ANALYSE D'UN NOUVEL ALGORITHME ET CONCEPTION D'UNE LIBRAIRIE EN C# PERMETTANT LA CONVERSION DES NOMBRES DÉCIMAUX EN NOMBRES À VIRGULE FLOTTANTE (IEEE 754)

MÉMOIRE PRÉSENTÉ COMME EXIGENCE PARTIELLE
DE LA MAÎTRISE EN TECHNOLOGIE DE L'INFORMATION

PAR
CARL VERRET

FÉVRIER 2022

RÉSUMÉ

En programmation, l'utilisation des nombres à virgule flottante pour représenter les nombres réels est pratique courante. Le traitement de ces nombres implique des opérations coûteuses puisqu'il faut lire et convertir les données avec précision. Ces opérations peuvent représenter un goulot d'étranglement lors du traitement de données à haut volume, par exemple l'échange de données en format CSV (*comma-separated values*), JSON (*JavaScript Object Notation*), etc.

Une nouvelle approche algorithmique permettant d'accélérer la vitesse de lecture des nombres à virgule flottante a récemment été mise au point par Lemire (Lemire, 2021b). La performance de cet algorithme a été démontrée à l'aide du langage C++ dans une librairie nommée fast_float (Lemire, 2021a). Ce projet de maîtrise a pour but de vérifier le potentiel de cet algorithme avec le langage C#.

Dans une approche de programmation basée sur les essais, nous avons analysé et adapté la librairie fast_float en langage C#. Bien qu'elle ne soit pas aussi rapide que sa version programmée en C++, notre adaptation de l'algorithme est jusqu'à neuf fois plus rapide que la librairie standard du langage C#. Nous avons validé le comportement des fonctions et l'exactitude des opérations à l'aide des fichiers et des cas d'essais les plus utilisés dans le domaine du traitement des nombres.

La difficulté de ce projet réside à plusieurs niveaux. Le langage C# est un langage de haut niveau qui a d'abord été conçu dans une optique de convivialité et de modularité et non de performance. L'optimisation du code est une activité itérative complexe qui nécessite de multiples connaissances techniques : programmation avancée en C# et en C++, analyse du code *JIT*-assembler, analyse de la performance, etc. Notre librairie tire avantage des opérations vectorisées (technologie SIMD), particulièrement efficaces, mais sous-documentées pour le langage C#.

La démarche d'optimisation du code réalisée dans ce projet s'applique à tout programme écrit en langage C# où la performance représente un enjeu. Notre librairie permet d'améliorer considérablement la vitesse de lecture des nombres flottants. Nous avons soumis la mise en oeuvre décrite au sein de ce mémoire à Microsoft pour adoption au sein de sa plateforme .NET. Acceptée en février 2022 par les ingénieurs de Microsoft, notre contribution sera bientôt utilisée par des milliers de programmeurs et des millions d'utilisateurs de partout dans le monde dès que la version 7.0 du framework sera rendue publique.

ABSTRACT

In programming, it is common practice to use floating point numbers to represent real numbers. Processing these numbers involves costly operations since the data must be read and converted accurately. These operations can represent a bottleneck when processing high volume data, for example exchanging data in CSV (comma-separated values), JSON (JavaScript Object Notation) format, etc.

A new algorithmic approach to speed up the reading speed of floating point numbers has recently been developed by Lemire (Lemire, 2021b). The performance of his algorithm has been demonstrated using the C++ language in a library named fast_float (Lemire, 2021a). This master's project aims to verify the potential of this algorithm with the C# language.

In a test driven development approach, we analyzed and adapted the fast_float library in C# language. Although not as fast as its C++ version, our adaptation of the algorithm is up to nine times faster than the standard C# language library. We validated the behavior of functions and the correctness of operations using the most widely used files and test cases in the field of number processing.

The difficulty of this project relies on several levels. C# language is a high-level language that was first designed with ease of use and modularity in mind, not for performance. Code optimization is a complex iterative activity that requires multiple technical knowledge : advanced C# and C++ programming skills, analysis of JIT-assembler code, benchmarking tools, etc. Our library takes advantage of vectorized operations (SIMD technology), which are particularly efficient but under-documented for C#.

The code optimization process carried out in this project may applies to any program written in C# where performance is an issue. Our library considerably improves the reading speed of floating point numbers. We have submitted our implementation to Microsoft for adoption within its .NET platform. Accepted in February 2022 by Microsoft engineers, our contribution will soon be used by thousands of programmers and millions of users from all over the world as soon as version 7.0 of the .NET framework is released.

AVANT-PROPOS

Je tiens tout d'abord à remercier sincèrement mon professeur encadrant, M. Daniel Lemire, pour son aide tout au long de mes travaux. Avant d'entreprendre la maîtrise, j'ai eu la chance de suivre quelques cours avec lui et j'ai tout de suite apprécié ses enseignements. C'est en écoutant une de ses conférences que j'ai été intrigué et interpellé par le sujet de mon mémoire. N'ayant jamais eu l'occasion de travailler en optimisation du logiciel, il s'agissait pour moi d'un défi de taille : un chemin ardu et parsemé de doutes. Heureusement, M. Lemire est un professeur exceptionnellement engagé. Le soutien qu'il m'a offert fut à la hauteur du défi qui m'attendait. Il est rigoureux et ne ménage pas les encouragements. Lors d'une conversation, il m'avait décrit ses travaux comme étant de la programmation "très avancée". C'est exactement le créneau dans lequel il excelle. Je lui exprime ma gratitude pour m'avoir permis de travailler sur ce projet concret et valorisant.

Je remercie le personnel de l'Université TELUQ pour les services administratifs et académiques que j'ai reçus. En particulier Mmes Isabelle Marquis et Mélanie Samson du Service des études pour m'avoir épaulé avec efficacité et gentillesse à chaque occasion. L'enseignement que j'ai reçu fut actuel et diversifié. Le niveau de difficulté des cours était relevé et correspondait tout à fait à mes attentes. Je garderai un excellent souvenir de mon passage à votre établissement. En fin de parcours, j'ai eu le privilège de recevoir la bourse d'excellence pour études au 2e cycle. Je remercie l'Université TELUQ pour cette reconnaissance. Merci également aux membres du comité d'évaluation pour l'attention que vous avez porté à ce mémoire et pour les commentaires pertinents que vous m'avez transmis.

J'adresse mes remerciements à mon employeur, le Vérificateur général du Québec et plus particulièrement à mes supérieurs MM. Simon Gauvin et Jean-Pierre Fiset qui m'ont encouragé dans ce retour aux études. Votre généreux programme de soutien aux études m'a été précieux. J'ai eu la chance de travailler pendant de nombreuses années avec M. Gauvin et je le considère non seulement comme un excellent patron mais également comme un ami et un complice de belles réalisations.

J'exprime ma reconnaissance aux experts de la communauté GitHub et de chez Microsoft qui se sont intéressés à notre travail, en particulier MM. Günther Foidl, Egor Bogatov et Tanner Gooding pour leur contribution inestimable. Côtoyer ces experts fut une belle occasion pour moi d'apprendre de nouvelles connaissances de

programmation et d'optimisation. Malgré près de 25 années d'expérience dans le domaine de la programmation, il m'en reste encore beaucoup à apprendre.

Finalement, c'est à ma famille et à mes amis proches que je tiens à dire merci. D'abord à ma conjointe Mélanie Tardif, qui sait pertinemment l'effort que requièrent des études graduées pour avoir elle-même emprunté ce chemin avec brio. Je la remercie pour ses encouragements et sa compréhension pour toutes ces heures que j'aurai consacrées à ce travail. À mes fils Philippe et William qui m'ont vu pas mal occupé ces derniers mois ; je souhaite leur avoir démontré que l'effort mène aux résultats et que s'investir avec passion dans un projet est enrichissant. Merci à mes parents qui m'ont toujours encouragé au dépassement. Par leur propre exemple, ils m'ont montré que ce qui vaut la peine d'être fait mérite d'être bien fait. Je n'oublie pas mes amis Jean-François Daigle et Jean-François Gagné à qui j'ai parlé à répétition de ce projet qui me tenait tant à cœur. De quoi vais-je bien vous parler maintenant ? :)

Par un matin de l'automne 2020 m'est venue l'idée un peu folle d'un retour aux études. C'est en réponse à cette pandémie où toutes les activités parascolaires et sociales ont fait place au télétravail et au confinement que j'ai décidé d'investir tout ce temps qui serait le mien dans un projet constructif. Ayant toujours autant d'intérêt pour l'informatique et les études, je savais que "ça allait bien aller" !

Pour Philippe et William,

Sachez saisir le positif dans chaque situation.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES EXTRAITS DE CODE

CHAPITRE I

INTRODUCTION

La mission de nombreux systèmes informatiques repose sur une arithmétique impliquant les nombres réels. Ces opérations, dont l'efficience et l'exactitude s'avèrent cruciales, sont effectuées à l'aide d'une représentation établie en 1985. La norme IEEE-754 définit la représentation par les nombres à virgule flottante (également appelés nombres flottants). Cette norme est aujourd'hui reconnue et adoptée par tous les acteurs de l'industrie. Toutefois, puisqu'il s'agit d'une approximation, l'utilisation de cette norme requiert des opérations arithmétiques complexes lors de la lecture et de l'écriture des nombres afin d'en préserver l'exactitude et gérer les cas d'exception. Certains programmes manipulent une quantité considérable de nombres à virgule flottante. Par exemple, la transmission et le traitement à haut volume de données en format texte CSV (*Comma Separated Values*) et JSON (*JavaScript Object Notation*).

La conversion décimale des nombres flottants a fait l'objet de nombreuses recherches au cours des quelques 60 dernières années (Ashenhurst & Metropolis, 1959). La question est analysée dans les deux directions (lecture et écriture), l'objectif étant toujours le même : trouver l'algorithme le plus rapide pour lire et écrire ces nombres tout en conservant l'exactitude. Ce problème est toujours d'actualité : alors que le volume et le débit des données à traiter ne cessent d'augmenter,

peu d'innovations sont répertoriées depuis les années 1990 en ce qui concerne la méthode de calcul utilisée pour lire et convertir ces nombres.

En février 2021, Lemire a publié un article dans lequel il s'est intéressé au problème de la lecture et de la conversion des nombres décimaux en nombres à virgule flottante (Lemire, 2021b). Son article, qui présente l'algorithme qu'il a mis au point, est accompagné d'une librairie nommée fast_float. Écrite en langage C++, sa librairie offre des performances largement supérieures à toutes les solutions de remplacement existantes.

La majorité des langages de programmation modernes proposent une implémentation des nombres flottants sous la norme IEEE-754. C'est le cas du langage C#, développé et mis au point par Microsoft au cours des 20 dernières années. Suite à l'analyse du code de la librairie standard du langage C#, nous estimions qu'il serait possible d'obtenir des performances supérieures à l'algorithme en place. Nous avons voulu démontrer par ces travaux de maîtrise les gains en performance procurés par la nouvelle approche.

1.1 Objectifs et questions de recherche

L'objectif principal de ces travaux de recherche est de programmer en langage C# une adaptation de la librairie fast_float (écrite en langage C++) et d'effectuer une analyse détaillée de sa performance. Les travaux effectués se sont déroulés en plusieurs étapes :

- Comprendre l'algorithme fast_float et les enjeux liés aux nombres flottants ;
- Programmer, tester et optimiser les fonctions de la librairie ;
- Mesurer les gains en performance par rapport à la librairie standard offerte par Microsoft ;

— Comparer les performances en fonction du langage (C# vs C++) et de l'encodage des caractères (UTF-8 et UTF-16).

Nous avons répondu à ces questions :

1. Est-il possible d'atteindre des performances supérieures à ce qui est offert par la librairie standard tout en conservant l'exactitude des calculs de conversion des nombres flottants ?
2. Est-ce que les performances de librairie csFastFloat sont équivalentes à celles de la librairie fast_float originale ?
3. Quels sont les facteurs à considérer lors de l'optimisation de la performance d'un programme écrit en langage C# ?

L'intérêt de ces questions de recherche, outre l'amélioration significative de la performance de la librairie standard de Microsoft, est d'identifier les facteurs qui permettent au programmeur d'influencer positivement l'exécution du code par le compilateur CLR (*Common Language Runtime*). Le langage C# n'a pas été conçu selon les mêmes attentes que le langage C++ au niveau de la performance.

1.2 Motivation et pertinence du projet

Puisque la conversion des nombres décimaux en nombres flottants est une opération fréquente et coûteuse, elle peut représenter un goulot d'étranglement dans les cas de traitement à haut volume. Alors que la bande passante et les capacités de stockage ne cessent d'augmenter, il y a peu d'amélioration en ce qui a trait aux opérations arithmétiques de conversion. Réduire le temps de traitement de ces nombres procurerait un gain significatif de performance.

Outre la librairie standard de Microsoft, nous n'avons recensé aucune librairie spécialisée dans le calcul des nombres flottants. Nous avons vérifié le code source des principales librairies spécialisées dans la conversion des nombres flottants

(traitement de fichiers JSON, CSV) et chacune d'entre elles utilise la librairie standard.

La démarche d'optimisation mise en œuvre dans le cadre de ce mémoire peut s'avérer tout à fait appropriée pour tout autre projet de développement où la performance représente un enjeu. Le dernier chapitre présente une liste de recommandations visant l'optimisation des programmes écrits en langage C#.

1.3 Contribution et résultats obtenus

Les travaux d'adaptation effectués dans le cadre de ce mémoire ont permis de mettre en place la librairie csFastFloat (*C-Sharp Fast-Float*). Notre implémentation en langage C# de l'algorithme a été publiée en février 2021 sur la plateforme collaborative GitHub. Cette plateforme internationale permet aux concepteurs de logiciel de contribuer à de multiples projets de type logiciel libre (*open source*). Cette tendance est répandue et les chefs de file de l'industrie comme Microsoft publient sur cette plateforme le code source de leurs librairies. Nous avons également publié la librairie csFastFloat sur le site NuGet.org où sont partagées des centaines de librairies utilitaires. Depuis sa mise en ondes, elle a été téléchargée plus de 1 800 fois.

La performance de la librairie csFastFloat est largement supérieure à la librairie standard de Microsoft (jusqu'à neuf fois plus rapide). L'algorithme et nos résultats ont été portés à l'attention des spécialistes de chez Microsoft. Notre suggestion d'inclure cet algorithme dans une prochaine version de la librairie standard a été acceptée. Les travaux visant à incorporer l'algorithme fast_float sont en cours.

1.4 Défis inhérents au travail de recherche

L'informatique en tant que science comporte une composante théorique et une composante expérimentale. Ce projet s'inscrit dans l'algorithmique expérimentale : nous cherchons à faire le lien entre un algorithme et la performance de sa mise en oeuvre. En effet, il est possible pour un algorithme efficace d'avoir une performance décevante ou avantageuse lorsqu'il est mis en oeuvre au sein d'un système donné, et c'est souvent par l'expérience qu'un tel constat est effectué. Par ailleurs, notre projet comporte aussi une composante de recherche en ingénierie puisque nous comparons expérimentalement différentes solutions à un problème pratique important.

L'analyse de la performance et l'optimisation d'un programme informatique est un processus itératif ardu. Les outils de mesure à notre disposition, aussi sophistiqués soient-ils, demeurent un élément de complexité avec lequel il faut composer. Nous mesurons des opérations qui s'effectuent en nanosecondes dans un environnement sensible aux fluctuations.

La nature même de la tâche à accomplir demeure le plus grand défi. Puisque les langages C++ et C# possèdent une syntaxe similaire et plusieurs points en commun, passer d'un langage à l'autre pourrait sembler facile à première vue. Or, ce n'est pas le cas pour plusieurs raisons. D'abord, il faut maîtriser l'algorithme et son implémentation afin de pouvoir en conserver l'esprit. Même s'il paraît simple en soi, l'algorithme fast_float comporte un bon nombre d'opérations arithmétiques pour lesquelles l'exactitude doit être garantie. De plus, le code à adapter est hautement optimisé et fait usage de fonctionnalités qui n'ont pas d'équivalence en langage C#. Notre objectif était clair : il ne suffisait pas de proposer une adaptation sans faille, il fallait tirer le maximum des capacités de l'écosystème .NET. À cet effet, nous avons mis à profit une technologie nommée SIMD (*Single Instruction, Multiple Data*) qui permet d'effectuer des opérations (fonctions) dites "vectorisées" où l'on

traite simultanément les données en blocs. Ce concept, qui se prête parfaitement à l'analyse des chaînes de caractères, a également été employé par Lemire dans sa version C++ sous une forme légèrement différente. Comme nous l'expliquerons dans les sections suivantes, Lemire a utilisé une technique appelée SWAR (*SIMD Within A Register*) ne nécessitant pas l'implication de ces fonction spécialisées. L'emploi des fonctions SIMD vectorisées nécessite une transformation des algorithmes afin de traiter les données en blocs. Ce qui ajoute au défi, c'est que l'usage de ces fonctions est très peu répandu en langage C# et que la documentation est minimale.

1.5 Plan du mémoire

Ce mémoire est constitué en deux parties. La première, État de l'art, débute par un portrait des connaissances en lien avec les travaux de ce mémoire. Nous y détaillerons le concept des nombres à virgule flottante, le contexte dans lequel la norme IEEE-754 a été mise en place ainsi que les principaux articles publiés en lien avec le sujet. Un chapitre sera également consacré aux aspects techniques des langages de programmation utilisés ainsi qu'à la technologie des tests unitaires. La seconde partie combine les différents éléments de contribution. Nous décrirons d'abord l'algorithme fast_float et son implémentation en langage C++. Par la suite seront abordés les travaux d'adaptation et de conversion vers le langage C#. Nous présenterons les résultats obtenus et en ferons l'analyse détaillée. Finalement, nous dresserons une liste de recommandations favorisant l'atteinte d'une performance élevée en langage C#.

Première partie

État de l'art

CHAPITRE II

PORTRAIT DES CONNAISSANCES

Une grande partie des travaux du mémoire consiste en la lecture et l'analyse de données exprimées en nombres décimaux afin de les convertir en nombres à virgule flottante. Dans ce chapitre, nous dresserons le portrait des connaissances en lien avec ce sujet. Dans un premier temps, nous définirons cette représentation des nombres réels et passerons en revue les principaux éléments de la norme IEEE-754 ainsi que le contexte dans lequel elle a été établie. Cette norme, et les différentes révisions dont elle a bénéficié, est le résultat du travail de plusieurs experts. Nous résumerons les articles pertinents en lien avec la portion de la norme qui concerne la lecture et l'écriture des nombres flottants. Bien que par convention cette représentation des nombres réels est largement adoptée par l'industrie, d'autres formes ont été définies. Nous terminerons en dressant une liste de quelques autres représentations des nombres réels.

2.1 Représentation des nombres réels

Pour l'humain, la lecture et l'écriture d'un nombre réel (entier, décimal ou fraction) ne posent aucun problème. Comme nous avons tous appris à compter en base 10, nous concevons assez facilement les valeurs 23 et 0.15. Nous associons aux n positions numériques les valeurs $x \times 10^n$ pour la portion entière et $x \times 10^{-n}$ pour

0	0	0	4	$\cdot$	3	0	0	0
$\cdots$	100	10	1	$\cdot$	$\frac{1}{10}$	$\frac{1}{100}$	$\frac{1}{1000}$	$\cdots$

Figure 2.1 : Représentation en base 10

0	1	0	0	$\cdot$	0	1	0	0
$\cdots$	4	2	1	$\cdot$	$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$	$\cdots$

Figure 2.2 : Représentation en base 2

la partie décimale (figure 2.1). Ainsi, le nombre 4.3 se calculera : $4 \times 1 + 3 \times \frac{1}{10} + 0 \times \frac{1}{100} \ldots = 4.3$. Pour ce qui est des fractions, nous utilisons et comprenons la valeur $\frac{1}{3}$ sans la représenter de façon exacte. Au besoin, nous pouvons recourir à la notion de valeur périodique (par exemple, $0.\overline{33}$).

Dans la mémoire de l'ordinateur cela n'est pas aussi simple puisque les calculs mathématiques s'effectuent en base 2. Les valeurs sont calculées selon $x \times 2^n$ et $x \times 2^{-n}$ (figure 2.2). Les bases binaire et décimale ne sont pas commensurables (toutes les valeurs de la base 10 ne peuvent pas être représentées exactement par la base 2). Par exemple, la valeur du nombre 0.3 s'exprimera en binaire 0.01001100110 où la dernière portion "0110" sera périodique. Avec les séquences périodiques, le problème qui survient en est un de capacité de stockage. Que ce soit sur 32 ou 64 bits, il y une limite au nombre de 1 ou de 0 qu'il serait possible d'intégrer pour s'approcher de la valeur 0.3. Le tableau 2.1 présente une approximation du nombre 0.3 en base binaire en utilisant 12 chiffres après la virgule.

