PSL Week - Frugal AI 2025: Practical work guide

This document presents instructions and questions to guide the Frugal AI PSL week participants for the practical work sessions. All the materials (slides and codes) can be recovered from the following GitHub repository: github.com/Deyht/frugal_ai
Do not leave the file in the "Downloads" directory, as this will be automatically erased regularly. Either copy your work into "Documents" or into another personal storage space.

To validate the PSL Week, each participant must submit a practical report for evaluation. The questions presented here should be used as general guidelines, but each participant is free to explore each subtopic in greater or lesser detail than what is proposed. In all cases, it is essential to present your approach to each question or subject and to construct critical observations of the obtained results.

All the produced codes and figures, as well as the report, if written numerically, must be sent in the form of a compressed archive by the end of the last session to: david.cornu@obspm.fr or deposited in the following online repository: share.obspm.fr/s/9DGDKtf6TXpxyZL. The archive must be named so that the participant can be easily identified (e.g., `surname_name_frugal_ai.zip`).

Part I: Optimization and HPC

This first part is about high-performance computing with a focus on the **matrix-multiplication operation at the heart of AI/ML models**. The objective will be to implement and evaluate increasingly complex versions of this operation to maximize numerical efficiency and minimize energy consumption. We will notably evaluate the impact of the programming language, explore low-level optimization, CPU parallelization, and GPU acceleration.

We will implement a classical matrix multiply operation between an $M \times K$ matrix A and 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 note that we will mostly follow the triple loop algorithm and ignore possible alternatives, such as the Strassen algorithm. These sub $\mathcal{O}(n^3)$ implementations are mostly interesting for very-large matrices and cannot be optimized as deeply regarding the classical numerical system memory hierarchy.

This part of the practical work can be adapted to run on any system with only a few minor requirements (mostly Python with a few common packages, and a C compiler). However, we strongly recommend using the provided Linux-based systems in the room that have a unified hardware configuration and on which the practical work has been validated.

To ensure you have access to all the required packages, you should first run

```
conda activate simul
```

Starting codes for the different exercises are provided in the course archive at github.com/Deyht/frugal_ai.

This part was inspired by the matrix multiplication chapter of Algorithmica by Sergey Slotin.

System information and baseline energy measurement

1. Before proceeding, gather information about the system you will use. Assuming you run on a Linux-based system, the command `lscpu` will provide details about your system CPU. To interpret the results of this notebook, identify the number of cores/threads, the amount of L1/L2/L3 cache, and the supported instruction sets (notably check for AVX2 support). Some information might be missing, such as the distribution of cores in P and E cores for modern Intel CPUs, but it can be found by searching the CPU model name online. To get a more detailed view of the memory hierarchy of your system, you can run the `lstopo` command (requires the `hwloc` package).

2. Measure the baseline power draw of your system with a wattmeter (provided in the room). Discuss the limits of your baseline measurements regarding what peripherals are considered in the measurement. Test the variation in power draw for typical light use, such as browsing the web, watching a video, or editing a document. When measuring video playback power, try to measure the power draw with and without "video" hardware acceleration enabled. What do you observe?

   For the rest of this part, we will consider that the total energy consumed by a given computation can be approximated by $E = \Delta P \times T$, with E the energy in Joules, ΔP the increase in power draw compared to the baseline in Watts, and T the total time of the computation in seconds. To avoid continuously measuring the power-draw of all systems, we make the approximation that ΔP is constant regardless of the type of computation performed on a single core. Therefore, the consumed energy is considered proportional to the execution time. For a more complete view, please refer to the slides.

Python

3. Starting from the `matmul.py` script, implement a naive 3-loop version of this matrix multiplication in a function. The script already initializes the matrices A, B, and C at 32-bit precision (for future comparison with the C-language implementations). A test comparing the result (in terms of content of C) of your implementation with the `C=A@B` operation is also included so that you can verify your implementation. To evaluate the numerical efficiency of each implementation, we exclude allocation and initialization times and evaluate only the computation time by framing the function call within `time` commands. This first version will be highly unoptimized, so test your implementation with small matrix sizes first. Once your implementation produces the expected numerical results, report the execution time for a matrix size of $M = N = K = 512$, which will serve as a first baseline.

