MU4MA016 course notes

Didier Smets

September 18, 2024

WEEK 1 : SEPT. 5TH 2024

The course being entitled **Algorithms** and **Data Structures** for **Computer programming**, in this introductory lecture we will briefly discuss each of these three words, trying to highlight by some simple examples the kind of questions which will interest us in the following.

1 Algorithm

The following algorithm was originally described by Euclid in order to find the greatest common divisor (gcd) between two positive integers a and b.

Algorithm 1 Euclid's algorithm (original version)

```
Require: $a, b$ are positive integers
  function GCD($a$, $b$)
      while $a \neq b$ do
          if $a > b$ then
              $a \leftarrow a - b$
          else
              $b \leftarrow b - a$
          end if
      end while
      return a
  end function
```

The above description is called `pseudo code`. It is intended to be read my humans, and it must be unambiguous for humans. Instead, computer programs which we will tackle later, are intended to be read by computers (compilers or interpreters), which implies further restrictions on them (both syntaxic and in terms of the operations available).

Among the properties of algorithms, we will be particularly interested in the following :

- Correctness : that is the least one should ask for, but it needs not be overlooked !
- Complexity : both time and space complexity (to be explained below).
- Ease of implementation : because in the end it will need to be turned into a program.

Correctness

In the case of Euclid's algorithm we could argue along the following lines :

1. After each step of the while loop, a and b remain both positive[1] Indeed they must be different to enter the loop and the smaller is subtracted from the larger.
2. Because of the previous claim, the sum $a + b$ decreases strictly after each step of the loop. In particular, there can be at most finitely many steps of the loop and therefore the algorithm will always terminate.
3. Let c be the actual greatest common divisor of a and b. Then after each step of the loop, c is still a divisor of a and b. Indeed, if a number divides two other numbers, then it also divides their difference. Since the algorithm eventually returns (the modified version of) a, let's call it d, it follows that c divides d.
4. We next claim that d divides (the original values of) a and b. To prove this we argue as above, but backward. First, at the end of the while loop we have $a = b = d$, and therefore d divides both a and b. Now if d divides a and b after some (arbitrary) step of the loop, it also did at the previous one. This is because if a number divides two other numbers, it also divides their sum (and running a step backward correspond to changing one of them to the sum of the two).
5. In summary, we have shown on one hand that c divides d (in particular $c \leq d$), and on the other hand that d divides a and b. By definition of c being the greatest common divisor of a and b, it follows that $c = d$, which proves the correctness.

In many cases, proving algorithm correctness proceeds by showing that some form of invariant is preserved during the process (especially when some loops are involved). In the example above, the invariant was the divisibility by c.

Complexity

As mentioned already, for all the algorithms we will study, we shall discuss both **time complexity** and **space complexity**. These notions only make rigorous sense within a so-called *computation* and a *memory* model, but in short :

[1]Note to french speaking students : in mathematical english **positive** means "strictement positif"; the french "positif" is instead translated as **non negative**.

- The time complexity measures how much computations the algorithm needs to perform in order to provide its result. Since the number may depend on the input values, it is often useful to distinguish between the worst case scenario and some form of average case.
- The space complexity measures the amount of local storage required by the algorithm in order to perform its computations, in addition to the storage of its inputs.

In practice the two notions often pull in opposite directions, and the optimum is a matter of compromise.

In the case of Euclid's algorithm, space complexity is trivial: no temporaries are required since only updates of the original variables occur. Time complexity (assuming that comparison and arithmetic operations can all be performed in $O(1)$ time[2]) can easily be computed to be $O(\max(a, b))$, where the worst case happens when one of the two values, say b, is 1 (just think of the routing of the algorithm in that case, leading to a being decreased by 1 at each step until it reaches 1).

The following variant is actually often called the Euclid algorithm too, although it was not presented by Euclid in this form.

Algorithm 2 Euclid's algorithm (improved version)

```
Require: $a, b$ are positive integers
Require: $c$ a temporary storage for an integer
  function GCD($a$, $b$)
      while $b \neq 0$ do
          $c \leftarrow b$
          $b \leftarrow a$ modulo $b$
          $a \leftarrow c$
      end while
      return a
  end function
```

Exercise :

- Prove the correctness of this second version.
- Prove that its time complexity is $O(\log(\max(a, b)))$.
  Hint: proceed backward and compare the current values of a and b to the Fibonacci sequence defined by $F_0 = 0$, $F_1 = 1$ and $F_{k+2} = F_{k+1} + F_k$.

[2]As reasonable as it may look, in practice most implementations would implicitly require that the input integers fit in some range allowing them to be represented easily on the architecture, e.g. over 64bits, which by nature kills the whole meaning of $O()$ analysis, but that is what it is...

