université PARIS-SACLAY

Techniques avancées de génération de code pour la performance

Advanced techniques for high performance code generation

Thèse de doctorat de l'université Paris-Saclay

École doctorale n°580: sciences et technologies de l'information et de la communication (STIC)
Spécialité de doctorat: Informatique
Graduate School: Informatique et sciences du numérique. Référent: Faculté des sciences d'Orsay

Thèse préparée dans l'unités de recherche **Université Paris-Saclay, CNRS, Laboratoire interdisciplinaire des sciences du numérique, 91405, Orsay, France**, sous la direction de **Joel FALCOU**, Maître de conférences

Thèse soutenue à Paris-Saclay, le JJ mois AAAA, par

Jules PÉNUCHOT

Composition du jury

Membres du jury avec voix délibérative

Christine PAULIN-MOHRING Professeure des universités, Université Paris-Saclay	Présidente
David HILL Professeur des universités, Université Clermont-Auvergne	Rapporteur & Examinateur
Thierry GÉRAUD Professeur des universités, EPITA Research Laboratory	Rapporteur & Examinateur
Christine PAULIN-MOHRING Professeure des universités, Université Paris-Saclay	Examinatrice
Amina GUERMOUCHE Maîtresse de conférences, Affiliation	Examinatrice

THESE DE DOCTORAT

NNT : 2020UPASA001

Remerciements

Merci à **Christine Paulin** d'assurer la présidence du jury, aux rapporteurs **David Hill** et **Thierry Géraud**, ainsi qu'à **Amina Guermouche** d'examiner la thèse avec Christine Paulin.

Je tiens également à faire part de ma reconnaissance envers **Christine Paulin-Mohring** pour la création du Magistère d'Informatique qui m'a permis de découvrir la recherche dans un cadre très privilégié.

Merci également à **Jean-Thierry Lapresté** et **Daniel Étiemble** pour leurs relectures.

Merci, bien sûr, à **Joël Falcou**, pour son accompagnement bienveillant, valorisant, et dévoué depuis mon premier stage de recherche.

Merci à **Amal Khabou**, dont l'aide m'a été précieuse pour la préparation de ma première conférence.

Merci à **Hartmut Kaiser**, pour son accueil à LSU, **Adrian et Cory Lemoine**, qui m'ont aidé à traverser un événement difficile et marquant pendant mon séjour, et bien sûr **au reste de l'équipe Ste||ar** pour les amitiés que j'ai pu y nouer, et la gentillesse de chaque personne que j'y ai rencontrée.

Merci à **Paul Keir**, pour sa collaboration amicale et bienveillante.

Merci à **mes parents**, pour m'avoir transmis la passion de l'informatique, pour m'avoir soutenu à tout âge, y compris pendant mes années de thèse.

Antoine Lanco et Alexandrina Korneva, pour leur camaraderie, les décorations de fêtes, les fêtes dans les champs, et les champs de pizzas.

Marie Debard, pour son soutien moral lorsque je m'inquiétais pour le financement de ma thèse.

université PARIS-SACLAY
ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre: Techniques avancées de génération de code pour la performance
Mots clés: Métaprogrammation, compilation, C++, HPC, parallélisme, constexpr, SIMD, optimisation, programmation générique, programmation générative, templates, langage dédié

Résumé: En réponse à la demande croissante de puissance de calcul, les constructeurs de matériel proposent de nouvelles architectures parallèles de très différentes natures, et ayant chacune leurs propres API et modèle de programmation.

Cela rend les applications haute performance plus complexes à développer avec des méthodes de programmation traditionnelles, d'autant plus lorsque plusieurs architectures sont ciblées.

En réponse à cette demande, des bibliothèques exploitant la génération de code à la compilation furent développées pour faciliter la programmation haute performance avec des abstractions de haut niveau tels que les langages dédiés externes.

Dans cette thèse, nous explorons en premier temps les techniques de métaprogrammation existant à travers un large panel de langages, puis nous évaluons les techniques actuelles de métaprogrammation de templates C++ dans le cadre de la génération de noyaux de calculs de type BLAS.

Nous couvrons ensuite de nouvelles techniques de métaprogrammation basées sur l'exécution de code C++ à la compilation en implémentant des parsers pour deux langages: Brainfuck, et un langage appelé Tiny Math Language (TML). Le parser Brainfuck est fourni avec plusieurs backends de génération de code pour étudier différentes techniques de génération de code, tandis que TML est utilisé pour étudier la génération de code en utilisant des bibliothèques portables haute performance.

Pour évaluer l'impact de ces diverses techniques de métaprorgammation, nous proposons une nouvelle méthodologie de benchmarking qui exploite le profiler intégré de Clang, ce qui permet l'analyse de performance en temps de compilation au-delà des mesures de performance en boîte noire.

université PARIS-SACLAY

ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Title: Advanced techniques for high performance code generation
Keywords: Metaprogramming, compilation, C++, HPC, parallelism, constexpr, SIMD, optimization, generic programming, generative programming, templates, domain-specific language

Abstract: As a response to an increasing demand for computing performance, hardware manufacturers propose new parallel hardware architectures of very different nature, each with their own API and programming model. This makes high performance applications more complex to develop with traditional programming practices, especially when multiple architectures are targeted.

As a response, libraries leveraging compile-time code generation were developed to facilitate portable high performance programming with high level abstractions such as Domain-Specific Embedded Languages (DSELs).

In this thesis, we first explore existing metaprogramming practices across a wide variety of languages, then evaluate current C++ template metaprogramming techniques in the framework of BLAS-like kernel generation.

We then cover new metaprogramming techniques based on compile-time C++ code execution by implementing compile-time parsers for two languages: Brainfuck, and a language called Tiny Math Language (TML). The Brainfuck parser is provided with several code generation backends to study and compare different code generation techniques, whereas TML is used to study code generation using portable high performance computing libraries.

In order to assess the compile-time impact of these various metaprogramming techniques, we propose a new benchmarking methodology that uses Clang's built-in profiler, enabling compile-time performance scaling analysis beyond black-box benchmarking.

Contents

Techniques avancées de génération de code pour la performance

Résumé en Français

Depuis le début de l'informatique les architectures des ordinateurs n'ont cessé de croitre en complexité. Le nombre de transistor dans ces derniers double encore aujourd'hui tous les 3 ans, et la fin de ce doublement quasi-exponentiel n'est pour le moment pas en vue. En revanche, la capacité de traitement séquencielle a connu un brusque ralentissement avec la fin de l'augmentation des fréquences des processeurs: on parle de la fin du «free lunch».

La conception des processeurs a donc évolué vers des architectures parallèles, qui permettent de découper des traitements en sous-tâches pour les faire exécuter simultanément par plusieurs unités de traitement. La nature et la granularité du découpage peut varier selon les architectures:

- **Le modèle SIMD** consiste à avoir, au sein d'un cœur de processeur, des registres vectoriels permettant d'exécuter **une instruction** sur plusieurs valeurs dans un seul registre, de manière simultanée.

- **Le modèle multi-cœur** permet à plusieurs cœurs indépendants de fonctionner en parallèle. Ces cœurs partagent la même mémoire dans le cas des architectures à mémoire unifiée, mais il est également possible d'avoir plusieurs processeurs indépendants ne partageant pas de mémoire et nécessitant donc d'effectuer des transferts de l'un à l'autre dans le cas des architectures à mémoire non unifiée.

- **Les processeurs graphiques (GPU)** permettent d'exécuter des «warps» en parallèle, à condition que ceux-ci soient toujours synchrones. Dans le cas où des warps se désynchronisent, ces derniers doivent s'attendre avant de poursuivre leur exécution parallèle. Ce modèle est très proche du modèle SIMD.

De nombreuses manières d'exécuter du code de manière parallèle existent, et peuvent co-habiter au sein d'un même ordinateur, ce dernier pouvant faire partie d'un cluster.

La diversité des architectures nécessite de nouvelles adaptations pour que les applications dites «haute performance» puissent s'exécuter le plus rapidement possible sur le plus d'architectures possibles. De nouvelles techniques de programmation ont vu le jour, permettant d'écrire des programmes

dont le code peut être spécialisé lors de la compilation en fonction de données connues lors de la compilation; c'est le principe de l'évaluation partielle. D'autres techniques permettent de manipuler des programmes en entrée comme en sortie lors de la compilation, nous parlerons dans ce cas de **métaprogrammation**.

La métaprogrammation permet donc de compiler des programmes en exécutables optimisés pour des architectures ciblées. On retrouve ces techniques cette pratique dans de nombreux langages, l'exemple historique le plus notable étant Lisp, paru en 1960, dont le système de macros permet d'utiliser l'intégralité du langage Lisp pour générer des programmes Lisp. Le langage C, lui, permet d'injecter des tokens lors de la phase de preprocessing qui précède l'analyse syntaxique du code. En C++, des techniques de métaprogrammation plus riches basées sur son système de **templates** permet de définir des structures et fonctions paramétriques, dont les paramètres (des types ou des valeurs) sont connus à la compilation. Le langage évoluera par la suite pour permettre d'évaluer du code C++ à la compilation pour construire des paramètres de templates.

Ces techniques, en plus des aspects «haute performance» du langage C++, on fait naître un écosystème de bibliothèques métaprogrammées permettant d'optimiser automatiquement des programmes pour des architectures très diverses: les architectures SIMD, les processeurs multi-coeurs, les GPU, ou tout autre type d'accélérateur programmable.

Certaines de ces bibliothèques utilisent la métaprogrammation pour proposer des langages dédiés, reposant la plupart du temps sur la surcharge d'opérateurs pour implémenter des langages basés sur la syntaxe C++. D'autres, comme Compile Time Regular Expressions, utilisent la métaprogrammation de templates pour implémenter des analyseurs syntaxiques permettant de reconnaître des langages arbitraires.

Depuis peu, la mémoire dynamique peut être utilisée dans les fonctions exécutées à la compilation d'un programme C++. Nous souhaitons étudier l'impact de cette nouvelle fonctionnalité sur l'implémentation d'analyseurs syntaxiques de langages arbitraires, autant sur la conception des analyseurs syntaxiques que sur leur performance en temps de compilation. Comment peut-on exploiter la mémoire dynamique dans l'évaluation de code C++ à la compilation pour l'intégration de langages dédiés de syntaxes arbitraires pour la programmation «haute performance» ?

Génération de code SIMD performant

Pour répondre à ces questions, nous commencerons d'abord par montrer l'efficacité de la métaprogrammation C++ pour la programmation «haute performance» à travers l'implémentation d'une version générique d'une routine d'algèbre linéaire très répandue: la multiplication matrice-vecteur, avec une

matrice en disposition «column-major».

