Notes 8: GPUs

Chris Paciorek

2025-02-13

Table of contents

Introduction . . . 1
GPU overview . . . 2
Accessing a GPU on the SCF . . . 2
GPU availability . . . 3
Basic GPU offloading . . . 3
Matrix multiplication example . . . 4
Multiple dispatch with GPU operations . . . 5
Copying data to/from the GPU . . . 6
64- vs 32-bit numbers . . . 6
Interacting with NVIDIA's CUDA libraries . . . 7
Vectorized example . . . 8
GPU architecture . . . 9
Overview . . . 9
Glossary . . . 10
Threads, blocks and grids . . . 10
GPU execution and efficiency . . . 11
GPU monitoring - details . . . 11
Using `nvidia-smi` for hardware and activity . . . 11
`deviceQuery` for hardware information . . . 13
GPU kernels . . . 15
Basic example . . . 16
Multiple blocks . . . 16
Larger computations . . . 17
Final comments (for now) . . . 18
When to use the GPU . . . 18

Introduction

This document is the eighth of a set of notes, this document focusing on the parallelization via GPUs. The notes are not meant to be particularly complete in terms of useful functions (Google and LLMs can now provide that quite well), but rather to introduce the language and consider key programming concepts in the context of Julia.

Given that, the document heavily relies on demos, with interpretation in some cases left to the reader.

> ⚠ GPU code run separately from rendering
>
> In most cases, I have set the code chunks to not execute when rendering this document, since that would require rendering to HTML/PDF on a machine with a GPU. In some cases I've copied in output from running the chunk manually outside of the rendering process, sometimes as part of code comments.

GPU overview

(This is repeated from the last set of notes.)

Some features of GPUs:

- Many processing units (thousands) that are slow individually compared to the CPU but provide massive parallelism.
- They have somewhat more limited memory (though modern GPUs like the A100 can have 80 GB of GPU memory).
- They can only use data in their own memory, not in the CPU's memory, so one must transfer data back and forth between the CPU (the *host*) and the GPU (the *device*). This copying can, in some computations, constitute a very large fraction of the overall computation. So it is best to create the data and/or leave the data (for subsequent calculations) on the GPU when possible and to limit transfers.
- For large-scale work, a GPU server will have multiple GPUs, e.g., 8 A100s or 8 H100s. We won't discuss running code on multiple GPUs nor on GPUs distributed across multiple machines.
- We'll focus on NVIDIA GPUs and NVIDIA's CUDA platform for running code on the GPU, but other manufacturers produce other kinds of GPUs accessed via different interfaces.

Accessing a GPU on the SCF

We have one (somewhat old) GPU on the `gpu` partition that is available for all to use equally.

After logging into an SCF standalone/login machine such as `arwen`, `gandalf`, or `radagast`, you can ask for access to a GPU through the SLURM scheduler.

```
srun -p gpu --gpus=1 --pty bash
nvidia-smi
```

A large variety of other, more powerful GPUs, are available through "partitions" that were purchased by faculty research grants. You're welcome to use them, but your job could be preempted (killed) at any moment by higher-priority jobs.

The partitions with GPUs include `jsteinhardt`, `yss`, and `yugroup`. Most of the GPUs are in `jsteinhardt`.

Here's how you'd request an A100 GPU in the `jsteinhardt` partition.

```
srun -p gpu -jsteinhardt --gpus=A100:1 --pty bash
nvidia-smi
```

One can also use `sbatch` to run a background job.

More details on using the SCF cluster and the GPUs in it are available in our quick start guide.

GPU availability

If you're on a machine with a (NVIDIA) GPU, you should be able to run `nvidia-smi` from the command line to see what GPUs are available to you. Depending on how the system is set up, if you're running through a scheduler such as Slurm, you may not see all the GPUs that are physically on the machine.

The environment variable `CUDA_VISIBLE_DEVICES` will show the ID(s) of the GPU(s) available to you. You generally won't need to use this, but it can be useful to check this variable.

```
#| eval: false
echo $CUDA_VISIBLE_DEVICES
```

We can check in Julia. Here's what we would see if a GPU is available (also requiring that CUDA.jl is set up to use a GPU even if one is available).

```
using CUDA
CUDA.has_cuda_gpu()
```

```
true
```

> **!** Invoke `using CUDA` on a machine with a GPU
>
> I invoked `using CUDA` on a machine without a GPU and it caused some pre-compilation that then prevented me from using the GPU on a machine with a GPU, giving messages like "Error: CUDA.jl could not find an appropriate CUDA runtime to use". Untangling that took some effort (in particular manually removing CUDA-related Julia packages via `rm -r ~/.julia/compiled/v1.10/CUDA*`).
> So if you're on a cluster or other system with multiple machines (such as the SCF), some with and without GPUs, it's best if you only use `CUDA` on a machine with a GPU.

