

Optimisation de DME



HPC - Projet final

2025-06-10

Aubry Mangold et Samuel Roland

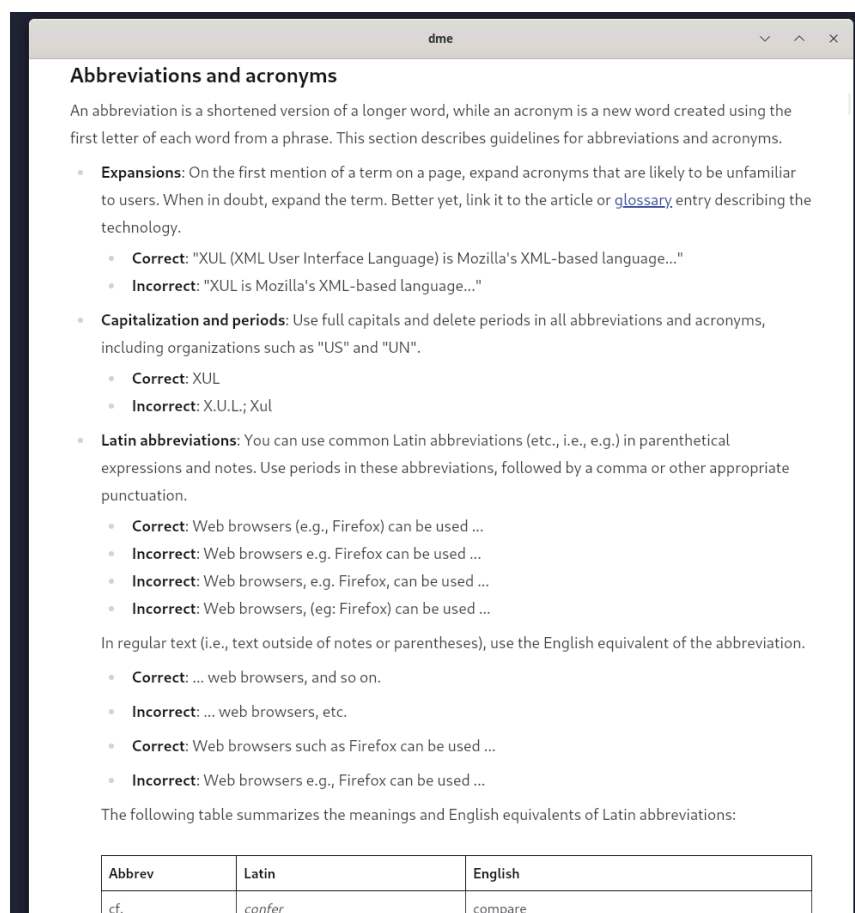
Table of Contents

Optimisation de DME	1
HPC - Projet final	1
Introduction	2
Rust	4
Tests	4
Benchmarking	5
Alternatives non retenues	5
Pull request	5
Optimisation de la colorisation syntaxique	6
Fonctionnement de Tree-Sitter	6
V1: programme initial et problème de performance	6
V2: mise en cache des grammaires	8
Optimisation de la recherche	9
Optimisation de l'installation d'une grammaire	10
Conclusion	11

Introduction

DME (Delightful Markdown Preview) a pour but de faciliter l'expérience autour de l'édition du Markdown, notamment pour la colorisation des snippets de code et la recherche de fichiers Markdown sur le disque pour faciliter switcher de fichiers directement depuis l'interface. Le preview se fait en convertissant le Markdown en un output HTML et CSS (pour le thème).

DME se découpe en une librairie `dme-core` dans le sous dossier `app/core` qui stocke toute la logique coeur de l'application. Nous allons nous concentrer uniquement sur cette partie pour ce projet d'optimisation. Cette librairie est ensuite utilisée par une application de bureau (avec le framework Tauri) pour bénéficier des fonctionnalités depuis une interface graphique. Tauri nous permet de faire des applications web dans des applications desktop en mettant ensemble VueJS (framework frontend) et Rust en backend.