Tableau 2.1 : Représentation de la partie décimale d'un nombre en binaire

Position n	Valeur x	2^{-n}	$x \times 2^{-n}$
1	0	0,5	0
2	1	0,25	0,25
3	0	0,125	0
4	0	0,0625	0
5	1	0,03125	0,03125
6	1	0,015625	0,015625
7	0	0,0078125	0
8	0	0,00390625	0
9	1	0,001953125	0,001953125
10	1	0,000976563	0,000976563
11	0	0,000488281	0
12	0	0,000244141	0
			Total 0,299804688

2.2 Nombres à virgule flottante

L'arithmétique sur les nombres flottants assistée par l'ordinateur ne date pas d'hier. La première implémentation répertoriée serait l'ordinateur électromécanique Z3 mis au point par Zuse en 1941 (Wikipedia, 2021f). Dans les années qui ont suivi, cette arithmétique est devenue fréquente et pratiquement tous les concepteurs d'ordinateur proposaient leur propre implémentation. Par exemple, les premiers ordinateurs UNIVAC et IBM 704 supportaient les nombres flottants différemment (répartition des bits, etc.). Deux formats de nombres binaires ont été mis en place : fixe et flottant.

Les nombres à virgule fixe correspondent à la représentation entier/fraction à laquelle on a fait référence précédemment. On segmente la série de bits disponibles en deux sections. La première constitue la partie entière du nombre et la deuxième la partie fraction. L'attribution des bits pour l'une ou l'autre des parties aura un impact direct sur l'étendue et la précision qu'il est possible de couvrir avec ces

partie fixe					partie fraction			
0	1	1	0	//	0	1	1	0
16	4	2	1		1/2	1/4	1/8	1/16

Figure 2.3 : Répartition des bits au format fixe

mantisse					exposant			
0	1	1	0	//	0	1	1	0
16	4	2	1		8	4	2	1

Figure 2.4 : Répartition des bits au format flottant

nombres. Accorder plus de bits à la partie fraction augmentera la précision au détriment de la portée (*range*).

Les nombres binaires à virgule flottante utilisent la notation scientifique en base 2. En mémoire, la série de bits est divisée en deux. La première partie est affectée à la mantisse et la deuxième à l'exposant. Augmenter les bits de la mantisse accentuera la précision mais diminuera l'étendue de l'exposant. Pour certaines représentations, seule la partie fraction de la mantisse sera conservée en mémoire. Pour les nombres normalisés, la partie entière (qui possède la valeur 1) est implicite et sera considérée comme un bit caché (*implicit bit*). Les figures 2.3 et 2.4 illustrent une répartition des bits pour chaque format sans pour autant évoquer un standard précis.

2.3 Norme IEEE-754

Avant l'arrivée d'un standard, la répartition des bits, la détermination du signe et la détection des exceptions variaient d'un fabricant de microprocesseurs et d'un langage à l'autre procurant ainsi certains ennuis aux programmeurs. L'exécution du même code sur une autre plateforme pouvait donner un résultat différent. Les variations s'exprimaient à deux niveaux : matériel (processeurs) et logiciel (librairies comportant les routines de calcul). Généralement, l'arithmétique était confiée à certaines librairies logicielles spécialisées. Il était parfois possible d'ajouter, en option, un module physique de calcul arithmétique (*Floating Point Unit*).

Vers la fin des années 70, la concurrence entre les fabricants de microprocesseurs est féroce. Alors que chaque fabricant implémente l'arithmétique sur les nombres flottants à sa façon, quelques acteurs de l'industrie tentent de s'imposer comme leader (Intel, Motorola, Zilog et National semiconductors pour ne nommer que ceux-là). Au moment de développer le coprocesseur Intel 8087, les dirigeants de la compagnie Intel ont en tête l'objectif suivant : obtenir les mêmes résultats arithmétiques sur toutes les plateformes où le coprocesseur est utilisé. Ils confient cette tâche ambitieuse à Wiliam Kahan. Avec son souci du détail et sa forte personnalité, Kahan était tout à fait désigné pour mener à terme ce mandat. Rapidement, ses travaux attirent l'attention des autres fabricants et quelques rencontres de travail sont tenues pour tenter de mettre en place la solution la plus adéquate. À l'époque, le coût et la disponibilité de chaque transistor étaient déterminants. Aidé de Coonen (son étudiant de l'époque) et Stone (un professeur en visite à Berkeley), Kahan mettra en place ce qui sera la base du standard IEEE-754 (Wikipedia, 2021g ; Severance, 1998). On le surnomme d'ailleurs à cet effet le "père des nombres à virgule flottante". En 1981, il conclut ses travaux en publiant un article intitulé "Pourquoi avons-nous besoin d'un standard arithmétique pour les nombres à virgule flottante". (Kahan, 1981).

Adoptée en 1985 par le IEEE (*Institute of Electronics and Electrical Engineers*) (IEEE, 1985), la norme IEEE-754 a établi les règles à suivre pour la représentation, la manipulation (addition, soustraction, multiplication, division et racine carrée), la gestion des exceptions ainsi que la procédure d'arrondissement. L'établissement et l'application de cette norme se sont étalés sur plusieurs années. On rapporte quelques événements catastrophiques liés aux erreurs de calcul qui auraient pu être évités par l'application du standard. Les plus cités sont l'échec en 1991 du système de défense Patriot (n'ayant pas réussi à bloquer un missile Scud et causant la mort de 28 personnes), l'écrasement en 1996 de la fusée Ariane 5 ainsi que

la dérive d'un index boursier ayant duré 22 mois à la Bourse de Vancouver en 1983. Heureusement, ces règles sont désormais respectées par tous les fabricants de microprocesseurs modernes et implémentées dans la majorité des langages de programmation. Elles permettent de traiter avec précision les nombres réels très grands et très petits.

Représentation en mémoire

La norme IEEE-754 prévoit la répartition et l'usage de chacun des bits en fonction de la notation scientifique et des trois composantes de celle-ci : signe, mantisse et exposant en base 2 (tableau 2.2). Le bit affecté au signe du nombre permet de traiter les nombres positifs et négatifs. La mantisse représente la partie fraction des chiffres significatifs du nombre et la partie entière (assumée à la valeur 1 pour les nombres normalisés) est implicite. On qualifie cette partie de bit caché ou implicite puisqu'elle fait partie du calcul mais n'est pas conservée en mémoire. Pour l'exposant, puisqu'il est nécessaire de représenter à l'aide d'une variable non signée des valeurs positives et négatives, la stratégie utilisée est la suivante : décaler la valeur de l'exposant en y ajoutant un seuil. Tout chiffre au-delà du seuil représentera un exposant positif et vice versa. La valeur du seuil est fixée en fonction du nombre de bits consacrés à l'exposant. En 32 bits, puisqu'il y a 8 bits pour l'exposant, le seuil sera 127 ($2^8 - 1$). Pour le format 64 bits, où il y a 11 bits pour l'exposant, ce sera 1023 ($2^{11} - 1$). En anglais, on désigne cette valeur *biased exponent*. La plage et la précision varient en fonction de la taille en octets de la variable. La plupart des langages de programmation possèdent leurs propres types de données permettant de représenter les nombres flottants.

Tableau 2.2 : Représentation en mémoire d'un nombre flottant selon la norme IEEE-754

	Variable float (32 bits)	Variable double (64 bits)
Signe `+/-`	1 bit	1 bit
Mantisse (sans le bit implicite)	23 bits	52 bits
Exposant	8 bits	11 bits

Conversion en nombre binaire flottant

La conversion d'un nombre décimal en nombre binaire flottant s'effectue en quelques étapes simples. Cet exemple, calculé en fonction d'une variable sur 32 bits, illustre la séquence à effectuer :

1. Calculer le bit du signe (0 si positif) ;
2. Convertir la valeur décimale en binaire ;
3. Représenter le nombre binaire en notation scientifique ;
4. Ajuster l'exposant en fonction du seuil (par exemple $127 + \text{exposant}$) ;
5. Conserver sur 23 bits la partie fraction du nombre à titre de mantisse (par exemple 1.1010 donnera 1010).

Pour reconstituer la valeur décimale d'un nombre binaire normalisé, on utilisera la formule $(-1)^{\text{signe}} \times (1 + \text{mantisse}) \times 2^{(\text{exposant}-127)}$.

Opérations mathématiques

Pour l'addition et la soustraction, il faut d'abord représenter les deux nombres selon le même exposant. Suite à l'opération, on procédera aux arrondissements requis. Pour la multiplication (et la division), les mantisses seront multipliées (divisées) et les exposants additionnés (soustraits).

Gestion des exceptions

Certaines exceptions sont prévues pour tenir compte des nombres 0 et infini (positif, négatif) ainsi que la situation où la valeur n'est pas un nombre représentable (*NaN : not a number*) comme c'est le cas par exemple pour $\sqrt{-5}$ et $\frac{0}{0}$. Toute opération invalide retournera le résultat NaN. La norme couvre également les situations de débordement où le nombre est trop petit ou trop grand pour être représenté à l'aide du nombre de bits *(underflow et overflow)*.

Procédures d'arrondissement

Avec ce modèle de représentation, certaines valeurs et résultats d'opérations sur les nombres réels ne peuvent être exprimés avec exactitude. Par exemple, lorsque la valeur binaire d'un nombre s'exprime par une séquence périodique, il faut alors procéder à une coupure ainsi qu'à un arrondissement. Instaurée en 1985 puis révisée en 2008 et 2019, la norme IEEE-754 prévoit les procédures d'arrondissement suivantes pour une valeur x (Muller et al., 2010) :

- Arrondissement vers l'infini négatif : la valeur sera égale au plus grand nombre flottant (possiblement l'infini négatif) inférieur ou égal à x ;
- Arrondissement vers l'infini positif : la valeur sera égale au plus petit nombre flottant (possiblement l'infini positif) supérieur ou égal à x ;
- Arrondissement vers zéro (tronquer les données) : la valeur sera égale au plus grand nombre possédant une magnitude inférieure à la magnitude x. Ainsi, la valeur sera arrondie vers le haut pour les nombres négatifs et vers le bas pour les valeurs positives ;
- Arrondissement vers la valeur la plus près : la valeur sera égale au nombre flottant le plus près de x. Pour les cas où x est situé exactement entre deux

nombres flottants consécutifs l'option par défaut sera d'arrondir au nombre pour lequel la mantisse est un nombre pair (*round to nearest even*).

2.4 Articles pertinents

Cette section présente, en ordre chronologique, une sélection d'articles pertinents en lien avec ce mémoire. Ces articles, déterminants et complémentaires, portent sur la lecture et l'écriture des nombres flottants. À la fin de cette section, le tableau 2.4 résumera les différents articles en fonction des deux opérations (lecture et écriture). L'article du professeur Lemire et son algorithme fast_float seront présentés dans les chapitres suivants.

Au début des années 1980, Connen complète ses études sous la supervision de Kahan. Il publie un guide pour la mise en œuvre du standard IEEE-754 (Coonen, 1980) suivi d'une thèse de doctorat sur le même sujet. Son guide comporte deux parties. La première constitue une description des éléments d'arithmétique (formats, opérations, exceptions, arrondissements, débordements, etc.). La deuxième partie est une série de spécifications destinées à tout fabricant de processeurs désirant implémenter la norme IEEE-754. Dans sa thèse, Coonen dresse le portrait du contexte dans lequel cette norme a été développée.

Pendant des années, Steele et White se sont intéressés à la conversion décimale des nombres binaires (Steele & White, 1990). Ils ont étudié la question et mis en lumière plusieurs problèmes inhérents à la façon dont les nombres binaires étaient convertis. Leur article, paru dans la conférence PLDI de 1990, traite de la représentation décimale des nombres binaires flottants de façon à ne subir aucune perte d'information, ne produire aucune information non essentielle (chiffres superflus) et procéder correctement à l'arrondissement lorsque requis.

Dans l'optique de ne produire que le bon nombre de décimales (aucun *garbage*), ils ont établi le nombre maximal de chiffres à produire (noté N) pour convertir un nombre possédant (n) chiffres d'une base vers une autre sans perte de précision ($N = 2 + \lfloor n/\log_2 10 \rfloor$). Cette formule présume que les opérations d'arrondissement s'effectuent correctement. Ils proposent plusieurs algorithmes nommés "Dragon (1, 2, 3 et 4)". L'impact des travaux de Steele et White est considérable. Ils ont influencé les spécifications de conversion décimale des langages de programmation mis en place dans les années subséquentes (Java, C, JavaScript). Dans la même conférence, Clinger publie un article dans lequel il traite de la conversion des nombres décimaux en nombres binaires flottants (Clinger, 1990). Il explique que puisque les bases 2 et 10 sont incommensurables (il n'est pas possible de représenter exactement en base 2 tous les nombres en base 10), il faut s'en remettre à la meilleure approximation pour $w \times 10^q$ où w et q sont des entiers positifs. Il est à la recherche d'un algorithme efficient où l'approximation est la plus juste possible.

Son algorithme nommé Bellerophon (en référence au héros grec ayant battu "le dragon") procure une méthode rapide pour calculer la valeur d'un nombre flottant basée sur une table de puissances de 10 précalculées. Avec ces valeurs, qui peuvent être représentées exactement par des variables de type double (ou float), le calcul devient alors trivial. Il suffira de multiplier (ou diviser si l'exposant est négatif) la mantisse par 10 à la puissance de l'exposant. Cela implique toutefois une condition : l'exposant et la mantisse doivent être compris dans une certaine limite (tableau 2.3). Dans ce mémoire, nous désignerons ce calcul comme la méthode rapide de Clinger (*Clinger's Fast Path*). Avec l'algorithme Bellerophon, les nombres qui ne peuvent pas être traités par la voie rapide doivent passer par une solution un peu plus ardue. Il faut d'abord calculer une approximation pour ensuite basculer vers un second algorithme moins efficient qui converge vers la meilleure approximation possible. Les

Tableau 2.3 : Conditions requises pour la méthode rapide de Clinger

Conditions	float	double
Exposant minimal	-10	-22
Exposant maximal	10	22
Nombre de bits pour la mantisse	23	52

calculs de Clinger utilisent la précision arithmétique double étendue (Wikipedia, 2021a).

Un peu plus tard la même année, Gay publie un article où il vint raffiner le travail de Clinger (Gay, 1990) de deux façons. D'abord, sa solution permet une approximation rapide qui va au-delà des limites fixées par la méthode rapide de Clinger. De plus, elle ne nécessite qu'une arithmétique à précision double. Pour tout exposant entier supérieur à 22, il décompose la formule d'approximation ainsi : $(w \times 10^{q-22}) \times 10^{22}$ où q doit être tel à ce qu'il soit possible de représenter exactement la valeur 10^{q-22}.

En 1991, Golberg publie un ouvrage qui allait devenir une référence (Goldberg, 1991). Son article traite notamment de l'arrondissement des nombres flottants, du standard IEEE-754, des aspects systémiques (instructions, compilation et traitement des exceptions) et certains sujets spécifiques comme les erreurs liées à l'arrondissement et la conversion décimale des nombres binaires.

En 2010, Loitsch publie le résultat de ses travaux sur la conversion décimale des nombres flottants (Loitsch, 2010). Il y présente ses algorithmes nommés Grisù (en référence au dragon du dessin animé italien Grisù). Sa méthode de calcul permet de laisser tomber la précision double qui était requise précédemment par Steele, White et Gay. De façon simplifiée, l'idée est la suivante : tout nombre flottant où l'exposant est négatif peut s'exprimer par la forme $\frac{f_v}{2^{-e_v}}$. Il est possible de calculer la valeur de chaque chiffre (*digit*) du nombre v en déterminant un exposant décimal

Tableau 2.4 : Résumé des opérations couvertes par article

Référence	lecture	écriture
(Coonen, 1980)	oui	oui
(Steele & White, 1990)	non	oui
(Clinger, 1990)	oui	non
(Gay, 1990)	oui	non
(Goldberg, 1991)	oui	oui
(Loitsch, 2010)	non	oui
(Adams, 2018)	non	oui

t tel que $1 \leq \frac{f_v \times 10^t}{2^{-e_v}} < 10$. Le résultat entier de la division constitue le premier chiffre. Il suffit ensuite de prendre le reste de la division multiplié par 10 et répéter l'opération de division. Cet algorithme devient efficace lorsque l'on conserve en mémoire une table des valeurs approximatives de $\frac{10^t}{2^{e_t}}$. Il n'est alors plus nécessaire d'utiliser les coûteuses opérations arithmétiques à précision double.

En 2018, Adams présente ce qui est considéré aujourd'hui comme l'algorithme le plus rapide pour la conversion décimale des nombres flottants (Adams, 2018). Son algorithme est nommé Ryu (en référence au surnom japonais signifiant dragon). L'idée de base est la suivante. En premier lieu, on décode et analyse le nombre flottant à convertir sous la forme $f = (\text{-}1)^s \times m_f \times 2^{e_f}$. Ensuite, considérant les nombres flottants f^- et f^+ qui sont situés tout juste avant et après le nombre à convertir f, on évalue les points de milieu suivants : $\frac{f+f^-}{2}$ et $\frac{f+f^+}{2}$. Le reste de l'algorithme consiste à arrondir et tronquer le nombre f en fonction de ces points de milieu.

2.5 Autres représentations des nombres réels

De nos jours, pratiquement tous les ordinateurs utilisent la norme IEEE-754 pour représenter les nombres réels. Cela n'exclut toutefois pas qu'on reproche à

cette représentation des lacunes qui peuvent s'avérer déterminantes dans certains scénarios. La précision fixe qui ne convient pas à toutes les applications, la confusion qui peut résulter des multiples valeurs possibles des nombres non représentables (NaN), les erreurs d'arrondissement ainsi que les débordements (*underflow* et *overflow*) ne sont que quelques exemples. Les détracteurs de cette norme évoquent souvent l'argument que cette norme "one size fits all" est vieillissante puisqu'elle est fondée sur des travaux ayant eu lieu dans les décennies 70 et 80. On dénonce également le caractère énergivore des unités de calculs des nombres flottants (*Floating Point Unit*) qui peut représenter dans certains cas jusqu'à 50 % de la consommation énergétique du processeur (Tagliavini et al., 2018).

En 2010, Collishaw et al. proposent un amendement à la norme IEEE 754 pour y inclure les nombres à virgule flottante en base décimale (Cowlishaw et al., 2001). Les nombres flottants décimaux (*decimal floating point*) offrent plusieurs avantages à leurs semblables binaires. Tout d'abord, puisqu'ils sont conservés en base 10, aucune transformation n'est requise pour la lecture et l'écriture. Cela réduit considérablement les erreurs de conversion. Ce format est tout à fait adapté aux opérations monétaires puisqu'il est possible de représenter avec exactitude les valeurs telles que 1,10. Plusieurs langages de programmation proposent déjà ce type de représentation. C'est le cas pour le langage C# et son type de données `decimal` (sans toutefois respecter la norme IEEE 754). Puisqu'elles reposent sur une implémentation logicielle, les opérations sur les nombres flottants peuvent représenter des coûts de traitement additionnels.

En 2015, Gustafson propose ce qu'il décrit lui-même comme le résultat de "25 ans d'efforts pour proposer quelque chose de mieux que les nombres à virgule flottante". Il propose une toute nouvelle famille de représentation appelée Unums (*Universal numbers*) (Gustafson, 2017), un superensemble des standards IEEE 754 et 1788. Son concept sera bonifié à trois reprises depuis : types I (2015), II (2016) et

Posits/III (2017). Lors d'un séminaire dispensé en 2017, Gustafson détaille ses motivations et les problèmes qu'il adresse avec sa solution (Gustafson, 2021b). En représentation Posits, les bits sont utilisés de façon à retirer le maximum d'information par bit (entropie). Ils sont répartis en signe, régime, exposant et fraction. La formule de calcul est semblable aux nombres flottants : $(-1)^{\text{signe}} \times 2^{\text{exposant}} \times 1 + \text{fraction}$. La valeur de l'exposant est déterminée par les bits du régime et de l'exposant et n'implique pas l'utilisation d'un biais.

Cette représentation, basée sur des concepts algébriques et géométriques (en opposition aux calculs pour les nombres flottants), alterne entre les nombres exacts et les intervalles. Contrairement au standard IEEE 754 où la meilleure approximation possible d'un nombre réel est utilisée, cette représentation tolère un résultat inexact. On évaluera alors un nombre à l'aide d'un intervalle. Selon Gustafson, les Posits permettent d'obtenir une meilleure précision (par rapport à la norme IEEE 754) pour le même nombre de bits ou encore une précision équivalente en utilisant moins de bits. Par conséquent, le processus de traitement consomme moins d'énergie. Le nombre de bits requis pour représenter un nombre réel en Posits n'est pas fixé. Il s'agit d'une variante du modèle des nombres à virgule flottante effilés (*Tapered Floating Point)* proposé par Moris en 1971 (Wikipedia, 2021c) auquel un bit d'incertitude a été ajouté. On y utilise un nombre de bits variable pour représenter les nombres.