There are various other Julia CUDA package functions that we can explore for checking details of the GPU, such as `CUDA.total_memory()`.

Basic GPU offloading

Before we get into the complexities of how GPUs work and what one has to do to write one's own code to use the GPU, let's see how easily one can offload standard calculations (in particular linear algebra) to the GPU.

The main package we'll use is the CUDA package. It provides:

- the `CuArray` type for working with arrays on the GPU,
- wrappers for functions in NVIDIA's CUDA libraries (the NVIDIA CUDA toolkit is distinct from the Julia CUDA package) that allow one to use Julia functions/operators with CuArrays, and
- tooling for writing CUDA kernels in Julia.

In this section we'll see the first two, before looking at writing kernels in the last section of these notes.

Note that this kind of offloading of linear algebra and vectorized computations to the GPU can be done similarly with PyTorch, JAX and CuPy in Python.

Matrix multiplication example

Matrix multiplication is simple, but it's at the heart of a lot of algorithms, including deep learning, so it's a simple, but non-trivial example of a situation where we could offload a part of a computation to the GPU for big speedups.

First we'll set up the matrices, generating them on the host/CPU (and they'll therefore be in CPU memory) and transferring to GPU (device) memory.

```
using BenchmarkTools
using CUDA
using LinearAlgebra

n = 7000;

x = randn(n, n);
y = randn(n, n);
x_gpu = CuArray(x);
y_gpu = CuArray(y);

## These use 64-bit numbers:
typeof(x)
# Matrix{Float64} (alias for Array{Float64, 2})
typeof(x_gpu)
# CuArray{Float64, 2, CUDA.Mem.DeviceBuffer}

CUDA.total_memory()
# 85097971712         # This is using an A100 GPU with 85 GB GPU memory.
CUDA.used_memory()
# 1600000000
n*n*8*2               # "Theoretical" memory use.
# 1600000000
```

Now let's compare the speed of the multiplication on CPU (host) vs. GPU (device). As usual we'll put our computation into functions to allow for JIT compilation and avoid use of global variables whose type can change.

```
LinearAlgebra.BLAS.set_num_threads(1);  # To keep things simple but perhaps unfair to the CPU.

function matmult(x, y)
    z = x * y;
    return z
end
```

```
@btime out = matmult(x, y);
# 10.280 s (2 allocations: 373.84 MiB)

@btime CUDA.@sync z_gpu = matmult(x_gpu, y_gpu);
# 41.270 ms (51 allocations: 1.19 KiB)

z_gpu = matmult(x_gpu, y_gpu);
typeof(z_gpu)
# CuArray{Float64, 2, CUDA.DeviceMemory}
z[1]
# ERROR: Scalar indexing is disallowed.
# Invocation of getindex resulted in scalar indexing of a GPU array.

# Moving the result back to the host.
z = Array(z_gpu);
z[1]
```

So the speedup was about 250-fold!

In class we'll use `nvidia-smi` and `top` to monitor the GPU and CPU.

> **!** Timing GPU code
>
> Some comments on timing GPU code:
> - I believe it's reasonable to use `@btime` to time GPU code
> - We could also use `CUDA.@time`.
> - Since GPU code is executed asynchronously, we should use `CUDA.@sync` or `synchronize` to make sure the execution is finished.

Multiple dispatch with GPU operations

Given that we use `*` with two `CuArray` matrices, Julia's multiple dispatch system is clearly involved in determining what code to call to execute the matrix multiplication on the GPU.

Digging around a bit in the source code for the `LinearAlgebra` and `CUDA` packages, it looks like the result of the dispatch is to call `LinearAlgebra.generic_matmatmul!`, with the method that works for CuArrays in the `lib/cublas/linalg.jl` file in the CUDA package. That presumably invokes the cuBLAS versions of the BLAS functions found in `lib/cublas/libcublas.jl` and `lib/cublas/wrappers.jl` where we see calls out to native C-based cuBLAS functions. But I don't see where the C code is.

At a high level that makes sense, though the details of all the wrapping and type inheritance/dispatch that is going on might be hard to trace through carefully.