In practical applications, a logarithmic complexity is a really strong and desirable one, in comparison e.g. here to the linear complexity of the original version. Even worse would be a quadratic, cubic, general polynomial or even non polynomial (abbr. as NP) complexity.

2 Data Structure

Consider the following two variants of the birthday-matching problem (i.e. determine if at least two among a class of students have the same anniversary date, omitting year).

Algorithm 3 Birthday match using sets

```
Require: As input, a set of students, called $class$
Require: As temporary storage, a set of students called $visited$
  Initialize $visited$ as the empty set
  for all student $a$ in $class$ do
      for all student $b$ in $visited$ do
          if $birthday\_match(a, b)$ then
              return true
          end if
      end for
      Insert $a$ into $visited$
  end for
  return false
```

We have implicitly assumed here that we have a notion of *Set*, over which we can *Iterate*, and which we can grow by making use of some *Insert*.

An **Abstract Data Structure (ADS)** is an abstract description of

1. What kind of data can be stored (possibly referring to other existing ADS)
2. What operations can be performed on these data

In terms of time complexity, in worst case scenarios we will need to apply $O(N^2)$ (where N is the number of students in class) times the birthday match check. We should also take into account the time complexity associated to the *Iterate* (for all) and *Insert* functions. These later must either be postulated (as we did for the arithmetic operations in Euclid's), or guaranteed by some implementation of a Concrete Data Structure within a computing and memory model.

In terms of space complexity, the set *visited* may grow at most to a size of N. With the same caution as above regarding theory vs implementation, we will say that the space complexity is therefore $O(N)$.

Note that whenever $N > 365$, the algorithm could exit early with *true*, by an immediate application of the pigeon-hole principle[3] Not only this underlines the caution that should always accompany asymptotic analysis in finite frameworks, but it also suggests a very different variant :

Algorithm 4 Birthday match using an array

```
Require: As input, a set of students, called $class$
Require: As temporary storage, an array $A$ of 365 boolean
  Initialize all entries of $A$ with false
  for all student $s$ in $class$ do
      Let $k$ be the anniversary date of $s$ (view as an integer $1 \leq k \leq 365$)
      if $A[k] = true$ then
          return true
      else
          $A[k] \leftarrow true$
      end if
  end for
  Return false
```

The time complexity of this variant is now only $O(N)$ (with the same caution as before), and the space complexity is fixed, independently of the size of the class, therefore $O(1)$.

Exercise : Replace 365 by an integer M (imagine we compare some ID numbers instead of birthdays), and assume that both M and N get large but also that $M \gg N$ (by this we mean that M is much larger than N) and that a spatial complexity of $O(M)$ is forbidden because it would be too large in practice. Are we forced to resort to the first version of the algorithm, or could we do better ? (*Hint: hash tables and sorted trees are both in the menu for later*).

The data structures that we shall go over in this course include : arrays, stacks, queues, lists, hash tables, (binary search) trees, heaps and graphs. Different implementations may use different concrete data structures to represent the same abstract data structure. The abstract notion of Set or of Dictionary, is often implemented either via a hash table, or with some form of tree.

3 Computer Programming

We are not going to study computer programming per se[4], but we are going to study how to efficiently implement (good) algorithms on computers, using specific languages.

[3]If there are more pigeons than cages, at least two pigeons must share the same cage. In french: “Principe des tiroirs”.

[4]In particular this is not a course on the latest hype in C++ !

As scientific programmers, there are essentially two choices, which differ by their strong points and limitations.

- Using a high level language, offering a number of high level native data structures, and potentially also an interactive interpreter. This is the case e.g. of Python and its CPython interpreter, or of Julia with its just-in-time (JIT) compiler. This is the easiest to start with, and it can be efficient too when appropriate libraries (coded in low level languages) are available for the task of choice, like e.g. Numpy just to name one. This is also a very good test bed for profiling a new algorithm on small data sets, before turning to a lower level language for performance on large scale data sets.

- Using a lower level language, giving a closer access to the hardware or at least to the virtual memory, and which can be compiled into efficient machine code due to compiler optimizations. This is the case e.g. of C/C++, and the one we will use in this course.

The syntax and grammar of C will not be taught in class. There are very good references online, C grammar is easy, and those who have never met with it will catch-up quickly in the first TP classes. From C++ (kind of a superset of C) we will only ever borrow a few features that *may* sometimes be handy.

Although lower level languages are said to be closer to the hardware, in this course for the most we will only need to understand an take into account the following. The storage memory is presented to us by the operating system (OS) as a large addressable 1D array, the so-called *virtual memory*, and

1. **The access time needed to bring back some data from RAM to the CPU can be much larger than the typical time needed by the CPU to perform some computation/instruction** (one usually measures time in so-called CPU cycles). As of today, it can be by a factor 100x to 1000x, with respect e.g. to an arithmetic operation.