Son concept a été implémenté en quelques langages de programmation (Julia, C++) et certains progiciels tel MatLab. Il n'a toutefois pas encore fait l'objet d'une implémentation matérielle. Sa proposition a été mal reçue par Kahan. Dans un texte publié en 2015 (Kahan, 2021) ainsi que lors d'un débat organisé en 2016 (Gustafson, 2021a), Kahan critique avec véhémence l'approche de Gustafson. Il la juge coûteuse (en termes des transistors qu'elle impliquerait pour une implémentation matérielle) et appuyée sur des concepts mathématiques

insatisfaisants (seules des notions algébriques et géométriques sont requises). Il accuse Gustafson de faire preuve d'une incompréhension délibérée. De nombreuses études comparant les avantages et inconvénients des Posits sont parues dont celle de De Dinechin et al. (Dinechin et al., 2019). Elles convergent principalement vers la même conclusion : les Posits ne pourraient pas remplacer tels quels les nombres flottants du standard IEEE. Comme l'indique De Dinechin, lorsque les Posits font mieux que les nombres flottants de la même taille, il offrent une précision supérieure. Toutefois, dans les cas où ils sont moins bons (selon certaines exceptions problématiques), leur précision varie dramatiquement. Les travaux de Gustafson ont relancé l'intérêt au sujet de la représentation des nombres naturels. Entre autres, dans un article publié en 2018, Lindstrom et al. proposent un framework modulaire permettant de généraliser la représentation des nombres réels (Lindstrom, Lloyd & Hittinger, 2018). Leur modèle générique est basé sur les représentations IEEE et Posits et met à profit les encodages de Elias (méthodes d'encodage des entiers positifs mis au point dans les années 70)(Elias, 1975).

CHAPITRE III

ÉLÉMENTS DE PROGRAMMATION

Les langages C++ et C# font partie des langages de troisième génération. En opposition aux deux premières générations (machine et assembleur), les langages de cette génération sont portables (indépendants de la plateforme d'exécution), offrent un niveau d'abstraction supérieur (Wikipedia, 2021b) et permettent dans bien des cas la programmation orientée-objet. Deux autres niveaux de classification sont également répertoriés. La quatrième génération regroupe les langages à domaines spécifiques comme l'interrogation des banques de données avec le langage SQL (*Structured Query Language*). La cinquième génération comporte les langages de programmation à contraintes (à base de règles) et à inférences. Son usage est principalement réservé au domaine de l'intelligence artificielle. C'est le cas par exemple du langage Prolog. Le niveau d'abstraction offert par un langage peut représenter un impact sur la performance d'un programme puisque certaines opérations sont prises en charge par le compilateur.

Ce chapitre présente une introduction aux deux langages de programmation utilisés dans le cadre de ce mémoire. Les langages C++ et C# ont été conçus à plus de 20 ans d'intervalle. Nous exposerons pour chaque langage le contexte et les motivations pour lesquels ils ont été créés, leur mécanique de compilation ainsi que le support offert pour l'utilisation des nombres flottants au niveau de leur librairie standard.

Une section est également consacrée à la technologie des tests unitaires, élément clé de la réussite de tout projet de programmation.

Bien que tous les deux classés dans la même génération, ces langages sont à la fois similaires pour leur syntaxe et différents dans leurs objectifs. Alors que le langage C++ est orienté vers la performance, le C# est orienté vers la convivialité et offre de nombreuses fonctionnalités qui relèvent davantage de la quatrième génération. Mentionnons par exemple la composante LINQ (*Langage INtegrated Query*) qui simplifie l'itération et la projection sur les collections (éléments en mémoire) et les banques de données. Quotidiennement indispensables à tout programmeur, ces fonctionnalités visent à simplifier et accélérer la mise en place de solutions logicielles et non la performance de haut niveau.

3.1 Langage C++

Le langage C++ a été inventé par Stroustrup au début des années 1980 (Stroustrup, 2021a). Au cours de ses études universitaires, il travailla avec le langage Simula, premier langage orienté-objet dont l'origine remonte à la fin des années 60 (Sklenar, 2021). Dans une entrevue réalisée en 1994, Stroustrup raconte en détails les différentes motivations qui l'ont amené à inventer un nouveau langage (Stroustrup, 2021b). Avec Simula, les ennuis qu'il rencontre sont nombreux : organisation du programme, lenteur d'exécution, portabilité et coûts d'exploitation. Lui vient alors l'idée d'ajouter ce paradigme de programmation à un langage beaucoup plus rapide. Il cible alors le langage C, la meilleure option disponible à ses yeux. Ce langage est efficient et portable. Il est autant utilisé dans l'industrie qu'au niveau académique.

L'extension qu'il conçoit sera d'abord appelée "*C with classes*". En 1985, il publie le livre "Programming with C++" qui devient rapidement une référence. Le langage est tellement populaire qu'un comité ISO est formé afin d'en produire une norme

(ISO 14882) qui sera publiée en 1998 (International Organization for Standardization, 2021). Ce langage a par la suite évolué graduellement et, à ce jour, nous en sommes à la version 20.

Considéré comme l'un des langages les plus rapides et les plus performants, le C++ est compilé directement en langage machine. Le cycle de compilation est composé de trois étapes : le prétraitement (*preprocessing*), la compilation et l'édition des liens (*linking*). La première étape permet de mettre en place le code brut à compiler. Le code sera dépouillé de tous ses commentaires et les instructions appelées "directives" seront traitées pour prendre en charge l'ajout de fichier d'en-tête (`#include`), la définition des types (`#define`) et certaines inclusions basées sur la définition symbolique (`#ifdef`). À cette étape seront exécutées l'analyse syntaxique (lexicale) et sémantique. Par la suite, le compilateur transformera le code brut en code *assembler* dans un ou plusieurs fichiers appelés "objet". Finalement, le produit de la compilation sera réorganisé, c'est-à-dire que les liens entre les fichiers objet seront résolus (définitions, symboles, adresses mémoire) afin de produire un résultat final (*output*). Deux types de fichiers peuvent alors être produits : les librairies (parties de code destinées à être jointes à d'autres programmes) et les programmes autonomes (*executable*).

Les programmes C++ peuvent être compilés et exécutés sur toutes les plateformes d'exploitation. Il existe des compilateurs tout à fait gratuits, l'un des plus connus étant GCC (GNU compiler collection). L'utilisation des trousses d'outils visant à faciliter le cycle de développement (compilation, essais unitaires, paquetage, installation) est une tendance à la hausse. À titre d'exemple, on peut citer le logiciel CMake dont l'usage est de plus en plus répandu. En plus de sa version gratuite (il s'agit d'un logiciel libre), CMake offre une version payante personnalisée.

Le langage C++ est fortement typé (toutes les variables doivent avoir un type défini). En ce qui a trait aux nombres à virgule flottante, deux types de données sont utilisés : float (précision simple sur 32 bits) et double (précision double sur 64 bits). Le traitement des nombres flottants (lecture et écriture) fait partie des opérations offertes dans la librairie standard. À cet effet, deux fonctions sont offertes : `std::strtod` pour la lecture (*string to decimal*) et `std::to_string` pour l'écriture. L'extrait de code 3.1 illustre le passage aux deux formes avec ces fonctions de base.

Extrait de code 3.1 : Traitement des nombres flottants avec les fonctions de base en C++

```
#include <iostream>
#include <cstdlib>

using namespace std;

int main()
{
  char maChaine[] = "1.23";
  char *finChaine;
  double resultat;
  // lecture
  resultat = strtod(maChaine,&finChaine);
  // ecriture
  cout << "resultat : " << to_string(resultat);
  return 0;
}
```

En langage C++, la responsabilité de la gestion de la mémoire est entièrement confiée au programmeur. Cela lui confère un contrôle total sur l'allocation et l'accès.

L'utilisation des pointeurs et des références est fréquente et la validation des bornes (*bound checking*) est assumée par le programmeur.

3.2 Langage C#

Le langage C#, mis en place par la société Microsoft, a été rendu public en 2002 dans le cadre du dévoilement de la plateforme de développement appelée .NET. Tout comme les langages Java et C++ dont il est largement inspiré, C# est un langage orienté-objet (il supporte l'encapsulation, l'héritage, le polymorphisme et l'abstraction) et fortement typé (toutes les variables doivent avoir un type défini). Il s'agit d'un langage robuste et facile à apprendre (Mahesh, 2021).

C# est un langage de troisième génération largement utilisé. Cela s'explique par sa convivialité, la rapidité avec laquelle on peut arriver au résultat et sa ressemblance avec les langages Java et C++. En mars 2021, le site TIOBE.com classait le C# au cinquième rang des langages les plus populaires (TIOBE, 2021), tout juste derrière C, Java, Python et C++ (figure 3.1). Ses usages sont multiples : applications Web avec la plateforme ASP.NET (*Active Server Pages*), applications Windows, applications mobiles ainsi que développement de jeux vidéo. À cet effet, il est à noter que le C# s'intègre parfaitement avec l'engin Unity, populaire dans le milieu de la conception des jeux vidéo. Le langage en est à sa version 9.0 publiée en 2020 en même temps que la toute dernière plateforme de développement .NET 5.0. Chaque version apporte son lot de fonctionnalités additionnelles et Microsoft publie une version tous les deux ou trois ans.

Contrairement au langage C++, le C# n'est pas compilé directement en langage machine. Il est d'abord traduit en langage intermédiaire MSIL (*Microsoft Intermediate Language*) qui sera à son tour interprété par l'environnement d'exécution CLR (*Common Language Runtime*). Plus précisément, il sera analysé par le compilateur

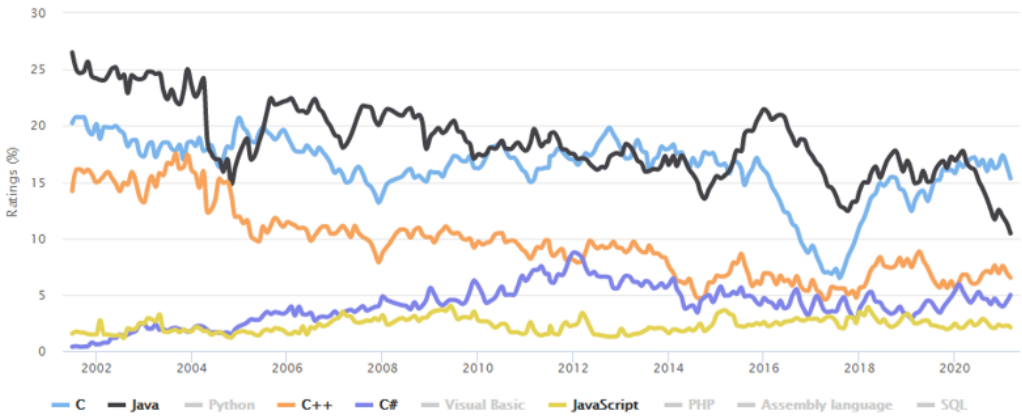


Figure 3.1 : Popularité des langages (source www.tiobe.com)

Ryu JIT qui transformera alors le langage intermédiaire en langage machine. Le cycle complet est illustré à la figure 3.2. Plusieurs langages peuvent être compilés de cette façon dans l'écosystème Microsoft. Entre autres, on retrouve Visual Basic, C# et F#. Cette mécanique de compilation existe également pour d'autres langages comme Java où il y a d'abord une transformation en *ByteCode* puis interprétation avec la machine virtuelle Java.

Ce cycle de compilation implique quelques considérations. D'abord, le développeur profite d'une certaine indépendance face à l'environnement dans lequel le code sera exécuté. De base, le code source qui est produit en C# est portable puisqu'il sera interprété par l'hôte sur lequel il sera exécuté. Dans certains scénarios, il est avantageux d'utiliser la compilation juste à temps JIT (*Just In Time*) puisqu'elle permet au compilateur d'optimiser les décisions en fonction de l'état du système ou encore de l'utilisation de certaines parties du code. La compilation JIT utilise également moins de mémoire que sa forme traditionnelle.

Comme c'est le cas pour le langage C++, la librairie standard offre les types de données et les opérations nécessaires à la manipulation des nombres flottants. Le langage C# comprend les types `float` (32 bits) et `double` (64 bits) équivalents

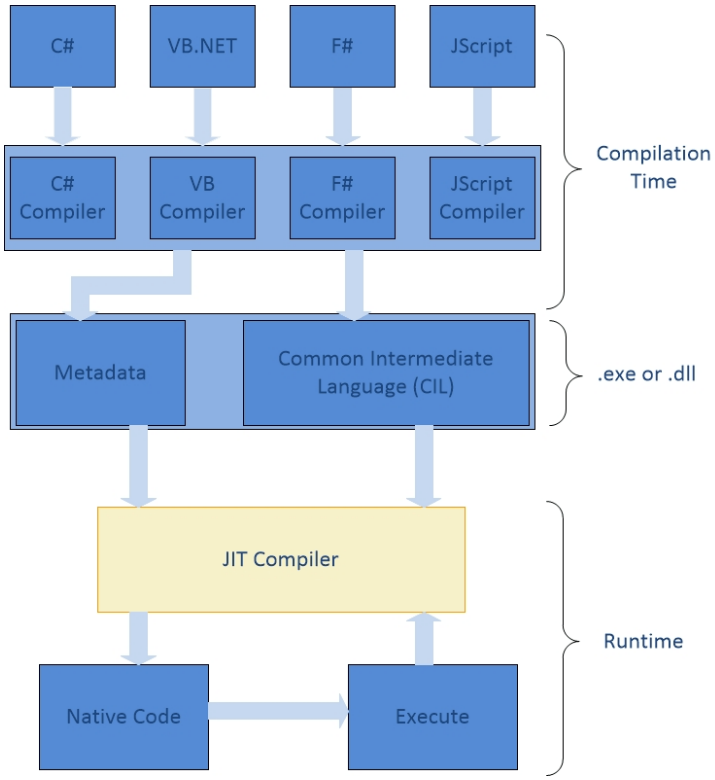


Figure 3.2 : Cycle de compilation du CLR (source https ://www.telerik.com)

aux mêmes types pour le C++ (Microsoft, 2021g). Chaque type possède sa plage et sa précision qui correspondent à la norme IEEE. Pour lire et écrire les nombres, il suffit d'utiliser la fonction `double.Parse` (lecture) et `ToString` pour l'écriture. L'extrait de code 3.2 illustre le passage aux deux formes avec ces fonctions de base.

Extrait de code 3.2 : Traitement des nombres flottants avec les fonctions de base en C#

```
int main()
{
  string maChaine = "1.23";
  double resultat;
  // lecture
  resultat = double.parse(maChaine);
  // ecriture
  Console.WriteLine( resultat.ToString());
  return 0;
}
```

Le langage C# prévoit également l'utilisation d'un second type de nombres à virgule flottante en base décimale : le type `decimal`. Stocké en mémoire sur 128 bits, sa précision est largement supérieure aux types `float` et `double`. À titre comparatif, la précision du type `decimal` est de 28/29 chiffres par rapport à environ 15/17 pour le type `double`. Ce type de données convient parfaitement aux opérations financières et monétaires. Toutefois, en conséquence de sa nature axée sur la précision, la plage approximative du type decimal est considérablement inférieure au type double. Alors que le type de données double couvre approximativement l'intervalle $[\pm 5.0 \times 10^{-324}, \pm 1.7 \times 10^{308}]$, la plage du type decimal est limitée à l'intervalle $[\pm 1.0 \times 10^{-28}, \pm 7.9 \times 10^{28}]$.

En C#, le développeur n'a pas à se soucier de la gestion de la mémoire pour l'allocation et le recyclage (*garbage collector*). Cela peut représenter un enjeu lorsqu'on cherche à optimiser un programme.

3.3 Tests unitaires

Depuis une vingtaine d'années, les tests unitaires font partie du quotidien des équipes de développement. Qu'il s'agisse d'une fonction singulière ou d'une partie du programme, ces petits bouts de code permettent d'automatiser la vérification du comportement du programme. Certaines méthodes de développement accordent une importance primordiale aux tests unitaires. C'est le cas de la technique de développement basée sur les essais (*TDD-Test Driven Development*). Cette technique populaire, que l'on attribue à Beck (Wikipedia, 2021d), est souvent associée aux nouvelles méthodes de développement comme Agile et XP (*Extreme Programming*). Elle consiste à réaliser, pour chaque fonction, les étapes suivantes :

— Écrire les tests correspondant à la fonction ;
— Écrire le code de la fonction ;
— Surveiller la réussite des tests ;
— Remanier le code au besoin (*refactoring*).

Pour chaque projet, la mise en place des tests unitaires représente un investissement. Les quelques heures consacrées au départ seront largement récompensées tout au long du projet. C'est ce qu'illustre Khorikov dans son ouvrage sur le sujet (figure 3.3) (Khorikov, 2020). Même si au départ la courbe d'avancement peut sembler à l'avantage des projets sans test, l'effort de maintenance finira nécessairement par être supérieur avec le temps. En particulier lors des changements qui surviennent systématiquement en cours de route. Plusieurs activités de maintenance telles la correction des anomalies, l'ajout d'une fonctionnalité ou le remaniement apporteront un certain désordre dans le code. On appelle ce phénomène l'entropie logicielle (*software entropy*) (Canfora et al., 2014). L'exécution des tests unitaires permettra au développeur d'être "informé" dès qu'une partie du logiciel n'a plus le même

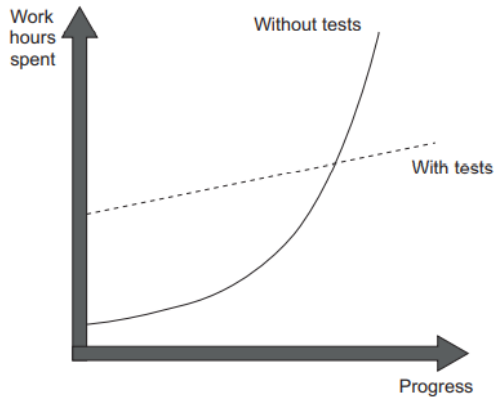


Figure 3.3 : Efforts investis pour les projets avec et sans tests unitaires

comportement. Il pourra alors remédier à la situation en apportant les corrections requises.

Anatomie d'un test unitaire

À la base, tous les tests unitaires sont construits de la même façon. Il s'agit d'une séquence en trois phases qu'on identifie en anglais par l'acronyme 'AAA' (*arrange, act, assert* - initialiser, agir et mesurer). L'extrait de code 3.3 illustre un exemple de test unitaire. Dans son ouvrage, Khorikov dresse une liste des quatre piliers des meilleurs tests unitaires (Khorikov, 2020) :

— Ils offrent une protection contre la régression (lorsque ce qui fonctionnait avant ne fonctionne plus à la suite d'une modification).
— Ils résistent aisément au remaniement du code (*refactoring*).
— Ils ne visent que les parties importantes du code et offrent une rétroaction rapide.
— Ils offrent la meilleure valeur en fonction de leur coût de maintenance.

Les tests unitaires permettent également la mise en place d'un processus d'intégration continue (Petersen, 2021). Les environnements de développement modernes mettent à profit l'automatisation des tests unitaires pour répondre à la question ultime : est-ce que le logiciel fonctionne encore ? Comme nous le détaillerons dans la partie contribution, nous avons accordé une grande importance aux tests unitaires dans le cadre des travaux de ce mémoire.

Extrait de code 3.3 : Exemple de test unitaire

```
[Trait("Category", "Smoke Test")]
[Fact]
public void cas_compute_float_64_1()
{
// Arrange
  for (int p = -306; p <= 308; p++)
  {
    if (p == 23) p++;
// Act
    var am = FastDoubleParser.ComputeFloat(q: p, w: 1);
    double? d = FastDoubleParser.ToFloat(false, am);
// Assert
    Assert.True(d.HasValue)
    Assert.Equal(d, testing_power_of_ten[p + 307]);
  }
}
```

Deuxième partie

Contribution

CHAPITRE IV

ADAPTATION ET PROGRAMMATION EN LANGAGE C# DE L'ALGORITHME FAST_FLOAT

Théoriquement, deux langages de programmation complets au sens de Turing sont réputés être interchangeables. Tout ce qui peut être programmé dans l'un peut être traduit dans l'autre (Wikipedia, 2021e). Dans la pratique, la conversion d'un programme est une activité ardue où plusieurs facteurs entrent en considération (complexité de l'algorithme, type de programmation utilisée, optimisation du code, spécificités des langages). Des outils spécialisés (appelés *transpilers* - contraction des mots *translate* et *compile*) permettent de convertir des instructions d'un langage vers un autre. Un peu à la manière des robots traducteurs (par exemple Google Translate), ces services sont intéressants pour un court extrait mais perdent vite de leur pertinence pour un ouvrage complet. Contrairement au robot traducteur, le programmeur expérimenté saura reconnaître l'intention sous-jacente derrière un programme et conservera l'esprit initial. Non seulement il est nécessaire de comprendre l'algorithme à adapter, mais il faut également maîtriser les capacités et les limites des langages qui sont en cause.

Ce chapitre présente les travaux d'adaptation et de programmation réalisés dans le cadre du mémoire. Dans un premier temps, nous exposerons les grands concepts de l'algorithme fast_float et de la librairie originale fast_float écrite en langage

C++. Ensuite, nous parcourrons les activités de développement et les différentes particularités qui ont été adressées lors de l'adaptation. La dernière section détaillera l'application concrète de notre librairie en l'intégrant à une solution existante de traitement de fichiers à haut volume.