L'implémentation repose sur une couche d'abstraction pour les registres et instructions SIMD permettant de connaître leur taille à la compilation. En exploitant cette donnée, nous pouvons générer des noyaux de calcul s'adaptant à la taille de ces derniers pour assurer un niveau de performance très élevé pour une multitude de jeux d'instructions SIMD: SSE4.2 et AVX2 pour x86, et NEON pour ARM.

Nous comparons ensuite les performances du code généré avec des implémentations en assembleur déclinées pour chaque micro-architecture fournies par la bibliothèque OpenBLAS en l'appliquant sur des matrices de tailles allant de 4×4 à 512×512. Malgré des résultats peu concluants en SSE4.2, les résultats se montrent très positifs en AVX2 où nous obtenons des résultats supérieurs ou équivalents à OpenBLAS. Quant à ARM, les résultats nous y obtenons des performances supérieures à celles d'OpenBLAS quelle que soit la taille de la matrice.

Ce chapitre a fait l'objet d'une publication scientifique à la conférence internationale **High Performance Computation & Simulation 2018**.

Nouvelle méthodologie pour l'analyse des temps de compilation

Nous proposons ensuite une nouvelle méthodologie pour l'analyse des temps de compilation des métaprogrammes C++, ainsi que des outils implémentant cette méthodologie de manière reproductible. Cette méthodologie repose sur l'utilisation des données de profiling de Clang pour permettre des analyses ciblées, et permet de réaliser des études de temps de compilation sur des métaprogrammes paramétrables, et donc d'observer les temps de compilation en fonction de la taille de ces derniers.

Le projet d'outillage **ctbench** qui l'accompagne repose sur CMake comme langage d'interface principal car il est très largement utilisé au sein de la communauté C++. Il permet de générer des graphes dans différents formats, y compris vectoriels, avec un système de paramétrage lui permettant de cibler et grouper des catégories d'événements dans le processus de compilation. Le moteur de génération des graphes est modulaire et permet l'ajout de nouveaux types de graphes via une interface de programmation clairement documentée. Il s'accompagne d'une bibliothèque C++ pour exploiter les hiérarchies de fichiers générées par **ctbench**.

Cette nouvelle méthodologie permet de mieux comprendre l'impact des techniques de métaprogrammation sur les temps de compilation en permettant d'observer finement chaque étape de la compilation étant donné que les mesures du profiler de Clang sont accompagnées de métadonnées relatives aux symboles C++ traités lors de la compilation.

Ce chapitre a fait l'objet d'une publication scientifique au **Journal of Open Source Software**, et d'une présentation à **CPPP 2021**.

Des nouvelles méthodes pour la génération de code

C++ 20 a rendu possible l'allocation de mémoire dynamique dans les programmes C++ exécutés à la compilation. Toutefois, la mémoire allouée ne peut être utilisée directement comme paramètre de template et il faut donc trouver des solutions pour exploiter son contenu autrement.

Une première solution consiste à passer, pour chaque élément contenu dans une structure dynamique, une fonction qui renvoie ledit élément. Cette option a un coût car les résultats intermédiaires ne peuvent pas être stockés, ainsi la fonction générant la structure devra être rappelée pour chaque élément de la structure. Cette manière de passer les éléments d'une structure arbitraire en paramètre de template est baptisée «pass by generator», et permet de générer du code à partir d'une structure dynamique sans nécessiter de représentation intermédiaire. Elle peut toutefois être utilisée pour générer une représentation intermédiaire sous forme d'arborescence de templates de types (appelées «expression templates»), ce qui est commun dans les bibilothèques de métaprogrammation C++.

Une autre solution consiste à sérialiser une structure dynamique pour la convertir en tableau de taille fixe, en remplaçant les pointeurs vers les sous-éléments par des indices pointant à l'intérieur du tableau dans lequel la structure est sérialisée. De cette manière, la structure peut être transformée en tableau de taille fixe ne contenant aucun pointeur vers de la mémoire dynamique allouée à la compilation, et ce tableau peut être utilisé directement en paramètre de template pour parcourir la structure et générer du code. Cette technique nécessite plus d'effort, mais on peut s'attendre à ce qu'elle soit moins coûteuse en temps de compilation car elle ne nécessite d'appeler la fonction de génération de la structure que deux fois.

Étude de cas: intégration du langage Brainfuck dans C++

Pour étudier ces nouvelles techniques de métaprogrammation, nous les appliquons pour l'implémentation d'un analyseur syntaxique et de générateurs de code pour le langage Brainfuck. L'implémentation du parser est triviale, et sa sortie est un arbre syntaxique sous forme d'arbres de pointeurs, similaire à ce qui est utilisé dans les compilateurs développés en C++. Les générateurs de code utilisent les techniques présentées dans le chapitre précédent pour transformer l'AST en code C++.

Nous effectuons ensuite des benchmarks pour comparer les différentes solutions: des benchmarks basés sur des programmes existants, l'un étant un Hello World! et l'autre affichant la fractale de Mandelbrot, ainsi que des benchmarks synthétiques effectués à l'aide de **ctbench** pour étudier le passage à l'échelle des différentes techniques plus en finesse.

Lors de ces benchmarks, nous constatons que la méthode dite «pass by generator» induit des temps de compilation quadratiques en raison du nombre d'appels à la fonction de parsing, appellée elle-même par les fonctions génératrices pour chaque nœud de l'arbre de syntaxe. Cette technique est suffisante pour des programmes de petite taille

La méthode consistant à sérialiser l'arbre, quant à elle, n'appelle la fonction de parsing que deux fois. De fait, sa complexité est linéaire et permet de compiler l'exemple affichant la fractale de Mandelbrot qui compte environ 11000 nœuds en moins de 20 secondes.

Nous en concluons donc que la sérialisation d'une AST permet d'obtenir des résultats

Ce chapitre a fait l'objet d'une présentation à **Meeting C++ 2022** dans le cadre d'un projet commun de recherche avec Paul Keir et Andrew Gozillon.

Application: intégration d'un langage mathématique dans C++

Les conclusions tirées du cas de Brainfuck nous permettent d'implémenter un autre analyseur syntaxique pour un langage mathématique que nous appelons Tiny Math Language (TML). Nous décidons d'implémenter un parser basé sur l'algorithme Shunting yard d'Edsger Dijkstra dont la sortie est la formule en notation polonaise inverse. Cette représentation permet ensuite de générer le programme correspondant à l'aide d'un métaprogramme template stockant les résultats intermédiaires dans une pile sous forme de tuple.

L'analyseur syntaxique est facilement paramétrable: il permet de définir des fonctions, opérateurs (avec leur associativité et leur précedence), des parenthèses, et des variables.

Nous décidons ensuite d'utiliser le parser et le générateur de code pour mesurer les temps de compilation pour des formules mathématiques de 1 à 41 symboles. Dans un premier cas, nous l'utilisons pour effectuer une série d'addition sur des entiers natifs. Dans un autre cas, nous l'utilisons pour effectuer des opérations sur des vecteurs de la bibliothèque **Blaze**. Par surcharge d'opérateurs, les types de Blaze génèrent l'arborescence de formules mathématiques pour ensuite générer le code les évaluant. Ce cas représente donc l'utilisation à la compilation d'un parser pour la génération de code «hautes performances». Enfin nous introduisons un troisième cas consistant simplement à générer les formules Blaze sans utiliser le parser en amont.

Dans tous les cas mesurés, nous obtenons des temps de compilation inférieurs à 20 secondes, y compris pour des formules de 41 symboles. La complexité liée au parsing et à la génération de code est quadratique, mais les temps de compilation demeurent raisonnables dans la mesure ou une formule Blaze comptant un seul symbole nécessite environ 5 secondes de temps de compilation.

Part I

Metaprogramming for High Performance Computing

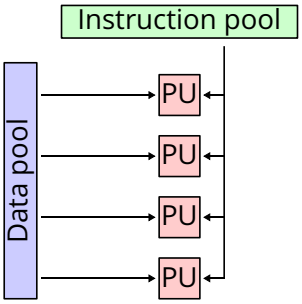
Introduction

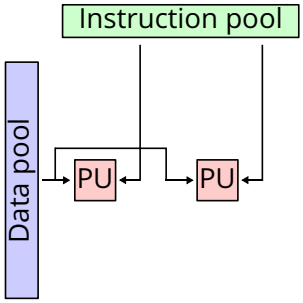
This thesis is about metaprogramming techniques for parallel code generation. It aims to study the boundary between compile-time parsing of Domain-Specific Embedded Languages (DSELs) and high performance code generation in C++.

The main motivation is to provide tools, libraries, and guidelines for embedding mathematical languages in C++ with the hope that it can be useful to build a cohesive set of tools to make generic High Performance Computing (HPC) more accessible to non-expert audiences. This goal is even more important as new computing architectures emerge over time. Developing high performance programs requires tuned implementations that can be achieved by either specializing implementations for the target platforms, or using libraries that provide specialized abstractions at various levels and for various domains.

Years after its introduction, Moore's law is still alive and well: even though computing power itself does not evolve as quickly, transistor count is still doubling every three year. While transistor count has been increasing almost exponentially, the throughput computing performance of single Central Processing Unit (CPU) cores has not. This lead to the conclusion that "free lunch" is over, and future high performance applications must adapt to increasingly parallel architectures to increase compute throughput [57, 41].

Vector registers and instructions allow single instructions allow instructions to be executed on several values at a time. This is known as the **Single Instruction Multiple Data stream (SIMD)** execution model. It is featured across all major instruction set architectures: **AVX** and **SSE** on x86, **NEON** and **SVE** on ARM, **VSX** on PowerPC, and **RVV** on RISC-V.





Multi-core architectures enables multiple CPU cores to execute different tasks simultaneously. These cores may or may not share the same memory pool. If so, the architecture is qualified as a **Unified Memory Architecture (UMA)**. Otherwise, the architecture is qualified as a **Non-Unified Memory Architecture (NUMA)**.

Source of the graphs: https://en.wikipedia.org/wiki/Flynn's_taxonomy

Graphical Processing Units (GPUs) can be used for general purpose computing. They can execute "warps" in parallel. Warps are similar to threads, except for the fact that they must be synchronous at all time for their execution to be carried simultaneously. When warps are not synchronous, their execution is carried separately until they are synchronous again. GPUs can be programmed via many Application Programming Interfaces (APIs), including but not limited to:

- **CUDA**: NVIDIA's proprietary C++-based language and library ecosystem, which is the leading solution for GPU computing. While NVIDIA provides their own proprietary compilers, the Clang/LLVM compiler can also compile CUDA programs.