Copying data to/from the GPU

If we copy a lot of data back and forth to the GPU, that can end up being a large fraction of the time of a GPU-based calculation. We'd like to:

- generate data on the GPU if we can,
- keep data on the GPU for multiple steps in a calculation, and
- avoid copying output back to the device if possible.

Let's see how the time of copying compares to the time of doing the matrix multiplication.

```
x = randn(n, n);

# To the GPU:
@btime CUDA.@sync x_gpu = CuArray(x);
# 61.138 ms (10 allocations: 288 bytes)
CUDA.@time CUDA.@sync x_gpu = CuArray(x);
#  0.071553 seconds (12 CPU allocations: 320 bytes) (1 GPU allocation: 373.840 MiB, 0.03% memmgmt tim

# From the GPU:
CUDA.@time CUDA.@sync out = Array(z_gpu);
# 0.329900 seconds (7 CPU allocations: 373.840 MiB)
```

So the transfer back takes much longer than the actual calculation!

The time for the transfer back to the host includes the time to allocate space on the host. We'd have to do more work to disentangle the transfer time from the allocation time (e.g., timing `y = Vector{Float64}(undef, n);`).

64- vs 32-bit numbers

In most contexts, computation on a GPU will use 32-bit real numbers for efficiency. This reduces memory use and speeds up computation because half as many bytes are being allocated, copied, and computed with. Of course we want to keep in mind the reduced precision in case that could be a problem for certain computations.

Let's first simply see if matrix multiplication on the CPU is faster with 32-bit numbers.

```
using BenchmarkTools

function matmult(x, y)
    z = x * y
    return z
end

n = 5000;

x = randn(n, n);
y = randn(n, n);
```

```
@btime matmult(x, y);

x32 = convert(Array{Float32}, x);
y32 = convert(Array{Float32}, y);
sizeof(x)
sizeof(x32)

@btime matmult(x32, y32);
```

```
  1.629 s (2 allocations: 190.73 MiB)
  725.687 ms (2 allocations: 95.37 MiB)
```

Indeed it is quite a bit faster.

Now let's consider that on the GPU. We'll also show random number generation directly on the GPU to avoid the cost of transferring data to the GPU.

```
x64_gpu = CUDA.randn(Float64, n, n);
y64_gpu = CUDA.randn(Float64, n, n);
typeof(x64_gpu)
# CuArray{Float64, 2, CUDA.DeviceMemory}

x32_gpu = CUDA.randn(n, n);   # `Float32` is the default.
y32_gpu = CUDA.randn(n, n);
typeof(x32_gpu)
# CuArray{Float32, 2, CUDA.DeviceMemory}

@btime CUDA.@sync z64_gpu = matmult(x64_gpu, y64_gpu);
# 39.353 ms (51 allocations: 1.19 KiB)
@btime CUDA.@sync z32_gpu = matmult(x32_gpu, y32_gpu);
# 38.409 ms (51 allocations: 1.19 KiB)

@btime CUDA.@sync z64 = Array(z64_gpu);
# 298.258 ms (7 allocations: 373.84 MiB)
@btime CUDA.@sync z32 = Array(z32_gpu);
# 147.791 ms (7 allocations: 186.92 MiB)
```

I'm quite surprised the 32-bit multiplication is no faster. I'm not sure what is going on. (If you time the `randn` you should see that the 32-bit randn generation is about half the time of the 64-bit generation.)

It does make sense that the transfer back takes about half as long.

Interacting with NVIDIA's CUDA libraries

Working with objects on the GPU requires interfacing with functions provided by the CUDA Toolkit and its libraries such as `cuBLAS`, `cuRAND`, `cuFFT`, etc. These are provided for Julia with the CUDA package (see `CUDA/<version_string>/lib`) so all that you need installed on the machine with the GPU is the NVIDIA driver software and the Julia CUDA package. You don't actually need NVIDIA's

CUDA Toolkit.