Aperçu de DME ouvert sur un document Markdown, l'aperçu étant simplement une page HTML

Sample programs in c

File: baklava.c

```
#include "stdio.h"

int main (void)
{

    for (int i = 0; i < 10; i++)
    {
        printf (".*s", (10 - i), " ");
        printf (".*s", (i * 2 + 1), "*****");
        printf ("\n");
    }

    for (int i = 10; -1 < i; i--)
    {
        printf (".*s", (10 - i), " ");
        printf (".*s", (i * 2 + 1), "*****");
        printf ("\n");
    }

    return 0;

}
```

Un snippet de code C colorisé par Tree-Sitter et le thème Catpuccin Latte

Le projet DME utilise Tree-Sitter pour la colorisation syntaxique. Cette librairie qui est un générateur de parseur permet de créer un CST (Concrete Syntax Tree) qui décrit la structure d'un morceau de code sous forme d'un arbre de tokens avec un type associé. Ces tokens permettent ensuite notamment de faire de la colorisation syntaxique (syntax highlighting) dans les IDE (tel que Neovim, Zed, Helix et bientôt VSCode) basée sur des parseurs puissants plutôt que des systèmes de regexes. Cette colorisation avancée peut aussi être utilisée ailleurs, sur le web notamment mais demande l'intégration n'est pas triviale.

DME intègre un système de téléchargement des grammaires pour les langages à coloriser souhaités et charge ensuite dynamiquement une librairie partagée par grammaire depuis la librairie tree-sitter (crate Rust) embarquée.

Cette colorisation est généralement plus poussées et plaisante visuellement, c'est pour cette raison que cette complexité a été intégrée dans DME.

Un système de recherche permet ensuite de chercher un fichier Markdown sur son disque dur, en cherchant via son chemin ou n'importe quel titre d'un des documents. Le but est de pouvoir facilement switcher entre des fichiers Markdown sans devoir constamment ouvrir des dossiers à droite à gauche pour trouver le bon document. Cela ressemble au Ctrl+P dans VSCode qui donne un sélecteur fuzzy sur la liste des fichiers. Mais cela est étendu au contenu des titres et ne se restreint pas au dossier ouvert comme sur VSCode.



Aperçu de l'interface de recherche, montrant des résultats de recherches avec des titres qui ont matchés.

Rust

Le langage de programmation Rust est notablement différent du C à cause de son modèle mémoire. Il était par conséquent difficile de rapporter les concepts de bas niveau vus en cours à l'optimisation de DME. Nous avons donc dû nous adapter à la fois au modèle mémoire de Rust, mais aussi à la crate `tree-sitter` et aux autres bibliothèques adjacentes utilisées par le programme. Les différents types proposés par les bibliothèques sont parfois très différents de ce que l'on voit en C.

De manière générale, l'optimisation de DME s'est focalisée sur des concepts de plus haut niveau tels que la refactorisation de l'utilisation des objets au travers du programme ou encore la réorganisation du code pour éviter des appels inutiles.

Tests

Pour éviter les regressions et s'assurer de la stabilité une fois le code refactorisé, nous avons créé en plus des tests unitaires existants, une suite de tests d'intégration qui nous permettent de vérifier le comportement général du preview et de la recherche.

Nous avons développés ça dans `app/core/tests`, ils peuvent être lancés spécifiquement depuis ce dossier avec `cargo test`, une fois les tests unitaires passés ils se lancent. Voir `tests/large_preview.rs` et `tests/large_search.rs` si utile.

L'exemple suivant montre un test d'intégration qui vérifie que le preview d'un fichier Markdown avec des snippets de code fonctionne correctement. Il utilise la bibliothèque `comrak` pour parser le Markdown et `tree-sitter` pour coloriser les snippets de code.

```
#[test]
fn test_large_search_on_mdn_content_can_find_a_single_heading() {
    let repos = clone_mdn_content();
    let mut disk_search = DiskResearcher::new(repos.to_str().unwrap().to_string());
    disk_search.start();
    let search = "Array constructor with a single parameter";
    let results = disk_search.search(search, 20, None);
}
```

```

assert_eq!(
    results.len(),
    1,
    "Results should have only one result, contains\n{results:?}"
);
assert_eq!(results[0].title, Some(search.to_string()));
assert!(results[0].path.ends_with(
    "content/files/en-us/web/javascript/reference/global_objects/array/array/index.md"
));
}

```

Benchmarking

Comme à chaque laboratoire de HPC, nous avons commencé par développer un moyen de rapidement lancer des benchmarks de manière répétée et idéalement d'exporter les résultats.

