GPU_acceleration

August 10, 2021

1 GPU

Julia provides excellent GPU support.

GPUs are devices which can run thousands of threads simultanously in parallel.

GPU threads are slower and memory limited than the CPU threads.

However, there are so many of GPU threads.

Many tasks can be executed much faster on a GPU than on a CPU, if these tasks can be parallelized.

1.1 Check installed GPU device:

[ ]:
```
;nvidia-smi
```

1.2 Banchmark CPU performace

Let us create a large matrix and time how long it takes to square it on the CPU:

[ ]:
```
import BenchmarkTools

M = rand(2^11, 2^11)

function benchmark_matmul_cpu(M)
    M * M
    return nothing
end

benchmark_matmul_cpu(M) # warm up

@BenchmarkTools.btime benchmark_matmul_cpu($M)
@BenchmarkTools.benchmark benchmark_matmul_cpu($M)
```

For benchmarking, the math operations are in a function which returns `nothing`.

We need a "warm up" line. Since Julia compiles code on the fly the first time it is executed, the operation we want to benchmark needs to be exectured at least once before starting the benchmark Otherwise, the benchmark will include the compilation time.

Note that `$M` is used instead of `M` for banchmarking. This is a feature of the `@BenchmarkTools.btime` macro. It allows evaluation of `M` before benchmarking takes place, to avoid the extra delay that is incurred when benchmarking with global variables.

1.3 Banchmark GPU performace

GPU operations using CUDA:

[ ]:
```
import CUDA

M_on_gpu = CUDA.cu(M) # Copy the data to the GPU and create a CuArray

M_on_gpu = CUDA.CURAND.rand(size(M)) # or create a new random matrix directly␣
↪on the GPU

function benchmark_matmul_gpu(M)
    CUDA.@sync M * M
    return nothing
end

benchmark_matmul_gpu(M_on_gpu) # warm up

@BenchmarkTools.btime benchmark_matmul_gpu($M_on_gpu)
@BenchmarkTools.benchmark benchmark_matmul_gpu($M_on_gpu)
```

Important: * Before the GPU can work on some data, it needs to be copied to the GPU (or generated there directly). * The `CUDA.@sync` macro waits for the GPU operation to complete. Without it, the operation would happen in parallel on the GPU, while execution would continue on the CPU. So we would just be timing how long it takes to *start* the operation, not how long it takes to complete. * In general, you don't need `CUDA.@sync`, since many operations (including `cu()`) call it implicitly. And it is usually a good idea to let the CPU and GPU work in parallel. Typically, the GPU will be working on the current batch of data while the CPU works on preparing the next batch.

1.4 GPU memory status

Let's check how much RAM we have left on the GPU:

[ ]:
```
CUDA.memory_status()
```

Julia's Garbage Collector (GC) will free CUDA arrays like any other object, when there's no more reference to it.

However, `CUDA.jl` uses a memory pool to make allocations faster on the GPU, so don't be surprised if the allocated memory on the GPU does not go down immediately.

Moreover, IJulia keeps a reference to the output of each cell, so if you let any cell output a `CuArray` it will be staying in the memory.

To force the Garbage Collector to run, execute `GC.gc()`. To reclaim memory from the memory pool, use `CUDA.reclaim()`:

[ ]:
```
GC.gc()
CUDA.reclaim()
```

1.5 GPU loop fussion

Many other operations are implemented for `CuArray` (`+`, `-`, etc.).

Their dotted versions are implemented as well (`.+`, `exp.()`, etc).

Importantly, the Julia loop fusion also works on the GPU.

For example, if we want to compute `M .* M .+ M`, without loop fusion the GPU would first compute `M .* M` and create a temporary array, then it would add `M` to that array, like this:

[ ]:
```
function benchmark_without_fusion(M)
    P = M .* M
    CUDA.@sync P .+ M
    return
end

benchmark_without_fusion(M_on_gpu) # warm up

@BenchmarkTools.btime benchmark_without_fusion($M_on_gpu)
```

Instead, the loop fusion in Julia ensures that the array is only traversed once, without the need for a temporary array:

[ ]:
```
function benchmark_with_fusion(M)
    CUDA.@sync M .* M .+ M
    return
end

benchmark_with_fusion(M_on_gpu) # warm up

@BenchmarkTools.btime benchmark_with_fusion($M_on_gpu)
```

1.6 GPU coding

Julia allows you to write your own GPU operations!