Since Julia is calling out to CUDA functions to run code on the GPU, there may be functions that won't work on a CuArray, but the various things I tried all seem to work.

```
y_gpu = CUDA.randn(3);    # This uses cuRAND.
y_gpu + y_gpu
```

```
3-element CuArray{Float32, 1, CUDA.DeviceMemory}:
 -2.3132815
  0.11570158
 -1.0381625
```

```
sin.(y_gpu)
```

```
3-element CuArray{Float32, 1, CUDA.DeviceMemory}:
 -0.986312
 -0.784628
  0.7079291
```

However, we can't work with scalar elements of the array because that tries to do a non-vectorized calculation on a GPU-based array.

```
y_gpu[1]
```

```
julia> y_gpu[1]
ERROR: Scalar indexing is disallowed.
Invocation of getindex resulted in scalar indexing of a GPU array.
This is typically caused by calling an iterating implementation of a method.
Such implementations *do not* execute on the GPU, but very slowly on the CPU,
and therefore should be avoided.
```

Vectorized example

Here we'll offload a vectorized calculation to the GPU. Note that in this case we cannot set up the calculation as a for loop as one can't loop through scalar operations on GPU in that fashion (more about how to loop on a GPU in the discussion of GPU kernels).

We'll use `.=` for in-place calculation to avoid additional memory allocation.

```
function vec_calc(x)
  x .= tan.(x) .+ 3 .* sin.(x)
  return 0
end

n = 250000000;

x = rand(n);            # 2 GB of data.
x_gpu = CuArray(x);
```

```
# CPU calculation:
@btime vec_calc(x);     # Again a bit unfair to the host/CPU in that we don't exploit multiple CPU co
# 9.041 s (0 allocations: 0 bytes)

# GPU calculation:
# Try to flush out any compilation time.
test = CuArray([1.1, 2.2])
CUDA.@sync vec_calc(test);  # This seems to take surprisingly long.

CUDA.@time CUDA.@sync vec_calc(x_gpu);
# 0.099259 seconds (39.83 k CPU allocations: 2.770 MiB)
CUDA.@time CUDA.@sync vec_calc(x_gpu);
# 0.003745 seconds (88 CPU allocations: 3.672 KiB)
```

I'm not sure why the initial call on `test` takes so long. I'm also not sure what explains the big difference between running the calculation on the full vector the first and second times (including why the first call involves so many separate allocations). The first time represents a 91-fold speedup and the second time a surprising 2400-fold speedup.

I'm also not sure of what is going on behind the scenes in terms of the Julia interpeter and/or compiler processing the `vec_calc` code to allow it to be run via CUDA calls on the GPU. `@code_native vec_calc(x_gpu)` indicates use of the `GPUArrays` package, in particular use of `gpu_call` to run a kernel on the GPU. So I think the main question is) how Julia creates a Julia-coded kernel function (such as this example Julia kernel. (Once the kernel function is created, there seem to be tools to convert LLVM code to code that runs on the GPU (more later).

> Exercise
>
> See how small `n` can be before the time of doing the calculation on the GPU exceeds the time for doing it on the CPU.
> Then, consider how the transfer time to/from the GPU affects the trade-off (in this case presuming the only calculation you were doing was `vec_calc` and that you needed the full output back on the host for a later calculation that can't be done on the GPU.

GPU architecture

Now that we've seen some of what we can do with GPUs, let's consider some about how they work and what that implies about what calculations we can usefully offload to them.

Overview

GPUs have thousands of "slow" (slower than the CPU) cores. Their speed comes from massive parallelization and how the code is executed in parallel on the input data.

Each individual computation or series of computations on the GPU is done in a thread. Threads are organized into blocks and blocks of threads are organized in a grid. The blocks and grids can be 1-,

2-, or 3-dimensional. E.g., you might have a 1-d block of 256 threads, with a grid of 3 x 3 such blocks, for a total of 256×9=2304 threads.

The choice of the grid/block arrangement can affect efficiency. I'm not an expert at this level of detail but we'll see some about this in the kernel example below. Note that using more than 1-dimensional grids and blocks is purely for the conceptual convenience of the programmer and doesn't correspond to anything on the hardware. So for the most part we'll use a one-dimensional grid of blocks and a one-dimensional blocks of threads.

In general you'd want each independent calculation done in a separate thread, but one might want to do a sequence of calculations on each thread. In general, you'll want to pipeline together multiple operations within a computation to avoid copying from CPU to GPU and back. Alternatively, this can be done by keeping the data on the GPU and calling a second kernel.

Glossary

- *streaming multiprocessor* (SM): a set of GPU cores with registers and shared memory; each GPU has multiple SMs
- *threads*: parallel execution units
- *blocks*: threads are grouped into blocks
- *grid*: blocks are grouped into a grid
- *warp*: a group of (usually) 32 threads that operate together
- *kernel*: a function that runs on the GPU (usually one thinks of a C function, but we'll write Julia-coded kernels)
- *device*: the GPU and its memory
- *host*: the CPU and its memory

Threads, blocks and grids

Some facts about the organization of threads on a GPU.

