PSL Week - Green AI 2024: Practical work guide

This document presents instructions and questions that aim to guide the Green AI PSL week participants for the practical work sessions. All the materials (slides and codes) can be recovered from the following GitHub repository: https://github.com/Deyht/green_ai

To validate the PSL week, each participant must submit a practical report, which will then be evaluated. 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 we propose. 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 zip archive before the end of the last session to: david.cornu@obspm.fr

Part I: Optimization and HPC

This first part tackles the subject of high-performance computing for various classical applications that are extensively used in all modern AI models. Our objective is to find the most efficient way of implementing and using specific operations to improve numerical efficiency and reduce the amount of energy required to complete specific tasks.

System energy measurement

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

A - Sequential loop optimization

In this sub-part, we will optimize a function that fills a 3D data cube based on the content of another data cube following this equation.

$$B(i,j,k) = \sum_{x=-rad}^{rad} \sum_{y=-rad}^{rad} \sum_{z=-rad}^{rad} A(i+x, j+y, k+z) \tag{1}$$

The i, j, and k indexes represent the coordinates inside the cubes, and rad represents a radius inside which the content of A is summed and stored at a given location of B. We note that B is not defined for cells closer to an edge than the radius value, so it should be left empty. The corresponding codes are in `green_ai/opt_simple/`.

Python

1. Starting from the `loop_seq_start.py` script, implement a naive 6-loop version of this equation in a function. This first version will be strongly un-optimized, so start with small data cubes and a small summation radius (e.g., $v\_size = 64$ and $rad = 3$). For this problem, allocation and initialization times are negligible. The compute time can be measured using the simple command:

   time python3 loop_seq.py

   Use this command to evaluate the compute time of the script running your function.

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. The compute time can also increase when the system is hot. Choose either the hot or cold regime and use it for all your measurements. For the hot regime, rerun your computation several times and only take measurements after a few times. For the cold regime, use mostly small compute time and let a few tens of seconds between each measurement.*

2. Try changing the order of the loops. Does it change the compute time? For which loop order did you observe the best computation time and why?

3. Using a wattmeter, measure the increase in power draw you observe while this function is running. Ensure at least a few tens of seconds of computation for the power draw to stabilize. Compute the total energy consumed by your computation in Joules. Increase the size of v_size and rad independently. What effect does it have on your computation time and the total energy required?

4. Estimate roughly the number of computations and memory read that must be done as a function of v_size and rad. Note that the number of writing operations scale with v_size^3. Draw two scaling curves by selecting a few values of v_size and rad. How does it scale in practice? What do you expect to be the limitation of your current 6-loop implementation?

5. Try replacing the 3 inner loops with an optimized Numpy sum operation. You can verify that you obtain the same result to validate your new implementation. What is the speedup you obtain? Draw new scaling curves with a few points and compare them to the previous ones.

6. Try to find another way of writing the function to speed up the computation (without compilation). Describe it here if you found one, and compare the scaling curves to the previous curves.

Wait for correction

---

7. We now provide a script with three different implementations for our region-sum function. The first two can be compiled with Numba by adding the following lines:

```
from numba import jit
[...]
@jit(nopython=True, cache=True, fastmath=False)
def loop_naive(v_size, rad, A, B):
    [...]
```

Adjust v_size and rad values to have at least a few tens of seconds of compute time with the compiled versions and estimate their respective speedup compared to the non-compiled ones. How does the compile version scale with v_size and rad? What is the power draw of the compiled versions, and how does it compare to the non-compiled version? What version consumes the less energy?

8. Compare the compiled version of one of the first two functions with the `loop_vectorial_radius` non-compiled one to establish which one is the fastest. Does the choice of v_size and rad value affect which one is the fastest and why? Write your observations.

Compiled version in C

9. Starting from the `loop_seq_start.c` source code, implement the same 6-loop version of the equation. The obtained code can be compiled and executed using the following commands:

```
gcc loop_seq.c -o loop_seq
time ./loop_seq
```

What is the execution time of your naive C function? **We are using a compiled language now, recompile every time you change the content of your source code !!!**

10. Select a value for *v_size* and *rad* for which the computation time is in tens of seconds and use it to compare the execution time of your new C version with the fastest Python function and estimate the speedup. What is the difference between the C and the Python versions in power draw?