Tout le code est toujours compilé en `--release` ce qui active les optimisations du compilateur Rust. On ne risque pas d'oublier de le faire car nous avons mis un check qui panic seulement en mode debug.

```
debug_assert!(false, "This benchmark system MUST be compiled and run in --release mode only to have best performances.");
```

Nous avons ensuite du configurer le profile de compilation pour `--release` pour qu'il garde les symboles de debug (équivalent de `-g`)

En rajoutant cette section au `Cargo.toml` de bench.

```

[profile.release]
debug = true

```

Pour lister les benchmark il suffit de lancer le programme bench normalement.

```

cd app/core/bench
> cargo run --release

```

Listing available benchmarks

- `preview_code` : Different code snippets numbers in various languages
- `preview_md` : Large Markdown file without code snippets

To execute a benchmark run: `cargo run --release -- bench <id>`

Nous avons choisi de définir les paramètres de benchmark comme suit :

- Benchmark du preview via la fonction `markdown_to_highlighted_html(path: &str) -> Result<Html, String>`
- `preview_code`: en utilisant une fonction utilitaire `generate_large_markdown_with_codes(30, 15)`; nous pouvons définir un maximum de 30 code par langage et un maximum de 15 langages. Ce ne sont que des limites maximum mais dans tous les cas le fichiers est bien grand. Le fichier généré dans `target/large-30.md` contient ainsi **117** morceaux de code dans 8 langages: `c go haskell java javascript lua rust scala`. Nous n'avons pas réussi à installer toutes les grammairs des langages proposés ou certaines n'ont pas de snippet disponible dans le repository utilisés.
- `preview_md`: nous avons choisi un grand fichier `files/en-us/mdn/writing_guidelines/writing_style_guide/index.md` sans aucun morceau de code. Comme la génération du HTML par Comrak est très rapide, nous avons fait un dupliqué de 30 fois son contenu et mis son résultat dans `target/big_markdown.md` ce qui fait 1.8M de lignes.

Alternatives non retenues

La crate `criterion.rs` est un système de benchmark qui permet de faire des mesures plus précises que les systèmes tels que `hyperfine`. Elle permet de benchmarker des fonctions Rust de manière fine et de faire des comparaisons entre les versions. Nous avons essayé de l'utiliser mais nous ne l'avons pas retenue car les benchmarks prenaient trop de temps pour être réalisés.

Pull request

La pull request est disponible sur <https://github.com/samuelroland/dme/pull/10>

Optimisation de la colorisation syntaxique

Fonctionnement de Tree-Sitter

Tree-Sitter est une technologie de loin simple à appréhender, de nombreux d'éléments le composent. Si besoin d'avoir une vue générale du fonctionnement des étapes, voir le documents [docs.md](#) section Preview sur le repository.

En résumé, nous avons des grammaires qui définissent comment tokeniser un langage. Ces grammaires sont publiés sur des repository Git tel que [celui pour le css tree-sitter-css](#).

Elle contiennent un parseur généré en C (depuis la définition JavaScript ou JSON)

```
src> tree
├─ grammar.json
├─ node-types.json
├─ parser.c
├─ scanner.c
├─ tree_sitter
│   ├─ alloc.h
│   ├─ array.h
│   └─ parser.h
```

Il est ensuite possible de demander de le compiler en librairie partagée qui va être dynamiquement chargée au runtime au moment où on a besoin de coloriser un code d'un langage donné.

Voici les étapes par lesquels nous devons passer pour coloriser via la crate `tree-sitter` et `tree-sitter-highlight`, une fois une grammaire clonée et compilée.

- Créer un Loader qui définit le chemin final des librairies partagées
- Charger la grammaire désirée
- Créer un objet `HighlightConfiguration` à partir de la grammaire
- Construire un `TreeSitterHighlighter` qui encapsule la configuration et peut être utilisé par le renderer. Ce dernier passe ensuite pas un CST (Concrete Syntax Tree) pour coloriser les tokens et les transformer en HTML.

V1: programme initial et problème de performance

Nous avons utilisé `perf`, en lançant notre fonction via notre système de benchmark.

```
cd app/core/bench
cargo build --release
sudo perf record --call-graph dwarf -e cpu-cycles target/release/bench fn preview_md target/
large-15.md
hotspot perf.data
```

On observe qu'énormément de temps est passé dans l'initialisation de la `HighlightConfig`, partie qui va charger les fichiers de query (de requête sur l'arbre de tokens) qui permettent ensuite d'attribuer la bonne colorisation.