Threads:

- threads grouped into a warp (on A100 and possibly most others, 32 threads per warp)
- threads in a warp operate in lock step (so an `if` statement causes threads to pause to wait for other threads that follow other branches)
- GPU can switch between warps very quickly
- threads should be cache-aware (best to have adjacent threads use adjacent memory)

Blocks:

- 1, 2, or 3-dimensional block of threads
  - 2 and 3 dimensions are just for code clarity, not real
- threads in a block have access to some fast shared memory on chip (for A100 this is 49 KB)
- A100 GPU has up to 1024 threads per block

Grid

- 1, 2, or 3-dimensional grid of blocks (again for code clarity only)
- A100 GPU has up to 2147483647 blocks in the grid (I think)

GPU execution and efficiency

- GPUS are good at executing the same computation across many data elements at once (called "SIMD" for "single instruction, multiple data".
- Threads in a block have access to some shared memory, which is on the chip, so access is very fast.
- Threads operate in a warp of 32 threads.
- Threads in a warp operate in lock-step, so `if` statements can be inefficient (some threads have to wait while a branch of the `if` statement is run for other threads).
- GPUs can switch between warps very quickly, which would happen while a warp waits for data.
- Threads in a warp should use neighboring values in an array for efficiency (note our discussion of the cache in the CPU context).
- Computations on a GPU generally use single precision (32 bit floats) (or even 16 or 8 byte floats), which will often run much faster than with doubles (64 bit floats).

GPU monitoring - details

Using `nvidia-smi` for hardware and activity

Here's some example output from a machine with multiple GPUs, but only one of which is available to us:

```
nvidia-smi
```

```
Fri Jan  3 14:11:23 2025
+-----------------------------------------------------------------------------------------+
| NVIDIA-SMI 565.57.01              Driver Version: 565.57.01      CUDA Version: 12.7     |
|-----------------------------------------+------------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute M. |
|                                         |                        |               MIG M. |
|=========================================+========================+======================|
|   0  NVIDIA GeForce RTX 2080 Ti     On  |   00000000:1A:00.0 Off |                  N/A |
| 22%   23C    P8              2W /  100W |       1MiB /  11264MiB |      0%      Default |
|                                         |                        |                  N/A |
+-----------------------------------------+------------------------+----------------------+

+-----------------------------------------------------------------------------------------+
| Processes:                                                                              |
|  GPU   GI   CI        PID   Type   Process name                              GPU Memory |
|        ID   ID                                                               Usage      |
|=========================================================================================|
|  No running processes found                                                             |
+-----------------------------------------------------------------------------------------+
```

Note that it shows that the GPU is an NVIDIA GeForce RTX 2080 Ti that has 11264MiB of memory (i.e., ~11 GB), and that at the moment, no process is using the GPU.

Next, if I logon as an administrator, I can see all 10 GPUs on one of the SCF machines, and that 7 of

