Astro 585: HW 7

Codename: The Maxwell-Jüttner Distribution

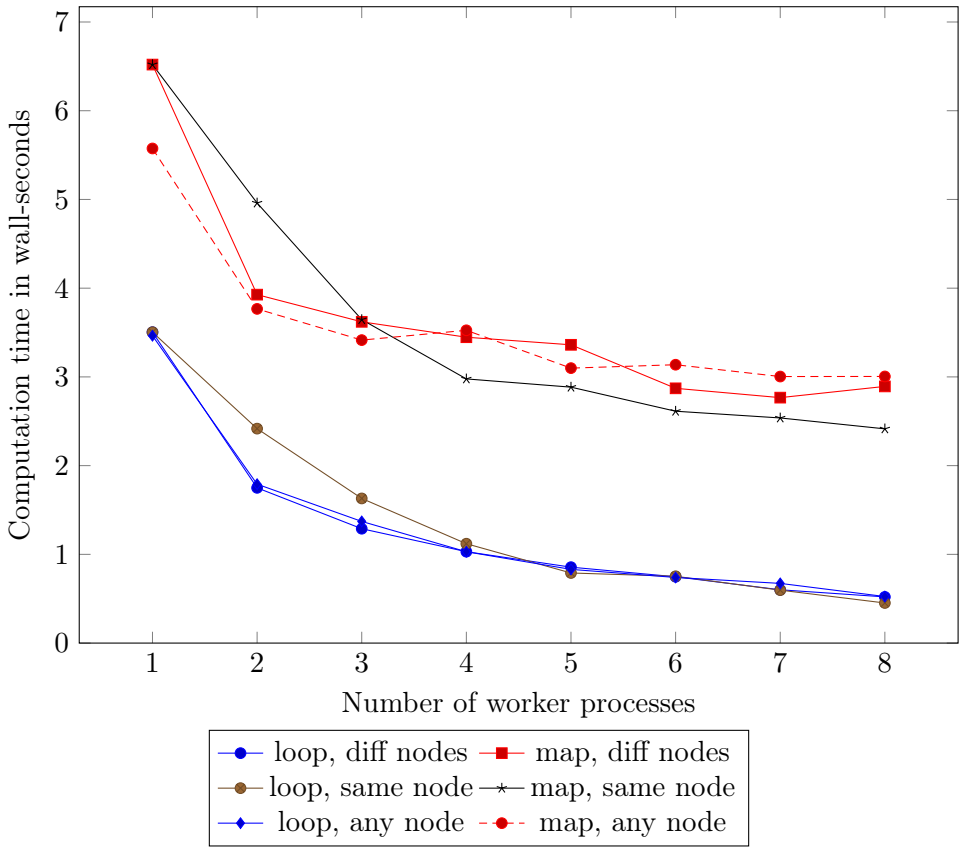
March 30, 2014

My git repository is here: `https://github.com/hsgg/astro585`, clone URL `https://github.com/hsgg/astro585.git`.

1 PBS, Clusters, LionXV

$\pi \approx 3.140\,996\,000\,000$. Hm, π...

The following, busy plot shows the computation time as function of number of worker threads, for both the loop and the map operations with 10^8 iterations. For each operation and number of workers, the code was run three times – all on one node, all on different nodes, and finally any combination that the pbs system saw fit to use.



Of course, it took longest to get started the job that needed 8 cores on a single node. Contention with others users is highest there.

Above 4 processors, the scaling is about linear, with not much to be gained from more processors.

I find a big difference between using map versus the hand-coded looping implementation, but not all that much between running on one node versus spreading it out.

For reference, these are the commands to run it:

```
$ cd hw7/
$ ./mkqsubscript.sh > q1_script.sh
$ chmod +x q1_script.sh
$ ./q1_script.sh
```