4. Now create a new matmul function where you replace the inner loop over K by a `Numpy` element-wise vectorial multiplication, replacing the k index with the slice ":" operator at the relevant index position for A and B. Also, replace the explicit sum loop with a `Numpy sum` function over the result vector. Measure the execution time of this new implementation using the same matrix size as before. What do you observe? What can explain the difference in computation time?

5. Python is an interpreted programming language, so conversion to machine code (compilation) is done on the fly on a per-line basis. However, libraries such as `Numba` can be used to compile compatible Python functions in a C-like fashion using a JIT (Just-In-Time compilation) formalism. To use `Numba` and compile a function, you can use:

```
from numba import jit

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

Measure the execution time of the compiled version of your two implementations and compare them to the non-compiled version. What is the speedup (time non-compiled over time compiled) for each version? Try explaining the speedup difference between the two matmul implementations.

Warning: functions with a JIT header are compiled the first time they are called, which will add an overhead time unrelated to the computation. To avoid measuring this, you should always call your compiled function once without the time beacons (e.g., for a small matrix), and then again to measure execution time.

6. To get a more complete view of the numerical efficiency of each implementation, we can estimate the number of floating-point operations per second or FLOPS. This quantity is obtained by dividing the total number of "useful" floating-point operations by the total time of computation. For matrix multiplication where $M = N = K$, the algorithmic numerical complexity is of the order of $\mathcal{O}(N^3)$, which implies that the computation time will grow cubically with the size of the matrices. From this, draw the evolution of the numerical efficiency in GFLOPS (10^9 FLOPS) as a function of the matrix size for your four implementations (naive and vectorized, non-compiled and compiled). You will draw the curve for sizes ranging from 128 to 1920, with a step size of 128. Feel free to automate the drawing of these curves for a selected size interval. Note that the largest size will require several minutes of computing time, so validate your approach in a smaller range or a larger step size first. Also, save your raw curve results so you don't have to rerun everything for future comparison graphs. The non-compiled naive version can take a very long time; you can choose to stop at 1024 for this specific curve.

7. Describe and try to interpret the different performance curves you obtained, addressing their differences and similarities.

8. To evaluate the remaining room for improvement, you will compare the best-performing previous implementation with various existing Python matmul implementations. For this, you will draw three new curves using the `@` operator, `numpy matmul`, and `numpy dot`. For this, you might need to declare functions with the same prototype as your previous implementation, but with the body replaced by the appropriate call. Since these implementations are fast, you can sample up to a size of 4096. Putting these new curves on the same graph as your previous implementations, estimate which one is the fastest, and what is the typical speedup at a matrix size of 1024 compared to your best Numba-compiled version? Also, discuss the global shape of these curves and explain why they are considered optimal. The common point of all these implementations is that they are written and optimized in a low-level language (C, or Fortran). Optimized matmul implementations are commonly found in BLAS (Basic Linear Algebra Subprograms) libraries such as OpenBLAS, which is the one used behind `numpy matmul` and `dot` functions (see `numpy linalg`).

   Warning: these implementations are likely to be parallelized, and we only want to estimate single-CPU performance for now. To be perfectly sure that you are only using one core, you can add the following lines at the very beginning of your Python script (before all other imports) (**COPY-PASTE DOES NOT WORK**):

   ```
   import os
   os.environ[”OMP_NUM_THREADS”] = ”1”
   os.environ[”OPENBLAS_NUM_THREADS”] = ”1”
   os.environ[”MKL_NUM_THREADS”] = ”1”
   os.environ[”NUMEXPR_NUM_THREADS”] = ”1”
   os.system(”export␣OMP_NUM_THREADS=1”)
   os.system(”export␣OPENBLAS_NUM_THREADS=1”)
   os.system(”export␣MKL_NUM_THREADS=1”)
   os.system(”export␣NUMEXPR_NUM_THREADS=1”)