the GPUs are in use:

```
Fri Jan  3 14:16:11 2025
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 535.104.05             Driver Version: 535.104.05   CUDA Version: 12.2     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  NVIDIA A100 80GB PCIe          On  | 00000000:4F:00.0 Off |                    0 |
| N/A   38C    P0              82W / 150W |   9887MiB / 81920MiB |      0%      Default |
|                                         |                      |             Disabled |
+-----------------------------------------+----------------------+----------------------+
|   1  NVIDIA A100 80GB PCIe          On  | 00000000:52:00.0 Off |                    0 |
| N/A   40C    P0              80W / 150W |  11833MiB / 81920MiB |      0%      Default |
|                                         |                      |             Disabled |
+-----------------------------------------+----------------------+----------------------+
|   2  NVIDIA A100 80GB PCIe          On  | 00000000:53:00.0 Off |                    0 |
| N/A   30C    P0              71W / 150W |  10039MiB / 81920MiB |      0%      Default |
|                                         |                      |             Disabled |
+-----------------------------------------+----------------------+----------------------+
|   3  NVIDIA A100 80GB PCIe          On  | 00000000:56:00.0 Off |                    0 |
| N/A   39C    P0              82W / 150W |  10027MiB / 81920MiB |      0%      Default |
|                                         |                      |             Disabled |
+-----------------------------------------+----------------------+----------------------+
|   4  NVIDIA A100 80GB PCIe          On  | 00000000:57:00.0 Off |                    0 |
| N/A   32C    P0              79W / 150W |   9899MiB / 81920MiB |      0%      Default |
|                                         |                      |             Disabled |
+-----------------------------------------+----------------------+----------------------+
|   5  NVIDIA A100 80GB PCIe          On  | 00000000:CE:00.0 Off |                    0 |
| N/A   58C    P0             154W / 150W |  53949MiB / 81920MiB |     99%      Default |
|                                         |                      |             Disabled |
+-----------------------------------------+----------------------+----------------------+
|   6  NVIDIA A100 80GB PCIe          On  | 00000000:D1:00.0 Off |                    0 |
| N/A   42C    P0              84W / 150W |  49121MiB / 81920MiB |      0%      Default |
|                                         |                      |             Disabled |
+-----------------------------------------+----------------------+----------------------+
|   7  NVIDIA A100 80GB PCIe          On  | 00000000:D2:00.0 Off |                    0 |
| N/A   24C    P0              41W / 150W |      4MiB / 81920MiB |      0%      Default |
|                                         |                      |             Disabled |
+-----------------------------------------+----------------------+----------------------+
|   8  NVIDIA A100 80GB PCIe          On  | 00000000:D5:00.0 Off |                    0 |
| N/A   35C    P0              79W / 150W |      7MiB / 81920MiB |      0%      Default |
|                                         |                      |             Disabled |
+-----------------------------------------+----------------------+----------------------+
|   9  NVIDIA A100 80GB PCIe          On  | 00000000:D6:00.0 Off |                    0 |
```

```
| N/A   42C    P0              86W / 150W |      7MiB / 81920MiB |      0%      Default |
|                                         |                       |             Disabled |
+-----------------------------------------+-----------------------+----------------------+

+---------------------------------------------------------------------------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|=======================================================================================|
|    0   N/A  N/A     37514      C   .../users/omer_ronen/mutemb/bin/python     9874MiB |
|    1   N/A  N/A     37509      C   .../users/omer_ronen/mutemb/bin/python    11820MiB |
|    2   N/A  N/A     37506      C   .../users/omer_ronen/mutemb/bin/python    10026MiB |
|    3   N/A  N/A     37522      C   .../users/omer_ronen/mutemb/bin/python    10014MiB |
|    4   N/A  N/A     37545      C   .../users/omer_ronen/mutemb/bin/python     9886MiB |
|    5   N/A  N/A    142017      C   python                                    53936MiB |
|    6   N/A  N/A    107884      C   ...2023/conda/envs/NODE/bin/python3.12    49108MiB |
+---------------------------------------------------------------------------------------+
```

`deviceQuery` for hardware information

The `deviceQuery` utility provided with CUDA will give details on the GPU hardware.

Here's an example with one of the SCF GPU servers, showing information about the processors available on the A100 GPU and the number of threads and blocks that are possible.

```
paciorek@saruman:~> deviceQuery
deviceQuery Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 10 CUDA Capable device(s)

Device 0: "NVIDIA A100 80GB PCIe"
  CUDA Driver Version / Runtime Version          12.2 / 12.2
  CUDA Capability Major/Minor version number:    8.0
  Total amount of global memory:                 81051 MBytes (84987740160 bytes)
  (108) Multiprocessors, (064) CUDA Cores/MP:    6912 CUDA Cores
  GPU Max Clock rate:                            1410 MHz (1.41 GHz)
  Memory Clock rate:                             1512 Mhz
  Memory Bus Width:                              5120-bit
  L2 Cache Size:                                 41943040 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 1
  Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total shared memory per multiprocessor:        167936 bytes
```

```
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 3 copy engine(s)
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Enabled
  Device supports Unified Addressing (UVA):      Yes
  Device supports Managed Memory:                Yes
  Device supports Compute Preemption:            Yes
  Supports Cooperative Kernel Launch:            Yes
  Supports MultiDevice Co-op Kernel Launch:      Yes
  Device PCI Domain ID / Bus ID / location ID:   0 / 79 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >
```

We can create the `deviceQuery` executable like this (this would need to be modified for other systems/machines):

```
module load cuda   # This gives access to the `nvcc` compiler.
git clone https://github.com/NVIDIA/cuda-samples
nvcc Samples/1_Utilities/deviceQuery/deviceQuery.cpp -I/usr/local/cuda-12.2/include \
   -ICommon -o ~/deviceQuery
```

Monitoring GPU usage

`gpustat` is useful for monitoring details of GPU usage.