Rather than using GPU operations implemented in the `CUDA.jl` package (or others), you can write Julia code that will be compiled for the GPU, and executed there.

This can be useful to speed up some algorithms where the standard kernels do not suffice.

For example, here's a GPU kernel which implements `u .+= v`, where `u` and `v` are two (large) vectors:

[ ]:
```
function worker_gpu_add!(u, v)
    index = (CUDA.blockIdx().x - 1) * CUDA.blockDim().x + CUDA.threadIdx().x
    index  length(u) && (@inbounds u[index] += v[index])
    return
```

```
end

function gpu_add!(u, v)
    numblocks = ceil(Int, length(u) / 256)
    CUDA.@cuda threads=256 blocks=numblocks worker_gpu_add!(u, v)
    return u
end
```

Important:

- The `gpu_add!()` function first calculates `numblocks`, the number of blocks of threads to start, then it uses the `CUDA.@cuda` macro to spawn `numblocks` blocks of 256 GPU threads and each thread executeds `worker_gpu_add!(u, v)`.
- The `worker_gpu_add!()` function computes `u[index] += v[index]` for a single value of `index`: in other words, each thread will just update a single value in the vector!
- The `CUDA.@cuda` macro spawns `numblocks` blocks of 256 threads each. These blocks are organized in a grid, which is one-dimensional by default, but it can be up to three-dimensional. Therefore each thread and each block have an `(x, y, z)` coordinate in this grid. See this diagram from the Nvidia blog post: .
- `CUDA.threadIdx().x` returns the current GPU thread's `x` coordinate within its block (one difference with the diagram is that Julia is 1-indexed).
- `CUDA.blockIdx().x` returns the current block's `x` coordinate in the grid.
- `CUDA.blockDim().x` returns the block size along the `x` axis (in this example, it's 256).
- `CUDA.gridDim().x` returns the number of blocks in the grid, along the `x` axis (in this example it's `numblocks`).
- So the `index` that each thread must update in the array is `(CUDA.blockIdx().x - 1) * CUDA.blockDim().x + CUDA.threadIdx().x`.
- As explained earlier, the `@inbounds` macro is an optimization that tells Julia that the index is guaranteed to be inbounds, so there's no need for it to check.

Hopefully, now writing your own GPU kernel in Julia will not seem like something only top experts with advanced C++ skills can do.

Let's check that the kernel works as expected:

[ ]:
```
u = rand(2^20)
v = rand(2^20)

u_on_gpu = CUDA.cu(u)
v_on_gpu = CUDA.cu(v)

u .+= v

gpu_add!(u_on_gpu, v_on_gpu)

@assert Array(u_on_gpu)  u
```

Note: the operator checks whether the operands are approximately equal within the float precision limit.

Let us benchmark the new custom kernel:

[ ]:
```
function benchmark_custom_assign_add!(u, v)
    CUDA.@sync gpu_add!(u, v)
    return nothing
end

benchmark_custom_assign_add!(u_on_gpu, v_on_gpu) # warm up

@BenchmarkTools.btime benchmark_custom_assign_add!($u_on_gpu, $v_on_gpu)
```

Let us see how this compares to `CUDA.jl`'s implementation:

[ ]:
```
function benchmark_assign_add!(u, v)
    CUDA.@sync u .+= v
    return nothing
end

benchmark_assign_add!(u_on_gpu, v_on_gpu) # warm up

@BenchmarkTools.btime benchmark_assign_add!($u_on_gpu, $v_on_gpu)
```

The new custom kernel is just as fast as `CUDA.jl`'s kernel!

Howvever, the new kernel would not work with huge vectors!

This is because there is a limit to the number of blocks & threads can be spawned.

To support huge vectors, each worker needs to run a loop like this:

[ ]:
```
function worker_gpu_add!(u, v)
    index = (CUDA.blockIdx().x - 1) * CUDA.blockDim().x + CUDA.threadIdx().x
    stride = CUDA.blockDim().x * CUDA.gridDim().x
    for i = index:stride:length(u)
        @inbounds u[i] += v[i]
    end
    return nothing
end
```

This way, if `CUDA.@cuda` is executed with a smaller number of blocks than needed to have one thread per array item, the workers will loop appropriately.

For more info, check out `CUDA.jl`'s documentation.