4.1 Algorithme fast_float

Lors de ses travaux sur l'analyse des fichiers JSON, pour lesquels il a mis en place une librairie qui permet de traiter à haut débit ces fichiers (Lemire, 2021c), Lemire s'est rapidement rendu compte que l'analyse des nombres flottants constituait un goulot d'étranglement. Alors que le traitement des données sous forme de texte s'exécutait à grande vitesse, les traitements impliquant des nombres flottants plafonnaient. Il constate également que contrairement à la conversion décimale des nombres flottants (écriture), l'analyse (lecture) n'a pas fait l'objet de nombreuses publications depuis l'article de Gay (Gay, 1990).

En 2021, Lemire publie un article et une librairie dans lesquels il présente le fruit de ses travaux : son algorithme nommé fast_float (Lemire, 2021b). En plus de fournir un résultat exact, cet algorithme permet de traiter à haute vitesse les nombres flottants puisqu'il ne repose que sur quelques opérations simples. Il s'applique pour tous les nombres dont la mantisse comporte au plus 19 décimales *(digits)*. Le traitement est séparé en deux parties : l'analyse de la chaîne à traiter et le calcul du nombre flottant. Cette section en présente les grands concepts.

Analyse de la chaîne de caractères

La première étape consiste à lire et valider la chaîne de caractères à traiter en fonction de l'un des formats numériques autorisés. En parcourant la chaîne de caractères, on assemble les trois éléments permettant le calcul du nombre

flottant : le signe (positif/négatif), la mantisse (w) et l'exposant (q). L'analyse de la chaîne prévoit également les quelques cas d'exception en détectant les valeurs infinies (positive et négative), la valeur 0 et les valeurs ne pouvant être représentées numériquement (NaN). L'analyse de la chaîne ignore les espaces devant le nombre et détermine le nombre total de caractères ayant été considérés dans l'analyse (*consumed characters*). Une fois l'analyse terminée, on vérifie l'opportunité d'appliquer l'une des deux méthodes suivantes : la méthode rapide de Clinger si les conditions le permettent et l'algorithme fast_float autrement. Rappelons que lorsque l'exposant et la mantisse se situent à l'intérieur de certaines limites, la méthode de Clinger revient à multiplier (ou diviser lorsque l'exposant est négatif) la mantisse par 10 à la puissance de l'exposant. La méthode rapide de Gay n'est pas mise à contribution dans l'algorithme puisque les gains qu'elle apporte ne sont pas significatifs par rapport à la complexité qu'elle implique. L'algorithme ne tient pas compte du signe du nombre à traiter pendant les calculs. On le considère toutefois lors de la dernière étape du traitement. Dans les rares cas où le nombre de décimales excède 19, une méthode de repli pour le calcul est mise à profit.

Calcul du nombre flottant

Un nombre binaire flottant s'exprime par la formule ($m \times 2^p$) où m représente une mantisse binaire et p la bonne puissance de 2. Les calculs arithmétiques de l'algorithme fast_float s'appuient sur l'équivalence de cette représentation ($m \times 2^p$) à la formule utilisée par Clinger ($w \times 10^q$). En décomposant le terme 10^q en $5^q \times 2^q$, on retrouve la formule suivante : $w \times 5^q \times 2^q = m \times 2^p$. Les variables q et w étant déterminées à l'étape précédente (exposant et mantisse), il suffira alors de trouver les valeurs de m et p (la séquence d'opérations mathématiques est présentée à l'algorithme 1).

Pour optimiser le calcul de la mantisse binaire, les puissances de 5 sont préalablement calculées et conservées en mémoire avec une précision de 128 bits. Une fois la mantisse et l'exposant obtenus, il ne reste plus qu'à calculer l'opération $m \times 2^p \times 2^{-\exp}$ (où exp prendra la valeur 52 pour le type `double` et 23 pour le type `float`), traiter le signe du nombre et réinterpréter le résultat (alors un entier long) en variable double (64 bits) ou en float (32 bits).

Algorithme 1: Calcul de la valeur de la mantisse et de l'exposant

1: Selon w (la mantisse décimale) et q (l'exposant décimal)
2: Calculer m :
3: - Normaliser la mantisse (selon le nombre de zéros non significatifs)
4: $l \leftarrow leading\_zeroes(w)$
5: $w \leftarrow 2^l \times w$
6: - Convertir la mantisse décimale en mantisse binaire :
7: Estimer et ajuster le produit $w \times 5^q$:
8: $z \leftarrow 5^q \times w \div 2^{64}$
9: Effectuer certains contrôles de débordement
10: - m prendra la valeur des 54 premiers bits significatifs de z
11: Calculer p :
12: - Calculer u (bits significatifs de z)
13: $u \leftarrow z \div 2^{127}$
14: - Établir l'exposant binaire
15: $p \leftarrow ((217706 \times q) \div 2^{16}) + 63 - l + u$
16: - Traiter les cas anormaux (exposants trop petits, valeurs sous-normales (*subnormals*)
17: - Procéder à l'arrondissement au besoin

4.2 Présentation de la librairie fast_float originale

Publiée par Lemire au début de l'année 2020 (Lemire, 2021a), la librairie fast_float permet de convertir les chaînes de caractères représentant des nombres décimaux. Programmée selon le standard C++ 17.0, elle permet de traiter des nombres selon deux formats : fixe et scientifique. La version 3.0 de la librairie a été mise en ondes récemment. Quelques ajustements ont été apportés. Par exemple, l'ajout d'un

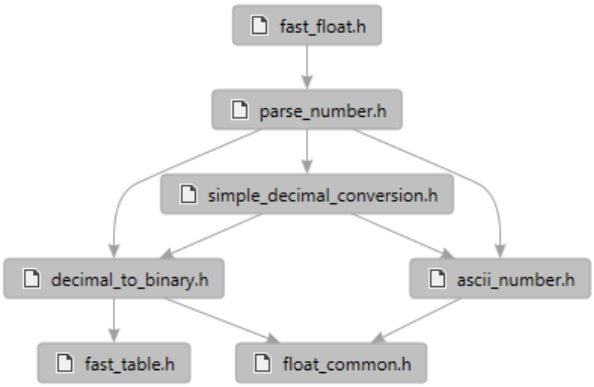


Figure 4.1 : Structure des fichiers d'en-tête de la librairie fast_float

paramètre permettant de spécifier le caractère utilisé comme séparateur décimal et l'optimisation de la méthode de repli pour les nombres comportant plus de 19 décimales. Cela ne représente toutefois aucun impact sur les travaux de conversion que nous avons effectués.

Organisation des éléments de la librairie

La librairie est conçue pour être intégrée au programme hôte. Elle se compose de quelques fichiers d'en-tête simples à référer (fichiers *header*) (Microsoft, 2021d). La structure hiérarchique de la librairie est illustrée à la figure 4.1. Les structures de données et fonctions sont regroupées dans un seul nom d'espace (*namespace*) `fast_float` et sont classées par fichier en fonction de leur mission. Le contenu de chaque fichier est présenté au tableau 4.1.

Les fonctions principales de la librairie sont les suivantes :

- `from_chars` : orchestration du traitement ;
- `parse_number_string` : analyse de la chaîne de caractères ;
- `compute_float` : calcul du nombre (algorithme fast_float).

Tableau 4.1 : Description du contenu des fichiers d'en-tête de la librairie

Fichier	Description du contenu
fast_float.h	Fichier principal à inclure au programme hôte. Ce fichier expose la structure de données pour la réponse, les formats numériques disponibles ainsi que le patron général de la fonction d'analyse.
parser_number.h	Orchestration principale pour l'analyse et le calcul du nombre flottant.
ascii_number.h	Fonctions d'analyse de la chaîne de caractères.
decimal_to_binary.h	Fonctions de calcul pour l'algorithme fast_float. Ces fonctions seront appelées lorsque la méthode rapide de Clinger ne peut être appliquée.
simple_decimal _conversion.h	Fonctions de repli (*fallback*) pour l'analyse des nombres lorsque la méthode de Clinger et l'algorithme fast_float ne sont pas applicables (lorsque le nombre à analyser possède plus de 19 décimales).
float_common.h	Paramètres, structures et fonctions communes pour l'algorithme.
fast_table.h	Valeurs précalculées des puissances de 5 selon l'intervalle $[-342, 308]$.

Utilisation de la librairie

Pour convertir une chaîne de caractères, il suffit d'appeler la fonction d'analyse `from_chars` en spécifiant les paramètres d'entrée sous la forme de pointeurs : début et fin de la chaîne à traiter et référence vers la variable où le nombre calculé sera conservé (extraits de code 4.1 et 4.2 provenant de la documentation de la librairie fast_float). Selon le type de la variable de retour (float, double), le compilateur sera en mesure d'appeler la fonction d'analyse avec les bons paramètres de calcul. Cela est rendu possible par l'utilisation des patrons de fonction dont nous parlerons plus tard dans cette section. En guise de retour, la fonction appelante recevra un pointeur situé à la fin de la chaîne traitée ainsi qu'une structure de données représentant l'erreur obtenue en cas de problème d'analyse. En cas de succès, le

résultat de la conversion sera emmagasiné à l'adresse de la variable fournie en entrée. La fonction d'analyse ne déclenche pas d'exception.

Extrait de code 4.1 : Différentes signatures de la fonction d'analyse

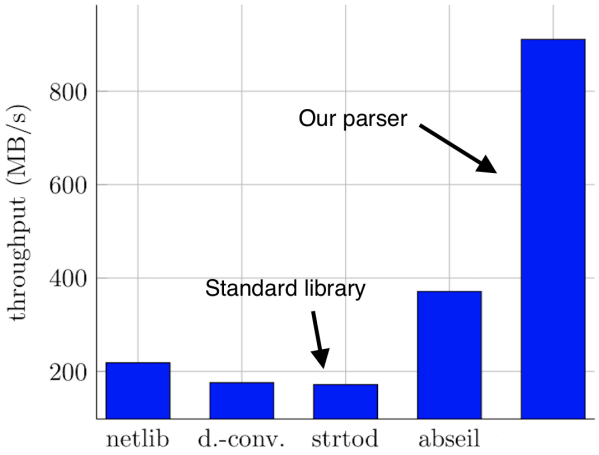
```
from_chars_result from_chars(const char* first,
                             const char* last,
                             float& value, ...);

from_chars_result from_chars(const char* first,
                             const char* last,
                             double& value, ...);
}
```

Extrait de code 4.2 : Appel de la méthode ParseDouble

```
#include "fast_float/fast_float.h"
#include <iostream>

int main() {
    const std::string input =  "3.1416 xyz ";
    double result;
    auto answer
         = fast_float::from_chars(input.data(),
                                  input.data()+input.size(),
                                  result);
    if(answer.ec != std::errc())
    {
    std::cerr << "parsing failure\n"; return EXIT_FAILURE;
    }
    std::cout << "parsed the number " << result << std::endl;
    return EXIT_SUCCESS;
}
```



Parsing speed for random 64-bit floating-point number in the uniform model on the AMD Zen 2 processor.

Figure 4.2 : Performance de la librairie fast_float vs librairies disponibles dans l'écosystème C++ (Lemire, 2021a)

Performance de la librairie

En plus de la fonction appelée strtod (*string-to-decimal*) qui provient de la librairie standard, il existe dans l'écosystème C++ plusieurs librairies spécialisées dans le traitement des nombres flottants. Parmi les plus réputées, on retrouve notamment netlib, double conversion (mise en place par Google) et abseil (également basée sur les travaux de Google) (NetLib, 2021 ; Google, 2021b, 2021a). Extraits de la documentation de la librairie, la figure 4.2 et le tableau 4.2 comparent sa performance par rapport aux librairies disponibles. Nous pouvons constater que la librairie fast_float est deux fois plus performante que sa plus rapide concurrente (abseil).

Tableau 4.2 : Vitesse et volume de traitement de la librairie fast_float vs librairies disponibles dans l'écosystème C++

Librairie	MB/s	MFloat/s
netlib	271.18	12.93
doubleconversion	225.35	10.74
strtod	190.94	9.10
abseil	430.45	20.52
fast_float	1042.38	49.68

Particularités de la librairie

La fonction `from_char` est responsable de l'orchestration du traitement. Elle tente d'abord l'analyse de la chaîne de caractères afin de construire la combinaison signe-mantisse-exposant. Par la suite, on vérifie la possibilité d'appliquer la méthode rapide de Clinger ou l'algorithme fast_float.

La séquence des instructions est conditionnée par différents paramètres qui varient selon le type de données à traiter (float ou double). Pour optimiser le code, Lemire a défini la structure de données nommée `binary_format` en utilisant une fonctionnalité du langage C++ appelée patron (*template pattern*) (Microsoft, 2021f). Cela permet de définir une structure générique qui pourra ensuite être déclinée pour plusieurs types de données, réduisant ainsi la redondance du code. La définition de cette structure est présentée à l'extrait de code 4.3. Pour illustrer les bénéfices de cette technique à l'aide d'un exemple concret, prenons les bornes pour les valeurs d'exposant permettant d'appliquer la méthode rapide de Clinger. Ces bornes ne sont pas les mêmes pour les types float et double (le minimum est -10 pour float et -22 pour double). Le patron de structure permet de généraliser le code en utilisant une seule condition `if` qui s'appliquera aux deux types de données (extrait de code simplifié 4.4).

Extrait de code 4.3 : Structure de paramètres binary_format

```
template <typename T> struct binary_format {
  static constexpr int mantissa_explicit_bits();
  static constexpr int minimum_exponent();
  static constexpr int infinite_power();
  static constexpr int sign_index();
  static constexpr int min_exponent_fast_path();
  static constexpr int max_exponent_fast_path();
  static constexpr int max_exponent_round_to_even();
  static constexpr int min_exponent_round_to_even();
  static constexpr uint64_t max_mantissa_fast_path();
  static constexpr int largest_power_of_ten();
  static constexpr int smallest_power_of_ten();
  static constexpr T exact_power_of_ten(int64_t power);
};
```

Extrait de code 4.4 : Utilisation et déclinaison d'un paramètre selon deux types de données

```
// condition avec parametre generique
..
if (binary_format<T>::min_exponent_fast_path() <= pns.exponent
...

// declinaisons du parametre min_exponent_fast_path
template <> constexpr
int binary_format<double>::min_exponent_fast_path() {
  return -22;
}
template <> constexpr
int binary_format<float>::min_exponent_fast_path() {
  return -10;
}
```

L'efficience des calculs repose sur l'accès aux valeurs précalculées pour les puissances de 5 et de 10. Ces valeurs, sur 128 bits, sont conservées en mémoire sous la forme de tableaux (*arrays*). L'extrait de code 4.5 illustre une partie des valeurs pour les puissances de 5. La première valeur du tableau correspond à l'exposant -342.

Extrait de code 4.5 : Valeurs précalculées des puissances de 5

```
const uint64_t power_of_five_128[]= {
        0xeef453d6923bd65a,0x113faa2906a13b3f,
        0x9558b4661b6565f8,0x4ac7ca59a424c507,
        0xbaaee17fa23ebf76,0x5d79bcf00d2df649,
        0xe95a99df8ace6f53,0xf4d82c2c107973dc,
        0x91d8a02bb6c10594,0x79071b9b8a4be869,
        0xb64ec836a47146f9,0x9748e2826cdee284,
        0xe3e27a444d8d98b7,0xfd1b1b2308169b25,

        ....
```

4.3 Conception et organisation de la librairie C#

La librairie csFastFloat a été mise en place à l'aide du logiciel Visual Studio 2019 selon la version du Framework .NET 5.0. En plus du code source de l'algorithme, on retrouve deux projets de tests : le premier contenant les tests unitaires et le deuxième les composantes permettant d'effectuer la mesure de la performance. Dans l'optique d'un partage et d'une réutilisation internationale, elle a été programmée en anglais (tout comme la librairie fast_float originale).

Les fonctions principales de la librairie sont les suivantes :

- `ParseDouble` et `TryParseDouble` : fonctions de façade[1] pour le type de données `double` ;
- `ParseFloat` et `TryParseFloat` : fonctions de façade similaires pour le type de données `float` ;
- `TryParseNumber` : orchestration du traitement complet d'analyse ;

1. Ces fonctions sont déclinées selon le format des paramètres d'entrée (string, char*, ReadOnlySpan) ainsi que pour les encodages sur 8 et 16 bits.

— `ParseNumberString` : analyse de la chaîne de caractères ;
— `FastPath` : méthode rapide de Clinger ;
— `ComputeFloat` : calcul du nombre (algorithme fast_float) ;
— `ToFloat` : conversion du résultat final en type `double` et `float`.

Utilisation de la librairie

Les fonctions d'analyse sont simples à utiliser. Il suffit d'appeler la méthode statique ParseDouble en spécifiant la chaîne à traiter (extrait de code 4.6). Des paramètres optionnels permettent de spécifier la forme du nombre attendu (scientifique ou décimal) ainsi que le caractère à utiliser pour la séparation décimale. La fonction peut retourner le nombre de caractères considérés lors du traitement d'analyse si l'on spécifie une variable de retour à cet effet (extrait de code 4.7).

Extrait de code 4.6 : Appel de la méthode ParseDouble

```
 string uneValeur = "1.45";
double resultat = FastDoubleParser.ParseDouble(uneValeur);
```

Extrait de code 4.7 : Prise en charge du nombre de caractères considérés

```
string uneValeur = "1.45";
double resultat =
FastDoubleParser.ParseDouble(uneValeur, out int nbCaracteres);
```

En entrée, trois types de données sont supportés : string, ReadOnlySpan<char> et char*. À cet effet, la signature de la fonction façade ParseDouble possède plusieurs surcharges (*overloads*). Deux versions de chacune des signatures sont offertes à l'utilisateur : avec ou sans la prise en compte du nombre de caractères considérés.

La librairie offre également ces fonctions sous le modèle TryParse (Microsoft, 2021c) qui tentera l'analyse et retournera un indicateur de réussite de l'opération

en plus du nombre flottant en cas de succès (figure 4.8). Contrairement à la fonction ParseDouble, la fonction TryParseDouble ne déclenche pas d'exception lorsque le traitement ne fonctionne pas.

Extrait de code 4.8 : Appel de la méthode TryParseDouble

```
string uneValeur = "1.45";
bool reussite =  FastDoubleParser
          .TryParseDouble(uneValeur, out double resultat);
```

Particularités et adaptations effectuées

Outre la fonction d'analyse de la chaîne de caractères, la librairie fast_float est composée de fonctions arithmétiques complexes dont le code est optimisé pour la performance. Heureusement, ces fonctions sont accompagnées d'une documentation abondante et détaillée. Après avoir étudié et analysé l'article de Lemire et le code de sa librairie fast_float, il fut alors possible d'entreprendre les travaux de conversion. Certaines fonctionnalités du langage C++ n'existent pas en C# et des adaptations ont été nécessaires pour tenter de conserver l'esprit et la performance de la librairie originale.

La première fonctionnalité à adresser est l'utilisation des patrons (*templates*) qui simplifie le code en généralisant le comportement des fonctions pour un type générique (Microsoft, 2021f). Cela se prête tout particulièrement pour l'algorithme puisqu'il est question de traiter deux types de données (float et double) en fonction de leurs spécificités respectives (par exemple la valeur maximale de l'exposant pour emprunter la méthode rapide de Clinger). Deux types de patrons sont utilisés pour consigner les paramètres de chaque type de données : structure et fonction. Ils sont illustrés aux extraits de code 4.3 et 4.4. En résumé, l'algorithme

fast_float emprunte un parcours générique conditionné par les patrons (structures et fonctions) correspondant au type de données à traiter (float ou double).

Les patrons n'existent pas en langage C#. Ce qui s'en approche le plus est l'utilisation des types génériques introduits à la version 2.0 du langage (MICROSOFT, 2021b). On peut définir une fonction comprenant un paramètre générique et y appliquer une contrainte pour indiquer qu'il doit hériter d'une classe (extrait de code 4.9). Toutefois, deux problèmes se posent. D'abord, il est impossible de limiter un paramètre générique à une classe qui est non héritable (*sealed*) comme c'est le cas pour tous les types de base (string, bool, int, float, etc.). De plus, la syntaxe du langage ne permet pas de spécifier le type de sortie d'une fonction à l'aide d'un type générique (pour retourner un float ou un double selon le cas). Ces limitations expliquent pourquoi il ne fut pas possible d'avoir un seul "chemin" pour traiter à la fois les types double et float comme c'est le cas pour la librairie fast_float. Deux classes statiques ont été mises en place : FastDoubleParser pour le traitement du type double et FastFloatParser pour le type float. Le comportement des fonctions de chaque classe est exactement le même.