We can install is using the Python `pip` installer.

```
pip install --user gpustat
```

```
~/.local/bin/gpustat
```

```
saruman                  Fri Jan  3 14:33:36 2025  535.104.05
[0] NVIDIA A100 80GB PCIe | 42°C,   0 % |  9887 / 81920 MB | omer_ronen(9874M)
[1] NVIDIA A100 80GB PCIe | 39°C,   0 % | 11833 / 81920 MB | omer_ronen(11820M)
[2] NVIDIA A100 80GB PCIe | 29°C,   0 % | 10039 / 81920 MB | omer_ronen(10026M)
[3] NVIDIA A100 80GB PCIe | 38°C,   0 % | 10027 / 81920 MB | omer_ronen(10014M)
[4] NVIDIA A100 80GB PCIe | 32°C,   0 % |  9899 / 81920 MB | omer_ronen(9886M)
[5] NVIDIA A100 80GB PCIe | 60°C, 100 % | 68155 / 81920 MB | arjunpatrawala(68142M)
[6] NVIDIA A100 80GB PCIe | 47°C,  14 % | 36377 / 81920 MB | zhangyunzhe2023(36364M)
```

```
[7] NVIDIA A100 80GB PCIe | 24°C,   0 % |     4 / 81920 MB |
[8] NVIDIA A100 80GB PCIe | 34°C,   0 % |     7 / 81920 MB |
[9] NVIDIA A100 80GB PCIe | 42°C,   0 % |     7 / 81920 MB |
```

There are various flags. I'm not familiar with the details but this invocation shows more information about the processes running on each GPU:

```
~/.local/bin/gpustat -fup
```

```
saruman                  Fri Jan  3 14:35:20 2025  535.104.05
[0] NVIDIA A100 80GB PCIe | 47°C,   0 % |  9887 / 81920 MB | omer_ronen/37514(9874M)
    37514 (  90%,   14GB): /scratch/users/omer_ronen/mutemb/bin/python -m mutemb.utils.data --chrom 1
[1] NVIDIA A100 80GB PCIe | 41°C,   0 % | 11833 / 81920 MB | omer_ronen/37509(11820M)
    37509 ( 100%,   10GB): /scratch/users/omer_ronen/mutemb/bin/python -m mutemb.utils.data --chrom 2
[2] NVIDIA A100 80GB PCIe | 29°C,   0 % | 10039 / 81920 MB | omer_ronen/37506(10026M)
    37506 (  10%,   11GB): /scratch/users/omer_ronen/mutemb/bin/python -m mutemb.utils.data --chrom 3
[3] NVIDIA A100 80GB PCIe | 38°C,   0 % | 10027 / 81920 MB | omer_ronen/37522(10014M)
    37522 ( 120%,   37GB): /scratch/users/omer_ronen/mutemb/bin/python -m mutemb.utils.data --chrom 4
[4] NVIDIA A100 80GB PCIe | 32°C,   0 % |  9899 / 81920 MB | omer_ronen/37545(9886M)
    37545 ( 100%,   26GB): /scratch/users/omer_ronen/mutemb/bin/python -m mutemb.utils.data --chrom 5
[5] NVIDIA A100 80GB PCIe | 60°C, 100 % | 68155 / 81920 MB | arjunpatrawala/142017(68142M)
   142017 ( 100%, 4010MB): python generate_jacobians.py
[6] NVIDIA A100 80GB PCIe | 44°C,   0 % | 27707 / 81920 MB | zhangyunzhe2023/228882(27724M)
   228882 (  50%, 1144MB): /scratch/users/zhangyunzhe2023/conda/envs/NODE/bin/python3.12 -Xfrozen_mod
[7] NVIDIA A100 80GB PCIe | 24°C,   0 % |     4 / 81920 MB |
[8] NVIDIA A100 80GB PCIe | 35°C,   0 % |     7 / 81920 MB |
[9] NVIDIA A100 80GB PCIe | 42°C,   0 % |     7 / 81920 MB |
```

GPU kernels

A *kernel* is a function that can be run on the GPU.

Next we'll see that we can actually write a kernel in Julia and then call that kernel. Kernels are functions that encode the core computational operations that are executed in parallel.

In other languages, the basic mode of operation with a GPU when you are writing your own GPU code is to write a kernel using CUDA (basically C) code and then call the kernel in parallel via C, R, or Python code. In Julia, we can write the kernel using Julia syntax (though many operations (particularly non-numerical ones) will not run on the GPU…).

