Hardware for Signal Processing
TP 1–2

HSAINI Anas – REZZOUQA Mohamed

Part 1 : C++ Multi-threading (Wallet)

Context and objective

The goal of this part is to practice multi-threading in C++ using `std::thread` and synchronization using `std::mutex`. This is done with a simple example of a wallet, inspired by a video game context.

The wallet contains rubies that can be credited or debited *one by one*, with an animation speed of 10 rubies per second (so 100 ms per ruby). The main idea is to make this animation non-blocking, so the game does not freeze while the wallet is being updated.

Project organization

The C++ project is located in the folder `wallet_cpp`. The compilation and execution are done using CMake.

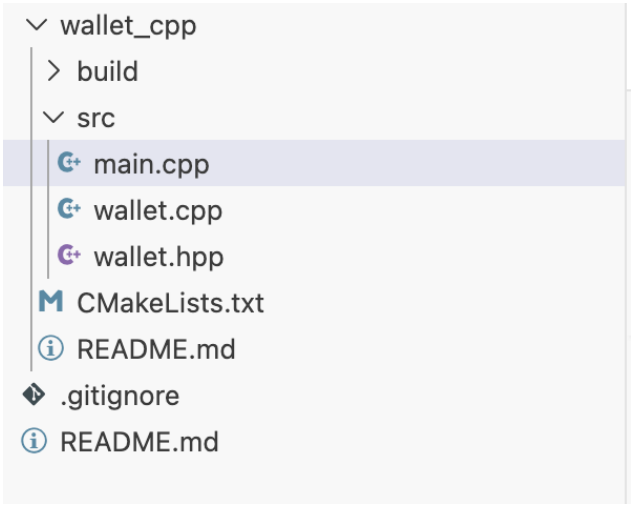

FIGURE 1 – Wallet_cpp project structure in VS Code.

```
(base) rezzouqa@Mickael wallet_cpp % cmake -S . -B build
cmake --build build

-- Configuring done (0.1s)
-- Generating done (0.0s)
-- Build files have been written to: /Users/rezzouqa/Documents/ENSEA/25_26/Hadware/TP_HSP
[100%] Built target wallet_demo
```

FIGURE 2 – Compilation and executable generation using CMake.

1 1.2 – Sequential version

In the sequential version, all operations are executed in the main thread. The methods `credit` and `debit` update the number of rubies one by one, with a delay of 100 ms between each update, in order to simulate a smooth animation.

Code extract

```
void Wallet::credit(unsigned int val) {
  for (unsigned int i = 0; i < val; ++i) {
    ++rupees;
    std::cout << "+1 rupee (rupees=" << rupees << ")\n";
    std::this_thread::sleep_for(std::chrono::milliseconds(100));
  }
}
```

```
(base) rezzouqa@Mickael wallet_cpp % ./build/wallet_demo seq

=== Mode: seq ===
Initial: 10
+1 rupee (rupees=11)
+1 rupee (rupees=12)
+1 rupee (rupees=13)
+1 rupee (rupees=14)
+1 rupee (rupees=15)
-1 rupee (rupees=14)
-1 rupee (rupees=13)
-1 rupee (rupees=12)
-1 rupee (rupees=11)
-1 rupee (rupees=10)
-1 rupee (rupees=9)
-1 rupee (rupees=8)
-1 rupee (rupees=7)
-1 rupee (rupees=6)
-1 rupee (rupees=5)
-1 rupee (rupees=4)
-1 rupee (rupees=3)
Final: 3
```

FIGURE 3 – Execution in sequential mode.

Question : Why would we want to parallelize this code in a game context ?

Answer :

In a video game, it is important that animations such as gaining or losing rubies do not block the rest of the program (inputs, AI, rendering, etc.). With a purely sequential version, the game cannot do anything else during the transaction. Using parallelism allows the game to continue running while the rubies animation is playing.