Extrait de code 4.9 : Signature d'une fonction possédant un paramètre générique

```
public static T maFonction<T>(T param) where T: MaClasse
{
// ..
}
```

L'encodage est la représentation numérique d'un caractère sur une longueur déterminée de bits (généralement 8 ou 16). Il existe plusieurs familles d'encodage (UTF, ASCII, etc.) comportant une table de valeurs pour chaque caractère. Par exemple, le caractère "a" prendra la valeur 61 en UTF et 97 en ASCII. Les langages C++ et C# supportent tous les deux l'encodage UTF. En présence d'un encodage

sur 8 bits, il est possible d'optimiser la détection et l'analyse d'une suite de 8 chiffres consécutifs dans une chaîne de caractères. De base, c'est le cas du langage C++ puisqu'il traite ses caractères sur 8 bits (encodage UTF-8). La situation est différente pour le langage C# où deux longueurs d'encodage sont supportées. Par défaut, les chaînes des caractères sont encodées sur 16 bits (UTF-16) (Microsoft, 2021a). Deux séries de fonctions sont disponibles dans la librairie csFastFloat (une pour chaque longueur).

Pour améliorer le niveau de performance avec les chaînes encodées sur 16 bits, nous avons mis en place une fonction qui effectue la détection et l'analyse des caractères en bloc. Nous avons employé une technologie appelée SIMD (*Single Instruction, Multiple Data*) qui permet d'effectuer des opérations dites "vectorisées" où l'on traite simultanément une portion des données. Bien qu'elles soient sous-documentées et complexes à mettre en place ces opérations ont procuré à notre librairie un gain considérable en performance. Le concept algorithmique est simple : nous chargeons dans un vecteur un bloc de 8 caractères pour déterminer en peu d'étapes si ce bloc est composé de nombres entre 0 et 9. En cas de succès, nous appliquons une transformation sur ce vecteur pour en constituer un nombre entier. L'extrait de code simplifié 4.10 illustre la détection d'une séquence de 8 chiffres à l'aide des instructions SIMD.

Extrait de code 4.10 : Détection de 8 caractères consécutifs à l'aide des vecteurs

```
// lire en un trait 8 caracteres
 Vector128<short> raw = Sse41.LoadDquVector128((short*)start);
 Vector128<short> ascii0 =
      Vector128.Create((short)(48 + short.MinValue));
 Vector128<short> after_ascii9  =
      Vector128.Create((short)(short.MinValue + 9));
 // soustraire et masquer la valeur '0'
 Vector128<short> a  =
      Sse41.Subtract(raw, ascii0);
 // verifier que toutes les positions sont inferieures a '9'
 Vector128<short> b = Sse41.CompareLessThan( after_ascii9, a);
  // en cas de succes, b ne contient que des 0
 if (!Sse41.TestZ(b, b))
 {
   return false;
 }
```

4.4 Tests unitaires et assurance qualité

L'utilisation de la méthode de développement basée sur les essais (*Test Driven Developpement*) (Jones, 2021) se prête tout particulièrement aux projets de conversion de code. Les tests unitaires permettent de vérifier et surveiller le comportement des fonctions (cas réguliers, limites et exceptions). Débuter par l'écriture des tests permet de mieux structurer le code source et d'obtenir rapidement une rétroaction pour chaque fonction programmée.

Au total, plus de 220 tests unitaires ont été programmés dans le cadre des travaux de ce mémoire afin de tester avec minutie chacune des fonctions. Parmi cette banque de tests, nous retrouvons les tests utilisés par Lemire lors de ses travaux

(Lemire, 2021a). Inclure ces tests et les cas d'essais qu'ils contiennent permet de tirer profit de l'expérience de Lemire et de vérifier les limites et problématiques qu'il a rencontrées dans ses propres travaux.

Les tests unitaires traitent avec succès les fichiers assemblés par Tao (Tao, 2021a). Employé de Google, ce mathématicien et informaticien de renommée internationale s'est intéressé aux travaux de Lemire et publia en 2020 un billet sur son blogue à cet effet (Tao, 2021b). Il a rassemblé et conçu des fichiers qui permettent d'exécuter une série exhaustive de tests liés à l'analyse des nombres flottants. Certains de ces fichiers proviennent des projets d'équipes reconnues (Google, IBM, GO) axés sur le traitement des nombres (IBM, 2011 ; Google, 2021b ; Tecent, 2021 ; Adams, 2021). Analyser sans erreur les fichiers de Tao confère un niveau de certitude élevé en ce qui a trait au comportement de la librairie. Il est à noter que certains fichiers produits par Lemire font désormais partie de la collection de Tao.

Nous avons également testé le comportement de notre librairie en utilisant les cas d'essais répertoriés par Abraham pour la librairie strtod (Abraham, 2021). Il s'agit d'une centaine de cas permettant de tester les limites liées à la conversion et au formatage des nombres flottants. Il est intéressant de mentionner que, contrairement à la librairie csFastFloat, la librairie standard de Microsoft n'arrive pas à résoudre plusieurs des éléments qui font partie du fichier d'Abraham.

CHAPITRE V

MÉTHODE ET OUTILS UTILISÉS POUR L'ANALYSE DE LA PERFORMANCE

Optimiser la performance d'une application est une tâche complexe qui comporte plusieurs étapes. Elles s'inscrivent dans un processus itératif dans lequel on tente de comprendre et d'améliorer les performances logicielles. En déterminant les chemins critiques, il est alors possible de remanier le code et d'explorer différentes avenues pour améliorer le temps d'exécution et l'utilisation de la mémoire. Ce chapitre traite du processus et des outils qui ont été utilisés pour étudier et améliorer la performance de la librairie. La démarche utilisée pour effectuer l'analyse et les principaux aspects à considérer lors de l'optimisation d'un programme écrit en langage C# y sont détaillés. Les recommandations formulées s'avèrent pertinentes pour le développement de toute application où la performance constitue un enjeu.

Les programmes conçus et exécutés à l'aide du Framework .NET ne sont pas compilés en langage machine. Ils sont pris en charge par l'environnement CLR (*Common Language Runtime*). Les responsabilités de l'environnement sont partagées en deux : l'engin d'exécution et le module de gestion de la mémoire. L'engin d'exécution (*execution engine*) est responsable de la compilation JIT, du système de types et du traitement des exceptions. Le module de gestion de la mémoire (*garbage collector*) est pour sa part responsable de l'allocation et du recyclage de la

mémoire (Kokosa, 2018). Les interactions entre les deux parties sont multiples et essentielles à la performance. Par exemple, le compilateur JIT fournit au module de gestion de la mémoire des informations statistiques sur l'utilisation des variables en temps réel. Pour maximiser la performance de la librairie, l'emphase a été portée sur ces deux aspects du CLR : l'utilisation de la mémoire (*garbage collector*) et l'optimisation du code JIT-asm généré (*execution engine*).

Une transformation considérable du code peut survenir lors du passage vers le langage intermédiaire. Par exemple, lorsqu'un programmeur utilise des fonctions de haut niveau (comme celles de la librairie LINQ), ces opérations seront traduites en plusieurs opérations de bas niveau. Bien qu'elles puissent représenter une économie appréciable de temps pour le développeur, ces fonctions peuvent représenter un enjeu pour la performance puisqu'il est impossible de contrôler avec précision quelles seront les instructions produites. C'est la principale raison pour laquelle ces librairies de haut niveau n'ont pas été utilisées dans le cadre de ce mémoire.

5.1 Utilisation de la mémoire

En langage C#, la mémoire est allouée de deux façons : sur la pile (*stack*) et sur le tas (*heap*). La mécanique d'allocation est assumée par le framework et le développeur n'a généralement pas à s'en soucier. Toutefois, l'utilisation de l'une ou l'autre de ces options aura des conséquences sur la performance du programme. Les coûts d'accès et de maintenance des objets en mémoire ne sont pas les mêmes pour chaque option.

La pile est utilisée par le système pour conserver la trace d'exécution du programme et une nouvelle couche (*frame*) est ajoutée à la pile pour chaque appel de fonction. Comme son nom l'indique, l'allocation sur la pile s'effectue en mode "dernier arrivé, premier sorti" (*LIFO - Last In, First Out*) et seules les données situées dans la

dernière couche de la pile sont accessibles. Lorsque l'exécution d'une fonction se termine, sa couche est détruite et retirée de la pile. On réfère souvent à la pile comme une série de petits casiers ordonnés les uns sur les autres où seulement le casier sur le dessus de la pile peut être accessible.

Pour sa part, le tas est utilisé pour emmagasiner et accéder rapidement aux variables et objets en mémoire. Contrairement à la pile, les données du tas sont accessibles en tout temps. La maintenance (réorganisation, nettoyage, etc.) du tas est confiée au *garbage collector* et représente une charge pour le système. Il est possible d'imaginer le tas comme une surface de travail où tout est accessible en tout temps et pour laquelle il faut refaire périodiquement l'épuration et le classement pour y libérer de l'espace.

La première fonctionnalité du langage C# employée pour minimiser l'implication du *garbage collector* et accélérer l'allocation de la mémoire est l'utilisation des structures. À l'inverse des classes, qui sont beaucoup plus souvent utilisées, les structures sont allouées sur la pile et non sur le tas, allégeant ainsi la tâche du *garbage collector* (Kokosa, 2018). La libraire csFastFloat comporte de nombreuses structures. Celles-ci sont notamment utilisées pour conserver les données recueillies lors de l'analyse de la chaîne de caractères ainsi que pour le calcul du nombre flottant (extrait de code 5.1).

Extrait de code 5.1 : Exemple de structure

```
internal struct value128
  {
    public ulong low;
    public ulong high;

    public value128(ulong h, ulong l) : this()
    {
      high = h;
      low = l;
    }
  }
```

Nous avons tiré profit de la structure de données `ReadOnlySpan<T>` (Microsoft, 2021e). Cette structure, également allouée sur la pile, permet de stocker les données dans une zone contiguë de la mémoire et ne sera jamais déplacée sur le tas. Elle a été employée pour chaque situation où l'on aurait normalement conservé les données dans un tableau (extrait de code 5.2). Il faut mentionner que les gains en performance pour ce type de données ne sont disponibles pour le moment que pour le type de données `byte`. Comme son nom l'indique, l'accès aux données n'est permis qu'en lecture seule.

Extrait de code 5.2 : Utilisation de la structure ReadOnlySpan

```
private static ReadOnlySpan<byte> powersTable
  => new byte[19] {
      0,  3,  6,  9,  13, 16, 19, 23, 26, 29,
      33, 36, 39, 43, 46, 49, 53, 56, 59
    };
```

L'utilisation des pointeurs et des références est une pratique courante en C++. En langage C#, il faut spécifier au compilateur notre intention d'accéder directement

à la mémoire en marquant chaque fonction de l'attribut `unsafe` et prévenir la relocalisation accidentelle de la variable pointeur (*garbage collecting*) avec l'instruction `fixed` (extrait de code 5.3). L'attribut `unsafe` permet l'utilisation des pointeurs pour effectuer des opérations arithmétiques (extrait de code 5.4), des déplacements ainsi que des affectations (*casting, reinterpretation*).

Extrait de code 5.3 : Initialisation d'un pointeur à l'aide de l'instruction fixed

```
public static unsafe double ParseDouble(ReadOnlySpan<char> s)
{
  fixed (char* pStart = s)
  {
  // ...
  }
}
```

Extrait de code 5.4 : Opération arithmétique sur les pointeurs

```
// p et pstart sont des pointeurs char *
// calcul de la distance entre p et pstart
 answer.characters_consumed = (int) (p - pstart);
```

Une partie importante de l'algorithme repose sur des données précalculées conservées en mémoire (tables des puissances de 5 et de 10, etc.). L'accès à ces données a été optimisé par l'emploi de la fonction `GetArrayDataReference` qui retourne l'adresse de la position 0 du tableau (*array*). La stratégie consiste à accéder directement à la zone mémoire où sont conservées les données en calculant un déplacement (extrait de code 5.5). Cela prévient la vérification systématique et coûteuse des limites lors de l'accès à la mémoire (*bounds checking*). Nous illustrerons concrètement l'impact sur le code généré par cette optimisation à la section suivante.

Extrait de code 5.5 : Accès optimisé à la mémoire sans vérification des limites

```
static ushort maFonction(int deplacement)
{
  ref ushort tableRef =
      ref MemoryMarshal.GetArrayDataReference(maZone);
  return Unsafe.Add(ref tableRef, (IntPtr)(uint) deplacement);
}
```

5.2 Optimisation du code JIT-asm

Pour optimiser les performances de la librairie, il est nécessaire de comprendre le pourcentage du temps de traitement qui est accordé à chaque fonction. Les outils de profilage sont conçus à cet effet. Deux outils ont été utilisés dans le cadre du projet : l'outil Performance Profiler inclus dans le logiciel Visual Studio et l'outil vTune développé par la compagnie Intel (Intel, 2021). Le profilage a été effectué selon les étapes suivantes :

- Mise en place d'une application autonome utilisant la librairie csFastFloat ;
- Exécution de l'application sous la surveillance de l'outil de profilage ;
- Examen du rapport d'exécution et détection des fonctions problématiques (*hot path*) (figure 5.1).

Le logiciel Visual Studio 2019 permet l'analyse de la performance à même les tests unitaires. Cela simplifie le processus puisqu'il n'est pas nécessaire de mettre en place une application autonome. L'outil vTune permet pour sa part de consulter le code JIT-asm ayant fait l'objet de l'examen. L'information recueillie dans le rapport d'examen permet de comprendre et d'identifier quelles sections du code pourraient être améliorées (temps d'exécution et utilisation de la mémoire). On

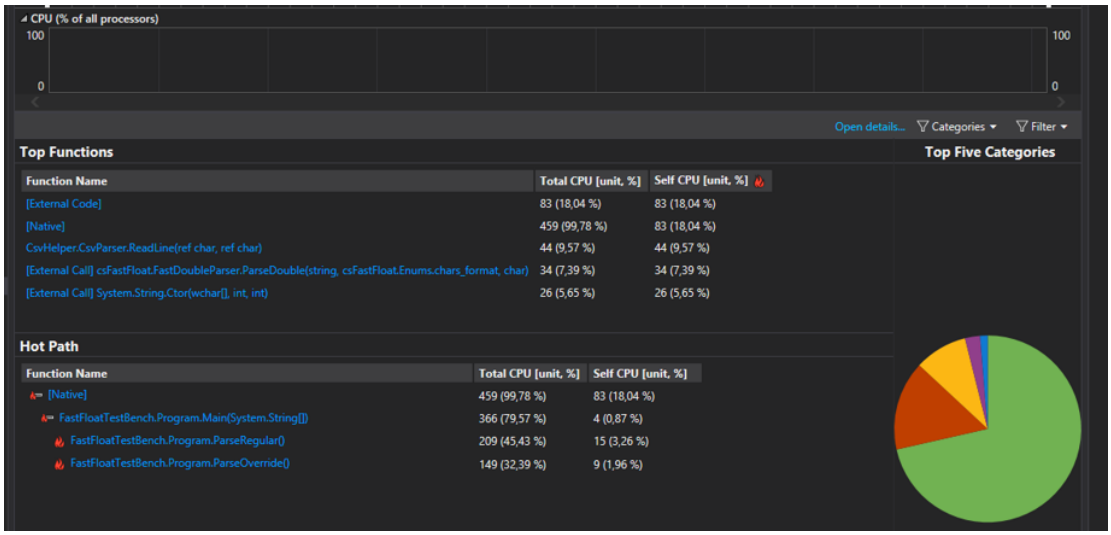


Figure 5.1 : Rapport sommaire de l'examen

peut alors apporter des ajustements et reprendre l'analyse afin de vérifier l'impact des modifications.

Outils d'analyse du code JIT-asm

L'ordre des conditions et le choix des instructions utilisées dans une fonction C# auront un impact sur le code *assembler* généré. Il est nécessaire d'étudier le code JIT-asm généré pour comprendre l'impact d'un choix. Dans le cadre de ce projet, quelques outils complémentaires ont été employés :

— le logiciel vTune cité précédemment ;
— l'outil en ligne sharplab.io (Shchekin, 2021) ;
— l'affichage du code *assembler* à même Visual Studio ;
— l'outil en ligne Compiler Explorer (Godbolt, 2021).

Tous ces outils fonctionnent de la même façon. Ils analysent le code source et affichent le rendu en code *assembler*. Les trois premiers ont été utilisés pour le

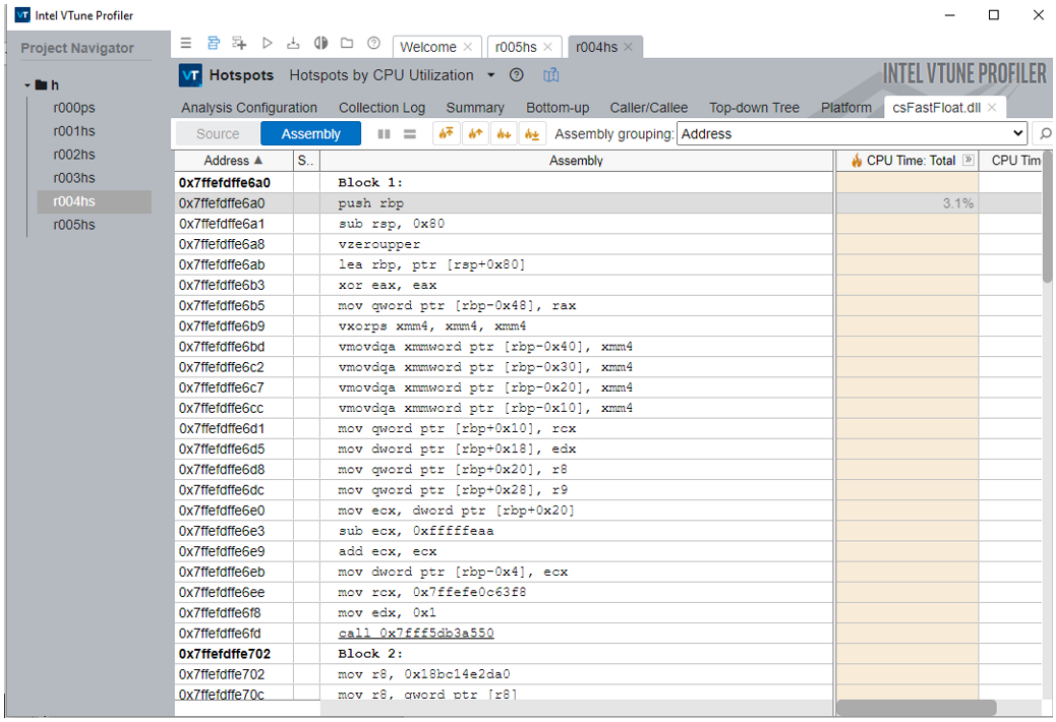


Figure 5.2 : Affichage du code JIT-asm avec vTune

code écrit en langage C# et le troisième pour étudier certaines fonctions de la librairie fast_float écrite en C++.

Avec l'outil vTune, l'analyse du code JIT-asm s'effectue à la suite à l'examen de performance. On peut alors explorer et consulter sur demande le code *assembler* de n'importe quelle fonction (figure 5.2). L'avantage de cet outil est la disponibilité du code pour l'ensemble de la librairie (*assembly*). Toutefois pour mesurer l'impact d'un changement, il faut exécuter à nouveau l'examen de profilage. Cette opération peut prendre plusieurs minutes.

Le deuxième outil, sharplab.io, offre la conversion instantanée d'une fonction en code JIT-asm (figure 5.3). Cela représente un avantage lorsque l'on veut connaître l'impact d'un choix (instruction ou permutation) en comparant le rendu de deux fonctions. En revanche, cet outil a été conçu pour analyser un échantillon de code et

Figure 5.3 : Analyse d'une fonction avec sharplab.io

non un ensemble de fonctions réparties en plusieurs classes (et sur plusieurs fichiers). Cela peut représenter une manipulation du code pour satisfaire les exigences du compilateur en ligne.