   [...]
   ```

Compiled version in C

In this part, we will analyze multiple implementations of the matrix-multiplication operation in C, ranging from low to high levels of optimization and complexity. Even though we will primarily evaluate performances for square matrices, we will implement matrix multiplication so that the M, N, and K dimensions are exposed. Always verify that your custom implementations produce the correct results for non-square matrices.

While the C programming language supports multi-dimensional array indexing (but not vectorized operations), most implementations (and low-level C libraries) use a flattened matrix representation. This choice more explicitly reveals the data layout in system memory and allows us to identify potential cache-miss effects more easily. For our implementations, we will adopt column-major indexing as it is used in most low-level linear algebra libraries (inherited from Fortran). For a matrix with M rows and N columns, i and j indexing the rows and columns respectively, the $C(i, j)$ element of the flat matrix encoded in column-major can be accessed with $C[j \times M + i]$.

We provide a `matmul.c` source code file as a starting point for your custom C implementation. Our global objective will be to progressively align with the OpenBLAS implementation of the `sgemm` (Single precision GEneral Matrix Multiply) operation to achieve a similar level of performance. For this reason, the starter code includes the calculation using OpenBLAS and internally measures the compute time (excluding matrix allocation and initialization as before). Since OpenBLAS is parallelized by default, you need to manually set the number of threads to one so both its power draw and execution time are comparable to your sequential implementation. For this, you can call the following lines in the terminal in which you will execute the code:

```
export OMP_NUM_THREADS=1
export OPENBLAS_NUM_THREADS=1
```

Since we are working in plain C now, the source code must be recompiled manually whenever you change it. The compilation command using the GCC compiler for a given `source_file.c` is:

```
gcc matmul.c –o matmul –lopenblas –lm
```

The `-o` option specifies the executable name, and `-lopenblas -lm` links the precompiled OpenBLAS and Math libraries. Still, we arranged the C code so it can read the matrix size from a command-line parameter, allowing it to be changed without recompiling. To launch the compiled code for a matrix of size 1920, you can run:

```
./matmul 1920
```

To facilitate interaction with the C code, we provide an example Python wrapper script that compiles and runs the C executable and extracts the execution time it returns. You can build upon this wrapper to draw the performance curves in this C programming part.

Matmul v1: Naive triple nested loop

9. Implement the same 3-loop naive version of the matrix multiplication in C. What is the typical execution time of your code for a matrix size of 1920? Compare and discuss the compute performance difference with your previous naive Numba-compiled Python matmul function. What fraction of the OpenBLAS performance have we reached?

10. For now, we are using the compiler's default configuration. While this default behavior depends on the exact compiler used (e.g., the Intel compiler is more aggressive than GCC by default), most compilers have a relatively ”safe” default, with most available automatic optimizations turned off. Try adding optimization flags to your compilation line, namely `-O1`, `-O2`, or `-O3`, and evaluate the compute time, still for a matrix size of 1920. Which one provides the best speedup, and what is the current fraction of OpenBLAS performance?

11. Draw the performance curve of this new implementation as a function of the matrix size using the best optimization flag and compare it to your previous non-OpenBLAS Python implementations. What do you observe?

    The current limitation of this matmul v1 implementation stems from an inefficient use of the system's memory hierarchy. It means we are hitting a memory bandwidth threshold, indicating we are far from the theoretical peak compute capability of a single modern CPU core. In other words, we need to reduce and mutualize access to global memory and reuse data as much as possible once it is loaded from the RAM into the different levels of CPU cache and CPU registers. The following optimizations will all follow this objective.

Matmul v2: Inner loop accumulator

