# Green AI - ESPCI 2024: Practical work guide

This document presents instructions and questions regarding the practical work sessions. All the materials (slides and codes) can be recovered from the following GitHub repository: https://github.com/Deyht/green_ai_espci

This repository contains simplified versions of the materials from the PSL week on the same subject. The participants must provide a report where they answer the questions and describe their observations. The deadline for the report will be specified during the session.

All the produced codes and figures, as well as the report, if written numerically, must be uploaded as a single archive (.zip, .tar, ...) named after the participant's full name and practical work part (e.g., `surname_name_pw_report_partA.zip`) at the following link: https://share.obspm.fr/s/iiSEz5BDsKk2b5d

# Part A: Optimization and HPC

This first part tackles the subject of high-performance computing for the matrix multiplication operation that is extensively used in all modern AI models. Our objective is to find the most efficient way of implementing and using this specific operation to improve numerical efficiency and reduce the amount of energy required.

We will implement a classical matrix multiply operation of an $M \times K$ matrix A with a $K \times N$ matrix B to obtain an $M \times N$ matrix C. The content of an element of C is given by:

$$C(i,j) = \sum_k A(i,k) \times B(k,j) \tag{1}$$

The indices i, j, and k go through M, N, and K, respectively. We provide Python scripts and C source codes that contain different implementations of this operation in `green_ai_espci/opt_matmul/`.

## Python

1. In the `matmul.py` script, we provide the code that allocates the three matrices and initializes A and B to random values. We also provide two hand-written implementations of the matrix multiplication operation, `matmul_naive` and `matmul_numpy_sum`. For this problem, allocation and initialization times are negligible, so the compute time can be measured using the command:

   time python3 matmul.py

   Use this command to evaluate the compute time (user time) of the script running the `matmul_naive` function for a small matrix size (e.g., $M = N = K = 512$). This will be your reference time for computing the speedup of subsequent versions. Ensure that only one compute operation is uncommented when doing compute time measurements.

   ***Note:*** *Compute time predictions can vary due to other loads on the system. Always run your compute time estimations a few times to average the variability.*

2. Change the script to execute the second implementation `matmul_numpy_sum` and evaluate the compute time for the same matrix size. What is the speedup relative to `matmul_naive`?

3. The two handwritten functions can be compiled with Numba by adding the following lines:

```
from numba import jit
[...]
@jit(nopython=True, cache=True, fastmath=False)
def matmul_naive(A, B, M, N, K):
    [...]
```

In the provided script, the lines are present and should just be uncommented. Note that the compilation with jit adds time at the first execution, so execute at least twice before measuring the time. Measure the new time of both functions compiled with Numba. What is their respective speedup compared to your reference time? What do you observe? Which one is the fastest?

4. To have a better representation of the computing efficiency of different implementations, we can estimate the number of floating-point operations per second (FLOPS). This can be done by dividing the total number of "useful" operations done by the total time of computation. For the matrix multiplication operation where $M = N = K$, the number of operations is simply $N^3$. Using one of the two Numba-compiled implementations, draw a curve representing the performance in GFLOPS as a function of the problem size from 256 to 2048 by steps of 256. What do you observe? Try explaining the shape of the curve.

5. Now try comparing the compiled handmade implementations with optimized matrix multiplication operations in Python using the `@` operator and the `matmul` and `dot` functions from Numpy. We note that these functions are likely parallelized by default, so the comparison with the handwritten version is unfair. To force the execution on a single core, use the following line in the same terminal on which you run your code: **export** `OMP_NUM_THREADS=1`. Running on a single core, the `real` and `user` time returned by the `time` command should be identical.

   Which of the optimized implementations is the fastest for a large matrix size of $M = N = K = 2048$, and what is the typical speedup compared to your best Numba-compiled implementation? Draw the same performance curve for the fastest optimized operation. Since the computation is much faster, draw the curve up to $M = N = K = 4096$. How does it compare to the performance curve of the Numba-compiled hand-written implementation? What are your explanations for the shape of this new performance curve?

