Hardware for Signal Processing

12/2025

I. C++ Multi-threading

1. Sequential Version : Implementation and Execution

Objective

The objective of this part is to implement a sequential version of a wallet system that manages an amount of in-game currency (rupees). This version serves as a baseline before introducing parallel execution and synchronization issues.

Code Structure and Implementation

The program is structured into three separate files, following standard C++ design practices :
wallet.h : It defines the data member and the public interface of the class, without any implementation.

This separation ensures modularity and allows the class to be reused safely.

wallet.cpp : This file contains the implementation of the methods declared in wallet.h, Each function modifies or accesses the rupees variable in a strictly sequential way.

main.cpp : contains the global logic of the program, simulating a simple game scenario.

Steps performed:

- A wallet is initialized with 100 rupees.
- A sale is simulated by adding 50 rupees.
- A purchase is attempted and validated based on the available balance.
- The final amount is displayed.

The Wallet class is used as a simple data container, while all decision-making logic remains in the main function.

Compilation and Execution

We Compile each source file : links them together and generate the executable main.exe
Then we execute the main programm :

```
C:\Users\kadid\OneDrive\Bureau\Hardware>cl /EHsc main.cpp wallet.cpp
Compilateur d'optimisation Microsoft (R) C/C++ version 19.50.35721 pour x64
Copyright (C) Microsoft Corporation. Tous droits réservés.

main.cpp
wallet.cpp
Génération de code en cours...
Microsoft (R) Incremental Linker Version 14.50.35721.0
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:main.exe
main.obj
wallet.obj

C:\Users\kadid\OneDrive\Bureau\Hardware>.\main.exe
Solde initial : 100 rubis
Apres vente : 150 rubis
Achat effectue.
Solde final : 30 rubis
```

The program outputs:

Solde initial : 100 rubis

Apres vente : 150 rubis

Achat effectué.

Solde final : 30 rubis

Question

This version is **fully sequential**:

- All operations are executed in a single execution flow.
- No concurrent access to the wallet occurs.
- The result is deterministic and reproducible.

This implementation is correct in a single-threaded context but does not reflect real game engines where several actions may modify the wallet at the same time (rewards, purchases, network events, UI updates).

This motivates the need for a **parallel version**, which will highlight concurrency issues and justify the use of synchronization mechanisms such as mutexes.

2. Parallelisation of Credits and Debits

Objective

The objective of this part is to parallelize the calls to the credit and debit methods by executing them in separate threads, running concurrently with the main thread.
This allows us to observe the issues that arise when multiple threads access a shared resource without synchronization.

Code Structure and Implementation

Instead of calling **credit()** and **debit()** sequentially in the main function, these operations are now executed using C++ threads (**std::thread**).

Each thread performs an operation on the same Wallet instance, while the main thread continues its execution.

In this version, **t1** credits the wallet, **t2** debits the wallet. Both threads access the same shared variable rupees.

Execution

In this execution, the final balance is always correct and equal to the expected value.

```
C:\Users\kadid\OneDrive\Bureau\Hardware>.\main.exe
Initial balance: 100 rupees
Final balance: 120 rupees

C:\Users\kadid\OneDrive\Bureau\Hardware>.\main.exe
Initial balance: 100 rupees
Final balance: 120 rupees

C:\Users\kadid\OneDrive\Bureau\Hardware>.\main.exe
Initial balance: 100 rupees
Final balance: 120 rupees

C:\Users\kadid\OneDrive\Bureau\Hardware>.\main.exe
Initial balance: 100 rupees
Final balance: 120 rupees

C:\Users\kadid\OneDrive\Bureau\Hardware>
```

However, this does not guarantee the absence of concurrency issues.
The program remains unsafe in a parallel context despite the consistent observed result.

Question

When the credit and debit methods are executed in parallel using threads, several problems may occur during execution.

The main issue is a race condition caused by multiple threads accessing and modifying the shared variable rupees at the same time. This can lead to **inconsistent or incorrect balances**, because the order of execution of threads is not guaranteed.

Another problem is **non-deterministic behavior** : the program may produce different results for the same inputs, depending on how the operating system schedules the threads.

Then one solution is to protect access to the shared resource using a **mutex**, ensuring that only one thread can modify the wallet at a time.
Other approaches include redesigning the program to **avoid shared data**.

3. Multi-threading and Mutex

Objective

The goal is to make the Wallet class thread-safe by protecting concurrent accesses using a mutex.

Code Structure and Implementation

- In **Wallet.h**, we add a **mutex**.
- In **wallet.cpp**, we **protect critical sections**

Execution

In this execution, the final balance is now consistent and deterministic across executions.

This confirms that race conditions have been resolved.

Question

Race conditions and data inconsistencies have been resolved. However, the use of a mutex introduces **contention**, which may impact performance when many threads are active.

If many threads attempt to debit simultaneously, most of them may fail due to insufficient balance, and threads may spend time waiting for the mutex, leading to r**educed efficiency**.

Not completely solvable in current architecture. A more suitable design would involve request queuing or condition variables, rather than direct concurrent access to the wallet.

4. Instant Wallet

Objective

The objective of this part is to avoid the limitation of the previous mutex-based implementation by introducing the concept of an instant wallet.

Two balances are now maintained:

- rupees : represents the physical balance, updated progressively.
- virtual_rupees : represents the effective balance, assuming all pending transactions are completed.

This allows multiple transactions to be validated immediately, even if the physical updates are still in progress.