2. In order to overcome this otherwise huge bottle-neck in computation, a number of physical levels of so-called caches are introduced in between the CPU and the RAM. You can think of these cache levels as smaller but faster access memories, and there are strategies (independent of us) to decide what is kept in cache (I like the analogy of RAM being the university library, the next level of cache is your home bookshelf, and the closest level of cache to CPU is your nightstand). **The important thing to remember as a programmer, is that if some data is put into cache, then the "nearby" data (in the virtual memory 1D array) is brought at the same time, and therefore it will be readily available provided our successive accesses to memory are sufficiently local**.

This notion of *locality* is well understood (and can be experienced!) in the following simple yet typical example of the computation of the product $A * B$ of two matrices, say of size $N \times N$.

Let's first implement it (in language C) in the way you were taught about the matrix product in Math classes. Note first that the memory being a 1D array, matrices need to be "flattened" to be recorded in memory. The so-called *row-major* convention, i.e. line after line, where the equivalent of what we write $A(i, j)$ in math is accessed in $A[i*N+j]$, is the most popular in C language implementations. The other obvious alternative, the so-called *column-major* convention, was most popular in Fortran language. Which algorithm works best for a matrix product is highly dependent of the choice of the convention, as we shall experience ! In the sequel we choose the *row-major* standard C convention. Note also that in C, indices for array indexing start at *zero*, not at 1 as we usually do in math, this eases address computations.

```
 1 void matrix_product1(float C[], const float A[], const float B[], int N)
 2 {
 3         for (int i = 0; i < N; ++i) {
 4                 for (int j = 0; j < N; j++) {
 5                         /* Compute C_{i,j} */
 6                         C[i * N + j] = 0;
 7                         for (int k = 0; k < N; k++) {
 8                                 C[i * N + j] += A[i * N + k] * B[k * N + j];
 9                         }
10                 }
11         }
12 }
```

When the loop counters i, j or k are incremented, the data locations involved in the actual product computation on line 8 vary differently. An increment of i leads to a jump of N (times the size of a float, not repeated later) in memory for C and A. An increment of k introduces a jump of N in the memory location for B, and finally an increment of j only induces increments of one (i.e. no jump, i.e. local). By far the most problematic of all these is the jump of N introduced by an increment of k : indeed it arises at every of the N^3 steps of the three nested loops. The increment of i causes the same jump of N, but it only occurs N out of the N^3 steps, which is negligible in proportion. This implementation, at least for large values of N, will therefore likely be memory bottlenecked.

Instead, let us propose the following alternative obtained for the most by a permutation of the loops in j and k :

```
 1 void matrix_product2(float C[], const float A[], const float B[], int N)
 2 {
        for (int i = 0; i < N; ++i) {
                /* Zero initialize C */
                for (int j = 0; j < N; j++) {
                        C[i * N + j] = 0;
                }
                for (int k = 0; k < N; k++) {
                        for (int j = 0; j < N; j++) {
                                /* Update C_{i,j} */
                                C[i * N + j] += A[i * N + k] * B[k * N + j];
                        }
                }
        }
}
```

Exercise : First convince yourself that this second version also actually computes the product of A and B ! Then analyze the memory accesses as above, and observe that the non local jumps only occur now N^2 out of N^3 times, which is negligible in proportion if N is large, and therefore should better avoid the memory latency problem.

Exercise : Check it in practice by testing e.g. with $N = 1000$, after you will be familiar with C and gcc in the first TP classes. In these tests, try both the compiler optimization `-O0` (i.e. no optimization) and `-O3` (most optimizations).

Proposed solution Here is a proposed test implementation, that will serve for those of you discovering C. Reproduce and test on your own !