- **HIP**: AMD's own open-source solution. Its API is meant to resemble CUDA's and is implemented through an open-source fork of the Clang/LLVM compiler.

- **SYCL**: a C++-based open standard for GPU and accelerator programming. It is mostly lead by Intel, but it does support all major GPU brands. The main SYCL implementation is provided by Intel OneAPI, which also features an open-source fork of Clang/LLVM.

- **OpenCL**: a C-based open standard for heterogeneous platforms.

Note that the execution model of GPUs are highly vectorized processors. However, vector instructions on CPUs are often used direclty through intrinsics, whereas GPU computing APIs offer much richer constructs to write compute kernels.

Other accelerators such as FPGAs, ASICs, DSPs, and many more.

To adapt to this variety of hardware and APIs, new abstractions were developed in the form of software libraries, compiler extensions, and even programming languages to help build portable high performance applications. Among these abstractions, a lot of them use **metaprogramming** for the development of high performing, easy to use, and portable abstractions.

Metaprogramming is the ability to treat code as an input or output of a program. Therefore, it allows the generation of programs specialized for specific hardware architectures using high level, declarative APIs. In C++, metaprogramming is mostly done through template metaprogramming, which is a set of techniques that rely on C++ templates to represent values using types to perform arbitrary computations at compile-time. Type templates can even be used to represent expression trees, which can be transformed into code. This kind of application is particularly useful for the implementation of C++-based DSELs that can build compile-time representations of math expressions and

transform them into optimized code. Without metaprogramming, these abstractions would require the introduction of compiler plugins or source-to-source compilers in the compilation toolchain, making it more complex.

However, template metaprogramming has been an esoteric practice since types are by definition not meant to be used as values. For this reason, C++ is evolving to provide first class support for metaprogramming: regular C++ code is now partially allowed to be executed at compile-time to produce results that can be consumed by templates. On the other hand, more C++ libraries start relying on C++ metaprogramming. As such, C++ metaprogramming becomes more mainstream, and the amount of compile-time computing is increasing, along with the complexity of C++ metaprograms.

This trend, as well as new C++ compile-time capabilities, raises new questions: how can we leverage compile-time C++ code execution to improve the quality and performance of C++ metaprograms, especially DSELs implementations for High Performance Computing code generation? Eventually, can we use them to parse DSELs directly instead of reusing the C++ syntax? And as the impact of these metaprograms on compilation times becomes an issue, how can we study their impact in a comprehensive and reproductible way?

In the first part of the thesis, we will take a look at the state of the art of metaprogramming across many languages and in C++. We will then see how it can be used to generate high performance linear algebra kernels and compare the performance of the generated kernels against hand-written assembly kernels.

In the second part we will focus on **ctbench**, a tool for the scientific study of C++ metaprograms compile-time performance. It implements a new benchmarking methodology that enables the study of metaprogram performance at scale, and through the lens of Clang's built-in profiler.

We will then use it to study novel C++ metaprogramming techniques based on compile-time C++ execution. The study will be conducted on two embedded languages that are parsed and transformed into C++ code, within the C++ compilation process itself.

The first language is Brainfuck, which is provided with several code generation backends to compare different code generation techniques. The second one is a simple math language called Tiny Math Language. It demonstrates the usability of compile-time C++ execution to implement a complete compiling toolkit for math formulas, within the C++ compilation process.

1 - Metaprogramming

In this chapter, I will first give an overview of metaprogramming in various languages. Then I will focus on the state of the art of C++ metaprogramming, and finally give examples of applications of such techniques being used in the context of HPC libraries.

1.1 . Metaprogramming styles and languages

Metaprogramming is not an new concept. It perpetuates itself in contemporary languages, with some being more widespread than others.

I will focus on partial specialization, which consists in separating static parts of programs from the dynamic parts in order to interpret the static parts at compile time and leave only the dynamic parts for the program's execution [34, 25]. When the target architecture is known during the compilation, partial evaluation can be leveraged to optimize critical parts of high performance applications by specializing (or tuning) these parts for it.

In C++, templates serve as an interface for partial specialization. However, many other ways to implement it exist across a wide range of languages spanning more than half a century.

- **The C and C++ preprocessor** acts as rudimentary token manipulation stage that was not originally made for metaprogramming, but it can be used to emulate complex logic, such as functional languages. As such, it can even be used for high performance code generation [67].

  In listing 1.1 we can see the Boost.Preprocessor library being used to emulate a tuple structure to select a variable qualifier depending on an arbitrary number.

Listing 1.1: C++ macro usage example [1]

```
#include <boost/preprocessor/facilities/apply.hpp>
#include <boost/preprocessor/tuple/elem.hpp>

#define CV(i)                                                  \
  BOOST_PP_APPLY(BOOST_PP_TUPLE_ELEM(                          \
      4, i,                                                    \
      (BOOST_PP_NIL, (const), (volatile),                      \
       (const volatile))))

CV(0) // expands to nothing
CV(1) // expands to const
```

- **Lisp macros** are known for being one of the oldest, yet one of the most powerful examples among all metaprogramming paradigms [40]. The List macros, as opposed to C and C++ macros, can make complete use of the Lisp language itself with powerful reflection capabilities. Anecdotally, common imperative language constructs such as the while loop can be implemented using macros as shown in listing 1.2.

Listing 1.2: Definition of the `while` Lisp macro[2]

```
(defmacro while (condition &body body)
  `(loop while ,condition do (progn ,@body)))
```

- **MetaOCaml** [38] implements quoting and splicing *i.e.* the ability to essentially copy and paste expressions, as well as staged compilation to evaluate statements at compile-time. This enables code generation to occur both at runtime and at compile-time, and extends OCaml's partial evaluation capabilities.

  Listing 1.3 shows an example of runtime specialization of a power function called `spowern`. It takes an integer N as an input, and returns a function that calculates the N^{th} power of its single input parameter.

Listing 1.3: Run-time specialization of the power function in MetaOCaml[3]

```
let rec spower : int -> int code -> int code =
  fun n x ->
    if n = 0 then .<1>.
    else if n mod 2 = 0
      then .< square .~(spower (n/2) x) >.
      else .<.~x * .~(spower (n-1) x)>.

let spowern : int -> (int -> int) code =
   fun n -> .<fun x -> .~(spower n .<x>.)>.;;
```

- **DLang** more or less extends the C++ Metaprogramming model. It leverages templates and compile time function evaluation just like its predecessor.

  Compile-time evaluation is much more permissive and mixins enable to generate code in a more direct way than C++. Dlang mixins allow injecting code in functions and structures in two ways: using template mixins which are pre-parsed constructs that can be injected later, as well as

[1] https://www.boost.org/doc/libs/1_85_0/libs/preprocessor/doc/ref/apply.html, 4th of June 2024

[2] https://lisp-lang.org/learn/macros, 4th of June 2024

[3] https://okmij.org/ftp/meta-programming/tutorial/, 4th of June 2024

string mixins that allow strings containing Dlang code to be compiled and inserted directly into programs.

Other metaprogramming constructs exists in Dlang, such as `static if` shown in listing 1.4, which is an equivalent of `if constexpr` in C++.

Listing 1.4: Use of `static if` in Dlang[4]

```
static if(is(T == int))
    writeln("T is an int");
static if (is(typeof(x) :  int))
    writeln("x implicitly converts to int");
```

- **Rust** proposes metaprogramming through macros, generics, and traits. In a similar way to Lisp macros, Rust macros can use the host language as a whole. Additionally, bits of code can explicitely be parsed and reflected upon at compile time.

  Here we can find an example taken from The Rust Programming Language book [39] where a simplified version of the `vec!` macro, which initializes a vector and pre-fills it with a static number of values, is defined:

  Listing 1.5 shows how the `vec!` standard macro can be defined. Note that the Rust standard library preallocates

Listing 1.5: Definition of the `vec!` macro[5]

```
#[macro_export]
macro_rules! vec {
  ( $( $x:expr ),* ) => {
    {
      let mut temp_vec = Vec::new();
      $(
        temp_vec.push($x);
      )*
      temp_vec
    }
  };
}
```

[4] https://tour.dlang.org/tour/en/gems/template-meta-programming, 4th of June 2024

[5] https://doc.rust-lang.org/stable/book/, 4th of June 2024

- **Terra**

  Terra[13] implements a very explicit metaprogramming model. The language is based on LUA, and exploits the dynamic nature of the language together with LLVM Just-In-Time (JIT) compilation to allow code generation to happen at runtime. It implements multi-staged compilation and splicing just like MetaOCaml.

  Listing 1.6 shows the definition of a metafunction that returns a quoted expression that adds one to an element, which is then spliced twice to generate a function called `doit`.

  Listing 1.6: Definition of the `addone` metafunction[6]

```
function addone(a)
    --return quotation that
    --represents adding 1 to a
    return `a + 1
end
terra doit()
    var first = 1
    --call addone to generate
    --expression first + 1 + 1
    return [ addone(addone(first)) ]
end
```

  Additionally, Terra can be embedded in other languages through its C API. Overall it is a very versatile and experimental take on metaprogramming, but the lack of interoperability with C++ templates makes it hard to justify its use for HPC applications.

- **LGEN**

  LGEN[53]: a compiler that produces performance-optimized basic linear algebra computations on matrices of fixed sizes with or without structure composed from matrix multiplication, matrix addition, matrix transposition, and scalar multiplication. Based on polyhedral analysis using CLoog, the generated code outperforms MKL and Eigen.

Other languages such as **Braid** [51] or **Julia** [8] implement staged metaprogramming, although Julia emits LLVM code directly, and Braid is unmaintained.

In the next section, we will take a deeper look at C++ as it has powerful metaprogramming capabilities, while providing bleeding edge support for all kinds of computer architectures through a decades-old ecosystem. Moreover, its development is still very active, with many significant metaprogramming proposals being adopted throughout the latest releases[50].

[6] https://terralang.org/getting-started.html, 4th of June 2024

1.2 . C++ language constructs for metaprogramming

C++ templates are an interesting feature for metaprogramming. As they are Turing-complete [60], one can design a set of **template metaprograms** [1] that allow the compiler to perform arbitrary computation at compile time, and generate C++ code fragments as an output. The resulting source code is then merged with the rest of the program and processed through the subsequent compilation stages.

Due to the fact that templates are instantiated at compile-time, they can only accept types and compile-time constants as parameters. Templates support pattern-matching and recursion thanks to partial template specialization, making it equivalent to a pure functional language [28].

This complex logic is often used to implement HPC libraries that embed high level declarative languages based on C++'s syntax, and provide high performance portable functionalities thanks to metaprogramming.