La fenêtre "Dissasembly Window" est un outil peu connu et peu documenté de l'environnement de développement Visual Studio qui permet d'afficher le code *assembler* généré lors de l'exécution d'un programme. Cette option est destinée à être utilisée en mode "débogage"[1]. En mode *debug*, le compilateur exécutera une traduction *assembler* non optimisée qui respecte la sémantique du programme C# original. En mode *release*, c'est une version optimisée qui sera exécutée (où le compilateur peut prendre la décision de transformer le code) et les points d'arrêts n'y sont habituellement pas considérés. En désactivant l'option "déboguer uniquement le code utilisateur" *(Just My Code)*, il est possible de capter les points

1. Où l'objectif est l'éxécution du programme pas à pas. Cela implique que le code soit accompagné de son descriptif appelé *symbol*.

```
Disassembly
Address: csFastFloat.Utils.TryParseEightConsecutiveDigits_SIMD(char*, out uint)
Viewing Options
  Show code bytes   Show address
  Show source code  Show symbol names
  Show line numbers
cmp         eax,1
jne         csFastFloat.Utils.TryParseEightConsecutiveDigits_SIMD(Char*, UInt32 ByRef)+02Ch (07FFDBA5F395Ch)
  272:       {
  273:         value = 0;
mov         rax,qword ptr [rbp+18h]
xor         edx,edx
mov         dword ptr [rax],edx
  274:         return false;
xor         eax,eax
lea         rsp,[rbp]
pop         rbp
ret
  275:       }
  276:
  277:
  278:       value = 0;
mov         rax,qword ptr [rbp+18h]
xor         edx,edx
mov         dword ptr [rax],edx
  279:       Vector128<short> raw = Sse41.LoadDquVector128((short*)start);
mov         rax,qword ptr [rbp+10h]
vlddqu      xmm0,xmmword ptr [rax]
vmovapd     xmmword ptr [rbp-10h],xmm0
  280:       Vector128<short> ascii0 = Vector128.Create((short)(48 + short.MinValue));
vmovupd     xmm0,xmmword ptr [csFastFloat.Utils.TryParseEightConsecutiveDigits_SIMD(Char*, UInt32 ByRef)+0120h (07FFDBA5F3A50h)]
vmovapd     xmmword ptr [rbp-20h],xmm0
  281:       Vector128<short> after_ascii9 = Vector128.Create((short)(short.MinValue + 9));
vmovapd     xmm0,xmmword ptr [rbp-10h]
vpsubw      xmm0,xmm0,xmmword ptr [rbp-20h]
vmovapd     xmmword ptr [rbp-30h],xmm0
vmovapd     xmm0,xmmword ptr [rbp-30h]
vpcmpgtw    xmm0,xmm0,xmmword ptr [csFastFloat.Utils.TryParseEightConsecutiveDigits_SIMD(Char*, UInt32 ByRef)+0140h (07FFDBA5F3A70h)]
vmovapd     xmmword ptr [rbp-70h],xmm0
vmovapd     xmm0,xmmword ptr [rbp-70h]
vptest      xmm0,xmmword ptr [rbp-70h]
je          csFastFloat.Utils.TryParseEightConsecutiveDigits_SIMD(Char*, UInt32 ByRef)+084h (07FFDBA5F39B4h)
  286:       {
  287:         return false;
xor         eax,eax
lea         rsp,[rbp]
pop         rbp
```

Figure 5.4 : Affichage du code *assembler* avec Visual Studio

d'arrêt (*breakpoints*) dans le code et ainsi interrompre l'exécution du programme à certains endroits pour afficher le code *assembler* en mode "final". Cette fenêtre est illustrée à la figure 5.4.

Le dernier outil, Compiler Explorer, propose sensiblement les mêmes services que les précédents pour une gamme différente de langages (C++, Java, Rust, Go, etc). Une fonctionnalité intéressante de cet outil est le regroupement des instructions (en langage *assembler*) par couleur pour chacune des lignes de code analysées. La figure 5.5 illustre l'utilisation de cet outil.

Pour expliquer la pertinence de l'analyse du code JIT-asm, nous avons comparé le rendu de deux approches pour l'accès aux données d'un vecteur. Les fonctions M1 et M2 accèdent toutes les deux à la position n du vecteur $powersTable$. La première

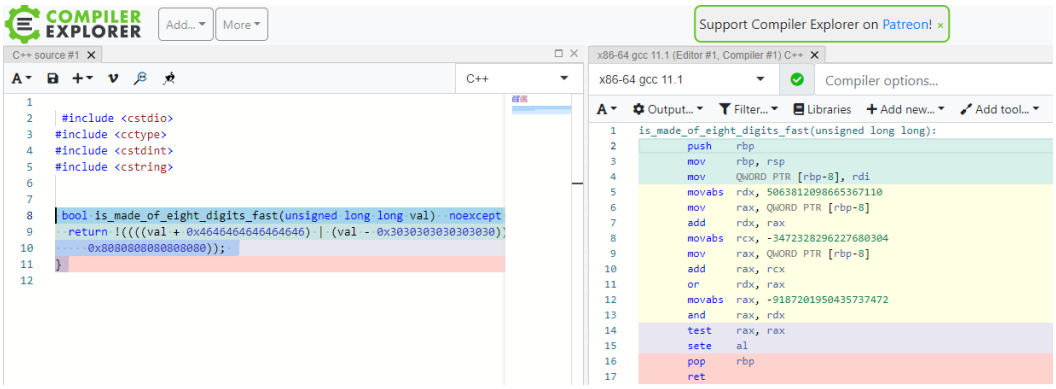


Figure 5.5 : Analyse d'une fonction C++ avec Compiler Explorer

utilise la méthode conventionnelle d'un indice entre crochets. La seconde calcule un déplacement en mémoire à partir d'une référence obtenue avec la méthode GetReference (extrait de code 5.6).

Le résultat obtenu lors de la compilation peut étonner (extrait de code 5.7). La fonction M1, ne comptant qu'une seule instruction C#, est traduite en six instructions JIT-asm. À l'inverse, la fonction M2 n'en compte que deux. En regardant de près, on constate que les quatre lignes supplémentaires de la première fonction (0000, 0003, 000d et 0012) constituent une vérification préventive sur le registre *ecx* par rapport aux bornes du vecteur *powersTable* (*bounds checking*). Pour la fonction M2, nous avons forcé l'accès direct en mémoire avec l'instruction AddByteOffset. Le reste des deux fonctions est le même : on capture dans le registre *eax* la valeur à l'adresse de base du vecteur (0x205e0990) + déplacement contenu dans le registre *ecx*.

Cet exemple d'optimisation démontre la nécessité d'examiner et de bien comprendre le code JIT-asm généré, en particulier lorsque des opérations sont effectuées de façon répétitive. On peut ainsi éliminer certaines vérifications et améliorer la performance.

Extrait de code 5.6 : Accès en mémoire sans vérification des bornes

```
static ReadOnlySpan<byte> powersTable => new byte[19]
{
  0,  3,  6,  9,  13, 16, 19, 23, 26, 29,
  33, 36, 39, 43, 46, 49, 53, 56, 59,
};

static byte M1(int n)
    =>  powersTable[n];

static byte M2(uint n)
{
  ref byte tableRef = ref MemoryMarshal
        .GetReference(powersTable);
  return Unsafe.AddByteOffset(ref tableRef, (nint)n);
}
```

Extrait de code 5.7 : Code JIT-asm généré avec et sans calcul des bornes

```
C.get_powersTable()
    L0000: mov dword ptr [ecx], 0x205e0990
    L0006: mov dword ptr [ecx+4], 0x13
    L000d: ret

C.M1(Int32)
    L0000: cmp ecx, 0x13
    L0003: jae short L000d
    L0005: movzx eax, byte ptr [ecx+0x205e0990]
    L000c: ret
    L000d: call 0x5dcb7610
    L0012: int3

C.M2(UInt32)
    L0000: movzx eax, byte ptr [ecx+0x205e0990]
    L0007: ret
```

CHAPITRE VI

RÉSULTATS OBTENUS

Le deuxième objectif principal du mémoire consiste à mesurer et analyser la performance de la librairie csFastFloat. Nous présenterons dans ce chapitre les résultats des expérimentations effectuées afin d'effectuer une analyse détaillée de la performance. En premier lieu, nous effectuerons une comparaison des résultats avec la librairie standard de Microsoft. Par la suite, nous analyserons la différence de performance entre les implémentations de l'algorithme fast_float (C++ et C#).

6.1 Mise en contexte

La librairie csFastFloat peut s'exécuter sur plus d'une plateforme. Elle supporte une multitude de version du Framework .NET (standard 2.0, core 3.1 et 5.0) et plus d'un système d'exploitation (Windows, Linux). Les essais de performance ont été effectués avec la version la plus récente du framework (.NET 5.0). Nous avons mesuré les performances à l'aide d'un serveur Linux hébergé sur une architecture Rome (processeur AMD et microarchitecture Zen 2.). Pour des fins de validations, les mesures de performance ont également été prises à l'aide d'un ordinateur de travail personnel (processeur Intel Core i7-10610U).

L'analyse comparative (*benchmarking*) est un outil essentiel qui permet de mesurer la performance d'un programme qu'on souhaite optimiser tout au long du cycle

de développement (Gordon, 2019). L'analyse peut porter sur une partie du programme ou sur une fonction individuelle. Mesurer la performance d'une librairie représente certains défis. Il faut minimiser les interférences et s'assurer que les résultats soient reproductibles. Pour garantir la fiabilité des résultats, les programmes mesurés doivent être compilés avec une configuration de type *release* (sans aucune information de débogage).

Nos analyses ont été effectuées à l'aide la librairie BenchmarkDotNet (Akinshin, 2021). Il s'agit d'une solution simple et fiable pour mesurer la performance logicielle. Cette librairie est largement adoptée, tant par les grandes compagnies comme Microsoft que par la communauté de développement GitHub. Elle prend en charge les opérations de démarrage (*warmup*) nécessaires pour assurer la fiabilité des résultats. Ces opérations sont primordiales puisque les résultats pourraient varier considérablement entre les premières exécutions du code et les exécutions successives. Rappelons que le code C# est transformé en langage intermédiaire pour être ensuite interprété par le *runtime* CLR. On cherche en quelque sorte à mesurer une version calibrée pour laquelle toutes les optimisations ont été apportées par le *runtime*. Suite à cette calibration, le programme en mémoire est considéré au sommet de sa performance (Barrett et al., 2017). La librairie BenchmarkDotNet est simple à utiliser. Sa configuration de base, qui convient à la grande majorité des scénarios d'essais, peut être personnalisée (nombre d'itérations, temps d'exécution, statistiques affichées au sommaire des résultats, etc.). En plus de la vitesse d'exécution, il est possible de mesurer l'empreinte du programme en mémoire. Cette librairie effectue une analyse statistique des résultats afin de mettre de côté les valeurs aberrantes (*outliers*) obtenues lors de l'analyse. Barett et al. mentionnent l'importance d'exclure ces résultats trop petits ou trop grands qui pourraient provenir d'une interférence (liée à la compilation ou à la gestion de

la mémoire) (Barrett et al., 2017). Tous les résultats sont présentés en millions de float par seconde.

Mettre en place les éléments du *benchmark* est une activité relativement simple. Il suffit de charger en mémoire le contenu du fichier à traiter pour ensuite le soumettre à une ou plusieurs fonctions d'évaluation. Un détail particulier à mentionner consiste au fait que les fonctions d'analyse doivent retourner une valeur pour ne pas être détectées comme du code mort (*dead code*) par le compilateur JIT. Les extraits de code 6.1 et 6.2 illustrent la simplicité des fonctions d'analyse.

Extrait de code 6.1 : Scénario d'analyse avec Double.Parse

```
[Benchmark(Description = "Double.Parse()")]
  public double DoubleParser_()
  {
    double max = double.MinValue;
    foreach (string l in _lines){
      double d = Double.Parse(l);
      max = d > max ? d : max;
    }
    return max;
  }
```

Extrait de code 6.2 : Scénario d'analyse avec FastDoubleParser.TryParseDouble

```
[Benchmark(Description = "FastFloat.TryParseDouble()")]
  public double FastParser_()
  {
    double max = double.MinValue;
    foreach (string l in _lines){
      FastDoubleParser.TryParseDouble(l, out double d);
      max = d > max ? d : max;
    }
    return max;
  }
```

Pour des fins d'apprentissage, nous avons également mis en place un programme rudimentaire d'analyse comparative (nommé BenchmarkHandCoded). Les résultats obtenus avec ce programme sont similaires à ceux de la librairie BenchmarkDotNet. La conception de ce programme nous a permis d'expérimenter et comprendre les techniques requises pour mesurer avec précision les événements dont la durée est microscopique. Ce programme n'effectue toutefois pas l'analyse statistique des résultats.

6.2 Stratégie utilisée pour déterminer les résultats obtenus

Dans ce chapitre nous présenterons les résultats obtenus selon trois scénarios. D'abord, une comparaison avec la librairie standard de Microsoft. Deuxièmement, nous comparerons les deux implémentations de l'algorithme fast_float (C++ et C#). Finalement, nous ferons la démonstration des gains procurés par l'algorithme fast_float dans le cadre de l'intégration à une librairie spécialisée dans le traitement à haut volume des fichiers de données.

Tableau 6.1 : Description des fichiers utilisés lors des comparaisons

Fichier	Description
canada.txt	Ensemble de données couramment utilisé pour l'analyse comparative. Il contient plus de 111 000 éléments représentant des coordonnées géographiques exprimées sur 64 bits.
mesh.txt	Ensemble emprunté aux données d'essais de la librairie sajson (Austin, 2021). Il contient des données relativement simples à traiter.
synthetic.txt	Données composées de 150 000 nombres flottants aléatoires.

Dans le cadre des travaux de ce mémoire, les opérations à mesurer s'effectuent en nanosecondes. Puisqu'il s'agit d'une unité de mesure microscopique, la stratégie que nous avons adoptée est la suivante : générer des fichiers de tests contenant des centaines de milliers de cas à traiter, charger ces données en mémoire (toutes les lignes du fichier) et mesurer le temps de traitement pour l'ensemble des lignes. On obtient alors le temps de traitement au total (que l'on peut diviser par le nombre de lignes pour déterminer les temps de traitement moyen et minimal par nombre). L'ensemble des comparaisons est basé sur trois fichiers dont la description est donnée au tableau 6.1.

Un élément qui s'est révélé préoccupant lors de l'analyse comparative est la légère variation qui est survenue systématiquement d'une séance d'évaluation à une autre. Illustrons ce phénomène à l'aide d'un exemple où en quelques séquences d'évaluation nous mesurons l'évaluation du fichier `canada.txt` par deux fonctions d'évaluation (a) et (b). Sans changer quoi que ce soit, il se pourrait que les résultats (exprimées par le volume traité par milliseconde) ne soient pas identiques d'une séquence à l'autre. C'est la raison pour laquelle il faut avant tout considérer le ratio entre les différentes fonctions et non pas les métriques basées sur les millisecondes. Les résultats de quelques séquences consécutives sont présentés au tableau 6.2. La situation devient plus corsée dans la phase d'optimisation du code où les

Tableau 6.2 : Variation des résultats entre les séquences d'analyse comparative

Sequence	Method	FileName	Min	Ratio	MFloat/s
1	a()	canada.txt	36.9 ms	1.00	2.99
1	b()	canada.txt	4.09 ms	0.12	24.38
2	a()	canada.txt	37.44 ms	1.00	2.97
2	b()	canada.txt	4.53 ms	0.12	24.52
3	a()	canada.txt	36.29 ms	1.00	3.06
3	b()	canada.txt	4.53 ms	0.12	24.53

améliorations microscopiques ne sont pas toujours significatives en ce qui concerne le ratio entre les fonctions. Il faut alors s'en remettre à l'analyse du code *assembler* généré et préférer l'option qui nous semble favorable (par exemple, sélectionner l'option qui génère le plus petit nombre d'instructions).

La réorganisation du code (*refactoring*) peut entraîner une variation des résultats. Nous avons observé une fluctuation allant parfois de 2 à 4 % en ne changeant que la disposition du code.

6.3 Comparaison avec la librairie standard

Une fois la conversion de l'algorithme achevée, il convient de répondre à la question de recherche visant à déterminer si les gains procurés par l'algorithme fast_float sont significatifs en langage C#. Nous avons ainsi mesuré le traitement du contenu des trois fichiers décrits à la section précédente. Les résultats de l'analyse sont spectaculaires (figure 6.1). Pour deux fichiers sur trois, la libraire csFastFloat traite le contenu entre 88 et 89 % moins de temps. Pour le fichier `synthetic.txt`, la librairie csFastFloat arrive à traiter plus de 29,31 millions de nombres par seconde par rapport à 3,11 millions par seconde pour la librairie standard.

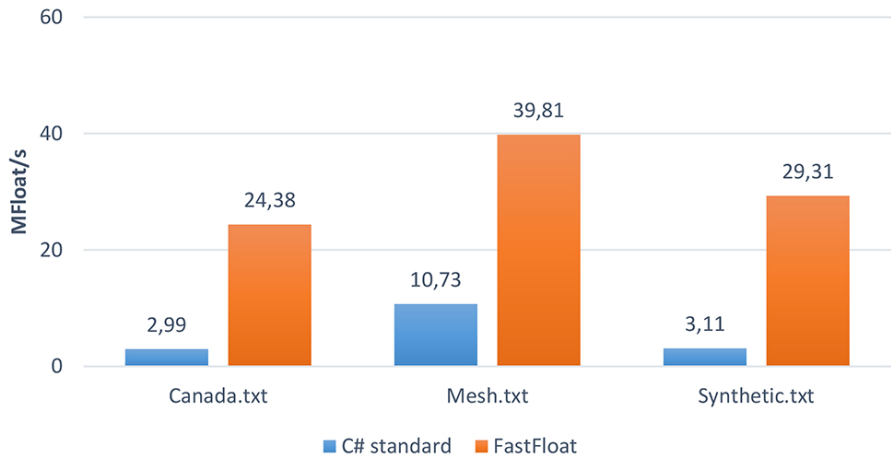


Figure 6.1 : Comparaison avec la librairie standard

À la lumière des résultats obtenus, nous avons soumis à Microsoft une demande de contribution afin que l'algorithme fast_float soit inclus dans la prochaine version du Framework .NET. Notre proposition a été accueillie favorablement. Les experts en charge de l'évolution de la plateforme de développement nous ont demandé de produire un certain nombre d'analyses comparatives supplémentaires. Ils nous ont également questionné au sujet de certains cas particuliers pour obtenir l'assurance qu'ils soient pris en charge par notre algorithme. Cette dernière requête étant adressée par la production des tests unitaires.

La première demande d'analyse qu'ils nous ont formulée concerne la comparaison de la performance pour deux scénarios : l'analyse des nombres encodés sur 8 bits (encodage UTF-8) et l'analyse des nombres avec le format de données `float` (sur 32 bits). Présentés aux figures 6.2 et 6.3, les résultats confirment la supériorité de notre librairie pour les deux scénarios.

Par la suite, ils nous ont demandé d'évaluer la vitesse de traitement pour des nombres comportant une mantisse d'une longueur de 6, 9, 15 et 17 décimales (ces nombres représentant des cas fréquents d'utilisation) pour lesquels la position de

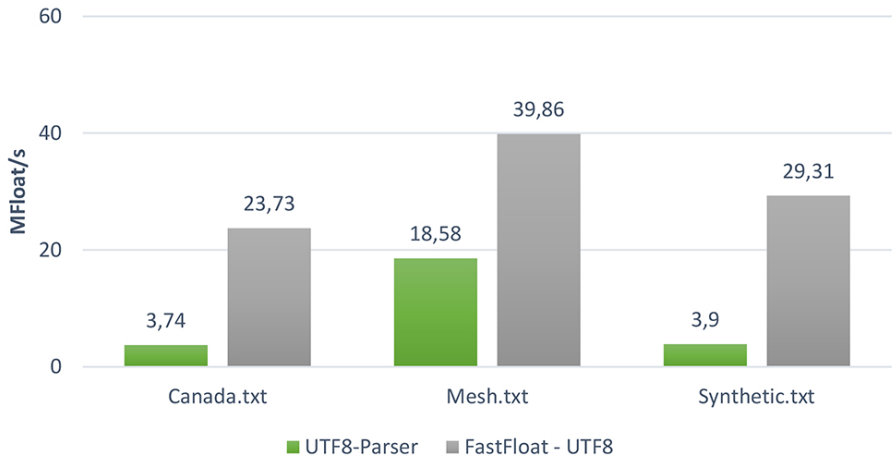


Figure 6.2 : Comparaison avec la librairie standard - UTF8

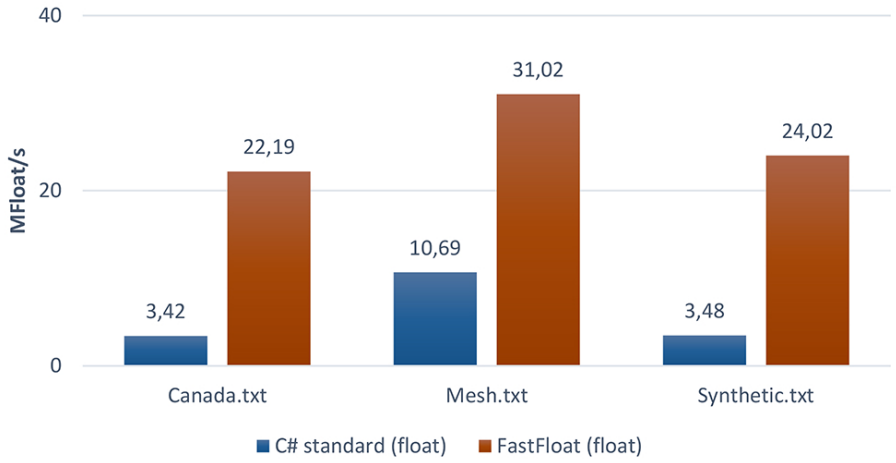


Figure 6.3 : Comparaison avec la librairie standard pour les données sur 32 bits

la virgule est variable (où le ratio partie entière-partie fraction varie). Nous avons assemblé une série de fichiers d'essais où les nombres flottants correspondent à une mantisse d'une taille de 6 à 17 décimales et comparé la performance pour le type de données `double`. Les courbes des volumes traités sont présentées à la figure 6.4.