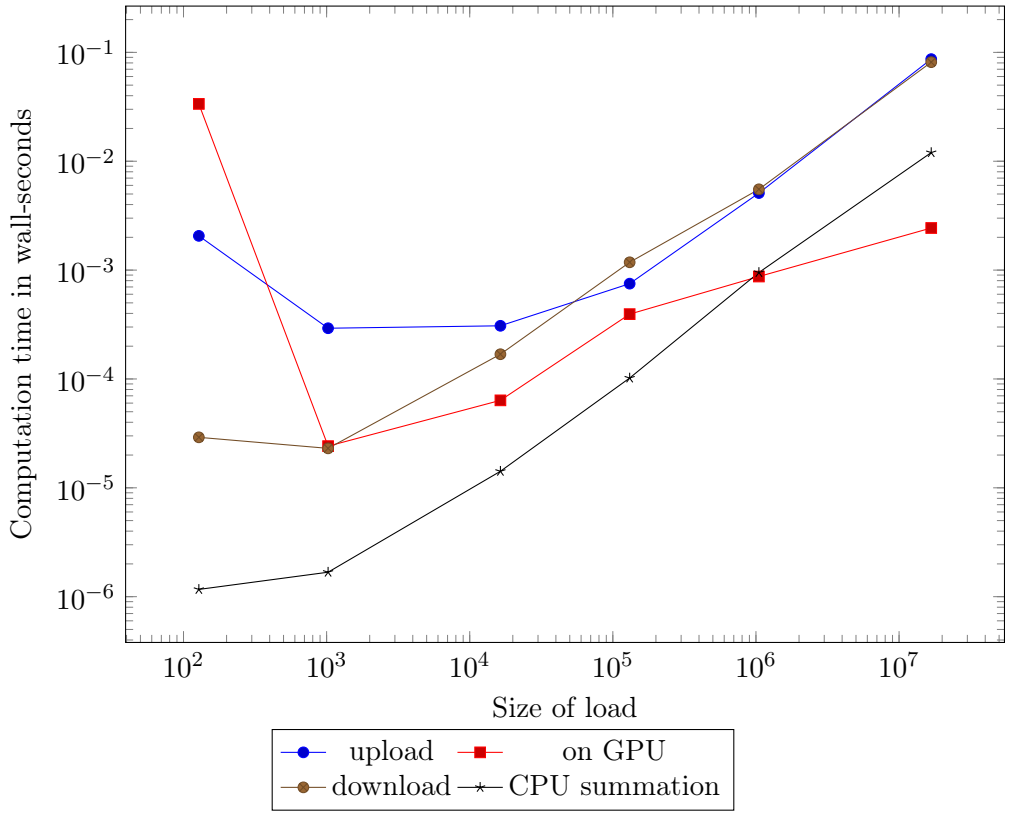
Then wait. To extract the numbers:

```
$ ./Q1_get_result.sh
```

Then recompile the LaTeX document `soln.tex` to recreate the graph.

2 CUDA

For this I used `tesla.rcc.psu.edu`.



For small data sets the GPU takes much of the time, although the data upload to the GPU also takes some time. I find it surprising that it takes longer than for larger datasets. Perhaps if I ran it multiple times, there would be a large scatter caused by interference from other processes.

For large datasets the GPU time becomes negligible, and the up- and download times become dominant, as the GPU can distribute the work efiiciently among its small cores.

Performing the summation on the GPU potentially increases the performance significantly, as the download to the CPU will essentially disappear. However, since the data still needs to be accessed for the summation, I expect the GPU to take more time doing the calculation than the CPU, so the speedup will be less. It depends on whether the CPU is the bottleneck in the download, or the GPU.

For reference, the commands to get the plots are:

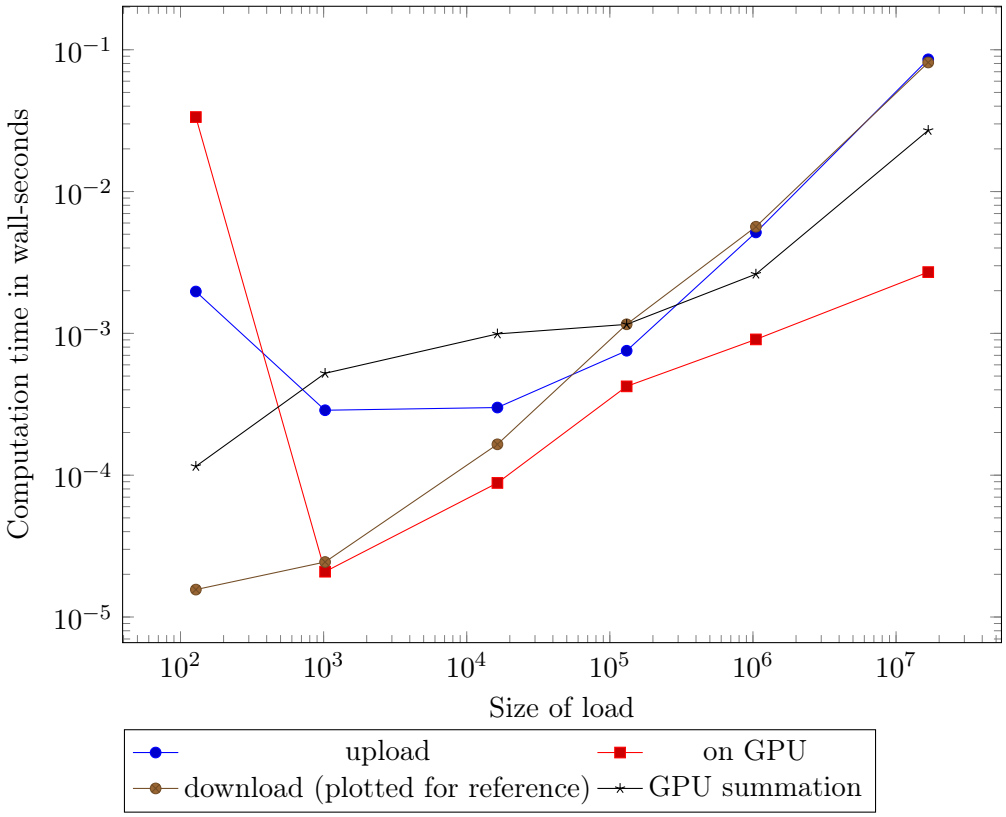
```
$ cd hw7
$ ./HW7_Q2.jl > q2.out  # must be run where GPU is available
$ ./Q2_parse_script.sh < q2.out > q2.dat
```

