HPCE CW1 - Making Matlab fast(er)

Issued: 2014/01/21

Due: 2014/01/04, 23:59

1 Goals

The goal of this coursework is to introduce you to a few key concepts and practical skills in a "safe" environment:

- Scalar versus Vector performance.
- Anonymous functions and higher-order programming.
- Very basic parallel programming over iteration spaces.

Some of these terms are taken from computer science, and have deep and wonderful underlying formalisms, but they are actually quite simple and understandable on a practical level. If you've done anything serious with matlab you've probably used most of them before, and the point here is to explicitly identify and classify them before we use them in less safe environments.

2 Environment and Tools

This work uses plain Matlab, so all department computers should have it installed, or you can work at home if you have it installed.

Parts of the exercises rely on having multiple processors available, so at some points you will need at least four logical processors to properly explore behaviour and performance. [1] There are plenty of machines with this many processor cores in college on level 5. Pretty much any recent machine will have two cores, which is enough to do the exploratory and development work, but for the final performance runs you need to try it on a machine with more cores.

3 Deliverables

This coursework is split up into a number of stages, each of which will rely on you writing one or more matlab functions or scripts. The names of the files (and

[1]For now we will skip over details like hyper-threading.

functions) will be given to you, but you are free to create extra helper functions or scripts if you wish. Turning around the feedback fast relies on automated tests, so double-check that your file-names are right before submitting.

As well as the code you will also need to include specific outputs generated by your code, which should be placed in the specified directory.

You do not need to include a written report, or any other documentation – your code should be appropriately commented using matlab documentation and comment conventions (so don't comment every statement for the sake of it), and if you really must communicate something beyond code for marking, you can include a "readme.txt".

4 Exercises

You are going to be speeding up fractal rendering using a number of techniques, all applied within matlab. Download and unzip the coursework spec, start matlab, then set the matlab current directory to wherever you unzipped it. You'll see a number of directories called "init", "q1", and so on, most of which are empty. You'll be populating these directories with your code. You'll also see a file called "setup.m", which sets up the search path so that matlab will look in the different directories for functions. Change to that directory and run the setup script. [2]

Important: When running your code, make sure that your working directory is the one containing "setup.m".

Now run "animate_julia" (with no arguments needed). With the search path setup, it should pick-up the function from the "init" directory automatically.

Switch to the figure window, stare at it for a bit, then try moving the mouse over the window. It is rendering a version of the Julia set, which is based on the iteration:

$$z_{i+1} = z_i^2 + c \tag{1}$$

Where the starting z_1 is the complex coordinate for a given pixel, and c is a complex constant for the entire frame. The pixel is coloured according to how far into the sequence we have to go before $|z_i| \geq 1$. For some pixels the sequence never escapes the unit circle, so we stop at some arbitrary `maxiter`. The code for this is in "julia.m". A second procedure "render_julia.m" is responsible for calculating the values of z for each pixel, but defers to `julia` to to the actual calculation.

You can use ctrl-c in matlab to quit out of the animation.

The animation procedure actually has three parameters: a width and a height to control the resolution of the image, and a maximum iteration count. If you specify higher values for those parameters, you'll see that the frame-rate starts to go down. Our goal is a fairly typical one: we want to increase the scale

[2]When you're done with the coursework, you may like to run "unsetup", to undo the search path.

of problems that are feasible to tackle (image resolution) and the quality of the solution (maximum iteration), but without decreasing performance (frame-rate). Indeed, we'd like to make everything better, so higher resolution, higher quality, and faster frame-rate.

4.1 Q1 - Testing and measurement

When starting any code optimisation task, you should always start by making sure you have:

- A well defined quantitative metric which defines performance, and an automated and repeatable method for applying that metric.
- An automated way to test that any optimised function still operates correctly.

We want both parts to be automated so that they are easy to run and do not rely on human judgement. Programmers will have pre-conceived ideas about whether an optimisation will work or not, and it is not uncommon to be so convinced a code change has made something faster, that they ignore that it has caused a performance regression (i.e. it is slower than before). Hard numbers make it more difficult to self-delude.

For this task, create two functions in the "q1" directory:

```
function [t,n,w,h,c,maxiter] = time_renderer(renderer,w,h,c,maxiter);
% time_renderer measures execution time for varying input parameters
%   renderer : handle to a function with signature ...
    renderer(w,h,maxiter)
%   w,h,c,maxiter : Points at which to measure execution time,
%                   Each can be a vector or a scalar, but if there ...
    are any
%                   vectors, then all vectors must have the same ...
    length.
%                   If no args are specified, the defaults should be
%                   w=round(2.^(4:0.5:10)); h=w; c=sqrt(2); maxiter=64;
%   Returns:
%     t : execution time per frame
%     n : Total pixels for each frame
%     w,h,c,maxiter : Parameters corresponding to each frame.

function [] = test_renderer(renderer);
% test_renderer checks that a renderer is functionally correct,
%   by checking it against the reference render_julia function
%   for various (w,h,c,maxiter) test-points.
%   The set of test-points chosen is up to the implementation,
%   and should be chosen to balance likelihood of error-detection
%   against execution time.
%   The function should fail using assert if any test fails, or
%   run to completion otherwise.
```