data/matrix_multiplication_test.c

```
#include <stdio.h>      // For printing to the console
#include <stdlib.h>     // For memory allocation (malloc), string to integer
                        // converion (atoi), and generating random numbers (rand)
#include <sys/time.h>   // For implementing a chrono (struct timeval and
#include <time.h>       // For time() function used in random seed generation
                        // gettimeofday). Requires a POSIX OS.

void matrix_product_v1(float *C, const float *A, const float *B, int N) {
        for (int i = 0; i < N; ++i) {
                for (int j = 0; j < N; j++) {
                        C[i * N + j] = 0;
                        for (int k = 0; k < N; k++) {
                                /* Update C_ {i , j } */
                                C[i * N + j] += A[i * N + k] * B[k * N + j];
                        }
                }
        }
}

void matrix_product_v2(float *C, const float *A, const float *B, int N) {
        for (int i = 0; i < N; ++i) {
                /* Zero initialize C */
```

```c
                for (int j = 0; j < N; j++) {
                        C[i * N + j] = 0;
                }
                for (int k = 0; k < N; k++) {
                        for (int j = 0; j < N; j++) {
                                /* Update C_ {i , j } */
                                C[i * N + j] += A[i * N + k] * B[k * N + j];
                        }
                }
        }
}

void timer_start(struct timeval *tv) { gettimeofday(tv, NULL); }

unsigned int timer_stop(const struct timeval *tv, const char *str) {
        struct timeval now;
        gettimeofday(&now, NULL);
        unsigned int mus = 1000000 * (now.tv_sec - tv->tv_sec);
        mus += (now.tv_usec - tv->tv_usec);
        if (str[0]) {
                printf("Timer %s: ", str);
                if (mus >= 1000000) {
                        printf("%.3f s\n", (float)mus / 1000000);
                } else {
                        printf("%.3f ms\n", (float)mus / 1000);
                }
        }
        return (mus);
}

/* When profiling with compiler optimization, it is important to ''do''
 * something with the result, all of it, in order to prevent the smart
 * compiler from just no computing what is not latter used.
 */
float check_and_avoid_lazy_optimizers(const float *C, int N) {
        float dummy = 0;
        for (int i = 0; i < N * N; ++i) {
                dummy += C[i];
        }
        return dummy;
}

/* We initialize A and B with random numbers in the interval [0,1] */
void initialize_matrices(float *A, float *B, int N) {
        srand(time(NULL));
        for (int i = 0; i < N * N; ++i) {
                A[i] = (double)rand() / RAND_MAX;
                B[i] = (double)rand() / RAND_MAX;
        }
}

int main(int argc, char **argv) {
        /* Read 1st program argument string and convert into a number*/
        int N = atoi(argv[1]);
        if (N <= 0) {
                printf("First argument N must be positive.\n");
                return (EXIT_FAILURE);
        }

        /* Allocate memory for storing A, B, and the resulting C = A * B */
        float *A = (float *)malloc(N * N * sizeof(float));
        float *B = (float *)malloc(N * N * sizeof(float));
        float *C = (float *)malloc(N * N * sizeof(float));
        if (A == NULL || B == NULL || C == NULL) {
                printf("Memory allocation failed ($N$ too large ?)\n");
                return (EXIT_FAILURE);
        }

        /* For measuring timings of both algorithm */
        struct timeval chrono;

        /* Test 1 */
        initialize_matrices(A, B, N);
        timer_start(&chrono);
        matrix_product_v1(C, A, B, N);
        timer_stop(&chrono, "Algo 1");
        printf("Sum of C : %f\n", check_and_avoid_lazy_optimizers(C, N));

        /* Test 2 */
        // initialize_matrices(A, B, N);
        timer_start(&chrono);
        matrix_product_v2(C, A, B, N);
        timer_stop(&chrono, "Algo 2");
        printf("Sum of C : %f\n", check_and_avoid_lazy_optimizers(C, N));

        /* Release memory to OS */
        free(C);
        free(B);
        free(A);

        return (EXIT_SUCCESS);
}
```

data/matrix_multiplication_test.c

We then compile it (first with zero optimization) :

```
@: gcc -Wall matrix_multiplication_test.c -o matrix_multiplication_test
```

and test it

```
@: ./matrix_multiplication_test 1000
Timer Algo 1: 4.159 s
Sum of C : 250005104.000000
Timer Algo 2: 3.229 s
Sum of C : 250005104.000000
```

We can observe a modest difference but not great. Let's introduce compiler optimization (level 3 = all).

```
@: gcc -Wall -O3 matrix_multiplication_test.c -o matrix_multiplication_test
@: ./matrix_multiplication_test 1000
Timer Algo 1: 1.031 s
Sum of C : 250005104.000000
Timer Algo 2: 141.192 ms
Sum of C : 250005104.000000
```

First as you can see the -O3 flag implied a great speed-up, but more importantly the _v2 version of the algorithm also now runs one order of magnitude faster than the _v1 one !

Remark. Although the matrices A and B were filled with random real numbers between 0 and 1, the sum of the coefficients of C seems to be an integer. How is that possible ?! The (pseudo)-random values stored in A_{ij} and B_{jk} are supposed to model a uniform distribution on $[0, 1]$, therefore

$$\mathbb{E}(sum(C)) = \sum_{i,j,k=1}^{N} \mathbb{E}(A_{ij}B_{jk}) = \sum_{i,j,k=1}^{N} \mathbb{E}(A_{ij})\mathbb{E}(B_{jk}) = \frac{N^3}{4},$$

where we have used that A_{ij} and B_{jk} are independent (they are different samplings of the random variable) and $\mathbb{E}(A_{ij}) = \mathbb{E}(B_{jk}) = \frac{1}{2}$ for a uniform variable over $[0, 1]$. For $N = 1000$, we expect therefore that $\mathbb{E}(sum(C)) = 2.5e8$, which indeed is an integer. But the error itself, in the example sampling above 5104, has no reason to be an integer. This is not at all related to the maths, but instead to the way real numbers (here 32 bits floating points) are stored in a computer, we will discuss that into more details later. Not all real numbers are represented (there would be infinitely many of them!), and it happens that within 32bits floats the closest float to $2.5e8$ is $2.5e8 + 1$. If you take $N = 100$ instead, you'll start observing fractional parts in the answer.