Code Structure and Implementation

- In **Wallet.h**, we add the new attributes and methods
- In **wallet.cpp**, we add the instant wallet logic.
- **virtual_rupees** is updated immediately and safely, allowing instant validation of transactions.
- Physical updates (credit / debit) are executed asynchronously in separate threads.
- **The balance returned to the user is always the virtual balance**, ensuring consistency from a logical point of view.

Execution

During execution, multiple debit operations can be validated immediately without blocking.

The displayed balance corresponds to the virtual balance and remains consistent from a logical point of view.

The program remains responsive even when several transactions are executed concurrently.

```
C:\Users\kadid\OneDrive\Bureau\Hardware>.\main.exe
Initial balance: 100
Final balance (virtual): 40

C:\Users\kadid\OneDrive\Bureau\Hardware>.\main.exe
Initial balance: 100
Final balance (virtual): 50

C:\Users\kadid\OneDrive\Bureau\Hardware>.\main.exe
Initial balance: 100
Final balance (virtual): 50

C:\Users\kadid\OneDrive\Bureau\Hardware>.\main.exe
Initial balance: 100
Final balance (virtual): 40

C:\Users\kadid\OneDrive\Bureau\Hardware>.\main.exe
Initial balance: 100
Final balance (virtual): 40

C:\Users\kadid\OneDrive\Bureau\Hardware>
```

The final virtual balance varies between executions, depending on the order in which concurrent transactions are processed.

Although the balance remains valid, the result is non-deterministic.

This shows that concurrency issues still exist despite the use of a virtual wallet.

Question

virtual_debit and **virtual_credit** must call debit and credit because the physical balance (**rupees**) still needs to be updated to reflect the real execution of transactions.

Running **debit** and **credit** in separate threads allows instant validation of operations without blocking the program.

However, the order of concurrent transactions remains non-deterministic, which can lead to different valid final balances.

II. Vectorizing maps in Pytorch

1. Implement the four operations

Objective

The goal of this part is to implement basic operations on symmetric positive definite (SPD) matrices using PyTorch, including vectorization, reconstruction, matrix square root, and matrix logarithm.

```
Original SPD matrix M:
 tensor([[7.2089, 4.2414, 1.9428],
        [4.2414, 3.4564, 0.3264],
        [1.9428, 0.3264, 1.3833]])

Vectorized M:
 tensor([7.2089, 4.2414, 3.4564, 1.9428, 0.3264, 1.3833])

Reconstructed M:
 tensor([[7.2089, 4.2414, 1.9428],
        [4.2414, 3.4564, 0.3264],
        [1.9428, 0.3264, 1.3833]])

Vectorize/Devectorize error: 0.0

||P @ P - M||: 2.3947793579282006e-06

||exp(L) - M||: 1.4404113244381733e-06
```

Exécution

Each operation is implemented using PyTorch tensor operations and eigendecomposition to ensure numerical correctness on SPD matrices.

Test

Simple tests confirm that reconstruction errors and numerical residuals are close to zero, validating the

2. Code optimization to improve computation time

Objective

The goal is to reduce computation time by avoiding Python loops and using batch-wise, vectorized PyTorch operations.

Exécution

The optimized implementations rely on **batch processing** and **vectorized tensor operations**.

This significantly reduces computation time compared to scalar or loop-based implementations.

Test

The numerical errors are identical for both the non-optimized and optimized implementations, confirming that optimization does not affect correctness.

However, the optimized batch-based versions significantly reduce computation time, especially for the matrix square root.

This demonstrates the efficiency gain brought by vectorization and batch processing in PyTorch.

```
SQRT errors: 0.00023275657440535724 0.00023275657440535724
SQRT times : 0.09725260734558105 0.002006053924560547
LOG errors : 0.00032672297675162554 0.00032672297675162554
LOG times  : 0.013670921325683594 0.00924539566040039
```

III. Measures of performance in computing

We compute FLOPS of our diffusion model (Lab 4 of TIV) using different methods given in the lecture presentation slides.

1. Analytic computation

Method

The computational cost of the diffusion model is estimated analytically by summing the FLOPs of each linear layer in the noise prediction network. For each Linear layer, FLOPs are computed using the standard formula **2×in_features×out_features**, while activation functions and other operations are neglected due to their relatively low cost.

Result

```
FLOPs for noise prediction network (SimpleMLP):

Layer                          | In       | Out      | FLOPs
-------------------------------------------------------------
time_embed.0                   | 50       | 50       | 5,000
time_embed.2                   | 50       | 50       | 5,000
net.0                          | 52       | 128      | 13,312
net.2                          | 128      | 128      | 32,768
net.4                          | 128      | 2        | 512
-------------------------------------------------------------
TOTAL FLOPs  : 56,592
TOTAL GFLOPs : 0.000056592
```

The total number of FLOPs for a single forward pass is **56,592**, corresponding to approximately 5.66×10^{-5} GFLOPs, which confirms that the model is computationally lightweight.

2. Automatic FLOP Estimation

Method

The computational cost of the diffusion model is also estimated using an automatic profiling tool, which counts the number of multiply–accumulate operations (**MACs**) executed during a forward pass. Since one MAC corresponds to one multiplication and one addition, the number of FLOPs is obtained by multiplying the MAC count by two.

Result

The profiler reports **28,296 MACs**, which corresponds to **56,592 FLOPs**. This value is exactly identical to the analytical estimation obtained in Method 1, confirming the correctness of the manual computation. Minor differences may occur in general due to counting conventions, but in this case both methods provide consistent results.

```
MACs: 28296.0
FLOPs: 56592.0
```