Then recompile the LaTeX document `soln.tex` to recreate the graph.

It seems to frequently and randomly fail at the synchronization points.

2.1 Summation on the GPU

Ah, a nice problem! The summations agree on the answer, something like 0.682 689 499 359 469 2, which is pretty close to what it should be.



I most likely didn't use the best algorithm, or the best choice of parameters for the algorithm, but it does seem to save a good deal of time, at least at large problem sizes, as can be seen by the fact

that the GPU summation time takes less than the download time. At small problem sizes the extra overhead makes the on-GPU summation take longer than the download, and so is anti-beneficial.

The part of the code that I inserted into `HW7_Q2_funcs.jl` is below. It successively sums up blocks of the input array `y_gpu`.

```
  println("Timing the summation being performed on the GPU")
  tic()
  percore = 16
  n_sums = n
  n_subsums = iceil(n_sums / percore)
  nonsum_gpu = y_gpu
  subsum_gpu = CuArray(Float64, (n_subsums))
  sumtmp_gpu = subsum_gpu  # keep for later freeing
  result_gpu = CuArray(Float64, (1))
  first_round = true

  while n_sums > 1
      sum_block_size = choose_block_size(n_subsums)
      sum_grid_size = choose_grid_size(n_subsums, sum_block_size)
      launch(sum_gpu_kernel, sum_grid_size, sum_block_size,
          (nonsum_gpu, subsum_gpu, n_sums, n_subsums, percore), stream=stream1)
      synchronize(stream1)  # Hm, Darve says global synchronization is unreliable...

      # setup next
      n_sums = n_subsums
      n_subsums = iceil(n_sums / percore)
      tmp = nonsum_gpu
      if first_round == true  # then nonsum_gpu = y_gpu
          first_round = false
          # Don't overwrite y_gpu, but we can recycle x_gpu
          tmp = x_gpu
          println("y_gpu no longer in danger")
      end
      nonsum_gpu = subsum_gpu
      subsum_gpu = tmp
  end
```

With CUDA kernels in `normal_pdf_gpu.cu`:

```
extern "C"   // ensure function name to be left alone
{

    __global__ void normal_pdf_gpu(const double *x, double *y, unsigned int n)
    {
// assumes a 2-d grid of 1-d blocks
unsigned int i = (blockIdx.y * gridDim.x + blockIdx.x) * blockDim.x + threadIdx.x;
```

```
        if(i<n)  y[i] = exp(-0.5*x[i]*x[i])*rsqrt(2.0*M_PI);
    }

    __global__ void sum_gpu(double *y, double *sumptr, unsigned int n,
            unsigned int n_subsums, unsigned int percore)
    {
        // assumes a 2-d grid of 1-d blocks
        unsigned int i = (blockIdx.y * gridDim.x + blockIdx.x) * blockDim.x
            + threadIdx.x;
        unsigned int j = i * percore; // first element that this thread will take care of
        unsigned int k;

        if (i >= n_subsums)
            return;

        sumptr[i] = 0.0;
        for (k = 0; k < percore; k++) {
            if (j + k < n)
                sumptr[i] += y[j + k];
        }
    }

    __global__ void get_sum_gpu(double *y, double *sumptr)
        // copy the result into a smaller array
    {
        sumptr[0] = y[0];
    }
}


/* vim: set sw=4 sts=4 et : */
```

3 The rest

Hm...somehow I cannot convince myself to do these...It seems like figuring out how to use this cluster and that cluster is useful, I suppose, but not technically or scientifically interesting. A conundrum.