Both these functions are higher-order functions, meaning they take functions as arguments. For example, we can do:

```
rr=@render_julia
```

The '@' operator creates a handle to the function `render_julia` and assigns it the variable `rr`. This handle behaves like the function it has been assigned, so we can then do:

```
rr(16,16);
```

which is equivalent to calling `render_julia` directly. We can then pass this function handle into `test_renderer`:

```
test_renderer(rr);
```

The parameter specification for `time_renderer` is relatively complex in terms of mixing vectors and scalars. The reason for this is that we want to be able to easily explore the effects of different parameters on timing. So for example, you can do:

```
[t,n]=time_renderer(@render_julia_v1);
plot(n,t);
```

to get an idea of how execution time varies with pixels calculated.

However, you can also do:

```
c=0:0.05:1;
[t,n]=time_renderer(@render_julia_v1, 128,128, c);
plot(c,t);
```

to explore how execution time varies with different c.

4.2 Q2 - Basic vectorisation

As it stands, the code is completely scalar, and won't work with vectors. Take the three files from the init directory, and copy them into q2. Rename the files (and the functions in them) to "animate_julia_v1.m", "render_julia_v1.m", and "julia_v1.m".

You now need to modify two parts:

- Modify `julia_v1` so that it can take a vector parameter vz. Note that you don't have to make it fast at this point, you are just giving it the capability to take vectors as inputs
- In `render_julia_v1` modify the loop nest so that it is vectorised over y (so remove the loop over y).

Modify the files as needed to ensure they all call each other, rather than the old version, then test and check performance using the two functions from the previous question.

To check that you are getting the right functions being called, you could try putting a break-point in `julia_v1`, then call `test_renderer(@render_julia_v1)`.

4.3 Q3 - Adding genericity

We're going to develop a few different versions of functions, so now we'll add some flexibility to the system so we don't have to keep copying all the files. Copy the "animate_julia_v1.m" and "render_julia_v1.m" files from the previous exercise into q3, and rename the files (and functions) to `animate_julia_v2` and `render_julia_v2`.

You should now change the two functions such that they can be parameterised with another function:

```
function [pixels]=render_julia_v2(juliaproc, w,h,c, maxiter)
%  render_julia_v2 : Renders julia set using a specified iteration ...
    procedure
%     juliaproc : A function of the form [viter,vz] = juliaproc(vz, ...
    c, maxiter)
%                   where vz can be a vector.
%     w,h,c,maxiter : Standard rendering controls

function [] = animate_julia_v2(renderer, w, h, maxiter )
% animate_julia_v2 : Animates julia set using the given rendering ...
    procedure
%     renderer : A function of the form ...
    [pixels]=renderer(w,h,c,maxiter).
%     w,h,maxiter : Standard rendering controls
```

Note that `julia_v1` is of the right form to pass to `render_julia_v2`, so you should be able to do:

```
image(render_julia_v2(@julia_v1,64,64));
```

you can also pass `render_julia_v1` to `animate_julia_v2`:

```
animate_julia_v2(@render_julia_v1)
```

However, the real payoff comes when we can do:

```
f = @(w,h,c,maxiter)( render_julia_v2(@julia_v1, w,h,c,maxiter));
animate_julia_v2(f,64,64)
```

This gives us the ability to swap renderers and iteration procedures, so we can tune them independently.

To save typing, you may find it useful to create some helper functions. For example, if you create the function:

```
function [ f ] = rjv2( jp )
```

```
f=@(w,h,c,maxiter)( render_julia_v2(jp, w,h,c,maxiter) );
end
```

then you could simply do:

```
f = rjv2(@julia_v1);
animate_julia_v2(f ,64,64)
```

or more, concisely:

```
animate_julia_v2(rjv2(@julia_v1),64,64);
```

Make sure to test your composite renderer with your test function. Also briefly explore the performance of the renderer – is there any overhead compared to `render_julia_v1`, particularly for small frame sizes?

4.4 Q4 - Initial vectorisation

Copy "julia_v1.m" into q4, and rename the file and function to "julia_v2.m". The function signature (i.e. the input parameters and return types) will stay the same, but now modify it to have a vectorised inner loop using masks.

The broad structure of the code will be a loop counting up to at most `maxiter`, with a body along the lines of:

1. Use the exit condition to form a mask for points which are still active.
2. If no points are active, quit the loop.
3. Perform the julia update step for all points.
4. Update just the active points (using the mask).

This version may or may not be faster depending on multiple factors, some of which will be machine dependent, but some are parameter dependent.