2 1.3 – Parallelization using threads

In this version, the functions `credit` and `debit` are executed in separate threads. This allows the main thread to continue its execution while the rubies are being animated.

Code extract

```
std::thread t1([&](){ w.credit(5); });
std::thread t2([&](){ w.debit(12); });

t1.join();
t2.join();
```

```
(base) rezzouqa@Mickael wallet_cpp % ./build/wallet_demo threads

=== Mode: threads ===
Initial: 10
+1 rupee (rupees=11)
-1 rupee (rupees=10)
+1 rupee (rupees=11)
-1 rupee (rupees=10)
+1 rupee (rupees=11)
-1 rupee (rupees=10)
-1 rupee (rupees=9)
+1 rupee (rupees=10)
-1 rupee (rupees=9)
+1 rupee (rupees=10)
-1 rupee (rupees=9)
-1 rupee (rupees=8)
-1 rupee (rupees=7)
-1 rupee (rupees=6)
-1 rupee (rupees=5)
-1 rupee (rupees=4)
-1 rupee (rupees=3)
Final: 3
```

FIGURE 4 – Multi-thread execution without synchronization.

Question : What problems can appear and what solutions can be considered ?

Answer :

When several threads access the same shared variable at the same time, inconsistencies can appear. This is called a race condition (more precisely, a data race). The operations `++rupees` and `-rupees` are not atomic, so updates can be lost depending on the thread scheduling.

In addition, the output on `std::cout` can be mixed, with interleaved lines, which makes the execution hard to read.

A classical solution is to protect accesses to the shared variable using a mutex (or atomic variables, but here the goal is to practice `std::mutex`).

3 1.4 – Synchronization with a mutex

To avoid uncontrolled concurrent accesses, a `std::mutex` is used to protect the critical sections where the variable `rupees` is read or modified.

Code extract

```
std::lock_guard<std::mutex> lock(m_rupees);
++rupees;
```

```
(base) rezzouqa@Mickael wallet_cpp % ./build/wallet_demo mutex

=== Mode: mutex ===
Initial: 10
+1 rupee (rupees=11)
-1 rupee (rupees=10)
+1 rupee (rupees=11)
-1 rupee (rupees=10)
-1 rupee (rupees=9)
+1 rupee (rupees=10)
-1 rupee (rupees=9)
+1 rupee (rupees=10)
+1 rupee (rupees=11)
-1 rupee (rupees=10)
-1 rupee (rupees=9)
-1 rupee (rupees=8)
-1 rupee (rupees=7)
-1 rupee (rupees=6)
-1 rupee (rupees=5)
-1 rupee (rupees=4)
-1 rupee (rupees=3)
Final: 3
```

FIGURE 5 – Multi-thread execution with synchronization.

Question : Are all problems solved with this approach ?

Answer :

The mutex guarantees data consistency by removing data races on `rupees`. However, from a game point of view, there is still a problem. If many debit threads are launched, the real balance is updated slowly during the animation.

But the game logic (for example, allowing or refusing a purchase) often needs an immediate decision. So memory consistency is fixed, but the gameplay logic is not fully correct yet.

4 1.5 – Instant wallet (virtual_rupees)

To solve the previous limitation, a second balance called `virtual_rupees` is introduced. It represents the logical balance of the player and is updated immediately during a transaction, while the real balance (`rupees`) is still animated progressively.