12. Create a new function with the same prototype to host a second custom matmul implementation. Here, we will use an "accumulator" trick. For this, declare a single float variable, initialize it at zero before starting the K-loop, and accumulate the result of all the individual products in this variable instead of C. Once the K-loop is finished, update the $C(i,j)$ cell once with the content of this accumulator variable. What is the resulting speedup and fraction of OpenBLAS performance for matrices of size 1920 using the different compiler optimization flags (none, `-O1`, `-O2`, `-O3`)? What do you observe when you compare these results to matmul v1?

13. Draw a new scaling curve for this matmul v2 using the best compiling setup. How does it compare to the previous one? Does it change the limiting factor?

Matmul v3: Transposed matrix and auto-vectorization

From a memory bandwidth perspective, the main limiting factor for our matmul v1 and v2 is that the required elements of one of the two matrices, A or B, are not continuous in system memory. In the inner loop, k addresses the elements of the i-th row of A and the elements of the j-th column of B. With our column-major convention, elements of B are continuous in memory, but those of A are not, causing a cache miss in the memory hierarchy.

14. Create a new matmul version that first transposes the matrix A to have index continuity in the loop following:

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

    Since this is a computation trick, the required time to perform the transposition must be included in the evaluated computation time (and therefore in the matmul v3 function itself), but it should be negligible compared to the matrix multiplication itself. Once you have a working version, re-explore the effect of the `-Ox` optimization levels on this new implementation for the same matrix size of 1920. What do you observe? What fraction of OpenBLAS's efficiency do we achieve now?

15. Produce a scaling curve for this new version using the best set of compilation options up to a size of 4096 (you can space the sampling for larger sizes) and compare it to the previous ones. What do you observe?

    Having restored memory continuity within the K-loop allows the compiler to automatically vectorize the computation, enabling multiple FMA (Fused Multiply-Add) operations to be done simultaneously on a single CPU core. The exact optimization flag level at which auto-vectorization arises will depend on the actual CPU model and GCC version. While this could be programmed explicitly, it would require much more programming work, and the performance results would be the same as with auto-vectorization. To gain even more performance, we can start adding some low-level optimization flags that are only relevant now that we have fixed the strong cache-miss issue for matmul v1 and v2.

16. We will explore the impact of three independent compiler flags:
    - `-march=native` exposes the exact CPU architecture and supported instruction set to the compiler (e.g., AVX2 support).
    - `-funroll-loops` tells the compiler that loops of known size can be expanded.
    - `-ffast-math` allows the compiler to aggressively rewrite and reorder math operations (e.g., `a * (b / c)` → `(a * b) / c`), and to approximate some computations.

    Try adding the different options to your compilation line, along with `-O3`, and see how they interact with each other. Explore all possible combinations and report the computation time for all of them for a matrix size of 1920. What do you observe? What is the fraction of OpenBLAS performances that you reach with the best combination?

17. Draw the performance curve of this matmul v3 version using the best set of compilation options up to a size of 4096 and compare it to the previous ones. Also include the OpenBLAS curve for comparison. Discuss the differences between the curves. What is the limiting factor now?

Matmul v4: Manual vectorization using GCC SIMD built-in vector support

While auto-vectorization is a powerful feature of the compiler, we will see how we can match the performance of matmul v3 with a manually vectorized implementation that uses built-in GCC vector data structures to abstract SIMD operations using the AVX2 instruction set. The following line declares a type vector composed of 32 Bytes (or 256 Bits, which is the default vector size for AVX2), which can store eight simple precision floating point values of 4 Bytes/32 bits each:

```
typedef float vec __attribute__ ((vector_size(32)));
```

With this, we can define variables and arrays of type `vec` that can be indexed to fill their content. Then, when performing operations directly between vectors, the compiler will translate the operation to the appropriate intrinsic vectorized FMA instruction for your CPU. While this datatype is specific to GCC, there are equivalent types in other compilers. Such low-level optimization becomes hardware-specific (CPU brand, architecture, supported instruction sets, etc.).

To help you start this new implementation, we provide a new `matmul_vect.c` source file that contains a few examples of how to manipulate the vector data types and guidelines to fill the matmul v4 implementation. This starter file also contains prototypes and guidelines for the following v5 and v6 implementations.

18. To implement this matmul v4 version, you must first create a vectorized version of A^T and B and fill them with the corresponding data. Then you do the classical loops over N and M but replace the inner accumulator by a `vec` variable. Then do a loop over the n_K vector you could create from the K dimension, and for each one perform an element-wise vector multiplication operation between the corresponding subvectors from A^T and B (automatically converted to an FMA instruction). Doing so is equivalent to parallelizing the K-loop across eight compute units, with each one storing its results in one of the eight accumulate vector values. Therefore, updating the $C(i, j)$ values requires summing all these vector elements before moving to the next cell in C.

19. Once you have a working implementation for matmul v4, estimate the compute time of this version for all optimization flags (none, `-O1`, `-O2`, `-O3`) at the same 1920 reference size and compare to the same results from matmul v3. What do you observe? Then compare the execution time using the following set of optimization `-O3 -march=native -ffast-math -funroll-loops` for both implementations? What do you observe? What fraction of OpenBLAS's efficiency do we achieve?

20. Draw the new scaling curve of this matmul v4 using the best set of compilation options up to a size of 4096 and discuss how it compares to matmul v3 and to the OpenBLAS curve.

Matmul v5: Optimum data reuse with in-register vectorized kernel

We will now change our approach for optimization. Instead of thinking from RAM to the cache, we will try to identify how we can fit the core operations of our matmul algorithm into the fastest cache level: the CPU registers. We will then scale up by optimizing the loading from progressively slower levels of CPU caches.

In the previous approach, we fixed a column of B (the outermost loop) to optimize its reuse, and then looped (the intermediate loop) over the rows of A. However, with this approach, we still need to reload all rows of A as many times as there are columns in B, leading to multiple loadings of the same data. In addition, as soon as either a row or a column exceeds a given cache level, multiple copies from lower levels (or RAM) will be required.

In contrast, the core of the new implementation will be a low-level kernel that loads a k_h subset of rows from A and a k_w subset of columns from B and computes all cross products involved in the corresponding sub-part of C. In other words, the kernel will perform the multiplication of the two submatrices $A(k_h, K)$ and $B(K, k_w)$ to fill a submatrix $C(k_h, k_w)$.

The first task of the kernel is to declare a 2D accumulator for the sub-region of C that it will fill. To optimize computation, this region must be explicitly declared as a set of vectors. Since we work in column-major order, we will vectorize along the k_h dimension, which must be a multiple of 8. The region should also be small enough to fit within the CPU's AVX2 register (refer to the slides). We recommend using $k_w = 6$ and $k_h = 16$, which will occupy 12 CPU vector registers. Considering that AVX2 declares 16 such registers, it leaves up to 4 vector variables for the computation.

The kernel's core will consist of three loops. Unlike previous implementations, the outer loop runs over K. For a given k value, we have k_h elements from the subset of A and k_w elements from the subset of B. The kernel role is to compute the element-wise product of these two 1D-arrays and add the result to the appropriate elements of C. To do so with a minimum number of vector registers, we can loop over the k_w dimension, selecting a single element and broadcasting its value to a β temporary vector. We then do a loop over the vectorized version of our k_h 1-D array, selecting one α vector at a time, and accumulating their dot product in the proper subpart of the already vectorized accumulator. Once all iterations of the three loops have been done, the accumulator values can be used to fill the corresponding subregion of C.

21. Implement the vectorized kernel we just presented with the help of the provided prototype in `matmul_vect.c` (also refer to the slide for illustrations). Then define a new matmul v5 function that calls this kernel in loops to cover all possible subregions of C corresponding to the kernel size. For this version to work correctly, the matrices' sizes must be multiples of the kernel size. Therefore, we recommend evaluating multiples of 48 of the matrix size (change the starting point and the step size when drawing scaling curves).