If you think about the graphical output, it is effectively a plot of the number of iterations taken for each point. You can explore this more effectively by doing:

```
surf(render_julia_v1(128,128,0.5,64));
```

which will show the executions as a 3D height map. If you look at the spikes, they represent points which took a very large number of iterations, while the surrounding pixels escaped very quickly. However, in the vectorisation from the previous question, we were stepping the entire vector of points forward until *all* points had finished. Until those spikes are processed, every iteration of the loop will have to deal with the full-width vector, even though most points in the vector are inactive after just a few iterations.

4.5 Q5 - Advanced vectorisation

Copy “julia_v2.m” into q5, and rename the file and function to “julia_v3.m”, keeping the same function signature. Modify the function so that it is still vectorised, but on each iteration it only operates over active points, rather than operating over all points and using masks to only update the active ones.

The code will still have a loop counting up to at most `maxiter`, but now you will be operating on a vector that changes in size, containing only the active points. So initially all points will be active, then your loop body will:

1. Check that the set of active points is not empty.
2. Apply the julia update function.
3. Detect any points which have finished.
4. Remove them from the active set.

As usual, you should see what the timing effects of this change are – is it better, worse, under what circumstances, and is the behaviour explainable?

Note that a big change here is that we have moved from a statically sized iteration space in `julia_v2`, where the size of the inner vector stays constant, to a dynamically sized iteration space. This has implications for hardware efficiency, particularly once we move into a GPU context: it is often much cheaper to have code that deals with fixed-size vectors, rather than taking the overhead of resizing the array.

4.6 Q6 - Code generation

The final optimisation of the julia proc we’re going to consider is compiling the function to native code, rather than using the interpreted matlab code. We’ll actually move back to scalar (non vectorised code) – if we *really* cared about speed we’d want to vectorise the native code too, but we’re trying to get high performance with minimum effort.

Copy “julia_v1.m” (i.e. the code which can take vector arguments, but is not vectorised) into q6, and rename it “julia_v4.m”. Now use the matlab `codegen` command to compile it into a native function called `julia_v4_mex`. The commands used to compile it should be recorded in a file in q6 called “compile_julia_v4.m” – whether “julia_v4_mex” appears in the q6 directory or the base directory doesn’t matter.

Some hints in this process are:

- Read the documentation for `codegen`.
- Note that you have to add something to the .m file you want to compile.
- You’ll need to make sure you have a C compiler setup – it should be ok by default, but if necessary look at the `mex` command.

- For this function, the size of the input can vary at run-time. You'll need to work out how to do this – look at the `–config` parameter to `codegen`, and `coder.config`. You might also wonder if this is a case of using "DynamicMemoryAllocation in matlab".
- Check the reports coming out of `codegen`, both for errors, and to see how it is getting converted into code.

Feel free to apply as many optimisations as you can find, with the restriction that you must stay in matlab world, so no editing of the the low-level C. Also, your code should be correct.

Play around with the performance – you should get a pretty decent speed-up here.

4.7 Q7 - Basic parallelism

The final think we'll try is to look at very basic parallelism using `parfor`. First, read the documentation for `parfor` in matlab. You'll see that there are quite a lot of restrictions on the types of statements you can put inside a parallel for loop, and particularly on the "outputs" of a `parfor`.

Copy "render_julia_v2.m" into q7, and rename it to "render_julia_v3.m". Modify this renderer so that it now generates the calls to the juliaproc in parallel.

You may find some problems with writing into the output array (errors about classified variables), in which case follow the error message/hints that matlab is giving you. Also remember to make sure that you call `matlabpool` to get the multi-processing to start up.

4.8 Q8 - Final performance testing

Hopefully you have been using your timing function to play around with performance as you go, but to finish we will look more carefully at performance. You should now have four different juliaproc implementations (`julia_v1` ... `julia_v4`) and two renderers. These combinations will all have different performance properties, which will vary by architecture and input parameters. The amount of parallelism used by matlabpool will also affect the performance of the parallel loop, so should be considered.

Create a script called "plot_times.m" in q8. This script should create *at most* four timing graphs as pdfs, exploring what you think best captures the tradeoff between the different functions. The script should be completely automated, and automatically do the timing runs and generate the pdfs. Make sure the pdfs work as standalone graphs, so they should have appropriate labels, titles, and so on. However, it is up to you precisely what you want to graph – pick wisely, and choose the axis ranges and scales to make sure you are telling a story.

Your submission should contain the pdfs generated on the machine you used, as well the script that generated them.

4.9 Submission

For submission, you should prepare a zip file which mirrors the structure of the original zip file, but with your files added. The matlab files need to match the names specified in the various questions – if they are mis-named, they will break the test-harness. The pdf graphs from Q8 will be looked at by a human, so they should have appropriate names.

When ready, submit the zip via blackboard.