11. Try to add `-O1` to your compilation line and re-run your code. What is the effect on the compute time? Do the same thing but with `-O2` and `-O3` independently. Try to remove the `-Ox` optimization but try adding the following compilation options instead, both individually and in combination: `-march=native`, `-fast-math`, and `-funroll-loops`. Try to combine these options with the different levels of `-Ox` optimization. Using the online documentation and the observed performances, can you identify which optimization is incompatible with `-O3` (if any) and discuss why? What is the speedup between our best-performing optimized compiled C version and the fastest Python function now?

12. Instead of updating the value of B directly in the inner loop. Try to declare a variable that you initialize to zero before starting the sub-cube *rad*-based loops and accumulate the sum in this variable instead. Fill the B cell with this value once at the end of the sub-cube loop before changing position in the cube. What is the effect of this change on the computation time, and why? Does the effect change regarding the choice of compilation optimization (none to `-O3`)?

13. Draw scaling curves for *v_size* and *rad* for the C version and the fastest Python version on the same graph. In there differences in the scaling? Try to sample a few relatively close points in the extreme regimes of *v_size*, very small (typically $v\_size < 32$) and very large (typically $v\_size > 512$) to evaluate if there are differences of behavior in the two implementations that depend on the amount of CPU cache on your system.

    Wait for correction

---

14. Compare the execution time of a similar code in Fortran. The Fortran code can be compiled identically by simply replacing `gcc` with `gfortran`. What is the compute time difference between C and Fortran on this problem? How does this difference scale with *v_size* and *rad*?

OpenMP parallelization

We now provide an OpenMP parallelized version of the previous C code in `loop_workshare.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 can then be 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`

15. Compiling this new code with OpenMP and executing it with only one thread, what is the effect on the compute time compared to the fully sequential version?

16. We now want to construct a speedup scaling curve by varying the number of execution threads. Firstly, select a *v_size* and *rad* setup that takes at least 1 minute to compute using the sequential code. You can then use the compute time of the 1-thread OpenMP compiled version as your reference to exclude the instruction overhead from the scaling measurement. Explore values of threads from 1 to 12 and draw the speedup curve. What do you observe?

17. You can search for your CPU specificities by using the command `lscpu` to obtain a model number and indications on the number of physical cores and logical threads of your CPU. You can also search for more information using the CPU model number. Considering that this algorithm should scale almost perfectly with the number of threads, what are the properties of your CPU that can explain your speedup curve? Provide your CPU model and specificities in your report and your observations for your curve.

18. Try to change the `schedule(dynamic, 1)` setup in the source code for a `schedule(static, N)` and draw a curve of the execution time as a function of N using the number of threads that you found to be the most efficient (still > 1). What do you observe, and how do you relate your observation to the expected behavior of the "schedule" clause?

19. What is the effect of CPU parallelization on the power draw for your system? Compare the total energy consumed in Joules for the same computation using 1 thread and 2 threads. Which one is the most energy efficient and why? Draw a new curve that shows the total energy consumed for the computation when you vary the number of threads from 1 to 12.

20. To go further, you can compare the execution time of the provided identical Fortran code. What is the compute time difference between C and Fortran on this problem? How does this difference scale with *v_size*, *rad*, and the number of threads?

Wait for correction

B - Matrix multiplication optimization

In this subpart, we will optimize the 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, following:

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

The indices i, j, and k go through M, N, and K, respectively. The codes are in `green_ai/opt_matmul/`.

Python

1. Starting from the `matmul_start.py` script, implement a naive 3-loop version of this matrix multiplication. This first version will be strongly un-optimized, so start with small matrices (e.f., $M = N = K = 256$). Allocation and initialization times are negligible for this problem, too, so it can be timed using the same `time` command as before. Use this command to evaluate the compute time using your function.

2. Measure the power draw of your system for this problem, does it change in comparison to what you hat for the sequential loop on a single core?

3. Try compiling your implementation using Numba. What is your typical speedup?

4. Try comparing your implementation with optimized matrix multiplication in Python, for example, using the `@` operator or the `matmul` and `dot` functions from Numpy. What is the fastest one, and what is the typical speedup compared to your Numba-compiled version?