## Optimized C implementation

We provide the `matmul.c` source code that contains six different implementations for the matrix multiplication with an increasing level of optimization (from v1 to v6). The `main` function allocates the three matrices and initializes them to random values. The matrices are flattened as 1D arrays, and we adopt the column-major indexing formalism.

***Note:*** *For a matrix with M rows and N columns, i and j indexing the rows and columns, the* `C(i,j)` *element of matrix encoded in flattened column-major can be accessed with* `C[j*M+i]`*.*

We added timers to measure the elapsed time between two markers in the code. Encapsulating the call to a function between these markers allows us to measure the corresponding computation time. We also added a call to the optimized SGEMM matrix multiplication function from the OpenBLAS library. This allows us to verify that our custom implementations are correct regarding computed values for the C matrix, and it also provides an optimization performance goal. Like for optimized library in pythons, OpenBLAS is parallelized by default, so you must use the same export to force it to run on only one CPU core: **export** `OMP_NUM_THREADS=1`

The different implementations' performance gets progressively closer to the OpenBLAS implementation of the `sgemm` operation. The provided code can be compiled using:

gcc matmul.c −o matmul −lopenblas −lm

**The code must be recompiled before execution every time you change the source!**

6. The `matmul_v1` function corresponds to a naive 3-loop implementation. Using $M = N = K = 1920$, what is the typical execution time of this v1? How does it compare to the time of your previous best naive Numba-compiled Python function? What are the GFLOPS for both versions at this given matrix size?

7. Try adding optimization flags to your compilation line. Measure the time and estimate the GFLOPS when compiling with `-O1`, `-O2`, and `-O3`. What do you observe?

8. The `matmul_v2` function adds an accumulator to compute the sum. Measure the computation time and the GFLOPS of this v2 for the four possible optimization flag (none, `-O1`, `-O2`, `-O3`). What do you observe regarding the effect of the optimization flags? Why does the use of an accumulator improve the performance?

9. The `matmul_v3` function transposes the matrix A to have index continuity in the k-loop following:

$$C(i,j) = \sum_k A^T(k,i) \times B(k,j) \tag{2}$$

   Compiling this function with `-O3`, what are the resulting compute time and GFLOPS, and how does it compare to the v1 and v2? Now try to add the following additional optimization codes in the compilation line `-O3 -march=native -ffast-math -funroll-loops`. What are the effects on the compute time and GFLOPS, and why? Try dropping off one option and identify the one that has the strongest effect on performance.

10. The `matmul_v4` function uses the GCC vector data structure to perform an explicit SIMD vectorization of the matrix multiplication operation. For this, it creates new vectorized versions of A and B and fills them with the corresponding data. The A matrix is still transposed (but on the fly with the conversion to the vector type) to ensure memory continuity. The sums of products required to fill C are then made using the FMA instruction by doing the operations on vector data structures. Compile the code calling this version with all the previous optimization flags. What are the effects on the compute time and GFLOPS, and how does it compare to the optimized v3? Explain your observations.

11. Produce scaling curves for the v1 to v4 versions representing the GFLOPS as a function of the matrix sizes from 48 to 1920 using only multiples of 48 (the step size is left to your appreciation, but you should have at least about ten points) and considering that $M = N = K$. For each version, use the set of optimization flags that resulted in the best performances (some flags are detrimental to the v1 and v2 versions). What do you observe? How can you explain the performance dependency with the problem size? What could be the limitation of this v4 version?

