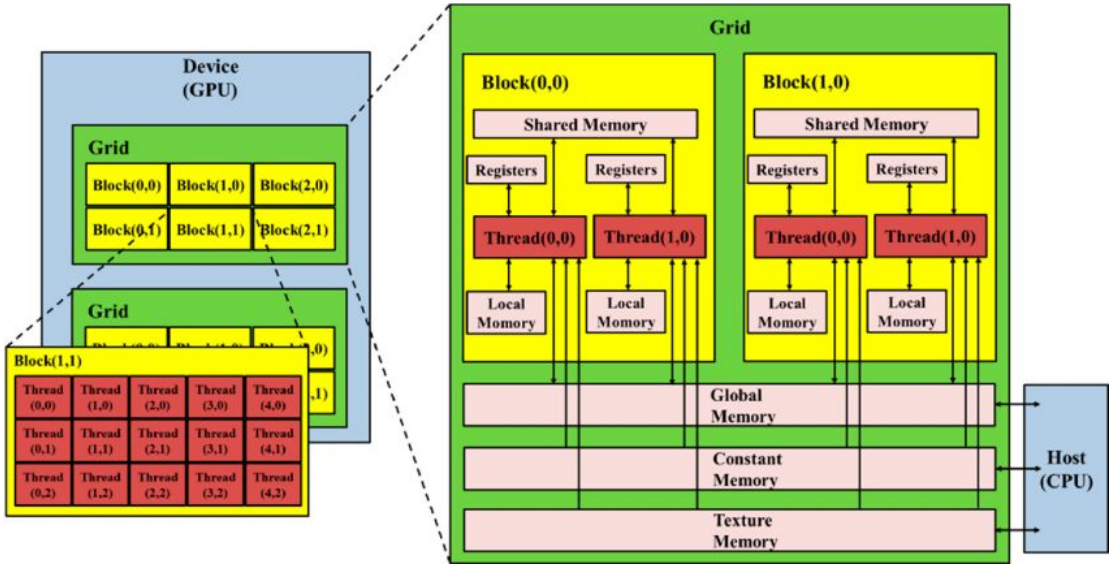
Computational Hierarchy

- Kernel
  - Basically just a function that runs on the GPU
  - Computation is broken up into many different threads
- Multiple levels of computation.
  - **Grid**: the effective work space for a kernel
  - **Block**: partitions of a grid that have their own memory pool
  - **Warps**: partitions of a block, independently scheduled, usually 32 threads
  - **Threads**: do some small unit of work



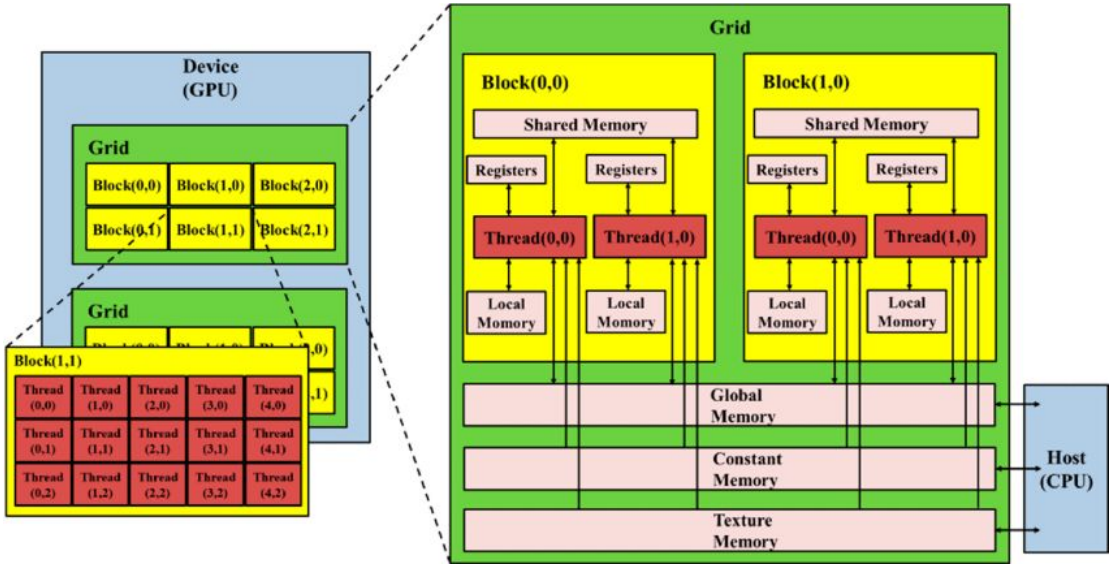
Memory Hierarchy

- Registers
  - Used by individual threads
  - Very fast access
- Shared Memory
  - Independent pools given to each block
  - Allocated in the kernel
  - Fast access
- Global Memory
  - Can be used by any thread
  - Usually for read/writing results or initial data
  - Slow access



Memory Hierarchy

- Local Memory
  - Allocated in the Global Memory pool
  - Can only be accessed by a single thread
  - As slow as global access
- Unified Memory
  - Allocated in RAM, outside GPU
  - Accessible by both GPU and CPU
  - Slowest access time
- Some other types, Constant/Texture but is not accessible through CUDA.jl