1.2.1 . C++ template metaprogramming

Listing 1.7 shows basic principles of C++ template metaprogramming. The `fibonacci_t` type template accepts an integer called N, and exposes the N^{th} element of the Fibonacci series as its `value` static member. The template has 3 definitions: a generic one to calculate elements for $N > 1$, and two specializations for elements of ranks 0 and 1.

Listing 1.7: Fibonacci series computation using C++ templates

```
template <unsigned N> struct fibonacci_t;

template <unsigned N> struct fibonacci_t {
  static constexpr unsigned value =
      fibonacci_t<N - 2>::value +
      fibonacci_t<N - 1>::value;
};

// Specializations for cases 0 and 1
template <> struct fibonacci_t<0> {
  static constexpr unsigned value = 0;
};

template <> struct fibonacci_t<1> {
  static constexpr unsigned value = 1;
};

std::array<int, fibonacci_t<5>::value> some_array;
```

Note that template parameters in 1.7 are values and not types. This is allowed since C++ 11 as Non-Type Template Parameters (NTTPs) were introduced along with the **constexpr** function and variable qualifier. Before NTTPs were introduced, even numerical values had to be represented using types.

The constexpr qualifier allows functions and variables to be used to produce NTTPs, or more broadly, to be used in **constant evaluations**. Constant evaluations are evaluations that result in the creation of compile-time constants such as NTTPs. In this example, the constant is the size of a static array.

This evolution of the C++ standard marks the beginning of **value-based metaprogramming**, in opposition to **type-based metaprogramming**. Although in this example, both types and values are involved in the compile-time computation.

1.2.2 . Different kinds of templates

C++ templates offer ways to output code for data structures, values, and functions. Ultimately, these kinds outputs constitute what metaprograms can or cannot generate.

- **Type templates** to generate data structures:

```
template <typename T> struct named_value_t {
  std::string name;
  T value;
};
```

- **Type alias templates** which can be used as abstractions on top of template types:

```
template <typename T>
using nested_named_value_t =
    named_value_t<named_value_t<T>>;
```

- **Function templates** to create generic functions:

```
/// Returns a value annotated with
/// its string representation
template <typename T>
named_value_t<T> make_named_value(T value) {
  return named_value_t<T>(std::to_string(value),
                          std::move(value));
}
```

- **Variable templates** to map template parameters to values:

```
/// Returns the default value of T annotated with
/// its string representation
template <typename T>
named_value_t<T> annotated_default_v =
    make_named_value(T{});

named_value_t<int> val = annotated_default_v<int>;
```

1.2.3 . Different kinds of parameters

Metaprograms can take many kinds of C++ constructs as inputs.

- **Type parameters** are the primary kind of parameter used in template metaprogramming in which types are used to represent everything from enumerations, arrays, or even functions.
- **NTTPs** are complementary with type parameters. They can be used to wrap values within types, which can be particularly useful in the context of template metaprogramming. For example a type template definition like `template<int I> integer_t{};` allows integers to be stored as types.
- **Parameter packs** make it possible for templates to have an arbitrary number of parameters, as long as types and values are not mixed together. Parameter packs also exist for function parameters, which can be useful for template parameter deduction.

  A type template definition like `template<typename ... Ts> tuple_t{};` can be used to store an arbitrary list of types as a single type.
- **Regular function parameters** can be used in certain conditions. Since C++ 11, functions and variables can be qualified as constexpr and used to produce NTTPs.

Listing 1.8: Compile-time parameter examples

```
// Type template parameter
template <typename T> T foo() { return T{}; }
// Non-type template parameter
template <int I> int foo() { return I; }
// Parameter pack
template <typename... Ts> auto foo() {
  return Ts{} + ...;
}
// Constexpr function parameter
constexpr int foo(int i) { return i; }
```

1.2.4 . Advanced C++ template mechanisms

- **Template specialization** allows specializing a template for a given set of type or value parameters. A template that has a single boolean parameter can, for example, expose two different implementations depending on the parameter's value.
- **Template parameter deduction** works as a form of pattern matching for function template parameters, types and values alike. It can filter parameter patterns, and deduce nested template parameters in those patterns.

- **Substitution Failure Is Not An Error (SFINAE)** is a C++ principle that consists in not triggering a compilation error when a type substitution cannot be done in a template instantiation. Instead the type, function, or variable will simply be disabled and the next candidate will be instantiated.
- **Parameter pack expansion** allows mapping operations and performing reductions on parameter packs.

Listing 1.9: Advanced C++ template mechanism examples

```
// Template specialization
template <unsigned I> unsigned foo() {
  return I + foo<I - 1>();
}
template <> unsigned foo<0>() { return 0; }

// Template parameter deduction
template <typename T1, typename T2>
int foo(std::tuple<T1, T2>) {
  return T1{} + T2{};
}

// sfinae
template <typename T>
std::enable_if<std::is_integral_v<T>, void> foo() {
  return T{};
}

// Parameter pack expansion
template <typename... Ts>
bool var = (std::is_integral_v<Ts> && ...);
```

1.2.5 . Compile time logic

Compile time logic can be achieved in many C++ constructs.

- **Computation using types**, also known as type-based metaprogramming. Types can be used to represent scalar values, arrays, and many more kinds of data structures including expression trees. They can also be used to represent functions, and whole programs can be built and executed at compile-time using type templates as their only building block.
- **Computation using values**, thanks to NTTPs. They allow a slightly more explicit way to write metaprograms, as values are represented by values instead of types. Not all values are accepted as NTTP. Until C++ 20, only integral values (*i.e.* integers, booleans, etc.) could be used as NTTP.

- **Computation using constexpr functions**, since C++ 11, although only a limited subset of C++ can run in constant evaluations. Memory allocations have been allowed in this context only since C++ 20.

The use of constexpr functions for compile time programming might be preferable for many reasons: they are familiar to all C++ developers (and, as such, are more maintainable), they allow the use of types to enforce semantics properly as opposed to type-based metaprogramming, and they often allow compile time programs to run much faster than pure template metaprogramming does, as we will see in the next chapters. In 1.10, we can see a constexpr implementation of the Fibonacci sequence computation. It produces the same output as `fibonacci_t` in 1.7.

Listing 1.10: constexpr Fibonacci sequence computation

```
constexpr unsigned fibonacci(unsigned n) {
  unsigned n0 = 1, n1 = 0;
  for (unsigned i = 0; i < n; i++) {
    unsigned n1_copy = n1;
    n1 = n0 + n1;
    n0 = n1_copy;
  }
  return n1;
}
```

Metaprogramming is essential to implement high level DSELs that translate into high performance programs through a series of transformations that occur at compile-time, akin to an additional user-defined compilation stage embedded into the compilation process itself.

1.3 . Domain specific embedded language

DSELs in C++ use template metaprogramming via the **expression template** idiom. Expression templates [64, 62] consist of a technique implementing a form of **delayed evaluation** in C++ [54]. They are built around the **recursive type composition** idiom [33] that allows the construction, at compile time, of a type representing the AST of an arbitrary statement. This is done by overloading functions and operators on those types so they return a lightweight object. The object encodes the current operation in the AST being built in its type instead of performing any kind of computation. Once reconstructed, this AST can be transformed into arbitrary code fragments using template metaprograms.

As of today, most C++ DSELs rely on expression template and therefore are limited to the C++ syntax. The boost.proto [44] library enables rapid prototyping of DSELs based on the C++ syntax.

New techniques are becoming more popular through the use of constexpr strings to embed arbitray DSELs. One major example is the Compile Time Regular Expression (CTRE) [20] that implements most of the Perl Compatible Regular Expression (PCRE) syntax as shown in listing 1.11. It relies on type-based metaprogramming to parse regular expressions and transform them into regular expression evaluators.

Listing 1.11: CTRE usage example[7]

```
struct date {
  std::string_view year;
  std::string_view month;
  std::string_view day;
};

std::optional<date>
extract_date(std::string_view s) noexcept {
  using namespace ctre::literals;
  if (auto [whole, year, month, day] =
          ctre::match<
              "(\\d{4})/(\\d{1,2})/(\\d{1,2})">(
              s);
      whole) {
    return date{year, month, day};
  } else {
    return std::nullopt;
  }
}
```

1.3.1 . A use case for Expression Templates

Combined with compile-time mechanisms such as function overloading, specialization, and operator overloading, expression templates can be used to implement expression level DSELs and convert complex mathematical expressions into high performance code[64]. There are two main libraries that are able to do this: Eigen [26] and Blaze [30].

Figure 1.1 shows how mathematical expressions can be encoded via expression templates: an operation involving matrices generates a type hierarchy that represents the operation itself, and the assignment of this expression to `x` triggers the generation of an optimized program that computes the operation.

They enable a whole range of optimizations:

- **Lazy evaluation** is easily implemented through the generation of access operators that calculate single element values in one to one element operations such as additions or substractions.

[7] https://github.com/hanickadot/compile-time-regular-expressions, 4th of June 2024

- **Basic Linear Algebra Subprograms (BLAS) functions** can be used on complex operations that benefit from highly optimized implementations. For example while element-wise operations such as vector additions might better use lazy evaluation, more complex operations such as matrix-matrix multiplications require more hardware-specific tuning.
- **Multi-threading** can be implemented via parallel assignment functions, with the help of views.
- **SIMD optimizations** can be implemented through vector access operators to combine the benefits of lazy evaluation and SIMD computing.
- **GPU support** has been demonstrated as feasible. An attempt was made to bring it to Blaze [48], although it was never fully implemented due to many roadblocks such as the requirement of adding `__device__` qualifiers in front of all Blaze functions used in CUDA kernels.