Code extract

```
bool Wallet::virtual_debit(unsigned int val) {
  {
    std::lock_guard<std::mutex> lock(m_virtual);
    if (virtual_rupees < val) return false;
    virtual_rupees -= val;
  }
  std::thread t([this, val](){ debit(val); });
  workers.push_back(std::move(t));
  return true;
}
```

```
(base) rezzouqa@Mickael wallet_cpp % ./build/wallet_demo virtual

=== Mode: virtual ===
Initial balance (virtual): 10

[SALE] virtual +5

[PURCHASE] virtual -12
+1 rupee (rupees=11)

(main) game keeps running...
-1 rupee (rupees=10)
+1 rupee (rupees=11)
-1 rupee (rupees=10)
+1 rupee (rupees=11)
-1 rupee (rupees=10)
+1 rupee (rupees=11)
-1 rupee (rupees=10)
+1 rupee (rupees=11)
-1 rupee (rupees=10)
-1 rupee (rupees=9)
-1 rupee (rupees=8)
-1 rupee (rupees=7)
-1 rupee (rupees=6)
-1 rupee (rupees=5)
-1 rupee (rupees=4)
-1 rupee (rupees=3)

Final balance (virtual): 3
```

FIGURE 6 – Instant wallet with non-blocking animation.

Question : Why use a virtual balance and run animations in separate threads ? What problems still remain ?

Answer :

The virtual balance allows us to immediately decide if a transaction is possible (for example, buying an item), which is essential for the game logic. The methods `credit` and `debit` are then only used for the visual animation, and they are executed in separate threads so the main program is not blocked.

However, a problem still exists : if many transactions are executed, a large number of threads can be created (one per operation), which can be costly in terms of performance. It is also important to correctly join all threads before destroying the wallet, otherwise the program may stop unexpectedly.

Conclusion – Part 1

This first part highlighted the main challenges of multi-threading and synchronization in C++. Starting from a simple sequential implementation, we progressively introduced threads, mutexes, and finally a design more adapted to a game context thanks to the use of an instant wallet with `virtual_rupees`.

Part 2 : SPD matrix vectorization with PyTorch

Objective

The goal of this part is to work with SPD matrices (Symmetric Positive Definite) using the PyTorch library. We implement several classical operations on these matrices, and then compare a simple implementation using Python loops with an optimized version using PyTorch vectorized operations.

The required operations are :

— vectorization of an SPD matrix (lower triangular part),
— reconstruction of the matrix from the vector,
— matrix square root computation,
— matrix logarithm computation.

Project organization

All the code for this part is located in the folder `spd_pytorch`.

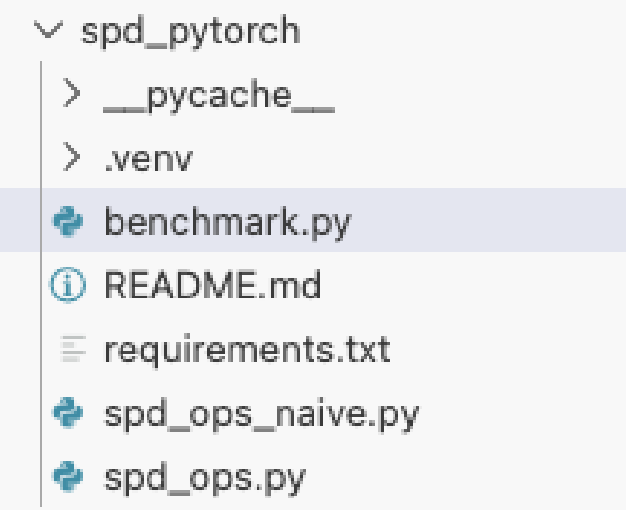

FIGURE 7 – File organization for the SPD PyTorch part.

The main files are :

— `spd_ops_naive.py` : simple implementations using Python loops,
— `spd_ops.py` : optimized versions using PyTorch operations,
— `benchmark.py` : script used to compare execution times.

5 2.1 – Naive implementation

First, the operations are implemented in a straightforward way, using Python loops over the matrix indices or over the batch dimension. This approach is easy to understand and to verify, but it becomes very inefficient when the batch size increases.

For example, to vectorize an SPD matrix, we explicitly browse the lower triangular part of the matrix using two nested loops.

```
for i in range(n):
  for j in range(i + 1):
    out2[b, k] = M2[b, i, j]
```

This type of code works correctly, but it is slow because Python is not efficient when handling many nested loops on large data.