Exercise : Compute that

$$\mathbb{E}(sum(C)^2) = \frac{1}{16}N^6 + \frac{1}{24}N^4 + \frac{1}{36}N^3$$

and therefore that

$$\Sigma(sum(C)) = \sqrt{Var(sum(C))} = \sqrt{N^4/24 + N^3/3} \simeq 0.2N^2 \text{ for N large.}$$

Run the test above multiple times, of with different values of N, and observe/check then that the deviation to the expected value is indeed of the order of the standard deviation $\Sigma(sum(C)) \simeq 0.2N^2$.

Hint : while evaluating terms of the form $\mathbb{E}(A_{ij}B_{jk}A_{i'j'}B_{j'k'})$, distinguish between the three cases : a) $(i, j, k) = (i', j', k')$, b) $(i, j) = (i', j')$ or $(j, k) = (j', k')$ but $(i, j, k) \neq (i', j', k')$, and finally c) $(i, j) \neq (i', j')$ and $(j, k) \neq (j', k')$. Recall also that whenever X is uniform over $[0, 1]$, $\mathbb{E}(X^2) = \int_0^1 x^2\, dx = \frac{1}{3}$.

Finally, let us compare our code performance with respect to Python, either using pure Python loops or using available scientific libraries, the most famous being Numpy (it is internally coded in C!).

data/matrix_multiplication_test.py

```python
import numpy
import time

for N in (10, 20, 50, 100, 200):

    A = numpy.random.random((N, N)).astype(numpy.float32)
    B = numpy.random.random((N, N)).astype(numpy.float32)
    C = numpy.zeros((N,N)).astype(numpy.float32)

    if (N <= 200):
        start = time.time()
        for i in range(N):
            for k in range(N):
                for j in range(N):
                    C[i,j] += A[i,k] * B[k,j]
        end = time.time()
        print("Pure Python for N = %4d : %.5fs" % (N, end - start))

for N in (1000, 2000, 4000):

    A = numpy.random.random((N, N)).astype(numpy.float32)
    B = numpy.random.random((N, N)).astype(numpy.float32)
    C = numpy.zeros((N,N)).astype(numpy.float32)

    start = time.time()
    tests = int(10000 / N) + 1
    for i in range(tests):
        C = A.dot(B)
    end = time.time()
    print("Using Numpy for N = %4d : %.5fs" % (N, (end - start) / tests))
```

data/matrix_multiplication_test.py

We obtain :

```
@: python3 matrix_multiplication_test.py
Pure Python for N =   10 : 0.00117s
Pure Python for N =   20 : 0.00923s
Pure Python for N =   50 : 0.07167s
Pure Python for N =  100 : 0.52314s
Pure Python for N =  200 : 4.19137s
Using Numpy for N = 1000 : 0.00737s
Using Numpy for N = 2000 : 0.04697s
Using Numpy for N = 4000 : 0.35011s
```

A number of remarks are in order :

1. For both algorithm, we can observe the expected behaviour of a cubic time complexity. Note that algorithm with a lower power of N do exist (see e.g. this Wiki link), but they are complex to implement and often come with large pre-factor constants. The most well-known is Strassen's algorithm, at $O(N^{\log_2(7)}) \simeq O(N^{2.8})$, but the present state of the art is $O(N^{2.37})$. Clearly any algorithm must be at least $O(N^2)$ (indeed at least the N^2 entries of C must be filled!), at least for a sequential algorithm, but it is open if one could go as low as $O(N^2)$.

2. There is a **huge** difference between the timings in pure Python and using Numpy, more than three orders of magnitude. As a matter of fact, for pure Python I had to limit N to a smaller value than a few hundreds in order to keep the script run time under control.

3. For $N = 1000$, Numpy is still 20 times faster that our current best implementation in `_v2` !

What can/should we do to improve our code to state of the art performance then ? There are easy improvements :

1. Allow the compiler to use the whole set of CPU instructions for the computer we are running on (this can be an issue if we wish to distribute our program to machines which would not have all this set). The gcc option is `-march = native`. A safer yet usually as effective solution is to require a common vector instruction set, like the MMX or AVX. We shall use the `-mavx` gcc switch.