V2: mise en cache des grammaires

On souhaite donc créer un cache global des `TreeSitterHighlighter`, indexé par la langue. Dans notre fichier `large-30.md`, cela implique de créer cet objet seulement 8 fois (1 fois par language) au lieu de 117 (une fois par code snippet). Cette optimisation permettrait donc de réduire considérablement les appels aux fonctions liées à Tree-Sitter et, par conséquent, d'améliorer la vitesse d'exécution du programme.

Enormément de difficultés ont été rencontrées lors de la gestion de durées de vies des variables en Rust et des manipulations autour de la librairie `tree-sitter`. Nous avons dû commencer par préparer le code en refactorisant le module `TreeSitterHighlighter` pour ne plus dépendre sur une référence à une `HighlightConfig`, ce qui a permis en cascade de retirer le besoin d'avoir un objet `Loader` passé de l'extérieur du module. Cela nous a permis de simplifier ensuite `ComrakParser` puisque nous n'avions plus cette référence mutable sur un `Loader`. Nous avons pu ainsi mettre en place notre cache.

Dans le fichier `app/core/src/preview/comrak.rs`, une `hashmap` protégée par un `RwLock` (système de lecteurs/rédacteurs vu en PCO) est utilisée pour permettre d'avoir des lectures concurrentes mais qu'un seul rédacteur à la fois et exclusif par rapport aux lecteurs. L'intérêt de cette technique par rapport au `Mutex` plus classique est que la plupart du temps sera dépensé pour faire des lectures, et que seul le premier usage d'un language demandera une écriture. Le wrapper `Lazy` est simplement un moyen d'avoir une instance globale qui est générée à la première utilisation.

Note: le code n'a pas pu être parallélisé, mais l'usage de `RwLock` dès le départ est une bonne pratique pour permettre une parallélisation future sans devoir refactorer.

```
// Global TreeSitterHighlighter cache indexed by language
static TSH_CACHE: Lazy<RwLock<HashMap<String, TreeSitterHighlighter>>> =
    Lazy::new(|| RwLock::new(HashMap::new()));
```

Le snippet suivant montre un aperçu simplifié de l'utilisation de ce cache :

```
impl SyntaxHighlighterAdapter for ComrakParser {
    fn write_highlighted(
        &self,
        output: &mut dyn Write,
        maybe_lang: Option<&str>,
        code: &str,
    ) -> io::Result<()> {
        // Shortened: skip when no lang

        // Open the cache in write
        let mut cache = TSH_CACHE.write().unwrap();
        let highlighter = cache.get_mut(&owned_lang);

        match highlighter {
            // We have a highlighter in cache, just use it
            Some(h_cached) => {
                output.write_all(h_cached.highlight(code).as_string().as_bytes())?;
            }
            None => {
                // Otherwise create a new one, use it and save it in CACHE
                let new_h = TreeSitterHighlighter::new(&owned_lang, &self.manager);
                match new_h {
                    Ok(valid_new_h) => {
                        output.write_all(valid_new_h.highlight(code).as_string().as_bytes())?;
                        cache.insert(owned_lang, valid_new_h);
                        drop(cache);
                    }
                    Err(_) => {
                        output.write_all(code.as_bytes())?;
                    }
                }
            }
        }
    }
}
```



```
    }
}
```

Résultat du benchmark

- preview_code: 0.3144s
- preview_md: 0.0466s

Nous parlions précédemment du fait que nous voulions avoir principalement des lectures, hors ayant codé un peu rapidement basé sur des exemples, nous avons toujours lockés le RwLock en mode `.write()` ce qui empêchera une vraie parallélisation des lectures. Nous avons pu corriger en accédant à ce mode seulement pour y aller sauver un nouvel objet `TreeSitterHighlighter`. A noter que `TSH_CACHE.write()` se trouve après la création de l'objet et qu'on drop directement le guard pour relâcher l'exclusion mutuelle, dans le but de passer le moins de temps possible en section critique.

```
    if let Some(lang) = maybe_lang {
        let owned_lang = lang.to_owned();
-       let mut cache = TSH_CACHE.write().unwrap();
-       let highlighter = cache.get_mut(&owned_lang);
+       let cache = TSH_CACHE.read().unwrap();
+       let highlighter = cache.get(&owned_lang);

        match highlighter {
            // We have a highlighter in cache, just use it
@@ -96,6 +96,8 @@ impl SyntaxHighlighterAdapter for ComrakParser {
            match new_h {
                Ok(valid_new_h) => {
                    output.write_all(valid_new_h.highlight(code).as_string().as_bytes());
+                   drop(cache);
+                   let mut cache = TSH_CACHE.write().unwrap();
                    cache.insert(owned_lang, valid_new_h);
                    drop(cache);
                }
            }
        }
    }
}
```