22. Once you have a working version, estimate the compute time for the reference size of 1920 using various compiler optimization options. Compare your result with the previous implementations. What do you observe? What fraction of OpenBLAS's efficiency do we achieve?

23. Using the best set of compiler options, draw a new scaling curve up to a size of 3840 for this matmul v5 version and discuss how it compares to the curves from previous versions. What is the limiting factor of this new version?

24. Try varying the kernel size, evaluating the number of required registers, and measuring the effect it has on compute time across at least two matrix sizes (small and large).

Matmul v6: Blocked matrix multiplication with register kernel

The previous version maximizes data reuse at the CPU register level. Still, the loop over the entire K dimension works well only if the submatrices $A(k_h, K)$ and $B(K, k_w)$ fit in lower levels of cache (typically L3/L2), reducing performance as the matrices grow. To ensure we work on a subproblem that fits in the caches, we can define a block of k indices of size b_K on which we run our kernel. By carefully selecting b_K so that the submatrices $A(k_h, b_K)$ and $B(b_K, k_w)$, we should be able to maximize cache efficiency. The only drawback is that a subset of k indices does not provide the complete solution for the corresponding C subregion. Therefore, the kernel must be modified to take the start and end positions of the current b_K block in the K dimension, and accumulate the results in C rather than setting the values. An additional inner loop over the b_K blocks must also be added to the function that calls the kernel.

While this formalism would already make better use of the caches, it is not yet fully aligned with the order of operations we performed to ensure that the cache hierarchy is fully optimized. Instead of relying on the kernel size to define the rows of A and the columns of B to load, we will adopt a blocking strategy that maximizes reuse of the loaded data and reduces overall load across all layers of the memory hierarchy. For this, we define a submatrix $C(b_M, b_N)$ where b_M corresponds to a subset of rows from A and b_N a subset of columns from B. These dimensions are chosen to be multiples of the kernel size, so independent kernel calls can process the corresponding C subregion. By combining this with the previous trick of working on a b_K-subset of K indices at a time, we can define a new global loop strategy to call the kernel. The principle is as follows: we start with a loop over the b_M subsets (set of A rows), then loop over the b_N subsets (set of B columns), and finally loop over the b_K subsets (sets of K indices). Doing so, we have selected submatrices from A, B, and C. Now, to apply the kernel to all the required positions, we add two loops: a first one over the number of kernels that fit in a b_M set, and a second one over the number of kernels that fit in a b_N set.

This strategy allows us to define submatrices of arbitrary size on which data reuse is maximized, relatively independently of kernel-size limitations. To achieve peak performance, the exact values of b_M, b_N, and b_K must be explored, keeping in mind that we want $b_K > b_N > b_M$ to respect the relative importance of reuse and that all submatrices should fit in the L2 cache at once for optimal reuse.

25. Based on the previous explanations and the slides, update the kernel and the matmul function v6 function starting from the provided code in `matmul_vect_start.c`. Once you have a working version, measure the computation time for the reference size of 1920 with all compiler optimization flags enabled. How does it compare to previous versions?

26. Explore different configurations for `l3`, `l2`, and `l1` (variables for b_M, b_N, and b_K), and report the corresponding compute time for a size of 1920. You can also explore changes in the kernel size, as it constrains the possible block sizes. Once you identify the best setup, express the compute time as a fraction of OpenBLAS's performance.

27. Using your most efficient setup, draw the performance curve of this matmul v6 implementation and compare it to the previous ones and to OpenBLAS' curve. What do you observe?

    We created an implementation that is very close to that in OpenBLAS. You can go check their source code (github.com/OpenMathLib/OpenBLAS) to observe that you will find many definitions of kernels of various sizes very similar to ours. The remaining few percent difference in performance between our implementation and OpenBLAS mostly lies in the automatic selection of the best kernel and blocking strategy for each matrix size, along with low-level optimizations specific to each major CPU architecture.

OpenMP parallelization

With the previous part, we got very close to the maximum possible performance of a matrix multiplication operation on a single CPU core. However, as discussed in the slides, the modern approach to improving CPU chip performance is to have multiple concurrent CPU cores on the same chip, sharing the same system memory. In this part, we want to explore how we can use the OpenMP API and paradigm to parallelize our matmul implementation.

The simplest approach for OpenMP parallelization is to distribute the work of a specific loop over multiple CPU cores (having one core work on a subset of the loop's elements). For this, OpenMP can be invoked as a "preprocessing directive" placed just before the loop to parallelize in the source code (see the slides for more details). In our case, we will use the following directive:

```
#pragma omp parallel for schedule(...) shared(...) private(...)
```

The schedule clause allows for specifying how the work should be distributed among the cores, the shared clause allow to specify which variable are shared between all cores, meaning that their value will be the same for all (this is the default for all variables declared OUTSIDE the parallel region), while the private clause specify which variable should be duplicated so each core has a different value, typically all "work" or intermediate variables (this is the default for all variables declared INSIDE the parallel region).

For the compiler to interpret the preprocessing directive, you must add the `-fopenmp` flag to your compilation line. Otherwise, the directive will be considered as a comment in your source code, and no parallelization will occur.

As we specified at the beginning of the section on C programming, the number of threads to create within the parallel regions is set outside the source code via the environment variable `OMP_NUM_THREADS`. These "logical" threads will spread across the physical CPU cores. Note that you could declare more logical threads than there are cores on your CPU, in which case multiple threads will try to run on the same core.

For a detailed view of OpenMP capabilities, we recommend reading the Idris course on OpenMP.

28. Using your most efficient version of matmul, try to add the OpenMP directive to one of the outer loops, taking care of which variables should be set as private. Try adding the parallelization directive to either the M or N loop, and measure the execution time using two threads for a reference size of 1920. What do you observe? Can you relate your observations to the fact that the L3 cache is shared for all CPU cores, while each core has its own L2 cache?

29. Try using the different scheduling strategies (`static`, `dynamic`, or `guided`) with different values of $\mathcal{N}$ using for example:

    ```
    #pragma omp parallel for schedule(strategy,N)
    ```

    What do you observe? What is the best setup?

30. We now want to evaluate how well our custom implementation scales with the number of OpenMP Threads. For a fixed matrix size of 1920, draw the evolution of the GFLOPS as a function of the number of threads.

31. To have a better idea of the scaling law, we also want to represent a speedup curve as a function of the number of threads. The speedup is computed as the compute time at a given number of threads over the compute time for a single thread. Before drawing the curve, compare the execution time of your matmul version with and without OpenMP for a single thread. What do you observe?

32. Now draw two speedup curves, one using the no-OpenMP time as a reference and another using the 1-thread OpenMP time as a reference. These curves should go from 1 thread to more threads than what is available in your CPU (get this information from the `lscpu` command). Discuss the shape of these curves regarding your understanding of your CPU model and configuration (provide the model of your CPU in the report).

    Using multiple CPU cores simultaneously on a given problem will reduce computation time but increase the power draw. However, because several parts of the CPU chip are shared, the increase in power draw from additional cores is usually not linear.

33. 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 a single thread against using P threads for your parallelized version? Estimate the same ratio for the parallelized OpenBLAS SGEMM function. What do you observe?

34. Verify the real evolution of the power draw of one of the systems provided in the room (or use the measurements from another group), and identify the sweet spot of optimal GFLOPS/Watt in terms of number of cores to use for these systems on this matrix multiplication operation.

GPU version with cuda

With the previous parts, we have achieved near-peak performance for matrix multiplication on a CPU chip and successfully distributed the work across multiple CPU cores. One of the key elements for achieving high performance was relying on the SIMD vectorization, notably through the AVX2 instruction set. However, while supporting such operations, CPUs are not dedicated to SIMD-like operations; instead, they support a wide range of applications. Therefore, computing chips explicitly designed for large vectorized problems should be particularly suited for matrix multiplication. This is notably the case for GPUs, which are composed of thousands of light cores with support for only a minimal set of instructions. This design makes them very efficient for handling large data volumes in a SIMD-like fashion. For a detailed view of GPU design and capabilities, have a look at the introduction section of the CUDA programming guide.

In this part, we want to evaluate the performance of the same matmul operation on an example GPU. For this, we provide a Jupyter Notebook (`GPU_matmul_demo.ipynb`) that should be run in a Google Colab environment for free access to a T4 GPU (Turing, 2560 CUDA cores, 16 GB RAM, peak FP32: 8.141 TFLOPS, and FP16: 65.13 TFLOPS, 70W peak consumption). This Notebook contains cells that write, compile, and benchmark different GPU matrix-matrix multiplication (matmul) implementations written in C/CUDA. The code and cells follow a similar scheme to our previous C matmul design with the Python wrapper.

35. The first part of the notebook evaluates the performance of the cuBLAS Nvidia library sgemm implementation. Evaluate the compute time for a matrix size of 4096 and compare it to the OpenBLAS single-core and multi-core CPU performance. What do you observe?

36. Draw the performance curve using cuBLAS with the T4, extending the range up to a matrix size of 16384. Extend your curve for the parallelized multi-core version of the OpenBLAS implementation for your CPU using the optimal number of threads. How do the two curves compare?