2. Modify our C code by adding the *restrict* keyword to the output array pointer C of our matrix_product functions. This tells the compiler that the place where we write C does not overlap (one says "aliases with") the one of A and or B, and avoids unnecessary copies.

3. Modify our C code by forcing a 256 bits alignment for our memory allocations of A, B and C. This can improve the use of vector instructions by the compiler (see 1) here above). To do so, we replace `malloc` by `aligned_alloc`.

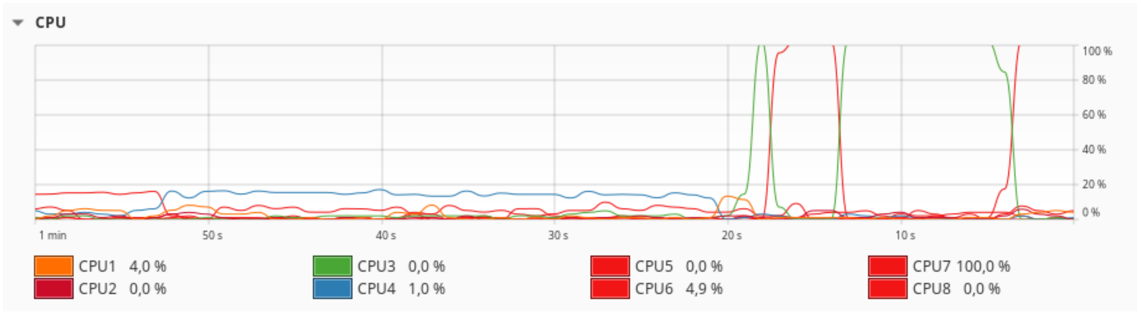
```c
void matrix_product_v2(float *__restrict C, const float *A,
                                              const float *B, int N) {
[...]
}

[...]
        float *A = (float *)aligned_malloc(32, N * N * sizeof(float));
        float *B = (float *)aligned_malloc(32, N * N * sizeof(float));
        float *C = (float *)aligned_malloc(32, N * N * sizeof(float));
[...]
```

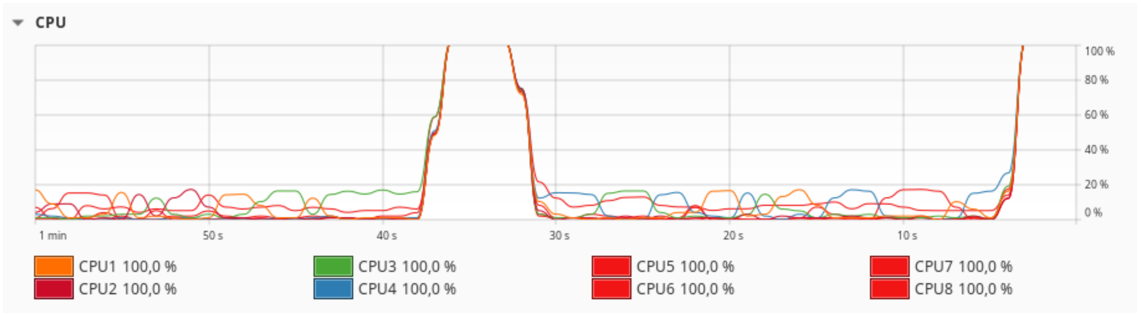
With these and the avx switch we obtain:

```
@: gcc -Wall -O3 -mavx matrix_multiplication_test.c -o
   matrix_multiplication_test
@: ./matrix_multiplication_test 1000
Timer Algo 1: 1.042 s
Sum of C : 250279296.000000
Timer Algo 2: 69.523 ms
Sum of C : 250279296.000000
```

We have improved from $140ms$ to $70ms$, that is a factor 2, not bad ! Be we are still missing a factor almost 10. Let us observe the Operating System Monitor, first during our own implementation :



and then during Numpy's implementation[5]



The road-map is clear : our code is sequential, it uses just one CPU core, while Numpy is making use of parallelism to use all the available 4 cores / 8 threads in my (modest) computer !

Numpy's (quite complex) source code can be browsed on Github here. One C source file for (one of the possible versions of) matrix multiplication is this one. It has many includes and internal preprocessor macros making it difficult to read though, you may even have difficulties to recognize a valid C code.

Turning our matrix multiplication code into a parallel algorithm is one of the very nice projects that you could implement for this course, especially for those of you that are in the HPC program. Either making use of CPU parallelism (using e.g. `OpenMP` or

[5]In both cases, we have increased N so has to have time to take a screenshot !

better the lower level `pthread`), or even going to the GPU if one is available on your computer (using e.g. `OpenCL`, `OpenGL/Vulkan` compute shaders, or the proprietary `CUDA`).