12. The `matmul_v5` and `matmul_v6` functions rely on the same kernel function. This kernel works at the scale of the CPU registers and relies on the fact that the same data are required for several operations in a matrix multiplication operation. By storing a chunk of data as registers and reusing them as much as possible before loading new data, this kernel should maximize memory throughput and start to reach compute limitations of the CPU. The kernel works on vector data structure registers so it can use vectorized FMA operations. The details of this kernel's inner workings are given in the course. The `matmul_v5` version simply decomposes the problem in chunks that have the size of the kernel and calls it for all the possible K at a given kernel location in C. Using the default kernel configuration and compiling the code with all the previous optimization enabled, what are the computation time and GFLOPs of this v5 for $M = N = K = 1920$?

13. The optimal size of the kernel is dictated by the available number of CPU vector registers. Try modifying the size of the kernel and search if there is a better configuration than the default one. If yes, provide your best configuration and explain why it is better than the default one. Do this search twice, first for a large problem size (e.g., $M = N = K = 1920$) and then for a small problem size ($M = N = K = 512$). What do you observe? Is the optimal configuration different?

14. Draw the performance versus problem size curve in the same way as before and compare it to the curves obtained for the previous versions. What do you observe, and what is still limiting this v5 implementation?

15. Try to invert the order of the i-loop and j-loop in `matmul_v5` and measure the compute time and GFLOPS. Do you observe an effect, and if yes, in what direction and why? Is this observation in agreement with the limiting point you identified in the previous question?

16. The `matmul_v6` function also uses the vectorized register kernel, but this time, it also tries to reuse the data stored in the different L-cache levels of the CPU. For this, it defines blocks in A and B with a size that is a multiple of the kernel size and calls the kernel for each possible sub-region. Doing so implies that a single call to the kernel is insufficient to obtain the final value of a cell in C and that the contribution must be accumulated over all the possible block positions in the K dimension. After compiling with all the optimization options, estimate the compute time and the GFLOPS of this last version for $M = N = K = 1920$. What is the remaining relative performance difference between this version on the reference OpenBLAS version? Draw the performance versus problem size curve for the v6 function and OpenBLAS. What do you observe? (be sure to do the required export so OpenBLAS runs only on one CPU core).

## OpenMP parallelization

We provide an OpenMP parallelized version of the previous C code in `matmul_para.c`. This new version can be compiled with OpenMP support by simply adding `-fopenmp` to your compilation line. The number of threads on which the code will be executed is then controlled by the environmental variable `OMP_NUM_THREADS`. You can adjust this value and re-run a code without recompiling as it is queried at execution time. You can set this variable to a value of X by using the command: **export** `OMP_NUM_THREADS=X`

17. Our objective is to evaluate how well our custom implementation scales with the number of OpenMP Threads. For a fixed problem size of $M = N = K = 1920$, evaluate the compute time and GFLOPS for the non-parallel version and the parallel version but with only one thread. What do you observe and why?

18. Still using a fixed problem size, draw a curve representing the achieved speedup as a function of the number of OpenMP threads ranging from 1 to 16. The speedup for a code running with N threads is defined as the compute time for 1 thread divided by the compute time for N threads.

19. To analyze the previous curve, you must first identify the physical properties of the CPU in your system, which can be done using the `lscpu` command line. From this, identify the number of physical cores and logical threads in your system and indicate them in your report. With this additional information, describe and explain the shape of the previous speedup curve.

20. Using multiple CPU cores at the same time on a given problem will both lower the computation time and increase the power draw. However, due to the mutualization of several parts of the chip, the increase in power draw induced by the involvement of additional cores is usually not linear and much lower than the power draw of the first core. If we consider that using all P physical cores of the CPU induces a doubling of the power draw, what would be the ratio between the energy consumed for the computation using 1 thread and P thread for the parallelized version of the `matmul_v6`? Estimate the same ratio for the parallelized OpenBLAS SGEMM function. What do you observe?

    ***Note:*** *The total energy consumed by a given computation can be approximated by $E = \Delta P \times T$, with $E$ the energy in Joules, $\Delta P$ the increase in power draw compared to the system baseline in Watts, and $T$ the total time of the computation.*