Structure of a Kernel

- Like a regular function with some constraints
- Cannot return a value, must be done in-place
- Each kernel has access to its own IDs
- Can allocated shared memory to be used by all kernels in the block
- Shared memory is allocated like in C, size/type must be known by compile time

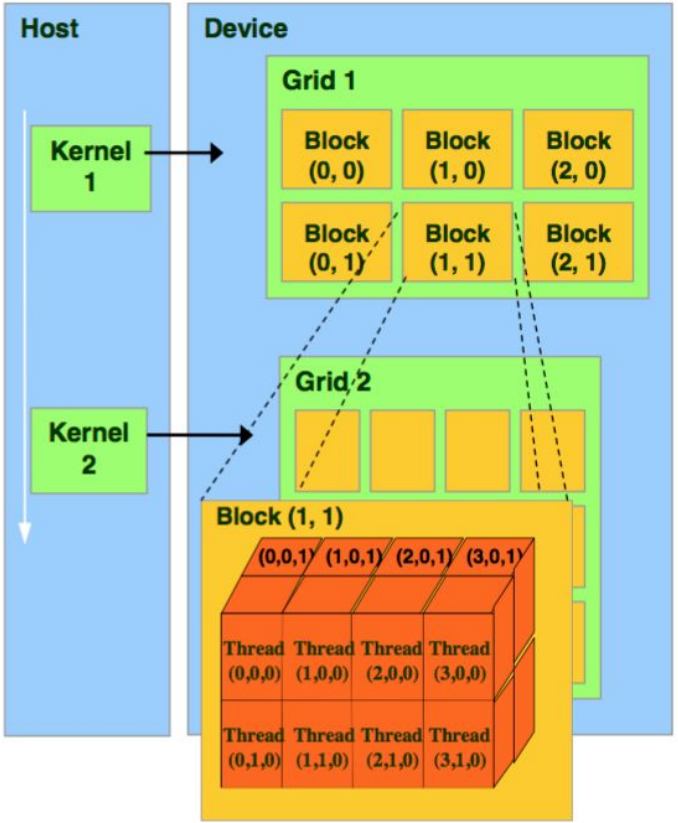
```
function gpu_add3!(y, x)
    index = (blockIdx().x - 1) * blockDim().x + threadIdx().x
    stride = gridDim().x * blockDim().x
    for i = index:stride:length(y)
        @inbounds y[i] += x[i]
    end
    return
end

numblocks = ceil(Int, N/256)

fill!(y_d, 2)
@cuda threads=256 blocks=numblocks gpu_add3!(y_d, x_d)
@test all(Array(y_d) .== 3.0f0)
```

Calling a Kernel

- Usually want to call with @cuda
- Will compile the CUDA kernel on first run can either run it immediately or save the function for later use. (**launch**)
- Can assign resources for kernel now(**threads,blocks**)
- Can assign shared memory to be used by kernel (**shmem**)
- Run async of the CPU, CuArrays will sync on access



Performance Programming

- DO NOT SCALAR INDEX
  - When accessing an spot in an array, do not a getindex call
  - If this happens the computation will be moved to the CPU
  - Should always run with CUDA.allowscalar(false)
- Can use CUDA.registers on a saved kernel to see how many registers it uses.
  - The more registers a thread needs the fewer threads a block can run
  - Can restrict number of registers using max_register, will spill contents into local memory
- Threads in a Warp should access adjacent memory locations
  - This is similar to CPU cache accessing
  - Called coalesced read/write in GPU terminology.
  - Threads in a Warp have their memory accesses combined into one

Performance Programming

- Use 32-bit integers and avoid StepRanges
  - Can usually use a while-loop
- Can use CUDA.registers on a saved kernel to see how many registers it uses.
  - The more registers a thread needs the fewer threads a block can run
  - Can restrict number of registers using max_register, will spill contents into local memory
- Try to remove exceptions/bounds checking
  - This is performant code
- Try to avoid branching conditions
  - Nature of GPU means there is wasted computation there
- CUDA.memory to see amount of memory used by kernel
  - Try to reduce amount of global/local memory being used
- Fuse Kernel calls
  - Individual kernels have to save their results in global memory.
  - If different kernels are operating on the same data serially, combine them into one single kernel

Kernel Example

```
function dec_cu_c_wedge_product!(::Type{Tuple{0,1}}, wedge_terms, f, α, primal_vertices)
  num_threads = CUDA.max_block_size.x
  num_blocks = min(ceil(Int, length(wedge_terms) / num_threads), CUDA.max_grid_size.x)

  @cuda threads=num_threads blocks=num_blocks dec_cu_ker_c_wedge_product_01!(wedge_terms, f, α, primal_vertices)
end


function dec_cu_ker_c_wedge_product_01!(wedge_terms::CuDeviceArray{T}, f, α, primal_vertices) where T
  index = (blockIdx().x - Int32(1)) * blockDim().x + threadIdx().x
  stride = gridDim().x * blockDim().x
  i = index
  @inbounds while i <= Int32(length(wedge_terms))
    wedge_terms[i] = T(0.5) * α[i] * (f[primal_vertices[i, Int32(1)]] + f[primal_vertices[i, Int32(2)]])
    i += stride
  end
  return nothing
```

Links for Further Reading

- https://cuda.juliagpu.org/stable/tutorials/performance/
- https://github.com/JuliaComputing/Training/blob/master/AdvancedGPU/2-2-kernel_analysis_optimization.ipynb
- https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html