Les benchmarks ne changent quasiment pas ce qui est normal puisque nous ne sommes pas encore en multithreadé. Running all benches

Résultat du benchmark

- preview_code: 0.3112s
- preview_md: 0.0464s

Optimisation de la recherche

Nous avons prévu d'optimiser la recherche mais le projet de PLM n'avait pas encore pu aller assez loin pour supporter une recherche stable et avec support de fuzzy matching et de tests solides (ce qui est difficile avec du fuzzy matching qui donne des résultats plus larges).

Nous avons quand même pu établir la mesure suivante qui nous permet de voir que la construction de l'index et la recherche de "abstraction" dans le repos de MDN, est déjà plutôt rapide.

```
> cargo run --release --bench general_keyword Compiling bench v0.1.0 (/home/sam/HEIG/year3/PLM/dme/app/core/bench) Finished release profile [optimized + debuginfo] target(s) in 0.78s Running target/release/bench bench general_keyword Running bench general_keyword Benchmark 1: target/release/bench fn general_keyword target/content Time (mean ± σ): 159.5 ms ± 10.3 ms [User: 164.0 ms, System: 184.3 ms] Range (min ... max): 144.6 ms ... 189.9 ms 40 runs
```

Si on ne lance que l'indexation et pas la recherche on obtient 125.5 ms ce qui montre que la recherche en elle-même n'est pas gourmande même avec le fuzzy matching mis en place.

En plus, un système de "streaming" des réponses permet d'envoyer les résultats au fur et à mesure, ce qui réduit encore le temps nécessaires avant l'apparition visuelle des premiers résultats.

Optimisation de l'installation d'une grammaire

Parmi nos grammaires, certaines sont plus ou moins lourdes. Hors, si on démarre DME et on installe immédiatement via l'interface graphique une douzaine de grammaires, cela peut prendre quelques minutes selon la quantité de grammars. Comme nous sommes impatient, nous pouvons lancer les git clone en parallèle, puisque Tree-Sitter et git travailleront dans des dossiers bien distincts (tant que toutes les grammaires concernent des langages différents).

Aperçu des poids des repository Git de qqes grammars 30M tree-sitter-bash 25M tree-sitter-c 155M tree-sitter-cpp 2.3M tree-sitter-css 545k tree-sitter-csv 2.2M tree-sitter-fish 13M tree-sitter-go 317M tree-sitter-haskell 914k tree-sitter-html 21M tree-sitter-java 50M tree-sitter-javascript 725k tree-sitter-json 1.4M tree-sitter-lua 2.9M tree-sitter-make 29M tree-sitter-markdown 106M tree-sitter-php 29M tree-sitter-python 1.1M tree-sitter-query 1.1M tree-sitter-regex 59M tree-sitter-rust 122M tree-sitter-scala 1.5M tree-sitter-toml 172M tree-sitter-typescript 1.3M tree-sitter-vue 2.0M tree-sitter-xml 4.0M tree-sitter-yaml

A première vu ça parait pas incroyable Tree-Sitter si c'est si lourd ces syntaxes... En fait, si on y regarde de plus près, ici avec celle de Rust cloné depuis <https://github.com/tree-sitter/tree-sitter-rust>, on voit que c'est le .git qui fait 49M la grande majorité du poids et que la librairie partagée rust.so ne fait que 1.1MB

```
> du -sh . 4.0K binding.gyp 64K bindings 8.0K Cargo.lock 4.0K Cargo.toml 4.0K CMakeLists.txt 4.0K
eslint.config.mjs 76K examples 4.0K go.mod 4.0K go.sum 40K grammar.js 4.0K LICENSE 4.0K Makefile 56K
package-lock.json 4.0K package.json 4.0K Package.resolved 4.0K Package.swift 4.0K pyproject.toml 12K
queries 4.0K README.md 1.1M rust.so 4.0K setup.py 6.6M src 160K test 4.0K tree-sitter.json
4.0K .editorconfig 49M .git 4.0K .gitattributes 36K .github 4.0K .gitignore
```

Comme tout l'historique et les branches séparées des branches principales ne sont pas du tout utile, ce qui compte c'est la dernière version sur main, nous pouvons demander à git de cloner seulement le dernier commit et seulement la branche principale.