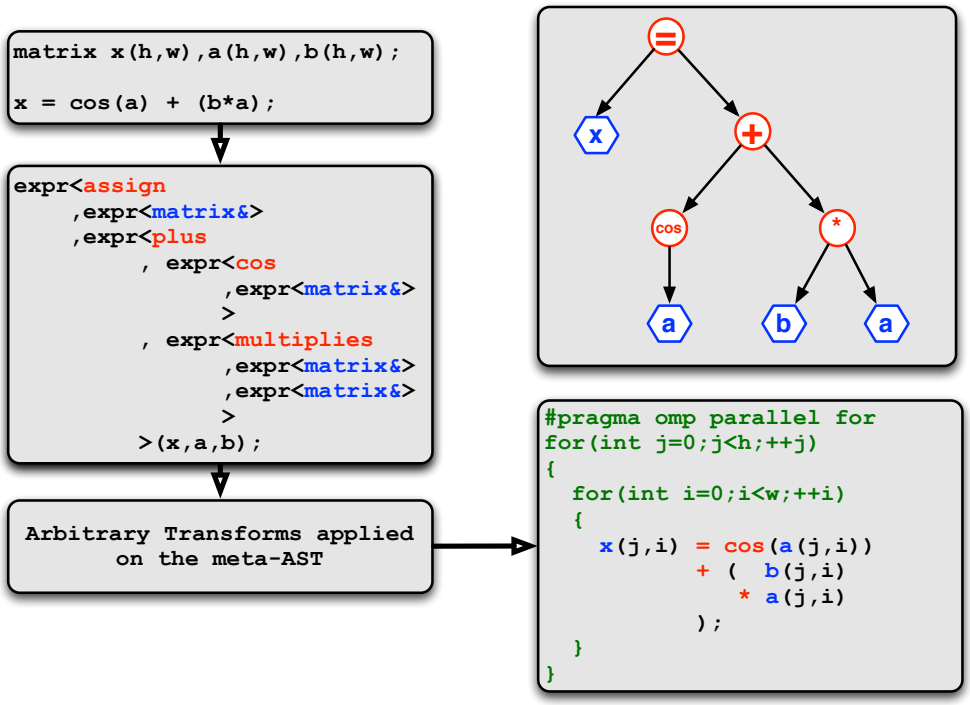


Figure 1.1: Expression template evaluation illustration[22]

To implement these optimizations in Blaze [30], expression templates and assignment function overloading are equally essential to implement evaluation strategies depending on the nature of the operations.

For example, element-wise operations are optimized through lazy evaluation thanks to element compute access operators, and via parallelism and vectorization implemented in the assignment functions. In the case of complex operations such as matrix-matrix products, optimizations are provided through assignment function overloading to call BLAS routines instead of using regular element-wise assignment.

Expression templates are an important building block for C++ math libraries that enable the creation of high level, portable DSELs that resolve into high performance code thanks to a combination of metaprogramming techniques. In the next section, we will see a collection of libraries that go beyond the idea of using templates for math code generation, and implement or enable the implementation of arbitrary compile-time programs.

1.4 . Metaprogramming libraries

1.4.1 . Type based metaprogramming

As previously said, C++ templates can be used as a compile-time functional language. Over time a range of metaprogramming libraries emerged, aiming to provide functionalities similar to regular ones, such as containers and algorithms for use in template metaprograms. Notable examples of such libraries are MPL [27], Brigand [29], and mp11 [15].

Listing 1.12: boost.mp11 code example

```
// A predicate is implemented as a structure template
template <int X> struct equals_to {
  template <typename Y>
  using apply = mp_bool<X == Y::type::value>;
};

// We store a number list as a type alias called ‘my_list‘
using my_list = mp_list_c<int, 0, 2, 4, 6, 8, 10>;

// ‘pos_of_6‘ is a type representation of the index of 6
using pos_of_6 =
    mp_find_if<my_list, equals_to<6>::apply>; // 3
```

Listing 1.12 shows a basic mp11 use case: we first define a list of arbitrary integers, and then we find the position of the first element of value 6 in the list.

Listing 1.13: Brigand code example

```
using my_list =
    brigand::integral_list<int, 0, 2, 4, 6, 8, 10>;

using pos_of_6 = brigand::size<brigand::find<
    my_list, std::is_same<brigand::_1,
                          brigand::integral_constant<
                              int, 6>>>>; // 3
```

Listing 1.13 shows the exact same task donw with the Brigand metaprogramming library.

All these libraries either enable template metaprogramming, or use template metaprogramming to achieve a specific goal. However with the introduction of constexpr programming, a new range of compile-time libraries aims to provide new capabilities for this new metaprogramming paradigm.

1.4.2 . Value based metaprogramming

In C++, constexpr functions can be executed at compile-time and since C++ 20, these functions can allocate memory, thus allowing dynamic containers to be implemented. Virtual functions were allowed too, further expanding constexpr programming capabilities. However, not all standard containers were made available directly as the standard has to be revised for every single one of them to be usable in constant evaluations. The **C'est** [36] library was meant to solve this temporary issue, filling the gap in the C++ standard by providing constexpr compatible standard containers.

As of today most of the containers implemented in **C'est** are available in up-to-date standard libraries, but it is still useful to write metaprograms in compilation environments that do not provide C++ 23 compatible compilers and standard libraries such as older versions of Debian or Red Hat Enterprise Linux.

Listing 1.14: **C'est** code example

```
constexpr std::size_t
find_pos(int element,
         cest::vector<int> const &elements) {
  return cest::find(elements.begin(), elements.end(),
                    element) -
         elements.end();
}

constexpr std::size_t pos_of_6 =
    find_pos(6, {0, 2, 4, 6, 8, 10});
```

It implements the same containers and algorithms as the C++ standard library, although all of them usable in constant evaluations. For example, `std::deque` is not usable in constant evaluations whereas its **C'est** equivalent

named `cest::deque` is. It was instrumental for this thesis as the research work I present here started a long time before C++ 23 was adopted and standard libraries as well as compilers started implementing it.

Similar to previous examples, listing 1.14 shows a compile-time program in which we find the index of the first element of value 6. Note that in this example, we are using properly typed values and functions instead of templates to represent values, predicates, and functions.

I contributed to the development of **C'est** by implementing a constexpr version of `std::unique_ptr`. This collaboration led Paul Keir and myself to speak together at Meeting C++ 2022 [37] where we talked about **C'est** and its use cases for constexpr programming research, including my own projects.

1.5 . Applications of metaprogramming for HPC

As we just saw, metaprogramming can bring significant benefits to libraries:

- **Performance**: notably in the case of CTRE. Regular expressions are usually interpreted at runtime, which adds a measurable overhead to text processing. CTRE shows leading performance, on par with Rust's regex library which also works by compiling regular expressions.
- **Language integration**: since these are C++ libraries, their APIs can take advantage of C++ operator overloading and lambdas. In Compile Time Parser Generator (CTPG), these are used to provide a domain-specific language that is close to what parser generators like YACC or Bison provide, though it is still regular C++ code which can be put inside any function body. Using a C++ API makes these libraries easier to learn as the syntax is already familiar to their users.
- **Streamlined toolchain**: as they only require to be included as headers. This avoids complicating compilation toolchains by requiring additional programs to be installed and integrated to the build system.

These qualities make metaprogramming a good candidate for the implementation of comprehensive HPC toolkits that would otherwise have slower implementations, or otherwise rely on compiler extensions like OpenMP.

As such, there are many C++ HPC libraries that use metaprogramming more or less extensively:

- **Eigen** [26] is the first major C++ library to implement Expression templates for the generation of high performance math computing. Expression templates is a C++ design pattern that consists in representing math expressions with type template trees. We will discuss them later in 1.3.

```
using Eigen::MatrixXd;
using Eigen::VectorXd;

int main() {
  MatrixXd m = MatrixXd::Random(3, 3);
  m = (m + MatrixXd::Constant(3, 3, 1.2)) * 50;
  std::cout << "m =" << std::endl << m << std::endl;
  VectorXd v(3);
  v << 1, 2, 3;
  std::cout << "m * v =" << std::endl
            << m * v << std::endl;
}
```

- **Blaze** [30] is a successor of Eigen that implements so-called "Smart Expression Templates" which extends upon the concept of expression templates implemented by Eigen. It aims to provide a more performant and extensible HPC library. However, Eigen is not set in stone and its design has since been updated.

```
int main() {
  using blaze::DynamicVector;
  using blaze::StaticVector;

  StaticVector<int, 3UL> a{4, -2, 5};
  DynamicVector<int> b(3UL);

  b[0] = 2;
  b[1] = 5;
  b[2] = -3;

  DynamicVector<int> c = a + b;

  std::cout << "c =\n" << c << "\n";
}
```

- **NT2** [24] is a research project that aims to provide a complete numerical toolbox that leverages metaprogramming to develop portable HPC applications with a Matlab-like interface while still achieving state-of-the-art computing performance.

```
int main() {
  nt2::table<double> x;
  nt2::table<double> y = nt2::ones(4, 4);

  x = 40.0 * y + 2.0;

  NT2_DISPLAY(x);
}
```

- **EVE** [23] provides generic abstractions over SIMD instructions as well as SIMD-optimized generic algorithms for the development of high performance and portable SIMD code [47].

```
int main() {
  eve::wide<float> x(
      [](auto i, auto) { return 1.f + i; });
  std::cout << "x      = " << x << "\n";
  std::cout << "2*x    = " << x + x << "\n";
  std::cout << "x^0.5 = " << eve::sqrt(x) << "\n";
}
```

- **HPX** [35] is a C++ parallel and distributed runtime library. It can execute small parallel tasks efficiently and distribute larger distributed tasks with a work following data execution model. Its parallel and distributed APIs as well as its parallel implementation of the standard library (based on its own parallel runtime) use metaprogramming for algorithmic genericity.

```
std::uint64_t fibonacci(std::uint64_t n) {
  if (n < 2)
    return n;

  hpx::future<std::uint64_t> n1 =
      hpx::async(fibonacci, n - 1);
  hpx::future<std::uint64_t> n2 =
      hpx::async(fibonacci, n - 2);

  // wait for the futures to return their values
  return n1.get() + n2.get();
}
```

- **Thrust** [7] implements GPU-accelerated equivalents of the Standard Library's algorithms, while CUB [43] provides GPU-optimized algorithm skeletons for generic programming on NVIDIA GPUs. AMD and Intel implement their equivalents for their own platforms, respectively ROCm and OneAPI.

```
int foo() {
  // generate random data serially
  thrust::host_vector<int> h_vec(100);
  std::generate(h_vec.begin(), h_vec.end(), rand);

  // transfer to device and compute sum
  thrust::device_vector<int> d_vec = h_vec;
  return thrust::reduce(d_vec.begin(),
                        d_vec.end(), 0,
                        thrust::plus<int>());
}
```

These libraries operate at many different levels: some of them provide high level declarative APIs for math computing, while others provide generic building blocks to write generic compute kernels.

1.6 . Conclusion

Metaprogramming surfaced in many forms and in many languages such as Lisp or C macros, long before C++ templates came to exist. It still persists as an advanced but widespread practice across many languages, ranging from basic preprocessing to dynamic code generation and compilation.

Template metaprogramming has been the de facto standard for C++ metaprogramming as many libraries use it to implement high performance or highly specialized libraries that benefit from compile-time evaluation and code generation. This is especially true for HPC libraries that benefit the most from partial evaluation.

The language C++ later evolved to provide host language availability for metaprogramming with the introduction and evolution of constexpr programming, similar to what Lisp macros have been pioneering. This new way of writing C++ metaprograms will be explored throughout the second part of this thesis. Meanwhile, we are going to investigate the actual benefits of such techniques by applying them to the basic building blocks of HPC applications: linear algebra kernels.