In a broader perspective, parallelization of algorithms in many different areas of Math/CS has been a very hot topic in recent years. This yields challenges both at the algorithmic level, and at the implementation level. They require specific courses and we shall not dive too deeply into it, yet they all build upon the concepts we shall develop in this course.

Week 2 : Sept. 12th 2024

4 Abstract vs Concrete Data Structure

It is always useful and also often important to separate what we will call the *interface* from what we will call the *implementation*. In the table below I have listed a number of concepts : all the ones in the left column correspond (and may be more or less understood as equivalent between each other) to the interface, while all the ones in the right column correspond to the implementation.

On the user side :	On the programmer side :
• Abstract Data Structure	• Concrete Data Structure
• Interface / Documentation	• Implementation
• Application Programming Interface (API)	• Source code
• Header files In C/C++, typically `.h` or `.hpp`	• Source files In C/C++, typically `.c` or `.cpp`

In vague terms the interface *should* desbribe :

- what kind of data it is designed to handle
- what actions on this data it can perform
- possibly offer some guarantee about algorithmic complexity of some of these actions (worst case, expected mean, etc).

It should *not* describe *how* these are implemented. One of the reasons for the latter is the ability to update/improve the implementation without breaking user code.

A key point to have in mind when designing interfaces and implementations is that less methods/actions available in the interface implies :

- less flexibility for the user
- less constraints for the implementation, and therefore possibly better performance

As a rule of thumb : one should not devise overcomplicated data structures, but choose the simplest one(s) that fit all the identified requirements for a task, but no more. Some people call it the KISS philosophy : Keep It Straight and Simple.

An example in the standard C library which one encounters in the very first steps of learning the language is the `FILE` abstract data structure for file IO (i.e. reading and writing to files).

Some typical code would look like

```c
#include <stdio.h>

// [some code here]

FILE *f;
f = fopen("example.txt", "r");
if (f == NULL) {
        // [produce error message and exit]
}

// [do something with f]

fclose(f);
```

What exactly is contained in FILE needs not be known to us[6], we shall use it only as a *opaque handle*. In the *header* file `stdio.h` we can find[7] (among other things) the **declaration**

```c
FILE *fopen(const char *__filename, const char *__modes);
```

which teaches us that the function `fopen` :

- requires two arguments, both of which being of type `char *` (used to represent character strings) and which won't be modified by the function itself (because they are marked with the keyword `const`).
- returns a pointer to a `FILE` data structure.

Most of the time, to access documentation for standard libraries you will instead refer to online documents, like e.g. `https://cplusplus.com/reference/cstdio/fopen/` in the above case.

Whenever we need to do something with the file, we must use the handle (it is called `f` is the sample code above) and pass it to the relevant function, as in

```c
// [do something with f]
char line[80];
fread(line, 80, f);
```

[6]It is actually a macro for a (complicated) C struct defined in stdio.c
[7]On Linux systems this file is found in `/usr/include/stdio.h`

5 Array based vs Pointer based implementation

The implementation of data structures can usually be divided into two families :

1. Array based
2. Pointer based

The difference between both amounts to the way the data is stored in virtual memory.

When array based, the data is stored in a *contiguous* way and it is assumed that each element of the array occupies the same amount of space in memory. The main implication of this is that to recover the element of index `k`, it suffices to fetch memory at the address being computed as the address of the start of the array plus `k` times the size of an array element. In other words, only the address of the start of the array needs to be recorded.

When pointer based instead, the data may be spread across memory. The implication is that each element of the data should be accompanied with a way to access the next and or the previous one(s), typically by storing its/their address(es).

We can therefore quickly foresee the advantages and limitations of both :

- Array based are potentially much faster because they allow for direct access to any element by its index and need not store addresses. On the other hand, the contiguity assumption has some implications on some operations (e.g. suppressing an element would imply a potentially large copy in order to avoid creating a gap, and the same would apply to some insertion in the middle). The question of how large the reserved memory chunk to store the array should be chose is also an important question, especially when it is not known a priori how large the data structure will be eventually (imagine we read elements from a file, and we only know we are finished when we reach its end).

- Pointer based have the exact opposite advantages and limitations. Adding and/or suppressing elements is conceptually and practically simpler, because it only implies modifying a couple of pointers, but that may come at the price of performance. Besides, direct access to the k-th element (whenever it makes sense, not all structures have a natural mapping to sequences, think e.g. of arbitrary trees) is typically not possible for them.

To provide an example of both, and having the goal of separation interface/implementation in mind, in the sequel we propose one common interface and two possible implementations for a *Stack* data structure. This is one of the simplest data structures, and it will therefore serve well our expository purposes. The first implementation will be pointer based, while the second will be array based. User code should should equally work in both cases, as a matter of fact there is nothing in the interface suggesting how it is implemented.