Compiled version in C

We provide the `matmul_start.c` source code as a starting point for your custom C implementation. For this exercise, **we strongly encourage using flattened versions of the matrix using the col major formalism! All the corrections will be provided using this format.**

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

The objective of this exercise is to get progressively closer to the OpenBLAS implementation of the `sgemm` operation. For this reason, the starter code includes the calculation with OpenBLAS and internally measures the compute time independently for OpenBLAS and your custom implementation. Openblas being 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. The provided code can be compiled using: `gcc matmul.c -o matmul -lopenblas`

5. Implement the same 3-loop version of the matrix multiplication. What is the typical execution time of your code for a specific M,N,K setup? Compare the compute performance with your previous naive Numba-compiled Python matmul function.

6. Try the difference `-Ox` optimization levels and the additional optimization flag we have seen in the previous exercise and quantify the effect they have on your compute time.

7. Create a new version and add the same "accumulator" trick used in the previous exercise to have only one writing operation in C for the loop over K. What is the resulting speedup?

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

$$C(i,j) = \sum_k A^T(k,i) \times B(k,j) \tag{3}$$

   Using the col-major formalism, the column indexes are continuous in memory. What is the resulting speedup?

9. Re-explore the effect of the `-Ox` optimization levels on this new implementation. What do you observe?

10. Produce scaling curves for your different versions by varying the matrix sizes and considering that M=N=K. Start low and increase the sizes progressively up to approximately 2000 using a step size sufficiently small to have a few tens of points. Do two independent curves for your last version using `-O2 -fast-math -funroll-loops` and `-O3 -fast-math -funroll-loops`; what do you observe. Try adding `-march=native` in both cases, what do you observe? What could be the limitation of your current version?

11. Measure the power draw for this last version and compare it to the power draw when using the naive and optimized Python versions and your naive C version. Do you see any differences, and why? Exclude the OpenBLAS version when measuring power draws by commenting the corresponding line to measure only your custom implementation. Try measuring the power draw of the OpenBLAS version only and discuss the result.

Wait for correction

---

12. Starting from the `matmul_vect_start.c` source code, try to implement a new version that declares new vector versions of A and B and fills them with the actual values of A and B. Now use the vector structure for performing the inner loop over K. What is the resulting speedup compared to the previous version using `-O3 -fast-math -funroll-loops` for both? Draw the new scaling curve and discuss it.

13. Try to implement the presented vectorized kernel and implement a new version that simply calls this kernel in loops that go through all the possible regions of C corresponding to the kernel size. Try two possible external loop orderings to see the effect on the computation time. Starting with this version, we recommend using a multiple of 48 to define the M, N, and K matrix dimensions so it is a multiple of the kernel size. Draw a new scaling curve for this version and discuss how it compares to the curves from previous versions. Try to vary the size of the kernel in terms of allocated registers and measure its effect on the compute time for at least two matrix size regimes (small and large).

14. Finally, try to implement the complete "blocked" version of the matrix multiplication that uses the various CPU caches. Try to vary the size of the different blocks and measure the effect on the compute time. What is your speedup compared to the previous version? Draw the new scaling curve and discuss it. How does this version compare to the OpenBLAS implementation regarding compute time and power draw?

15. Using the OpenMP directives presented in the previous exercise, try parallelizing the outermost loop of each of your matrix multiply implementations. What is the typical scaling you observe when varying the number of cores for each of them? Especially for your last blocked-vectorized version, do you observe a similar scaling to the OpenBLAS parallelization? Try it for different matrix sizes, and also try to push the number of thread higher than the number of physical CPU cores to see if OpenBLAS handle hyperthreading better than us.

16. Like in the previous exercise, evaluate the effect of CPU parallelization on the power draw for the matrix multiplication algorithm. Draw the curve of the total energy consumed as a function of the number of threads and find the optimum number to minimize energy consumption for your system. Discuss this result.

    Wait for correction

GPU version with cuda

Matrix multiplication operations can be efficiently accelerated on GPU hardware. In this section, we present an Nvidia CUDA custom version of a GPU sgemm kernel, inspired from CuTLASS and following the formalism presented in the slides. We compare our version with the CuBLAS closed-source version. The corresponding scripts are provided in `gree_ai/opt_matmul/GPU`. If your system is equipped with a dedicated Nvidia GPU hand has cuda installed, the scripts can be compiled using the following line after adapting the script name and the `arc_sm` value to match the compute capability of your GPU model.

nvcc −O3 gemm_custom.cu −o gemm_custom −arch=sm_86 −lcublas −lcudart

We also provide a Google Colab notebook to test the scripts on distant virtual environment. When doing so, the power draw can be approximated by the theoretical TDP of the reserved GPU (likely a T4, with a TDP of 70W).