2 - Code generation at low level

BLAS level functions[9] are the cornerstone of a large subset of applications. If a large body of work surrounding efficient and large-scale implementation of some routines such as General Matrix Vector multiply (GEMV) exists, the interest for small-sized, highly-optimized versions of those routines emerged.

In this chapter, we show that modern C++ generative programming techniques can deliver efficient automatically generated codes for such routines. The generated kernels are competitive with existing, hand-tuned library kernels with a very low programming effort compared to writing assembly code.

The results of this chapter have been published in Jules Pénuchot, Joel Falcou, Amal Khabou, "*Modern Generative Programming for Optimizing Small Matrix-Vector Multiplication*", 2018 International Conference on High Performance Computing Simulation (HPCS) [47].

2.1 . Context

The efforts of optimizing the performance of BLAS routines fall into two main directions. The first direction is about writing very specific assembly code. This is the case for almost all the vendor libraries including Intel MKL[32], AMD ACML[4] etc. To provide the users with efficient BLAS routines, the vendors usually implement their own routines for their own hardware using assembly code with specific optimizations which is a low level solution that gives the developers full control over both the instruction scheduling and the register allocation. This makes these routines highly architecture dependent and needing considerable efforts to maintain the performance portability on the new architecture generations. Moreover, the developed source codes are generally complex. The second direction is based on using modern generative programming techniques which have the advantage of being independent from the architecture specifications and as a consequence easy to maintain since it is the same source code which is used to automatically generate a specific code for a specific target architecture. With respect to the second direction, some solutions have been proposed in recent years. However, they only solve partially the trade-off between the abstraction level and the efficiency of the generated codes. This is for example the case of the approach followed by the Formal Linear Algebra Methods Environment (FLAME) with the Libflame library[68]. Thus, it offers a framework to develop dense linear solvers using algorithmic skeletons[11] and an API which is more user-friendly than LA-

PACK, giving satisfactory performance results. A more generic approach is the one followed in recent years by C++ libraries built around expression templates[63] or other generative programming[12] principles. In this section, we will focus on such an approach. To show the interest of this approach, we consider as example the matrix-vector multiplication kernel (gemv) which is crucial for the performance of both linear solvers and eigen and singular value problems. Achieving performance running a matrix-vector multiplication kernel on small problems is challenging as we can see through the current state-of-the-art implementation results. Moreover, the different CPU architectures bring further challenges for its design and optimization.

The quality and performance of BLAS like code require the ability to write tight and highly-optimized code. If the raw assembly of low-level C has been the language of choice for decades, our position is that the proper use of the zero abstraction property of C++ can lead to the design of a single, generic yet efficient code base for many BLAS like functions. To do so, we will rely on two main elements: a set of code generation techniques stemmed from metaprogramming and a proper C++ SIMD layer.

2.2 . Code generation via metaprogramming

As we saw earlier, metaprogramming is used in C++ [2],D [10], OCaml[58] or Haskell[52]. A subset of basic notions emerges:

- Code fragment generation: Any metaprogrammable language has a way to build an object that represents a piece of code. The granularity of this fragment of code may vary –ranging from statement to a complete class definition–but the end results is the same: to provide an entry level entity to the metaprogramming system. In some languages, such as MetaOCaml for example, a special syntax can be provided to construct such fragment. In some others, code fragment are represented as a string containing the code itself.

- Code processing: Code fragments are meant to be combined, introspected or replicated in order to let the developer rearrange these fragments and as a consequence to provide a given service. Those processing steps can be done either via a special syntax construct, like the MetaOCaml code aggregation operator, or can use a similar syntax than a regular code.

Now let's focus on language-based metaprogramming techniques so that the proposed method can be used in various compilers and OS settings as long as the compiler follows a given standard. With the standard C++ revision in 2014 and 2017, this strategy was renewed with three new C++ features:

- Polymorphic, variadic anonymous functions: C++ 11 introduced the notion of local, anonymous functions (also known as lambda functions) in the language. Their primary goal was to simplify the use of standard algorithms by providing a compact syntax to define a function in a local scope, hence raising code locality. C++ 14 added the support for polymorphic lambdas, *i.e.* anonymous functions behaving like function templates by accepting arguments of arbitrary types, and variadic lambdas, *i.e.* anonymous functions accepting a list of arguments of arbitrary size. Listing 2.1 demonstrates this particular feature.

Listing 2.1: Sample polymorphic lambda definition

```
// Variadic function object building array
auto array_from =
    [](auto... values) {
      // sizeof... retrieves the
      // number of arguments
      return std::array<double,
                        sizeof...(values)>{
          values...};
    }
// Build an array of 4 double
auto data = array_from(1, 2, 3., 4.5f);
```

- Fold expressions: C++ 11 introduced the ... operator which was able to enumerate a variadic list of functions or template arguments in a comma-separated code fragment. Its main use was to provide the syntactic support required to write a code with variadic template arguments. However, Niebler and Parent showed that this can be used to generate far more complex code when paired with other language constructs. Both code replication and a crude form of code unrolling were possible. However, it required the use of some counter-intuitive structure. C++ 17 extends this notation to work with an arbitrary binary operator. Listing 2.2 illustrates an example for this feature.

Listing 2.2: C++ 17 fold expressions

```
template<typename... Args>
auto reduce(Args&&... args) {
  // Automatically unroll the args into a sum
  return (args + ...);
}
```

Tuples Introduced by C++ 11, tuple usage in C++ was simplified by providing new tuple related functions in C++ 17 that make tuple a fully programmable struct-like entity. The transition between tuple and structure is then handled via the new structured binding syntax that allow

the compile-time deconstruction of structures and tuples in a set of disjoint variables, thus making interface dealing with tuples easier to use. Listing 2.3 gives an example about tuples.

Listing 2.3: Tuple and structured bindings

```
// Build a tuple from values
auto data = std::make_tuple(3.f, 5, "test");

// Direct access to tuple data
std::get<0>(data) = 6.5f;

// Structured binding access
auto&[a,b,c] = data;

// Add 3 to the second tuple's element
b += 3;
```

2.3 . C++ performance layer

The main strategies to get efficient small-scale BLAS functions are on one hand the usage of the specific instructions set (mainly SIMD instructions set) of the target architecture, and on the other hand the controlled unrolling of the inner loop to ensure proper register and pipeline usage.

Vectorization Vectorization can be achieved either using the specific instructions set of each vendor or by relying on auto-vectorization. In our case, to ensure homogeneous performances across the different target architectures, we relied on the Boost.SIMD[21] package to generates SIMD code for all our architectures. Boost.SIMD relies on C++ metaprogramming to act as a zero-cost abstraction over SIMD operations in a large number of contexts. The SIMD code is then as easily written as a scalar version of the code and deliver 95% to 99% of the peak performances for the L1 cache hot data. The main advantage of the Boost.SIMD package lies in the fact that both scalar and SIMD code can be expressed with the same subset of functions. The vector nature of the operations will be triggered by the use of a dedicated type – pack – representing the best hardware register type for a given type on a given architecture that leads to optimized code generation.

Listing 2.4 demonstrates how a naive implementation of a vectorized dot product can simply be derived from using Boost.SIMD types and range adapters, polymorphic lambdas and standard algorithm.

Note how the Boost.SIMD abstraction shields the end user to have to handle any architecture specific idioms and how it integrates with standard algorithms, hence simplifying the design of more complex algorithms. Another point is that, by relying on higher-level library instead of SIMD pragma, Boost.SIMD guarantees the quality of the vectorization across compilers and compiler versions. It also leads to a cleaner and easier to maintain codes, relying only on standard C++ constructs.

Listing 2.4: Sample Boost.SIMD code

```
template <typename T>
auto simd_dot(T *in1, T *in2, std::size_t count) {
  // Adapt [in,in+count[ as a vectorizable range
  auto r1 = simd::segmented_range(in1, in1 + count);
  auto r2 = simd::segmented_range(in2, in2 + count);

  // Extract sub-ranges
  auto h1 = r1.head, h2 = r2.head;
  auto t1 = r1.tail, t2 = r2.tail;

  // sum and product polymorphic functions
  auto sum = [](auto a, auto b) { return a + b; };
  auto prod = [](auto r, auto v) { return r * v; };

  // Process vectorizable & scalar sub-ranges
  auto vr = std::transform_reduce(
      h1.begin(), h1.end(), h2.begin(), prod, sum,
      simd::pack<T>{});

  auto sr = std::transform_reduce(
      t1.begin(), t1.end(), t2.begin(), prod, sum,
      T{});

  // Compute final dot product
  return sr + simd::sum(vr);
}
```

Loop unrolling The notion of unrolling requires a proper abstraction. Loop unrolling requires three elements: the code fragment to repeat, the code replication process and the iteration space declaration. Their mapping into C++ code is as follows:

- The code fragment in itself, which represents the original loop body, is stored inside a polymorphic lambda function. This lambda function will takes a polymorphic argument which will represent the current value of the iteration variable. This value is passed as an instance of `std::integral_constant` which allows to turn an arbitrary compile-time con-

stant integer into a type. By doing so, we are able to propagate the constness of the iteration variable as far as possible inside the code fragment of the loop body.

- The unrolling process itself relies on the fold expression mechanism. By using the sequencing operator, also known as operator comma, the compiler can unroll arbitrary expressions separated by the comma operator. The comma operator will take care of the order of evaluation and behave as an unrollable statement.

- The iteration space need to be specified as an entity supporting expansion via ... and containing the actual value of the iteration space. Standard C++ provides the `std::integral_sequence<N...>` class that acts as a variadic container of integral constant. It can be generated via one helper meta-function such as `std::make_integral_sequence<T,N>` and passed directly to a variadic function template. All these elements can then be combined into a `for_constexpr` function detailed in listing 2.5.

Listing 2.5: Compile-time unroller

```
template <int Start, int D, typename Body,
          int... Step>
void for_constexpr(
    Body body, std::integer_sequence<int, Step...>,
    std::integral_constant<int, Start>,
    std::integral_constant<int, D>) {
  (body(std::integral_constant<int,
                               Start + D * Step>{}),
   ...);
}

template <int Start, int End, int D = 1,
          typename Body>
void for_constexpr(Body body) {
  constexpr auto size = End - Start;
  for_constexpr(
      std::move(body),
      std::make_integer_sequence<int, size>{},
      std::integral_constant<int, Start>{},
      std::integral_constant<int, D>{});
}
```

The function proceed to compute the proper integral constant sequence from the Start, End and D compile-time integral constant. As `std::integral_sequence<N...>` enumerates values from 0 to N , we need to pass the start index and iteration 1 pragma are compiler-dependent and can be ignored increment as separate constants. The actual index is then computed at the unrolling site.