6 2.2 – Optimized version

In a second step, the same operations are rewritten using PyTorch vectorized functions. The idea is to avoid Python loops as much as possible and let PyTorch handle the computations directly on tensors.

For example, for the vectorization, we directly use the lower triangular indices provided by PyTorch.

```
idx = torch.tril_indices(n, n)
M_vec = M[..., idx[0], idx[1]]
```

This approach allows the whole batch to be processed in a single operation, which is much more efficient, especially when computations are done on GPU or using the MPS backend.

7 2.3 – Benchmark and comparison

A script called `benchmark.py` is used to compare the execution times between the naive and the optimized versions. The tests are performed on a batch of 256 SPD matrices of size 32×32.

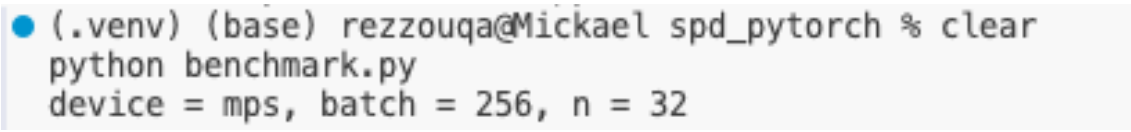

FIGURE 8 – Execution of the benchmark script.

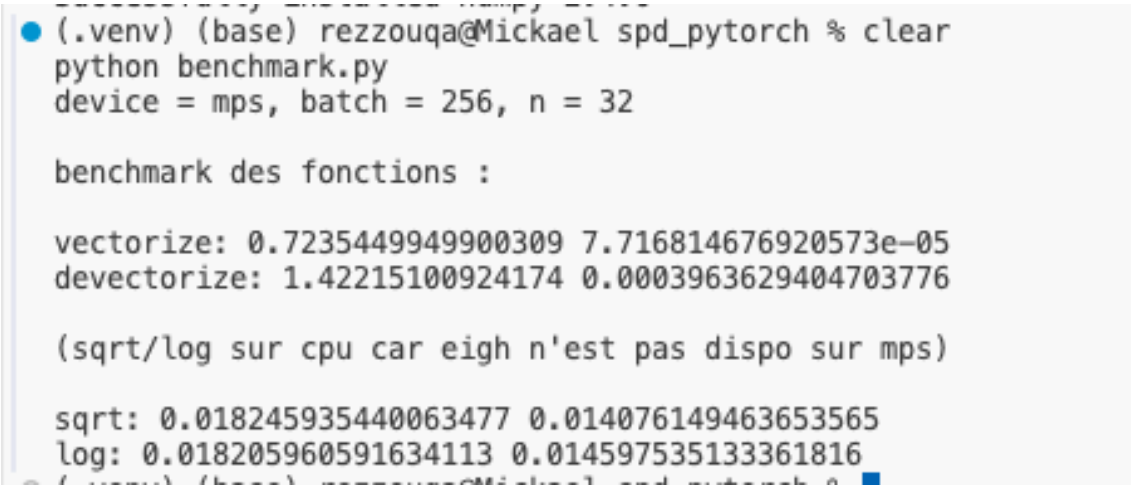

FIGURE 9 – Execution time comparison between naive and optimized versions.

Remark about execution on Mac M1

On our machine (Mac M1), the function `torch.linalg.eigh` is not available on the MPS backend. For this reason, the matrix square root and logarithm operations are executed on the CPU, while vectorization and reconstruction are still executed on MPS.

Results

We observe for example the following execution times :
— vectorize : about `0.65 s` (naive) versus `8.6e-05 s` (optimized),
— devectorize : about `1.28 s` (naive) versus `2.5e-04 s` (optimized),
— sqrt (CPU) : about `0.017 s` (naive) versus `0.013 s` (optimized),
— log (CPU) : about `0.017 s` (naive) versus `0.013 s` (optimized).

Analysis

The performance gains are very important for vectorization and reconstruction. This is mainly explained by the fact that Python loops are completely removed in the optimized version.