37. The Nvidia T4 has a peak power draw of 70W, which is very close to the power consumption of a mid-class desktop CPU. Try to identify the peak power consumption of your actual CPU from its spec sheet and estimate the optimal GFLOPS per watt for both the T4 GPU and your CPU (you can also assume your CPU consumes 70W if you do not find the actual value). Specify for which problem size the GFLOPS/W are optimal in both cases.

    The next part of the notebook implements a naive CUDA kernel to provide a practical demonstration of CUDA GPU programming. This version is similar to matmul v3 as it uses the accumulator trick and a transposed version of A. However, it does not directly expose the 3-loop structure. Using the SIMT formalism, it declares a CUDA kernel that is executed by all CUDA threads, each computing the value of a specific cell in C. This means that the kernel computes the current column and row of C from the flattened thread id ranging from 0 to (MxN) (excluded), performs the corresponding loop over K, and fills the corresponding C element. This approach prevents threads from competing for writing access over C.

38. Evaluate the performance of this naive CUDA kernel implementation for the reference size of 4096. What do you observe? Draw the performance curve for this implementation and compare it to the cuBLAS curve. What fraction of cuBLAS performance does it reach at best?

    The last part of the notebook implements a more advanced custom CUDA kernel that follows cuTLASS's matmul strategy. This implementation follows the same global strategy as before, aiming to maximize data reuse while accounting for the GPU's memory hierarchy. This implementation illustrates a few low-level optimization techniques for GPU programming.

39. Evaluate the performance of this last implementation for the reference 4096 size, then draw the performance curve in the same range that the one used for cuBLAS. Discuss the shape of the curves. What fraction of cuBLAS performance does it reach at best, and for what problem size?

Part II: CNN efficiency-based optimization

This second part addresses the optimization of Artificial Neural Network models for numerical efficiency and, therefore, energy efficiency.

For this exercise, we will use the ASIRRA (Animal Species Image Recognition for Restricting Access) dataset, which comprises 25000 labeled images of Cats and Dogs. The objective is to explore network structures, following the lecture's guidelines, to build a classification model that **maximizes classification accuracy under a given energy budget**. As discussed in the slides, both training and inference contribute to the model's environmental budget. To simplify the problem so it is compatible with a practical work session, we will assume the model's training cost is negligible and account only for electricity consumption during inference (which would match a long-lasting, broadly distributed model). We will also not factor in any infrastructure cost. Those are very strong approximations, and we invite the participants to refer back to the slide for a more complete view of the impact of ITC systems. At the end, for our practical case, we consider that the energy consumed is simply proportional to the execution time.

This part of the practical work will be carried out through Google Colab, granting each participant access to an Nvidia T4 GPU (Turing, 2560 CUDA cores, 16 GB RAM, peak FP32: 8.141 TFLOPS, and FP16: 65.13 TFLOPS, 70W peak consumption). Our energy budget will be based on the compute time required to process a set of 2048 test images, with a maximum budget of 300 ms. Your objective will be to design a model that achieves the highest possible classification accuracy (maximizing usefulness) while remaining under this budget. Note that a model reaching only 80% accuracy on this task (1 error out of 5 predictions) can barely be considered useful.

To help you get started, we provide a pre-written Jupyter Notebook] handling the data loading, model training, and inference based on a simple LeNET-5-like architecture. More information about the dataset and possible strategies for optimizing the model is provided directly in the notebook. The ANN model is built and trained using the custom CIANNA framework. You should be able to adapt the architecture with only minor adjustments to the appropriate notebook cells, taking inspiration from the provided example. You can refer to CIANNA's WIKI page for a complete framework description and access to the full list of supported layers and their configurations. You can also look at some example scripts provided in CIANNA's GitHub repository. Of course, you are also free to replace these parts with any other Deep Learning development framework you are familiar with (TensorFlow, PyTorch, etc.), as long as you correctly measure the inference time using a T4 GPU over Google Colab. Once you have a working model, you can enter its inference properties in the shared Google Sheet used as a dynamic leaderboard.

You can choose to tackle this practical work individually or by forming teams. Still, each participant must provide an individual report at the end of the week. In this part, the report format is free, but it must summarize your architecture research strategy, outline what worked and what did not, and provide possible explanations for these behaviors. Do not hesitate to provide detailed and technical information. You must also provide all the modified codes (or the modified notebook directly) so your inference result can be reproduced. You also have to provide the saving file for your best model. The saved files from other models can also be provided if you find them interesting. You can use hosting data services if the model is too large, or simply provide it with a USB Key before the end of the week.

Warning: Start by making a copy of the notebook inside your own Google Drive to avoid losing your work if the Colab session crashes. Also, Colab environments are temporary and not directly connected to your drive. If you want to save a trained model, you will need to download the corresponding `.dat` file from the `net_save/` repository in the Colab file explorer.