I have not investigated much to see what Julia does behind the scenes with the kernel code for it to be able to run on the GPU, but this blog post seems helpful. I *think* the steps involve going from LLVM code to PTX code (an assembly language for GPUs) that is then converted to GPU machine code. These steps involve tooling available outside of Julia, which seems like a reflection of the benefit of choosing to have Julia's compilation system use LLVM.

Basic example

Here's a basic example in which we'll do a calculation in place. We run 1000 scalar calculations using 1000 threads.

We use `@cuda` to compile and run the kernel.

```
function my_kernel(x)
  idx = threadIdx().x;   # What thread am I?
  if idx < length(x)
    x[idx] = tan(x[idx]) + 3*sin(x[idx]);
  end
  return
end

n = 1000;
x_gpu = CUDA.randn(n);
Array(x_gpu)[n]
# -1.5321726f0
@cuda threads=n my_kernel(x_gpu);
Array(x_gpu)[n]   # Check the computation was done by checking last element.
# -28.875708f0
```

There are limits on the number of threads we can use.

```
n = 2000;
x_gpu = CUDA.randn(n);
@cuda threads=n my_kernel(x_gpu);
# ERROR: Number of threads in x-dimension exceeds device limit (2000 > 1024).
```

Multiple blocks

We need to use at least as many threads as computations, and in addition to only being able to use 1024 threads in the x dimension, we can have at most 1024 threads in a block on the A100 GPU we're using. So we'll need multiple blocks.

```
function my_kernel(x)
  i = threadIdx().x;  # What thread am I within the block?
  j = blockIdx().x;   # What block am I in?
  idx =  (j-1)*blockDim().x + i;
  if idx <= length(x)
    x[idx] = tan(x[idx]) + 3*sin(x[idx]);
  end
  return
end

n = 2000;
nthreads = 1024;
x_gpu = CUDA.randn(n);
```

```
initial = Array(x_gpu)[n]
nblocks = Int(ceil(n/nthreads));

@cuda threads=nthreads blocks=nblocks my_kernel(x_gpu);
(initial, Array(x_gpu)[n])  # Check that calculation was done.
```

Let's do a smaller test run in which we can check on the thread and block indexing.

```
function my_kernel_print(x)
  i = threadIdx().x;  # What thread am I within the block?
  j = blockIdx().x;   # What block am I in?
  idx =  (j-1)*blockDim().x + i;
  if idx <= length(x)
    x[idx] = tan(x[idx]) + 3*sin(x[idx]);
    @cuprintln idx, i, j, blockDim().x, blockDim().y;
  end
  return
end

n = 200;
x_gpu = CUDA.randn(n);
nthreads = 100;
nblocks = Int(ceil(n/nthreads));
@cuda threads=nthreads blocks=nblocks my_kernel(x_gpu);
```

When we run this, notice the output seems to be grouped based on warps of 32 threads (apart from the last set since `n=200` is not a multiple of 32).

Larger computations

In many cases we'll have more tasks than the total number of GPU cores. As long as we don't exceed the maximum size of a block or grid, we can just ask for as many threads as we have tasks and rely on the GPU to manage assigning the tasks to the GPU cores.

We'd want to check that the number/dimension of the block here does not exceed the maximum block size. I didn't do that, but it ran, so it must have been ok!

Here we'll run the computation we ran earlier when we did not write our own kernel and just relied on Julia to offload to the GPU behind the scene.

```
n = 250000000;
x_gpu = CUDA.randn(n);
nthreads = 1024;
nblocks = Int(ceil(n/nthreads));
Array(x_gpu)[n]

# Run it once to flush out any compilation/transformation time.
y_gpu = CUDA.randn(5);
```

```
CUDA.@sync @cuda threads=nthreads blocks=nblocks my_kernel(y_gpu);

CUDA.@time CUDA.@sync @cuda threads=nthreads blocks=nblocks my_kernel(x_gpu);
# 0.002003 seconds (45 CPU allocations: 2.719 KiB)
Array(x_gpu)[n]
```

The 2.0 ms is reasonably comparable to the 3.7 ms when we just had Julia run the vectorized computation on the GPU (from the last time we ran it). That used 64-bit floats. When I reran the code above using 64-bit floats, the time was 5.2 ms.

Final comments (for now)

When to use the GPU

To effectively use the GPU, one generally wants to have a computation in which the same calculation is being done on many data elements. Things like the vectorized examples and matrix multiplication seen above, pixel-wise processing of images/video, simulations with many independent samples, etc.

Situations with a lot of conditionality (if-else branching), small data sizes, and lots of data transfer to/from the GPU are generally not good use cases for the GPU.