5602 Assignment 1

Kyle Parfrey — kyle@maths.tcd.ie

October 2020

Brief directions

The aim here is to explore the practical consequences of a processor's cache structure. You should write a C program to compute a vector triad, for vectors of doubles, of lengths 10 to 10^8. Print the data to a file, and create a plot that shows the change in performance as the vector length increases; this should be a plot of performance in GFLOPS (Y axis) vs vector length (X axis), and you will want to use a log scale for the vector length.

Use chuck to produce the data for the final plot. Explain the plot's various features with reference to the cache structure of chuck's processors.

Assorted helpful information

The vector triad operation is $\vec{A} = \vec{B} + \vec{C} * \vec{D}$, or:

```
for (int i=0; i < N; i++)
    A[i] = B[i] + C[i] * D[i];
```

Here `N` is the vector length. For each value of `N`, you'll want to do the above vector operation many times to get more reliable timing; this means wrapping that loop in another loop over "repetitions". Since you need to do this double-loop for each value of `N`, you'll need a final outer loop over those, giving three nested `for` loops in total.

Unfortunately, compilers are sometimes too clever for their own good, and will recognise that all of our repetitions are identical and optimise away our repetitions loop. You can prevent this from happening by putting some kind of an `if` test inside the repetitions loop but outside the inner loop along the vector. One option is to test if some value like `A[5]` is negative — if you've chosen the values of `B, C` and `D` such that this never occurs, the statement following the `if` test is never executed.

You should allocate memory for the four vectors dynamically:

```
double *A = malloc(N * sizeof(double));
```

Processes are generally given a limited amount of stack space, which we'll likely exceed for large `N`, resulting in memory errors. Fun activity if you've finished your plot early: use large statically allocated arrays (like `double A[N];`) to estimate the default stack size on chuck. There will be one bonus point for this (out of 16 — marks in this course will be expressed in hexadecimal, obviously).

To do the timing, you could use the `gettimeofday` function from the C standard library. This places the current time into a structure, with members for the second and microsecond. You can just call this function:

```
double walltime()
{
    struct timeval t;
    gettimeofday(&t, NULL);
    double wtime = (double) (t.tv_sec + t.tv_usec*1e-6);

    return wtime;
}
```

This returns the current time in units of seconds (the offset doesn't matter, since you'll just be measuring the time elapsed between two calls to this function). Use it like: `double time0 = walltime();`.

To write output into a file, you'll first want to open the file in write mode:

```
FILE *fptr = fopen("mydatafile.txt", "w");
```

The file pointer `fptr` can now be used to direct printf-style writes into the file. For every value of `N`, write the measured performance to the file:

```
fprintf(fptr, "%d   %lf\n", N, GFLOPS_measured);
```

To use the above functions, you'll need to `include` the following: `sys/time.h, stdlib.h, stdio.h`. Compile with level-1 optimisation: `gcc -O1 cachefun.c`. To find information on chuck's processors, run `lscpu` at the command line; the sizes of the various caches are near the bottom.

You can make the plot however you like. If you don't have a personal favourite yet, I'd recommend matplotlib — it's fairly easy to learn, and can make very complicated and very nice plots. If you have python on your personal machine, the easiest thing would be to copy the output file there and make the plot using matplotlib's interactive features. If not, you can run python scripts on chuck that will produce PDFs or images in your chosen format, which you'll be able to copy back to your own machine or open over SSH if you've enabled X forwarding.

If you're making the plot on chuck, you need to set the backend to something non-interactive, like `Agg`, before importing pyplot. Here's a very basic example that should work:

```
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt

plt.plot([1,2,3,4], [5,6,7,8])
plt.savefig("my_favourite_plot.pdf")
```

Matplotlib has good documentation — look at the pyplot tutorial at https://matplotlib.org/tutorials.

To make the plot, you'll need to load your saved data from the file into python. If you've saved this as two columns of text as I did above, you can load them into python with numpy's `loadtxt` function:

```
import numpy as np
data = np.loadtxt("mydatafile.txt")

col_0 = data[:,0]
col_1 = data[:,1]
```

The `loadtxt` function loads the data into a single array. You can get your two original columns back separately using the array slicing above.

There's a lot of useful information in Chapter 1 of Hager and Wellein's *Introduction to High Performance Computing for Scientists and Engineers*, specifically section 1.2.1. A PDF of this chapter is in the books section on Blackboard.