Finalement, ils nous ont demandé de produire une comparaison pour le traitement des nombres entiers (sans fraction décimale). Nous avons généré un fichier comprenant

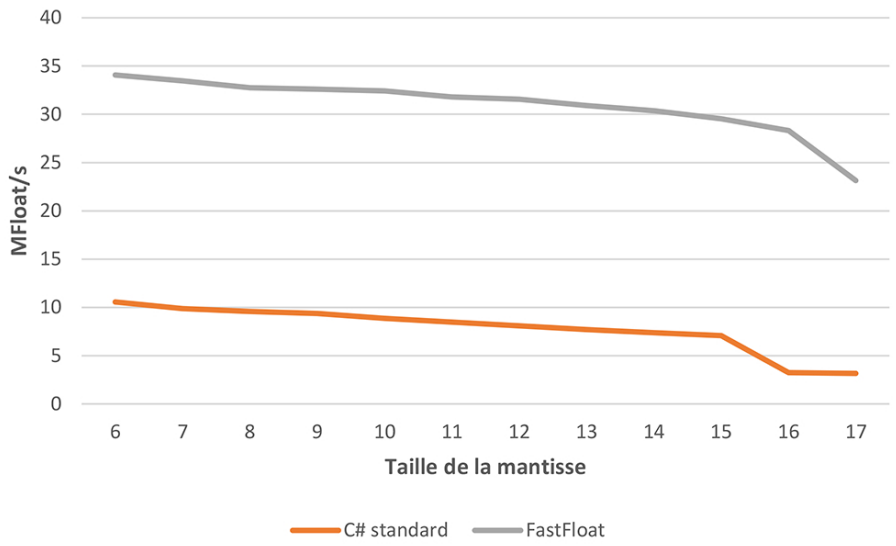


Figure 6.4 : Comparaison avec la librairie standard selon la taille de la mantisse

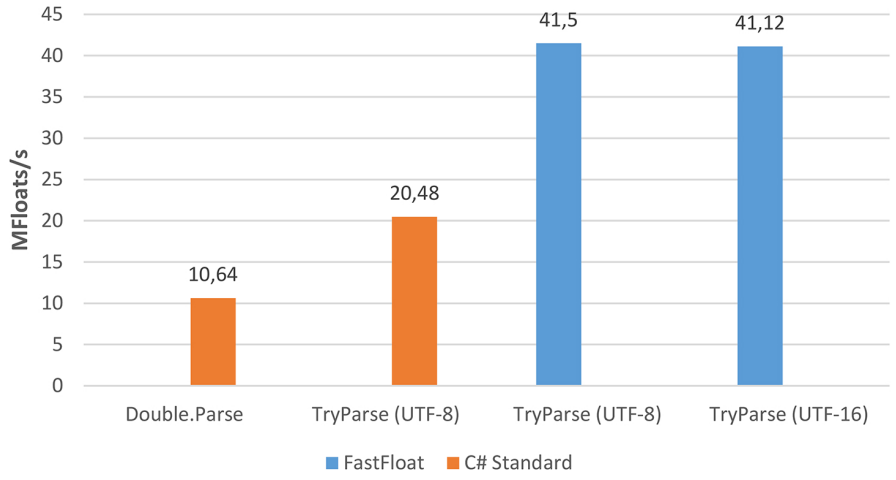


Figure 6.5 : Comparaison avec la librairie standard pour un ensemble de nombres entiers

150 000 nombres entiers aléatoires et comparé la vitesse de traitement pour les deux longueurs d'encodage (UTF-8 et UTF-16). Les résultats sont présentés à la figure 6.5.

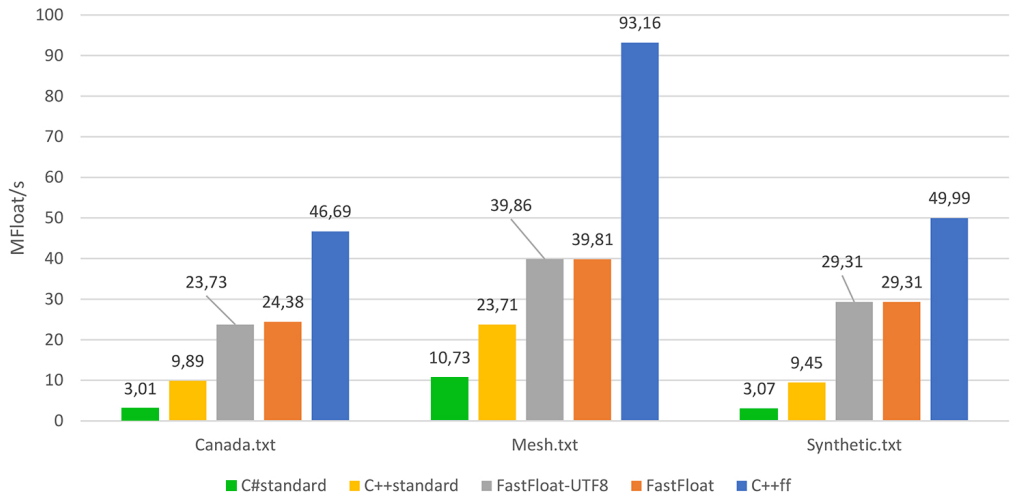


Figure 6.6 : Comparaison des librairies C++ et C#

6.4 Comparaison des versions C++ et C# de l'algorithme fast_float

L'un des objectifs du mémoire est de déterminer dans quelle mesure il est possible d'obtenir avec le langage C# une performance similaire à la librairie fast_float écrite en C++. À l'instar de ce qui avait été fait précédemment, nous avons comparé la vitesse de l'analyse du contenu des trois fichiers `canada.txt`, `mesh.txt` et `synthetic.txt`. Les résultats présentés à la figure 6.6 nous permettent de constater que même si la librairie csFastFloat est beaucoup plus rapide que la librairie standard de Microsoft des deux langages (C# et C++), elle ne permet pas d'atteindre le niveau de performance offert par la librairie fast_float écrite en langage C++. Nous avons tenté d'expliquer les raisons qui justifient ces résultats afin de mettre en place les éléments permettant de s'en approcher le plus près possible.

Avant d'approfondir l'analyse de la différence de performance, nous jugeons qu'il est à propos de mettre en lumière l'écart considérable entre la vitesse de traitement entre les librairies standard respectives pour chaque langage (notées C# standard

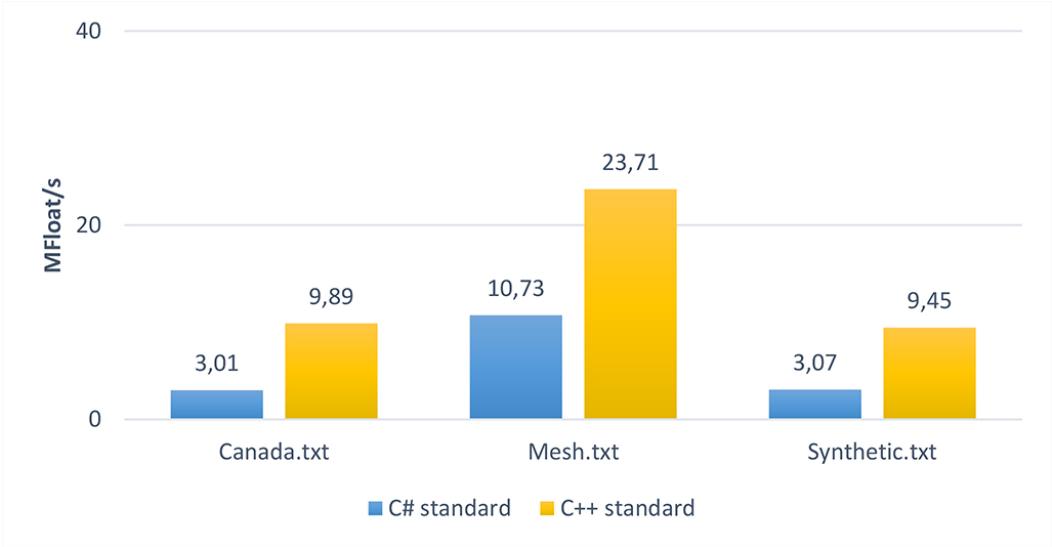


Figure 6.7 : Comparaison de la vitesse de traitement de la librairie standard des langages C# et C++

et C++ standard à la figure 6.7). Pour deux des trois fichiers traités, l'analyse et la conversion des nombres s'effectuent pratiquement trois fois plus rapidement en langage C++. Par conséquent, nous ne sommes pas surpris que la librairie fast_float originale soit environ deux à trois fois plus rapide que son adaptation en langage C#.

La conversion en nombre flottant comporte essentiellement deux étapes : l'analyse de la chaîne et le calcul du nombre (la conversion finale du résultat ne représentant qu'une fraction marginale du temps de traitement). Nous avons approfondi la comparaison entre les deux langages en isolant la portion analyse de la chaîne afin d'établir une base de comparaison n'impliquant pas les opérations arithmétiques complexes. Nous avons ainsi mesuré le volume des chaînes de caractères analysées pour les trois mêmes fichiers (figure 6.8). Comme c'est le cas pour le traitement complet, le volume traité avec le langage C++ est entre deux ou trois fois plus grand.

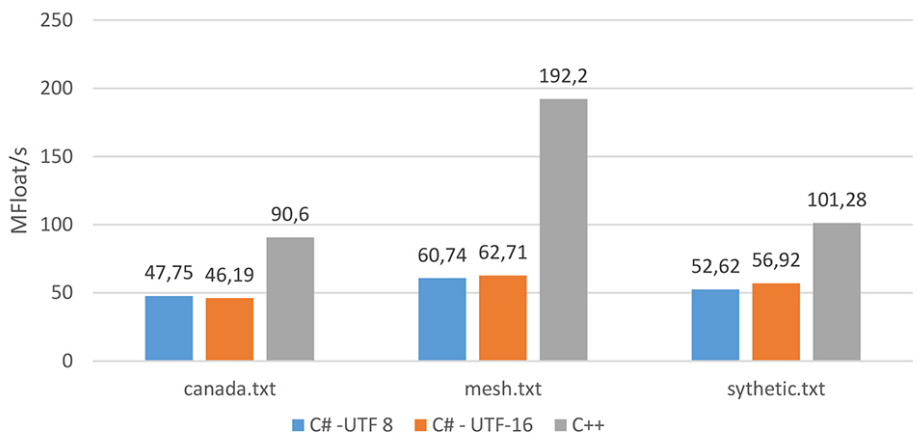


Figure 6.8 : Analyse des chaînes sans calcul arithmétique

Le traitement en boucle d'une chaîne de caractères est dans l'ordre de O(n) où n représente le nombre de caractères. Un des facteurs de succès des travaux de Lemire, c'est qu'il tire avantage d'une particularité des chaînes encodées sur 8 bits afin de détecter et d'évaluer en seulement quelques opérations la présence de 8 chiffres consécutifs situés après la virgule. Cette optimisation permet de réduire le passage dans les boucles de traitement et le nombre d'instructions nécessaires. Nous avons démontré la pertinence de cette optimisation en mesurant le volume traité pour un ensemble de 19 fichiers composés de 150 000 nombres aléatoires ayant un nombre de décimales fixe (allant de 1 à 19 décimales après la virgule [1]). La figure 6.9 illustre clairement l'augmentation du volume traité pour chaque multiple de 8 caractères. Dès les premières ébauches du code, nous avons souhaité inclure cette optimisation dans le cadre de la librairie csFastFloat. Or, en langage C# ce n'est pas aussi simple puisque de base les chaînes de caractères ne sont pas traitées selon l'encodage sur 8 bits.

1. Le seuil de 19 décimales est le nombre maximal de décimales où l'algorithme demeure optimal.

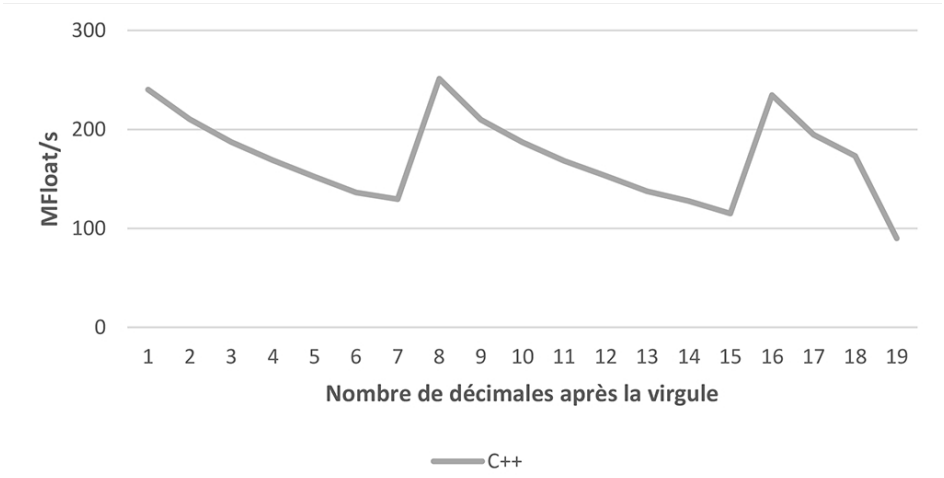


Figure 6.9 : Détection et analyse en groupe de 8 caractères avec C++

Tout au long des travaux du mémoire le code de la librairie a été bonifié à plusieurs reprises. Dès la deuxième version, nous avons décliné les fonctions de traitement pour supporter les deux longueurs d'encodage (8 et 16 bits). Pour le traitement sur 8 bits, nous avions inclus l'optimisation pour la détection et l'évaluation simultanées des 8 caractères. Il restait encore à trouver une solution pour améliorer la vitesse pour les chaînes sur 16 bits... À partir de la version 4.0, nous avons intégré la technologie SIMD et une toute nouvelle fonction d'évaluation en bloc. Cette technologie nous a permis d'atteindre un niveau de performance supérieur pour les chaînes encodées sur 16 bits. À titre comparatif, nous avons illustré à la figure 6.10 la différence du volume analysé avec et sans cette technologie pour les chaînes encodées en 8 et 16 bits. À partir du huitième caractère, on remarque une dégradation constante pour la version sur 16 bits sans l'apport des opérations "vectorisées".

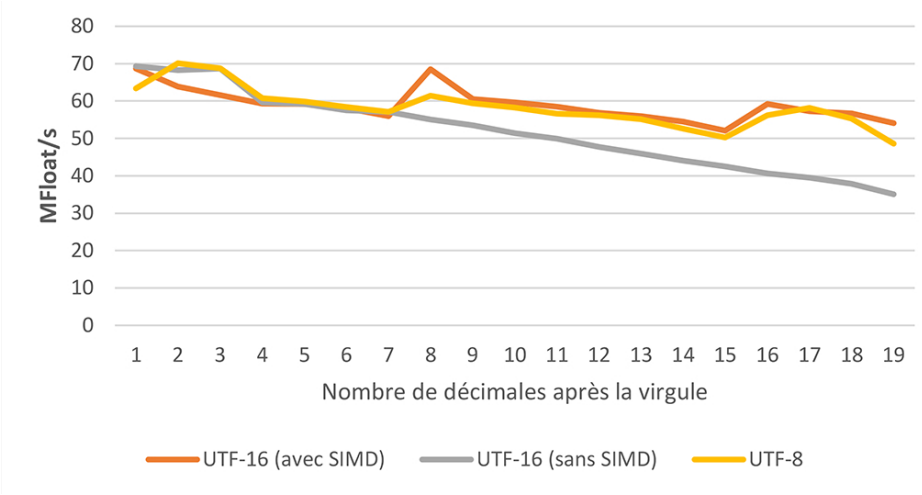


Figure 6.10 : Détection et analyse en groupe de 8 caractères avec C++

6.5 Intégration dans une librairie existante

Une application concrète des fonctions d'analyse des nombres flottants est la lecture et l'analyse d'un fichier CSV (*Comma Separated Values*). À cet effet, il existe des librairies spécialisées permettant de traiter ces fichiers à haut volume. Pour démontrer les bénéfices de l'algorithme, nous avons mesuré les gains qu'il procure en l'intégrant à une librairie existante. Notre choix s'est porté sur la librairie de traitement des fichiers CSVHelper écrite par Close (Close, 2021) puisqu'elle est actuellement l'une des plus rapides et des plus populaires (Verhagen, 2021).

L'intégration dans la librairie CSVHelper s'est avérée aisée puisque son code prévoit l'ajout et la personnalisation des modules d'analyse (*parsers*) pour chaque type de données. L'extrait de code 6.3 en démontre la simplicité.

Extrait de code 6.3 : Intégration à la librairie CSVHelper

```
public  class csFFDoubleConverter : DefaultTypeConverter
{
 // [...]
 public override object ConvertFromString
 (string text, IReaderRow row, MemberMapData memberMapData)
  => csFastFloat.FastDoubleParser.ParseDouble(text);
}
```

La comparaison s'est effectuée selon plusieurs fichiers :

— les fichiers `canada.txt`, `mesh.txt` et `synthetic.txt` utilisés précédemment ;

— les fichiers `w-c-*` contenant respectivement 100 000 et 300 000 occurrences de données sur la population mondiale obtenues par le site de partage de données OpenDataSoft (OPENDATASOFT, 2021).

Les résultats de l'analyse comparative sont présentés dans le tableau 6.3. On remarque que l'algorithme fast_float procure un gain de performance. Pour les trois premiers fichiers (canada, mesh et synthetic), le traitement est jusqu'à 53 % plus rapide. L'analyse comparative porte sur l'ensemble du traitement du fichier (lecture et calcul des nombres binaires). Ceci explique pourquoi nous avons un gain de 53 % au lieu de 89 % comme c'était le cas pour l'analyse précédente. Pour les deux autres fichiers, les gains sont plus modestes (entre 11 % et 13 %). Cela s'explique par la nature et l'étendue des données qu'ils contiennent. Ces données ne possèdent que sept chiffres décimaux significatifs. Tout comme l'algorithme fast_float, la librairie standard emprunte la méthode rapide de Clinger lorsque certaines conditions sont respectées. Par conséquent, la méthode de Lemire (algorithme fast_float) n'est pas aussi souvent mise à contribution pour les fichiers `w-c-*`.

Tableau 6.3 : Résultats de l'analyse avec BenchmarkDotNet (intégration)

Method	FileName	Min	Ratio	MFloat/s
Double.Parse()	canada.txt	83.64 ms	1.00	1.42
FastFloat.ParseDouble()	canada.txt	40.59 ms	0.47	2.95
Double.Parse()	mesh.txt	29.4 ms	1.00	2.71
FastFloat.ParseDouble()	mesh.txt	19.8 ms	0.71	4.02
Double.Parse()	synthetic.txt	111.87 ms	1.00	1.44
FastFloat.ParseDouble()	synthetic.txt	53.40 ms	0.46	3.00
Double.Parse()	w-c-100K.csv	189.44 ms	1.00	1.18
FastFloat.Parse()	w-c-100K.csv	167.09 ms	0.87	1.37
Double.Parse()	w-c-300K.csv	566.87 ms	1.00	1.19
FastFloat.Parse()	w-c-300K.csv	477.05 ms	0.89	1.33

CHAPITRE VII

DISCUSSION ET CONCLUSION

Dans le cadre de ce mémoire, nous avons adapté en langage C# et analysé la performance de l'algorithme fast_float. Les résultats obtenus sont excellents. Sous certaines conditions, la librairie csFastFloat est neuf fois plus rapide que la librairie standard offerte par Microsoft. Par conséquent, nous pouvons répondre à la question de recherche : il est tout à fait possible d'obtenir, en langage C#, un niveau de performance élevé avec l'algorithme fast_float. Nous avons démontré l'impact concret de nos travaux en intégrant notre librairie dans une solution existante de traitement de fichiers à haut volume.

L'atteinte de ce niveau de performance s'explique par plusieurs facteurs. D'abord, l'algorithme lui-même qui est beaucoup plus efficace que celui mis en place dans la version du Framework .NET 5.0. Nous avons tiré profit des fonctions du langage C# qui minimisent l'allocation de la mémoire et en optimisent l'accès. Nous avons également réduit le nombre d'instructions par l'utilisation des opérations vectorisées SIMD. La librairie que nous avons conçue n'est certes pas aussi rapide que sa version en langage C++, mais nous avons analysé et justifié cet écart de performance.

