Lecture 4: MATLAB — advanced use cases

(very) Short introduction to GPU computing on MATLAB

Juha Kuortti and Heikki Apiola

February 22, 2018

Aalto University
juha.kuortti@aalto.fi

Why GPUs

A GPU — or *massively parallel* approach to parallelism is somewhat different from what we saw previously.

- **CPU**
  - Low latency
  - Make sequentially dependent code run as fast as possible
  - Try to avoid memory access bottleneck with large caches
- **GPU**
  - Aim for high throughput
  - Finish as many instructions per clock cycle as possible
  - Need a lot of computation on chip $\rightarrow$ cannot afford large caches

Not all tasks are suitable for a GPU.

GPUs in MATLAB

Assuming that you have properly set up CUDA, (sorry AMD), the command that transfers your variable (generally, only matrices) to GPU memory is called `gpuArray`. The command broadcasts the variable to GPU, and afterwards you can do computations on it as you would with a regular variable.

Once you're done computing, you will need to get the variables back from the GPU memory. You'll do this with command `gather`.

Some basic demos

First — if you're on a laptop, you'll almost certainly will need to make a remote connection to a machine that will have CUDA enabled.

```
a = randn(3000);
b = a*ones(3000,1);
tic; x = a\b; toc
tic;
A = gpuArray(a);
B = gpuArray(b);
x = A\B; x = gather(x);
toc
```

Slightly more advanced demo

For massively parallel computation, `arrayfun` is a natural way to work, since it can assign one processor for every element of array.

So if you recall the `juliaDemo` from last lecture, we can do this on a GPU, and get a decent speedup. There are caveats, though — gpuArray cannot construct the the anonymous funtion from variables in workspace, for example.

What GPUs are not good for

GPUs are a powerful tool for computation, but when applied outside of their scope, they become more of a liability, the overhead eating away the speedup.

The speed multiplier provided by GPU is mainly due to the *coherent data access*. If your problem does not benefit from that, (say, doing a costly operation to few elements) the GPU cons will outweigh the pros.

So you think this isn't fast enough...

Just like we can make C-programs into MATLAB functions via `mex`, we can write CUDA natively and compile it via `nvcc` (i.e. system command). If written following the MATLAB convention, it can be called from MATLAB by using `parallel.gpu.CUDAKernel`.