To prevent unwanted copies and ensure inlining, all elements are passed to the function as a rvalue-reference or a universal reference.

A sample usage of the `for_constexpr` function is given in listing 2.6 in a function printing every element from a `std::tuple`.

Listing 2.6: Tuple member walkthrough via compile-time unrolling

```
template<typename Tuple>
void print_tuple(Tuple const& t) {
  constexpr auto size = std::tuple_size<Tuple>::value;
  for_constexpr<0, size>([&](auto i) {
    std::cout << std::get<i>(t) << "\n";
  });
}
```

Note that this implementation exposes some interesting properties:

- As `for_constexpr` calls are simple function call, they can be nested in arbitrary manners.
- Relying on `std::integral_constant` to carry the iteration index gives access to its automatic conversion to integer. This means the iteration index can be used in both compile-time and runtime contexts.
- Code generation quality will still be optimized by the compiler, thus letting all other non-unrolling related optimizations to be applied.

  One can argue about the advantage of such a method compared to relying on the compiler unrolling algorithm or using non-standard unrolling pragma. In both cases, our method ensure that the unrolling is always done at the fullest extend and does not rely on non-standard extensions.

2.4 . Application: the GEMV kernel

Level 2 BLAS routines such as GEMV have a low computational intensity compared to Level 3 BLAS operations such as GEMM. For that reason, in many dense linear algebra algorithms in particular for one sided factorizations such as Cholesky, LU, and QR decompositions some techniques are used to accumulate several Level 2 BLAS operations when possible in order to perform them as one Level 3 BLAS operation[5]. However, for the two-sided factorizations, and despite the use of similar techniques, the fraction of the Level 2 BLAS floating point operations is still important. For instance, for both the bidiagonal and tridiagonal reductions, this fraction is around 50%[59]. Thus, having optimized implementations for these routines on different architectures remains important to improve the performance of several algorithms

and applications. Moreover, small-scale BLAS kernels are useful for some batched computations[19].

Here, we consider the matrix-vector multiplication routine for general dense matrices, GEMV, which performs either $y := \alpha Ax + \beta y$ or $y := \alpha A^T x + \beta y$, where A is an $m \times n$ column-major matrix, α and β are scalars, and y and x are vectors. In this section, we focus on matrices of small sizes ranging from 4 to 512 as this range of sizes encompasses the size of most L1 cache memory, thus allowing a maximal throughput for SIMD computation units. The algorithm we present in listing 2.7 is optimized for a column-major matrix. For space consideration, we will only focus on the core processing of the computation, *i.e.* the SIMD part, as the computation of the scalar tail on the last columns and rows can be trivially inferred from there.

Our optimized code relies on two types representing statically-sized matrix and vector, namely `mat<T,H,W>` and `vec<T,N>`. Those types carry their height and width as template parameters so that all size related values can be derived from them. The code shown in listing 2.7 is made up of three main steps as detailed in figure 2.1:

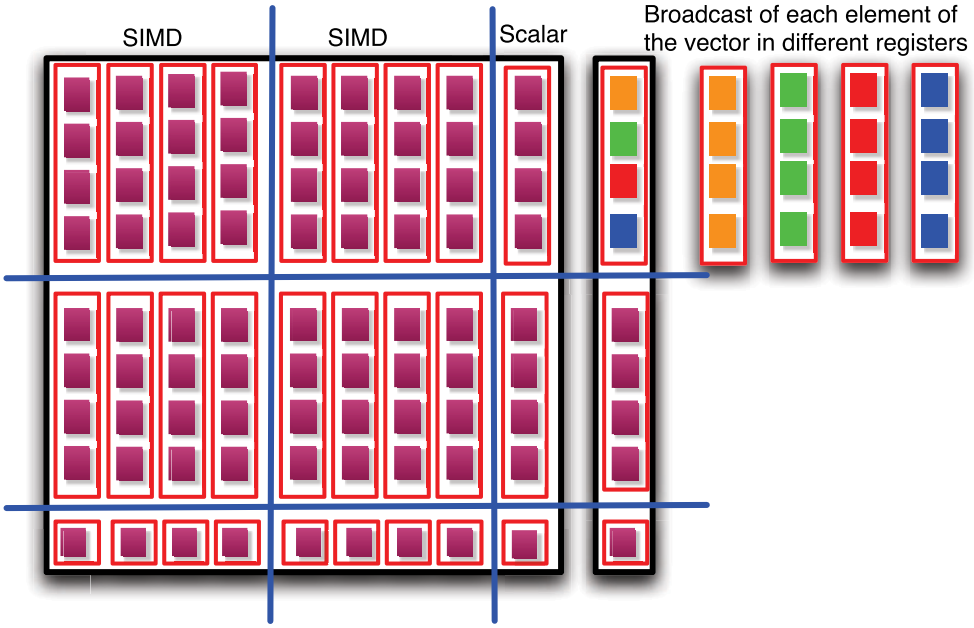


Figure 2.1: An example of matrix vector multiplication showing the SIMD/scalar computation boundaries. The matrix is 9×9 of simple precision floats so we can put 4 elements per SIMD register.

1. The computation of SIMD/scalar boundaries based on the static size of the matrix and the size of the current SIMD registers. Those computations are done in constexpr contexts to ensure their usability in the upcoming unrolling steps.

2. A first level of unrolling that takes care of iterating over all set of columns that are able to fit into SIMD registers. This unrolling is done so that both the corresponding columns of the matrix and the elements of the vector can respectively be loaded and broadcasted into SIMD registers.

Listing 2.7: Unrolled GEMV kernel

```
template <typename T, std::size_t H,
          std::size_t W>
void gemv(mat<T, H, W> &mat, vec<T, W> &vec,
          vec<T, W> &r) {
  using pack_t = bs::pack<T>;
  constexpr auto sz = pack_t::static_size;

  // Separating simd/scalar parts
  constexpr auto c_simd = W - W % sz;
  constexpr auto r_simd = H - H % sz;

  for_constexpr<0, c_simd,sz>( [](auto j) {
    pack_t pvec(&vec[j]);
    pack_t mulp_arr[sz];

    // Broadcasting vectors once and for all
    for_constexpr<0, sz>([&](auto idx) {
      mulp_arr[idx] = simd::broadcast<idx>(pvec);
    });

    // Walk through SIMD rows
    for_constexpr<0, r_simd>([&](auto I) {
      pack_t resp(&res[i + (I * sz)]);

      // Walk through columns
      for_constexpr<0, sz>([&](auto J) {
        pack_t matp(&mat(i + (I * sz), j + J));
        resp += matp * mulp_arr[J];
        simd::store(resp, &r[i + (I * sz)]);
      });
    }

    // Scalar code follows ...
}
```

3. A second level of unrolling that pass through all the available SIMD registers loadable from a given column. We rely on an overloaded `operator()` on the matrix to compute the proper position to load from. As usual with Boost.SIMD, the actual computation is run with scalar-like syntax using regular operators.

It is important to notice how close the actual unrolled code is to an equivalent code that would use regular for loops. This strong similarity shows that

modern metaprogramming matured enough so that the frontier between regular runtime programming and compile-time computation and code generation is very thin. The effort to fix bugs in such code or to upgrade it to new algorithms is comparable to the effort required by a regular code. The notion of code fragment detailed in Section II helps us to encapsulate those complex metaprogramming cases into an API based on function calls.

2.5 . Performance results of the generated GEMV codes

To validate our approach, we consider two main targeted processor architectures: an x86 Intel processor i5-7200 and an ARM processor AMD A1100 with Cortex A57. We compare the performance of the generated GEMV codes to that of the GEMV kernel of the OpenBLAS library based on GotoBLAS2 1.13[66]. We use GCC 7.2[55] with maximal level of optimization.

In the following experiments, we only show results for simple precision floats with column major data, but we obtained similar results for the double precision case, as well as the row major data. All the results displayed below are obtained using one thread. All those results has been obtained using Google Benchmark micro-benchmark facilities. Every experiments have been repeated for a duration of 3s, using the median time as an approximation of the most frequent measured time.

2.5.1 . On X86 Intel processor

In figure 2.2, we compare the performance of our implementation using the SIMD Extensions set SSE 4.2 and a similarly configured OpenBLAS GEMV kernel. The obtained results show that the performances of our automatically generated code is up to 2 times better for matrices of sizes ranging from 4×4 elements to 16×16 elements. However, for matrices of size 32×32 elements and 64×64 elements, the OpenBLAS GEMV kernel gives a better performances, especially for the 64×64 case. This is because the OpenBLAS library uses a dedicated GEMV kernel with specific optimizations and prefetching strategies that our generic solution can not emulate. Beyond this size (64×64 elements), the $L1 \rightarrow L2$ cache misses cause a performance drop for both our generated code and the OpenBLAS GEMV kernel. Nevertheless, our generated code sustains a better throughput for matrices of sizes 128×128 elements. For matrices of size 256×256 elements, the register usage starts to cause spill to the stack, showing that our solution can not be arbitrarily extended further to larger matrix sizes.