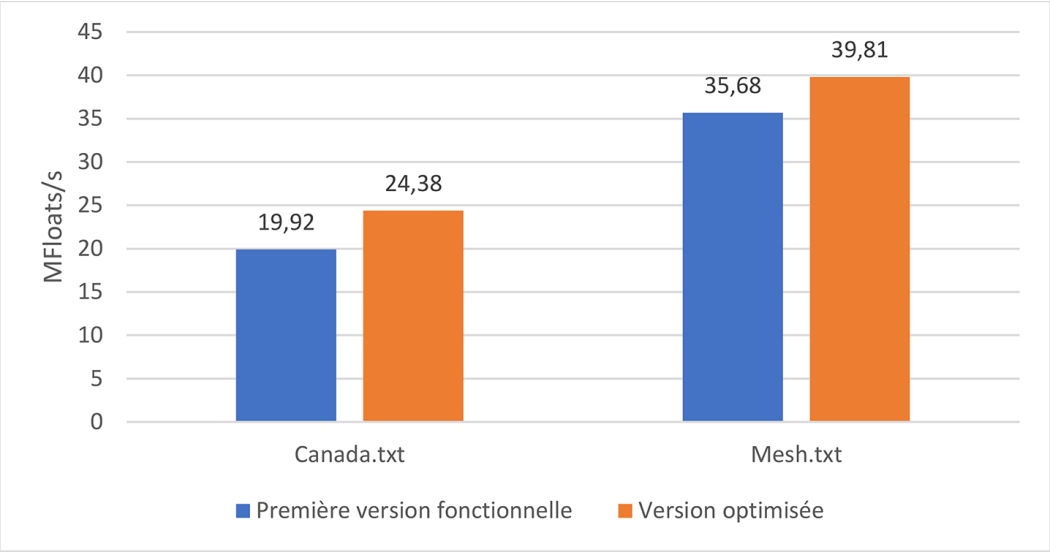


Figure 7.1 : Comparaison de la performance des différentes itérations de la librairie

7.1 Justification de la démarche d'optimisation

Comme nous l'avons décrit dans ce mémoire, le processus itératif qui mène à l'optimisation d'un programme est ardu. Les résultats qui sont présentés aux sections précédentes sont le fruit d'une somme considérable d'heures de travail et il serait intéressant de déterminer à quel point cela valait la peine d'y consacrer autant d'effort. Nous pouvons répondre à cette question, en comparant les performances de la première version "fonctionnelle" mise en place à la toute dernière version optimisée.

Comme nous pouvons le constater à la figure 7.1, l'optimisation des fonctions a permis d'améliorer la performance de 22 % pour le fichier canada.txt et d'environ 11 % pour le fichier mesh. Il importe de mentionner que la première version "fonctionnelle" offrait déjà une performance supérieure à la librairie standard. Cette version comportait déjà certaines des caractéristiques optimisées dont il est question dans ce mémoire.

7.2 Éléments de succès pour l'optimisation de la performance d'un programme écrit en langage C#

Tirées de l'expérience acquise lors de ce mémoire, les recommandations suivantes s'appliquent à la majorité des développements effectués en langage C#. Dans tous les cas, l'objectif est d'influencer positivement le comportement de l'engin d'exécution et du *garbage collector*.

La première recommandation porte sur l'utilisation des outils d'analyse de l'exécution (profilage et analyse comparative). Comme nous l'avons démontré dans la section des résultats, les informations recueillies lors de l'analyse comparative nous permettent d'identifier la performance et l'empreinte en mémoire des fonctions. Il serait avisé de mesurer les différentes fonctions développées tout au long d'un projet, en particulier lors d'un exercice de remaniement (*refactoring*) du code pour s'assurer de conserver le même niveau de performance. Les outils de profilage du code permettent également de connaître le pourcentage du temps de traitement qui est accordé à chacune des fonctions.

L'analyse du code JIT-asm généré par l'utilisation des outils que nous avons présentés fournit de précieuses informations quant à ce qui sera exécuté par le runtime CLR. Un bon exemple consiste au choix des instructions utilisées pour l'accès à la mémoire lors d'une boucle afin de prévenir la vérification systématique des bornes d'un vecteur. Il est possible de réduire significativement les étapes de validation exécutées (*bonds checking*) en recourant à certaines fonctions (comme la fonction `GetArrayDataReference`) ainsi qu'aux pointeurs. En langage C#, ces deux stratégies sont rarement utilisées par les développeurs mais elles s'avèrent cruciales lorsque la performance représente un enjeu.

Utiliser les types de données appropriées accélère et optimise l'accès et l'allocation de la mémoire. Par exemple, préférer les structures aux classes pour certaines

situations réduit l'implication du *garbage collector*. Lorsque possible, l'utilisation du type de données ReadOnlySpan<T> est également à privilégier.

Finalement, une optimisation qui peut s'avérer déterminante lorsqu'elle s'applique est l'utilisation des opérations vectorisées où l'on peut appliquer en une seule instruction une opération sur un bloc de données.

7.3 Perspectives et limites

La conception de la librairie csFastFloat s'est effectuée de façon itérative. Il fallait d'abord s'assurer du comportement des fonctions pour ensuite les optimiser progressivement. Passé un certain point, les bénéfices de ces optimisations sont parfois minces par rapport à l'effort qui doit être investi. Certaines idées ne procurent pas les bénéfices escomptés.

L'un des avantages procurés par le partage de notre code (ainsi que celui de la librairie fast_float originale) sur la plateforme GitHub est l'intérêt et la contribution des programmeurs qui continuent d'affluer depuis la publication. À ce jour, plus d'une centaine de programmeurs se sont intéressés à ce projet de maîtrise. C'est sans compter l'intérêt marqué de la communauté pour les travaux et publications Lemire.

Un exemple récent est l'amélioration significative apportée par un contributeur à la librairie originale. Il a optimisé le code de la fonction de repli utilisée pour les cas où la limite des 19 décimales est atteinte [1]. Bien qu'il s'agisse de cas relativement rares, il pourrait s'avérer pertinent de bonifier notre librairie en fonction de cette nouvelle approche pour les cas d'exception.

---

1. Cette fonction a été adaptée en langage C# mais ne fait pas l'objet de l'analyse détaillée de ce mémoire.

Nous avons reçu une demande de la part des experts de la compagnie Microsoft visant à décliner l'algorithme pour un troisième type de données (`half`). Nous y sommes parvenus très facilement lors des travaux d'intégration. Un élément à surveiller sera le dévoilement prochain de la version 7.0 du Framework .NET. Mesurer la performance de notre librairie dans ce nouvel environnement sera très intéressant. Nous évaluerons alors la possibilité de contribuer à nouveau au code du framework.

Au niveau de l'algorithme lui-même, il serait possible de vérifier dans quelles proportions nous pourrions étendre la vectorisation des opérations d'analyse des caractères. Actuellement, seule la fraction décimale est lue en bloc de 8 caractères. Lire la chaîne d'un trait et détecter le séparateur décimal (point ou virgule) pourraient représenter un gain. Encore faudrait-il que le nombre d'opérations requises pour lire et traiter le bloc (en tenant compte du séparateur décimal) soit inférieur au nombre d'opérations nécessaires pour lire et convertir chaque caractère un à un. Pour la même raison, il n'est peut-être pas avantageux d'utiliser les opérations vectorisées pour des chaînes dont la longueur se situe sous un certain seuil (par exemple une chaîne de 1 à 6 caractères).

Des travaux futurs pourront évaluer une approche appelant une librairie C++ à partir de C#. Le passage d'un segment de mémoire d'un langage à l'autre, étant donné une gestion automatique de la mémoire en C#, nécessite certaines précautions et pourrait engendrer des coûts au niveau de la performance et annuler les gains procurés par l'exécution d'un code C++ réputé plus rapide.

7.4 Conclusion

Depuis sa publication dans la communauté GitHub et sur le site NuGet.org, notre librairie a suscité l'intérêt. Elle s'intègre facilement à tout programme nécessitant

une fonction performante de lecture des nombres flottants. Au moment d'écrire ce mémoire, la librairie avait déjà fait l'objet de plus de 1 800 téléchargements.

L'impact des travaux réalisés dans le cadre de ce mémoire est concret. Nous avons partagé notre mise en oeuvre et nos résultats avec les ingénieurs de Microsoft afin que nos travaux soient intégrés au Framework .NET. Reçue très favorablement, notre proposition de contribution a été acceptée et les travaux d'intégration se sont conclus avec succès en février 2022. Ainsi, dès la publication de la version 7.0 du Framework .NET, ce sont des milliers de programmeurs et des millions d'utilisateurs qui seront en mesure de bénéficier des fruits de notre travail.

RÉFÉRENCES

Bibliographie

Adams, U. (2018). « Ryu: fast float-to-string conversion ». *ACM SIGPLAN Notices* 53.4, p. 270-282. doi : 10.1145/3296979.3192369.

Ashenhurst, R. L. et N. Metropolis (1959). « Unnormalized Floating Point Arithmetic ». *Journal of the ACM* 6.3, p. 415-428. doi : 10.1145/320986.320996.

Barrett, E. et al. (2017). « Virtual machine warmup blows hot and cold ». *Proceedings of the ACM on Programming Languages* 1 (OOPSLA), p. 1-27. doi : 10.1145/3133876.

Canfora, G. et al. (2014). « How changes affect software entropy: an empirical study ». *Empirical Software Engineering* 19.1, p. 1-38. doi : 10.1007/s10664-012-9214-z.

Clinger, W. D. (1990). « How to read floating point numbers accurately ». *ACM SIGPLAN Notices* 25.6, p. 92-101. doi : 10.1145/93548.93557.

Coonen, J. T. (1980). « Special Feature an Implementation Guide to a Proposed Standard for Floating-Point Arithmetic ». *Computer* 13.1, p. 68-79. doi : 10.1109/MC.1980.1653344.

Cowlishaw, M. et al. (2001). « A decimal floating-point specification ». Dans *Proceedings 15th IEEE Symposium on Computer Arithmetic. ARITH-15 2001*. 15th IEEE Symposium on Computer Arithmetic. ARITH-15 2001. Vail, CO, USA : IEEE Comput. Soc, p. 147-154. doi : 10.1109/ARITH.2001.930114.

Dinechin, F. de et al. (2019). « Posits: the good, the bad and the ugly ». Dans *Proceedings of the Conference for Next Generation Arithmetic 2019*. CoNGA'19. Singapore, Singapore : Association for Computing Machinery, p. 1-10. doi : 10.1145/3316279.3316285.

Elias, P. (1975). « Universal codeword sets and representations of the integers ». *IEEE Transactions on Information Theory* 21.2, p. 194-203. doi : 10.1109/TIT.1975.1055349.

Gay, D. M. (1990). « Correctly Rounded Binary-Decimal and Decimal-Binary Conversions ». Dans. AT&T Bell Laboratories Numerical Analysis Manuscript 90-10.

Goldberg, D. (1991). « What every computer scientist should know about floating-point arithmetic ». *ACM Computing Surveys* 23.1, p. 5-48. doi : 10.1145/103162.103163.

Gustafson, J. (2017). « Beating Floating Point at its Own Game: Posit Arithmetic ». *Supercomputing Frontiers and Innovations* 4.2. doi : 10.14529/jsfi170206.

IEEE (1985). « IEEE Standard for Binary Floating-Point Arithmetic ». *ANSI/IEEE Std 754-1985*, p. 1-20. doi : 10.1109/IEEESTD.1985.82928.

Khorikov, V. (2020). *Unit testing: principles, practices, and patterns*. OCLC: on1114498320. Shelter Island, NY : Manning. 282 p. isbn : 9781617296277.

Kokosa, K. (2018). *Pro .NET Memory Management: For Better Code, Performance, and Scalability*. 1st ed. 2018. Berkeley, CA : Apress : Imprint: Apress. 1 p. isbn : 9781484240274.

Lemire, D. (2021b). « Number parsing at a gigabyte per second ». *Software: Practice and Experience* 51.8, p. 1700-1727. doi : 10.1002/spe.2984.

Lindstrom, P., S. Lloyd et J. Hittinger (2018). « Universal coding of the reals: alternatives to IEEE floating point ». Dans *Proceedings of the Conference for Next Generation Arithmetic*. CoNGA '18. Singapore, Singapore : Association for Computing Machinery, p. 1-14. doi : 10.1145/3190339.3190344.

Loitsch, F. (2010). « Printing floating-point numbers quickly and accurately with integers ». *ACM SIGPLAN Notices* 45.6, p. 233-243. doi : 10.1145/1809028.1806623.

Muller, J.-M. et al. (2010). *Handbook of Floating-Point Arithmetic*. Boston : Birkhäuser Boston. doi : 10.1007/978-0-8176-4705-6.

Steele, G. L. et J. L. White (1990). « How to print floating-point numbers accurately ». Dans *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*. PLDI '90. White Plains, New York, USA : Association for Computing Machinery, p. 112-126. doi : 10.1145/93542.93559.

Tagliavini, G. et al. (2018). « A transprecision floating-point platform for ultra-low power computing ». Dans *2018 Design, Automation Test in Europe Conference Exhibition*, p. 1051-1056. doi : 10.23919/DATE.2018.8342167.

Webographie

Akinshin, A. (2021). *BenchmarkDotNet*. Repéré le 15 nov. 2021 à https://benchmarkdotnet.org/.

Godbolt, M. (2021). *Compiler Explorer*. Repéré le 15 nov. 2021 à https://godbolt.org/.

Gordon, S. (2019). *Introduction to Benchmarking C# Code with Benchmark .NET*. Steve Gordon - Code with Steve. Repéré le 15 nov. 2021 à https://www.stevejgordon.co.uk/introduction-to-benchmarking-csharp-code-with-benchmark-dot-net.

Gustafson (2021a). *Improved Version of The Great Debate @ARITH23 John Gustafson and William Kahan* [Fichier vidéo]. Repéré le 15 nov. 2021 à https://www.youtube.com/watch?v=KEAKYDyUua4.

GUSTAFSON (2021b). *Stanford Seminar: Beyond Floating Point: Next Generation Computer Arithmetic* [Fichier vidéo]. Repéré le 15 nov. 2021 à https://www.youtube.com/watch?v=aP0Y1uAA-2Y.

IBM (2011). *IBM Research | Cloud and Hardware Quality - Test Suite for IEEE 754R Compliance - Version 1.1*. Repéré le 15 nov. 2021 à https://www.research.ibm.com/haifa/projects/verification/fpgen/test_suite_download.shtml.

INTEL (2021). *Fix Performance Bottlenecks with Intel® VTune™ Profiler*. Intel. Repéré le 15 nov. 2021 à https://www.intel.com/content/www/us/en/develop/tools/oneapi/components/vtune-profiler.html.

INTERNATIONAL ORGANIZATION FOR STANDARDIZATION (2021). *Programming languages — C++ (ISO/IEC 14882:1998)*. ISO. Repéré le 15 nov. 2021 à https://www.iso.org/standard/25845.html.

JONES, M. (2021). *Test-driven development walkthrough - Visual Studio*. Repéré le 15 nov. 2021 à https://docs.microsoft.com/en-us/visualstudio/test/quick-start-test-driven-development-with-test-explorer.

KAHAN, W. (1981). *Why do we need a floating-point arithmetic standard?* Repéré le 15 nov. 2021 à http://people.eecs.berkeley.edu/~wkahan/ieee754status/why-ieee.pdf.

– (2021). *Commentary on "THE END of ERROR — Unum Computing"*. Repéré le 15 nov. 2021 à https://web.archive.org/web/20160801162357/https://people.eecs.berkeley.edu/~wkahan/EndErErs.pdf.

MAHESH, C. (2021). *Why Learn C#*. Repéré le 15 nov. 2021 à https://www.c-sharpcorner.com/article/why-learn-c-sharp/.

MICROSOFT (2021a). *char type - C# reference*. Repéré le 15 nov. 2021 à https://docs.microsoft.com/fr-ca/dotnet/csharp/language-reference/builtin-types/char.

MICROSOFT (2021b). *Classes et méthodes génériques*. Repéré le 15 nov. 2021 à https://docs.microsoft.com/fr-fr/dotnet/csharp/fundamentals/types/generics.

– (2021c). *Exceptions and Performance - Framework Design Guidelines*. Repéré le 15 nov. 2021 à https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/exceptions-and-performance.

– (2021d). *Fichiers d'en-tête (C++)*. Repéré le 15 nov. 2021 à https://docs.microsoft.com/fr-fr/cpp/cpp/header-files-cpp.

– (2021e). *ReadOnlySpan<T> Struct (System)*. Repéré le 15 nov. 2021 à https://docs.microsoft.com/en-us/dotnet/api/system.readonlyspan-1.

– (2021f). *Templates (C++)*. Repéré le 15 nov. 2021 à https://docs.microsoft.com/en-us/cpp/cpp/templates-cpp.

– (2021g). *Types numériques à virgule flottante - Référence C#*. Repéré le 15 nov. 2021 à https://docs.microsoft.com/fr-fr/dotnet/csharp/language-reference/builtin-types/floating-point-numeric-types.

OPENDATASOFT (2021). *World Cities Population*. Repéré le 15 nov. 2021 à https://public.opendatasoft.com/explore/dataset/worldcitiespop/table/?disjunctive.country&sort=population.

PETERSEN, J. V. (2021). *10 Reasons Why Unit Testing Matters*. Repéré le 15 nov. 2021 à https://www.codemag.com/Article/1901071/10-Reasons-Why-Unit-Testing-Matters.

SEVERANCE, C. (1998). *An Interview with the Old Man of Floating-Point*. Repéré le 15 nov. 2021 à https://people.eecs.berkeley.edu/~wkahan/ieee754status/754story.html.

SHCHEKIN, A. (2021). *SharpLab*. Sharplab.io. Repéré le 15 nov. 2021 à https://sharplab.io/.

SKLENAR, Y. (2021). *INTRODUCTION TO SIMULA*. Repéré le 15 nov. 2021 à http://staff.um.edu.mt/jskl1/talk.html.

STROUSTRUP, B. (2021a). *Bjarne Stroustrup's Homepage*. Repéré le 15 nov. 2021 à https://www.stroustrup.com/.

– (2021b). *The Design of C++* [Fichier vidéo]. Repéré le 15 nov. 2021 à https://www.youtube.com/watch?v=69ed0m889V4.

TAO, N. (2021b). *The Eisel-Lemire ParseNumberF64 Algorithm*. nigeltao.github.io. Repéré le 15 nov. 2021 à https://nigeltao.github.io/blog/2020/eisel-lemire.html.

TIOBE (2021). *Index for September 2021*. index | TIOBE - The Software Quality Company. Repéré le 15 nov. 2021 à https://www.tiobe.com/tiobe-index/.

VERHAGEN, J. (2021). *The fastest CSV parser in .NET | Joel Verhagen*. Repéré le 15 nov. 2021 à https://www.joelverhagen.com/blog/2020/12/fastest-net-csv-parsers.

WIKIPEDIA (2021a). *Floating-point arithmetic*. Wikipedia. Page Version ID: 1011839130. Repéré le 15 nov. 2021 à https://en.wikipedia.org/w/index.php?title=Floating-point_arithmetic&oldid=1011839130.

– (2021b). *Programming language generations*. Wikipedia. Page Version ID: 1037801800. Repéré le 15 nov. 2021 à https://en.wikipedia.org/w/index.php?title=Programming_language_generations&oldid=1037801800.

– (2021c). *Tapered floating point*. Wikipedia. Page Version ID: 1032192473. Repéré le 15 nov. 2021 à https://en.wikipedia.org/w/index.php?title=Tapered_floating_point&oldid=1032192473.

– (2021d). *Test-driven development*. Wikipedia. Page Version ID: 1043687706. Repéré le 15 nov. 2021 à https://en.wikipedia.org/w/index.php?title=Test-driven_development&oldid=1043687706.

– (2021e). *Turing-complet*. Wikipedia. Page Version ID: 183546511. Repéré le 15 nov. 2021 à https://fr.wikipedia.org/w/index.php?title=Turing-complet&oldid=183546511.

Wikipedia (2021f). *Z3 (computer)*. Wikipedia. Page Version ID: 1021248815. Repéré le 15 nov. 2021 à https://en.wikipedia.org/w/index.php?title=Z3_(computer)&oldid=1021248815.

– (2021g). *William Kahan*. Repéré le 15 nov. 2021 à https://en.wikipedia.org/wiki/William_Kahan.

Dépôts GitHub

Abraham, H. (2021). *numerics/strtod at master · ahrvoje/numerics*. Repéré le 15 nov. 2021 à https://github.com/ahrvoje/numerics.

Adams, U. (2021). *Converts floating point numbers to decimal strings*. Repéré le 15 nov. 2021 à https://github.com/ulfjack/ryu.

Austin, C. (2021). *SAJSON - Lightweight, extremely high-performance JSON parser for C++11*. Repéré le 15 nov. 2021 à https://github.com/chadaustin/sajson.

Close, J. (2021). *CsvHelper*. Repéré le 15 nov. 2021 à https://github.com/JoshClose/CsvHelper.

Google (2021a). *Abseil Common Libraries (C++)*. Repéré le 15 nov. 2021 à https://github.com/abseil/abseil-cpp.

– (2021b). *double-conversion*. Repéré le 15 nov. 2021 à https://github.com/google/double-conversion.

Lemire, D. (2021a). *fast_float number parsing library: 4x faster than strtod*. Repéré le 15 nov. 2021 à https://github.com/fastfloat/fast_float.

– (2021c). *simdjson : Parsing gigabytes of JSON per second*. Repéré le 15 nov. 2021 à https://github.com/simdjson/simdjson.

NetLib (2021). *C++ Network Library*. Repéré le 15 nov. 2021 à https://github.com/cpp-netlib/cpp-netlib.

Tao, N. (2021a). *Test Data*. Repéré le 15 nov. 2021 à `https://github.com/nigeltao/parse-number-fxx-test-data`.

Tecent (2021). *A fast JSON parser/generator for C++ with both SAX/DOM style API*. Repéré le 15 nov. 2021 à `https://github.com/Tencent/rapidjson`.