17. Using the provided `gemm_custom.cu` source code, measure the execution time of the CUDA GPU version for a matrix size that requires around one minute to run with the parallelized OpenBLAS CPU version. Estimate the speedup you obtain and draw the scaling curve for both the custom version and the CuBLAS implementation. If you use Colab, you might need to execute the scripts a few times in a row on the same setup to get rid of computing time variability.

18. Measure or estimate the energy consumption in Joules for the full computation of both the custom version and the CuBLAS implementation at a given size and compare them to the parallelized OpenBLAS CPU version. What can you say about GPU efficiency compared to CPU efficiency on the matrix multiplication task?

19. If your GPU is equipped with Tensor Core (all generations avec Volta, including T4), you can try to compile and run the provided `gemm_tc.cu` script. Compare the compute time between the FP32 and FP16 with FP32 accumulator version of CuBLAS gemmEX implementation. Evaluate the new energy consumption and compare it to the previous versions. Also, do you observe any drop in computing accuracy?

20. You can also try to modify the code to use FP16 computation with FP16 accumulator. This can be done by changing the alpha and beta scaling to their `half` variant and setting the compute type to `CUBLAS_COMPUTE_16F`. Do you observe any difference in computation time? Check if it was expected based on the spec sheet of your GPU. Do you observe any drop in computing accuracy?

Part II: CNN efficiency-based optimization

This second part tackles the subject of optimizing an Artificial Neural Network model through a metric that combines model accuracy, numerical efficiency, and model size.

For this, we will use the ASIRRA (Animal Species Image Recognition for Restricting Access) dataset that comprises 25000 labeled images of Cats and Dogs. We provide a reduced dataset version in the form of squared 256x256 RGB images in a single binary file. In this file, the first 12500 images are Cats, and the 12500 next are Dogs. The last 1024 images of each class are excluded from the training and constitute our reference test dataset. Your objective is to explore ANN structures following the guidelines from the lecture to build a classification model that maximizes the following score metric

$$S = \left(\frac{E_r}{E}\right)^{w_E} \times \left(\frac{T_r}{T}\right)^{w_T} \times \left(\frac{P_r}{P}\right)^{w_P} \tag{4}$$

where E is the classification error rate, T is the compute time, and P is the number of trained parameters of the model. The r indexed values represent the same quantities for a simple reference model for which the score result is 1.0. The three terms represent relative errors to this reference model, and the contribution of each part to the total score is weighted by the w_E, w_T, and w_P powers.

Our reference model is an architecture close to a LeNET 5 with small adjustments. The reference values for E_r, T_r, and P_r are provided with the scripts, and we set $w_E = 1$, $w_T = 0.95$, and $w_P = 0.1$. With these scaling factors, a model that improves the error rate by a factor of two while preserving the compute time and number of parameters will get a score of two. It goes almost the same way for improving the compute time by a factor of two while preserving the error rate. The number of parameters has a smaller effect but is here to prevent the appearance of highly parametric architectures.

Fully functional training and inference Python scripts are provided in `green_ai/opt_cnn/`. We recommend using the Google Colab notebook version of the scripts and running them using a T4 GPU, which was used to define the baseline compute time for our reference architecture. You can use another system to train your model, but inference time should be measured in Colab with a T4 for a fair comparison. In the `aux_fct.py` script, we provide several utility functions to download, load, visualize, and perform dynamic image augmentation. You might need to edit small parts of these functions to change aspects of the problem, like the input image size of the model (default 64x64) or the augmentation policy. Data can be visualized with the `pred_visual.py` script or with the corresponding cell in Colab.

We provide training and prediction scripts that make use of the CIANNA framework to construct the ANN model. You should be able to adapt the architecture of the model with only minor adjustments to the scripts as described in the lecture slides. You can refer to CIANNA's WIKI page for a complete framework description. 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. Once you have a working model, you can enter its inference properties in the shared Google Sheet used as a dynamical leaderboard.

You can choose to tackle this practical work individually or by forming a team. Still, each of you must provide an individual report at the end of the week. In this part, the report format is free, but it has to summarize your architecture research strategy, summarize your observation on 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 so your inference result can be reproduced. You also have to provide the save of your best model, and saves for other models if your find them interesting as long as they are small in size. 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.