For the matrix square root and logarithm, the improvement is more limited. In this case, the dominant cost comes from the eigenvalue decomposition (`eigh`), which is expensive in both implementations.

Conclusion – Part 2

This part clearly shows the interest of vectorization with PyTorch. When working with batches of matrices, using tensor operations instead of Python loops allows a strong reduction of computation time.

Part 3 : Performance measurement (FLOPS)

Objective

In this part, we try to estimate the computation cost of the diffusion model implemented during Lab 4 of the TVI course. Unlike the previous parts, which were mainly focused on the behavior or the quality of the model, we focus here on the computation aspect, using the FLOPs (Floating Point Operations) metric, as seen in the course.

The goal is mainly to obtain an order of magnitude of the model cost, and not an exact hardware-level value.

Studied model

The analyzed model corresponds to the score network used in the diffusion model. It is a MLP-based network that takes as input a noisy point $x_t \in R^2$ and a diffusion time t, and predicts the noise $\hat{\epsilon}_\theta(x_t, t)$.

The network is mainly composed of linear layers and activation functions. These operations represent most of the computation cost.

Method used

To estimate the FLOPs, we use the `fvcore` library, and more precisely the `FlopCountAnalysis` class. The idea is simple : we run the model on dummy inputs (batch size of 128, 2D data), and the tool counts the number of floating point operations performed during a forward pass of the network.

This method allows us to quickly estimate the computation cost without manually counting all operations.

Obtained result

The analysis was done on CPU. For one forward pass of the model, we obtain the following value :

$$FLOPs_{forward} \approx 6.357 \times 10^6$$

This corresponds to the number of floating point operations required to perform a single forward propagation through the network.

```
(.venv) (base) rezzouqa@Mickael spd_pytorch % "/Users/rezzouqa/Documents/ENSEA/
/spd_pytorch/.venv/bin/python" "/Users/rezzouqa/Documents/ENSEA/25_26/Hadware/T
/flops_lab4.py"
=== Calcul des FLOPs (Lab 4 diffusion) ===
Unsupported operator aten::mul encountered 2 time(s)
Unsupported operator aten::exp encountered 1 time(s)
Unsupported operator aten::sin encountered 1 time(s)
Unsupported operator aten::cos encountered 1 time(s)
FLOPs pour un forward pass : 6.357e+06
Attention: fvcore ne compte pas toutes les ops (sin, cos, exp)
mais l ordre de grandeur reste correct pour analyser le cout du modele
```

FIGURE 10 – Terminal output during the FLOPs computation using fvcore.

During execution, `fvcore` displays warnings indicating that some operations are not counted (for example `sin`, `cos` or `exp`). These operations are part of the time embedding of the model and are not supported by the tool.

The obtained value is therefore an estimation. However, since most of the computation comes from the linear layers of the MLP, the order of magnitude remains relevant to analyze the model performance.

Estimation of the total training cost

From the FLOPs value obtained for one forward pass, we can estimate the total training computation cost of the model.

According to the course, a common approximation is :

$$TrainingFLOPs \approx FLOPs_{forward} \times N_{samples} \times N_{epochs} \times 3$$

The factor 3 roughly corresponds to the forward pass, the backward pass, and the gradient computation.

In our case, by taking for example $N_{samples} = 10\,000$ and $N_{epochs} = 100$, we obtain :

$$TrainingFLOPs \approx 6.357 \times 10^6 \times 10^4 \times 10^2 \times 3 \approx 1.9 \times 10^{13}$$

This value gives an order of magnitude of the total computation cost required to fully train the diffusion model.

Conclusion

This part provided a concrete estimation of the computation cost of the diffusion model. Even if the network is relatively simple, we already observe several million FLOPs per forward pass, and a total training cost that quickly becomes large. This highlights the importance of considering computation performance, in addition to model quality, especially when moving to more complex architectures.