First the interface :

data/stack.h

```c
// A Header file for a Stack of integers API
#include <stdbool.h> /* For bool type   */
#include <stddef.h> /* For size_t type */

struct Stack; /* Declaration only of the type : struct Stack*/
typedef struct Stack Stack; /* Alias :
                             * Allows to write Stack instead of struct Stack */

/* Create a new stack */
Stack *stack_init();

/* Pushes a new element (int) on top of the stack */
void stack_push(Stack *s, int value);

/* Pop the top element from the stack */
/* NOTE : stack should NOT be empty   */
int stack_pop(Stack *s);

/* Dispose an existing stack */
void stack_dispose(Stack *s);

/* Check if stack is empty */
bool stack_is_empty(Stack *s);

/* Number of elements in the stack */
size_t stack_size(Stack *s);
```

data/stack.h

Next the pointer based implementation :

data/stack_pointer_based.c

```c
// A source file for a pointer based implementation of a Stack of integers
#include <stdbool.h>
#include <stddef.h>
#include <stdlib.h>
#include <assert.h>

#include "data/stack.h"

struct StackNode { /* Declaration and Definition of a struct StackNode */
        struct StackNode *next;
        int value;
};
typedef struct StackNode StackNode;

struct Stack { /* Definition of struct Stack */
        struct StackNode *head;
        size_t size;
};

```

```c
Stack *stack_init()
{
        Stack *s = malloc(sizeof(Stack));
        if (s != NULL) {
                s->head = NULL;
                s->size = 0;
        }
        return (s);
}

void stack_push(Stack *s, int value)
{
        assert(s != NULL);
        StackNode *new_node = malloc(sizeof(StackNode));
        if (new_node == NULL) {
                /* Fail silently */
                return;
        }
        new_node->next = s->head;
        new_node->value = value;
        s->head = new_node;
        s->size++;
}

int stack_pop(Stack *s)
{
        assert(s != NULL && s->size != 0);
        int res = s->head->value;
        StackNode *next_head = s->head->next;
        free(s->head);
        s->head = next_head;
        s->size--;

        return (res);
}

void stack_dispose(Stack *s)
{
        if (s == NULL) {
                return;
        }
        /* First free stack nodes if any */
        while (s->head != NULL) {
                StackNode *old_head = s->head;
                s->head = s->head->next;
                free(old_head);
        }
        /* Then free s itself */
        free(s);
}

bool stack_is_empty(Stack *s)
```

```c
{
        assert(s != NULL);
        return (s->size == 0);
}

size_t stack_size(Stack *s)
{
        assert(s != NULL);
        return (s->size);
}
```

data/stack_pointer_based.c

And finally the array based one :

data/stack_array_based.c

```c
// A source file for an array based implementation of a Stack of integers
#include <stdbool.h>
#include <stdlib.h>
#include <assert.h>

#include "data/stack.h"

struct Stack { /* Definition of struct Stack */
        size_t size; /* Number of elements in the stack        */
        size_t capacity; /* Capacity of the array holding elements */
        int *data; /* The data array itself                  */
};

Stack *stack_init()
{
        Stack *s = malloc(sizeof(Stack));
        if (s != NULL) {
                s->size = 0;
                s->capacity = 0;
                s->data = NULL;
        }
        return (s);
}

void stack_push(Stack *s, int value)
{
        assert(s != NULL);
        if (s->size >= s->capacity) {
                size_t new_cap = s->capacity == 0 ? 1 : 2 * s->capacity;
                int *new_data = realloc(s->data, new_cap * sizeof(int));
                if (new_data != NULL) {
                        s->data = new_data;
                        s->capacity = new_cap;
                } else {
                        return;
                }
        }
        s->data[s->size] = value;
        s->size++;
}

int stack_pop(Stack *s)
{
        assert(s != NULL && s->size > 0);
        s->size--;
        return (s->data[s->size]);
}

void stack_dispose(Stack *s)
{
        if (s == NULL) {
                return;
        }
        free(s->data);
        free(s);
}

bool stack_is_empty(Stack *s)
{
        assert(s != NULL);
        return (s->size == 0);
}

size_t stack_size(Stack *s)
{
        assert(s != NULL);
        return (s->size);
}
```

data/stack_array_based.c

Exercise : Assuming the cost of a memory allocation for a block of size M is $O(M)$, compute that using the doubling strategy for the array capacity in the previous implementation, the average cost for N successive pushes to the stack is only $O(1)$. Instead, realize that any strategy which would be based on a fixed additive capacity increase (rather than a multiplicative strategy) would lead to an $O(N)$ average cost. The factor 2 is not important though, any (possibly non integer) number $r > 1$ would accomplish the same goal.