# DVA248-VT22 : CPU Architecture Assignment

Solve all parts of the following exercises and submit your solutions on Canvas.

## Premise

Python is a *dynamically typed* and *interpreted* language. These are two of the reasons because Python programs perform remarkably slower than those written, for example, in C... But why? Let us consider, for example, the sum of two variables. For a C program, the compiler (which translates the source code into an optimized list of CPU instructions) knows the type of each variable since its declaration so that the proper instruction can be performed directly. The Python interpreter, instead, just knows that the current instruction involves two Python objects that must be identified before that the proper summation routine can be called; once the routine is done, a new Python object is created to hold the result.

A further important issue which penalizes the performance of the "object model" used in Python is that memory can be managed in a very inefficient way. While the items of a C array are stored contiguously in memory (with a consequent reduction of the access time due to the cache), the equivalent Python list consists of a set of scattered objects. For this reason, to appreciate the importance of the cache, we have to cheat with Python by replacing the interpreter with a compiler. To this end, the Numba module comes into play (students are invited to read the linked document to get an overview of what Numba is).

### Exercise 1:

Matrix multiplication is a build-block for many applications as well as a good example to experiment the proper usage of the CPU cache. Its simplest implementation consists in a triple nested loop iterating a number of times equal to the dimensions of the multiplied matrices. The following Python function multiplies two matrices, $\mathbf{A}^{n\times m}$ and $\mathbf{B}^{m\times p}$, then returns the resulting matrix $\mathbf{C}^{n\times p}$:

```
from numba import jit
import numpy
import time

@jit(nopython=True)
def ijk(A, B):
    n = A.shape[0]
    p = B.shape[1]
    m = A.shape[1]
    assert m == B.shape[0]
    C = numpy.zeros((n,p), dtype=numpy.uint32)
    for i in range(n):
        for j in range(p):
            for k in range(m):
                C[i][j] += A[i][k] * B[k][j]
    return C

@jit(nopython=True)
def slow(A, B):
    n = A.shape[0]
    p = B.shape[1]
    m = A.shape[1]
    assert m == B.shape[0]
    C = numpy.zeros((n,p), dtype=numpy.uint32)
```

```
    #your solution goes here
    return C

@jit(nopython=True)
def fast(A, B):
    n = A.shape[0]
    p = B.shape[1]
    m = A.shape[1]
    assert m == B.shape[0]
    C = numpy.zeros((n,p), dtype=numpy.uint32)
    #your solution goes here
    return C

def testMatrixProductFunction(A,B,f):
    print(f"\nTesting     : {f.__name__}(A,B)")
    tic = time.perf_counter()
    C = f(A,B)
    toc = time.perf_counter()
    print(f"Elapsed time: {toc-tic:0.3f} seconds")
    assert numpy.array_equal(C,numpy.matmul(A,B))

A = numpy.random.randint(10, size=(1000,1000), dtype=numpy.uint32)
print(f"A belongs to {A[0][0].__class__.__name__}^[{A.shape[0]}x{A.shape[1]}]")
B = numpy.random.randint(10, size=(1000,1000), dtype=numpy.uint32)
print(f"B belongs to {B[0][0].__class__.__name__}^[{B.shape[0]}x{B.shape[1]}]")

testMatrixProductFunction(A,B,ijk)
#testMatrixProductFunction(A,B,slow)
#testMatrixProductFunction(A,B,fast)
```

In Python, it is possible to specify which memory layout to adopt for storing the values of a matrix. The two main options are `'C'` (default) and `'F'` standing for, respectively, data stored in row-major (C-style) and column-major (Fortran-style) order. In other words, the C-style implies that the matrix items in the same row are stored contiguously in memory while the Fortran-style implies that the matrix items in the same column are stored contiguously in memory. If you are not sure about which order is used by a multi-dimensional array you can, for example, print its flags in this way:

```
print(A.flags)
```

a. Estimate the number of cache misses that occur accessing **A** in the function `ijk()`. Assume that each cache line can store 16 `numpy.uint32` values.

Now it's time to put your hands on the code. As first experiment, run the attached program and take note of the elapsed time. Then try to comment the `jit` decoration on top of the function and run the program again: don't worry, your machine is not broken and the program has not crashed... It just takes a "little bit" longer. Once you are done uncomment the `jit` line!

b. Complete the `slow()` function above by changing the order of the nested loops to obtain an equivalent function that runs slower than the original one. When you are done uncomment the related `testMatrixProductFunction` call for testing it.

c. Complete the `fast()` function above by changing the order of the nested loops to obtain an equivalent function that runs faster than the original one. When you are done uncomment the related `testMatrixProductFunction` call for testing it.

## Exercise 2:

If-statements can brake the ideal instructions' flow in the CPU pipeline with a consequent reduction of the overall performance. The next task deals with improving the performance of a function by

removing all its if-statements. The target function takes as input an array of 3D points and counts how many of them belong to each octant of the Euclidean three-dimensional coordinate system.

Your task is to complete the second function (called `OctantPopCountNoIf`) in which the chain of `if...elif...else` must be replaced with a function that directly calculates to which octant a point belongs based on the sign of its coordinates. Uncomment the last line to check the correctness of your solution.

*Hints*: assign a different bit value to each coordinate of every point according to its sign and concatenate those bits to form a number that defines the corresponding octant. To cast the result of a Boolean expression, `bexp`, into an integer use `int(bexp)`.

```
from numba import jit
import numpy
import time

@jit(nopython=True)
def OctantPopCount(points):
   assert points.shape[1] == 3
   counters = numpy.zeros(8, dtype=numpy.uint32)
   for p in points:
      if p[0]<0 and p[1]<0 and p[2]<0:
         counters[7] += 1
      elif p[0]<0 and p[1]<0 and not p[2]<0:
         counters[6] += 1
      elif p[0]<0 and not p[1]<0 and p[2]<0:
         counters[5] += 1
      elif p[0]<0 and not p[1]<0 and not p[2]<0:
         counters[4] += 1
      elif not p[0]<0 and p[1]<0 and p[2]<0:
         counters[3] += 1
      elif not p[0]<0 and p[1]<0 and not p[2]<0:
         counters[2] += 1
      elif not p[0]<0 and not p[1]<0 and p[2]<0:
         counters[1] += 1
      else:
         counters[0] += 1
   return counters

@jit(nopython=True)
def OctantPopCountNoIf(p):
   assert points.shape[1] == 3
   counters = numpy.zeros(8, dtype=numpy.uint32)
   #your solution goes here
   return counters

def testOctantPopCountFunction(points, f):
   print(f"\nTesting     : {f.__name__}()")
   n = points.shape[0]
   tic = time.perf_counter()
   counters = f(points)
   toc = time.perf_counter()
   print(f"Elapsed time: {toc-tic:0.3f} seconds")
   return counters

points = numpy.random.random_sample((2000000, 3)) - 0.5
c1 = testOctantPopCountFunction(points, OctantPopCount)
c2 = testOctantPopCountFunction(points, OctantPopCountNoIf)
#assert numpy.array_equal(c1,c2)
```