Ainsi la commande basique suivante

```
git clone https://github.com/tree-sitter/tree-sitter-rust
```

devient celle-ci

```
git clone --depth 1 --single-branch https://github.com/tree-sitter/tree-sitter-rust
```

Notre wrapper de Git GitRepos a une méthode from_clone définie comme suit.

```
pub fn from_clone( git_clone_url: &str, base_directory: &PathBuf) -> Result<Self, String> {
```

Nous l'avons changé pour supporter 2 nouveaux paramètres et pouvoir optionnellement activer ces optimisations, notamment en précisant le nombre de commit.

```
pub fn from_clone(
    git_clone_url: &str,
    base_directory: &PathBuf,
    only_latest_commits: Option<usize>,
    single_branch: bool,
) -> Result<Self, String> {
```

A noter que le fait de pull un seul commit n'empêche les git pull suivant de fonctionner comme attendu, cela ne récupérera que les commits plus récents.

Testons sans l'optimisation

```
GitRepos::from_clone( git_repo_https_url, &self.final_grammars_folder, None, false);
```

Benchmark 1: target/release/bench fn grammar_install <https://github.com/tree-sitter/tree-sitter-rust> Time (abs ⇒): 11.483 s [User: 4.585 s, System: 0.512 s]

Mean: 11.4832

Et après optimisation ?? > cargo run --release -- bench grammar_install Compiling dme-core v0.1.0 (/home/sam/HEIG/year3/PLM/dme/app/core) Compiling bench v0.1.0 (/home/sam/HEIG/year3/PLM/dme/app/core/

bench) Finished release profile [optimized + debuginfo] target(s) in 2.18s Running target/release/bench
bench grammar_install Running bench grammar_install Benchmark 1: target/release/bench fn
grammar_install <https://github.com/tree-sitter/tree-sitter-rust> Time (abs \approx): 1.522 s [User: 0.717 s, System:
0.091 s]

Mean: 1.5225

todo: commenter que forcément dépend surtout du réseau et taille de la grammaire.

A noter que le benchmark Rust a définie de manière assez concise dans `bench/src/grammars.rs` de la façon suivante

```
pub fn install_grammar(args: Vec) { let mut manager = TreeSitterGrammarsManager::new().unwrap();  
manager.install(&args[0]).unwrap(); }  
  
pub fn grammar_install_bench() { let mut manager = TreeSitterGrammarsManager::new().unwrap();  
manager.delete("rust").unwrap(); let link = "https://github.com/tree-sitter/tree-sitter-rust";  
run_hyperfine("grammar_install", vec![link], 1); }
```

Conclusion

L'optimisation de DME a été un projet intéressant qui a permis de se pencher sur les défis de performances liées à un autre langage que le C. Le modèle mémoire de Rust aura été particulièrement difficile à appréhender, notamment à cause des références et lifetimes. De plus, il a été difficile de naviguer une grosse base de code car une bonne quantité de refactorisation était nécessaire avant de pouvoir faire des optimisations significatives.

Beaucoup de temps a été investi dans la mise en place de l'infrastructure de test et de benchmarking car les scripts et outils développés au fur et à mesure du semestre ne se portaient pas immédiatement à un nouveau langage. Bien que des outils natifs à Rust existent (`criterion.rs`), ces derniers ne se sont pas avérés adaptés à nos besoins. La conception d'un module de benchmark directement intégré au programme s'est révélée judicieuse car la performance du code a pu être évalué très facilement au fur et à mesure des modifications. Cette technique est une bonne alternative à l'utilisation de scripts Bash, mais est peut-être restreinte à des langages modernes tels que le Rust qui permettent d'interfacer beaucoup plus facilement avec l'OS que C. De plus, l'outil `perf`, qui fonctionne tout aussi bien avec Rust que C, a été d'une importance primordiale pour identifier les goulots d'étranglement dans le code.

La recherche de fichier n'a pas pu être optimisée parce que le projet de PLM n'était pas encore assez avancé pour supporter une recherche stable et avec des tests solides. Cette fonctionnalité reste cependant une bonne piste à approfondir dans le futur.

De manière générale, la transposition des techniques apprises pour le C vers le Rust a été intéressante. Bien que le cours se concentre sur le C, les langages modernes tels que le Rust gagnent en part de marché et seront donc de plus en plus utilisés pour des projets de développement dont les performances sont critiques. Ce projet a été une bonne occasion de se familiariser avec l'optimisation d'un autre langage.