Exercises for Memory-Efficient Computing (answers)

For the solutions of the exercises, you may want to check the accompanying material:

- *Timings-escher.pdf*, a LibreOffice spreadsheet with all data and graphs.
- *Timings-escher.pdf*, a PDF version of the above document.

Finding optimal block

Exercise 0

- Which is the best block size? How it compares with L1 and L2 caches?

It can be *anything* between 10^4 and 10^6 bytes, depending on the architecture of the machine. It is usually comparable with the L2 cache size of your CPU, but the moral here is that, in order to find the best block sizes for your computations, there is no replacement for experimentation ;)

For your info, you can know the sizes for your caches with:

```
$ cat /sys/devices/system/cpu/cpu0/cache/index1/size
32K    # level 1 cache
$ cat /sys/devices/system/cpu/cpu0/cache/index2/size
256K   # level 2 cache
$ cat /sys/devices/system/cpu/cpu0/cache/index3/size
8192K  # level 3 cache
```

- How is the speed-up compared with a *raw* evaluation?

Approximately 2x or 3x.

- Why do you think the blocked calculation is faster than the raw one (for optimal blocksize)?

Basically because the blocking technique avoids bringing each operand each time from memory to cache every time it is used. Rather, it brings a block for every operand that fits in cache comfortably, do all the needed operations for this block and then you are done. As a consequence, the operands need to travel to the cache only once.

Optimizing arithmetic expressions

Exercise 1

- Set the *what* parameter to "numexpr" and take note of the speed-up versus the "numpy" case. Why do you think the speed-up is so large?

Numexpr basically follows the blocking technique as explained above. The additional speed-up is due to the fact that the virtual machine in numexpr is implemented at C level, and hence, it is faster than the above pure-python implementation.

Also, numexpr can do the `x**3 --> x*x*x` expansion, avoiding the expensive *pow()* operation (which is made in software).

Exercise 2

- Why do you think numpy is doing much more efficient with this new expression?

Because the *pow()* is computationally very expensive, and we removed it.

Note that NumPy cannot expand *pow()* too much as it would create too much (*big*) temporaries. As Numexpr temporaries are much smaller (fits in cache), it can expand pow() much more cheaply.

- Why the speed-up in numexpr is not so high in comparison?

The virtual machine of numexpr already made the *x**3 -> x*x*x* expansion automatically.

- Why numexpr continues to be faster than NumPy?

In this case, basically just because numexpr uses the blocking technique. Hence, this is a fair comparison on how much this technique can do for your computations.

Exercise 3

- Why do you think it is more efficient than the above approaches?

The reason is two-folded. In one hand, C code only need atomic temporaries that are kept in registers in CPU, as computation is done element-by-element, so no blocking technique is really needed here. Also, C-code does not have numexpr's virtual machine overhead, resulting in the fastest implementation.

Parallelism with threads

Exercise 4

- How the efficiency scales?

Scaling should approximately follow a negative exponential distribution. However, you will notice that the asymptote is different from zero. Such an asymptote is basically fixed by the memory bandwidth of the machine.

- Why do you think it scales that way?

The answer is that you are hitting the memory bandwidth limit in some of your computations. So, the reason is that the problem becomes bottle-necked by memory access when you use a large number of threads.

- How performance compares with the pure C computation?

You will see that numexpr can beat a pure C computation. This is for a good reason: numexpr uses several threads here, while C code does not. But see below.

Exercise 5

- How the efficiency scales?

It scales very similarly than numexpr, but it performs a bit better.

- Which is the asymptotic limit?

It is still closer to the bandwidth limit than numexpr one.

The reason for this is two-folded: the virtual machine in numexpr always has some overhead that C does not (i.e. dealing with blocks takes time), and also numexpr still have to deal with (small) temporaries in cache.

Exercise 6

- Compare the performance of this with polynomial evaluation.

y = x performs pretty similarly to the polynomial:

```
y = ((.25*x + .75)*x - 1.5)*x - 2
```

- Why it scales very similarly than the polynomial evaluation?

The bottleneck is in memory access, no CPU computing time.

- Could you have a guess at the memory bandwidth of this machine?

A rough estimation is that, if saturation point is around 0.01 seconds for computing an array of size:

```
8 * 1e7 ~ 76 MB
```

then the bandwidth for this is around:

```
(76 / .01) ~ 7600 MB/s ~ 7.5 GB/s
```