In figure 2.3, we compare the performance of our generated GEMV code using Advanced Vector Extensions AVX2 to the performance of a similarly configured OpenBLAS GEMV kernel. Again, the performances of our implementation are close to that of OpenBLAS and are even quite better for matrices of

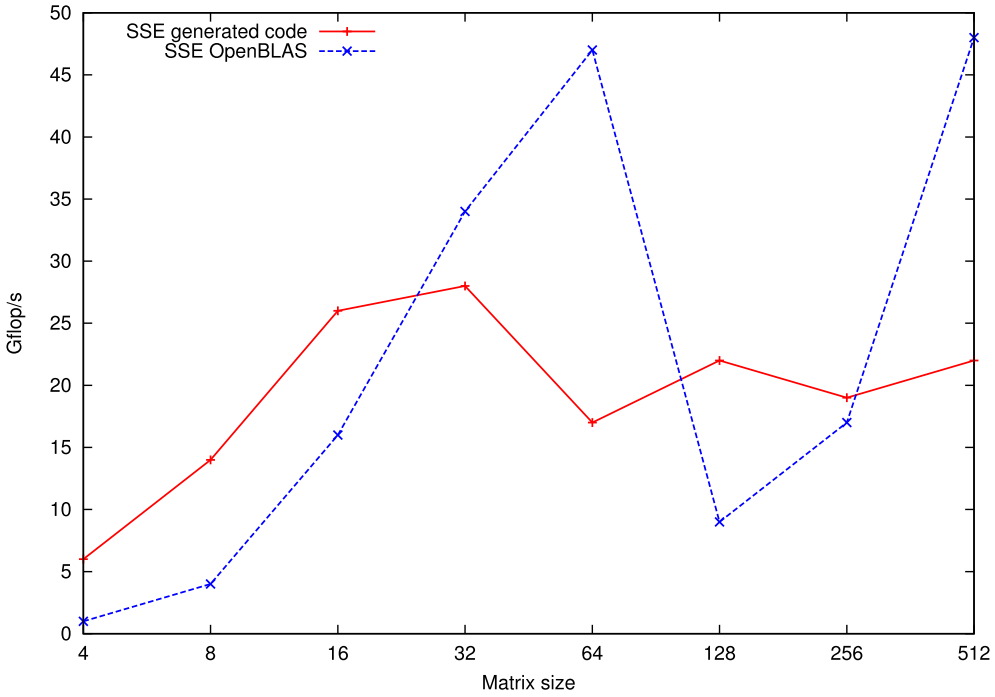


Figure 2.2: GEMV performance on Intel i5-7200 processor using SIMD Extensions set (SSE-4.2)

small sizes ranging from 4 to 16 elements. For example, for a matrix of size 8 elements,the automatically generated code has a performance that is 3 times better than the OpenBLAS GEMV kernel (15.78 Gflop/s vs 5.06 Gflop/s). Two phenomenons appear however. The first one is that the increased number of the AVX registers compared to the SSE ones makes the effect of register spilling less prevalent. The second one is that the code generated for the special 64×64 elements case[65] in OpenBLAS has a little advantage compared to our automatically generated code. Finally, we note the fact that, for matrices of size above 512×512 elements, we stop being competitive due to the large amount of registers our fully unrolled approach would require.

In both cases, the register pressure is clearly identified as a limitation. One possible way to fix this issue will be to rely on partial loop unrolling and using compile-time informations about architecture to decide the level of unrolling to apply for a given size on a given architecture.

2.5.2 . On ARM processor

The comparison between our automatically generated code and the ARM OpenBLAS GEMV kernel is given in figure 2.4. Contrary to the x86 Intel processor, we sustain a comparable yet slightly better throughput than the OpenBLAS GEMV kernel. The analysis of the generated assembly code shows that our method of guiding the compiler and letting it do fine grained optimiza-

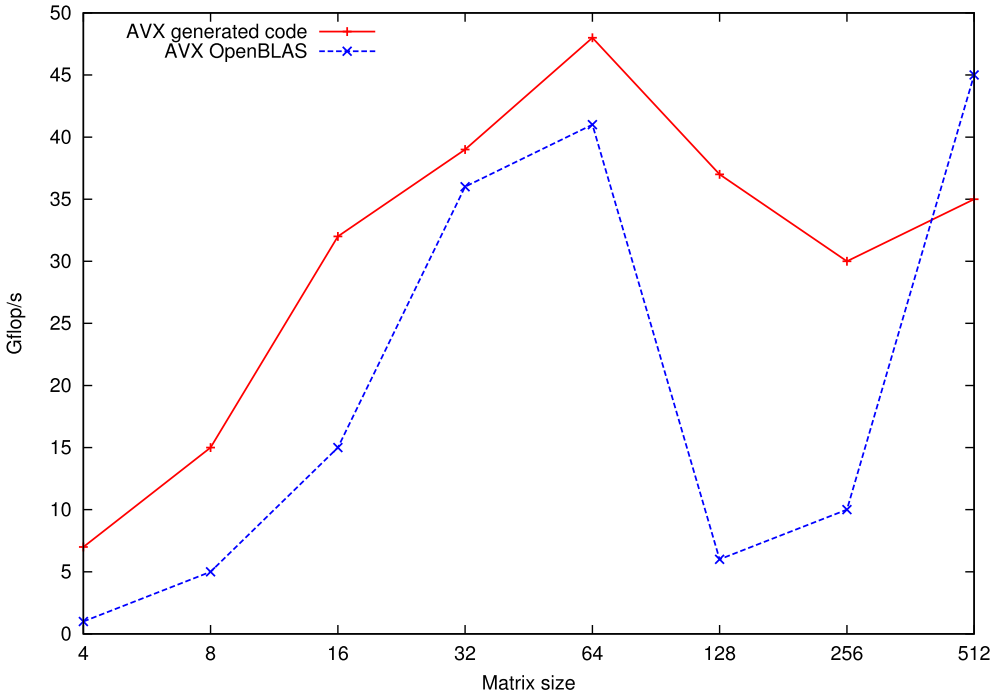


Figure 2.3: GEMV performance on Intel i5-7200 processor using Advanced Vector Extensions (AVX)

tions generates a better code than the hand-written assembly approach of the OpenBLAS library.

We exhibit performance drops similar to OpenBLAS due to $L1 \rightarrow L2$ misses. Register spilling also happens once we reach 512×512 elements. The combination of our template based unrolling and Boost.SIMD shows that it is indeed possible to generate ARM NEON code from high-level C++ with zero abstraction cost.

2.6 . Conclusion

This chapter presented the details of generating an optimized level 2 BLAS routine GEMV. As a key difference with respect to highly tuned OpenBLAS routine, our generated code is designed to give the best performance with a minimum programming effort for rather small matrices that fit into the L1 cache. Compared to the best open source BLAS library, OpenBLAS, the automatically generated GEMV codes show competitive speedups for most of the matrix sizes tested in this section, that is for sizes ranging from 4 to 512. Therefore through this section and the example of the level 2 BLAS routine GEMV, we showed that it is possible to employ modern generative programming techniques for the implementation of dense linear algebra routines that

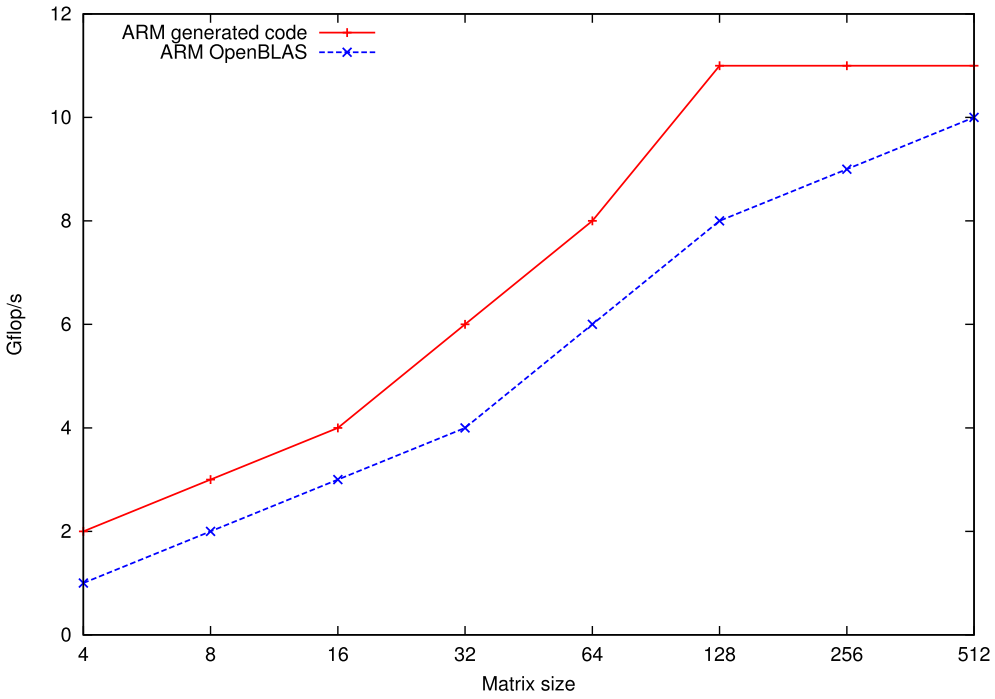


Figure 2.4: GEMV performance on ARM Cortex A57 processor

are highly optimized for small matrix sizes.

Now that we know that template metaprogramming is a viable technique for generating portable high performance kernels that can compete with hand written assembly, we will dive deeper into newer C++ metaprogramming techniques and apply them for compile-time parsing and code generation.

Part II

C++ metaprogramming beyond templates

3 - Compile time benchmarking methodology

With template metaprogramming libraries like Eigen[26], Blaze[30], or CTRE [20] becoming more widespread, there is an increasing interest for compile time computation. These needs might grow even larger as C++ embeds more features over time to support and extend this kind of practices, like compile time containers [17] or static reflection[61]. However, there is still no clear cut methodology to compare the performance impact of different metaprogramming strategies. As new C++ features allow for new techniques with alleged better compile time performance, no scientific process can back up those claims.

In this chapter, I introduce **ctbench**, which is a set of tools for compile time benchmarking and analysis in C++. It provides developer-friendly tools to define and run benchmarks, then aggregate, filter out, and plot the data to analyze them. As such, ctbench aims to be a foundational layer of a proper scientific methodology for analyzing compile time program behavior.

ctbench puts an emphasis on software quality. The goal was not just to develop a plotting tool for a single compilation time analysis, but to provide a repeatable process along with a robust implementation to improve compilation time performance analysis as a whole.

ctbench was presented at CPPP 2021 [45], and at Meeting C++ 2022 [37]. It was also published at the Journal of Open Source Science [46]. It is currently available as an open source project[1] and as a package through the Arch User Repository[2] and vcpkg[3].

3.1 . State of the art

C++ template metaprogramming raised interest for allowing computing libraries to offer great performance with a very high level of abstraction. Instead of building representations of calculations at runtime for interpretation, they are built at compile time to be transformed directly into programs.

As metaprogramming became easier with C++ 11 and C++ 17, it became more frequently used. Consequently, developers now have to bear with longer compilation times, often without being able to explain and improve them.

[1] https://github.com/jpenuchot/ctbench
[2] https://aur.archlinux.org/packages/ctbench-git
[3] https://vcpkg.io/en/packages

Therefore being able to measure compilation times becomes increasingly important and being able to profile and explain them as well.

This need turned into a variety of projects that aim to bring novel techniques to analyze the compile time performance of C++ metaprograms and metaprogramming techiniques beyond black box A/B comparisons. Moreover, many compile-time benchmarking tools are either interactive on benchmarking. As such, they are not adequate for the scientific study of compilation times, especially in the field of HPC where reproducibility is a known issue [6].

3.1.1 . Metabench

Metabench[18] instantiates variably sized benchmarks using Embedded Ruby (ERB) templating and plots compiler execution time, allowing scaling analyses of metaprograms. Its output is a series of web-based interactive graphs. The ERB templates are used to generate C++ programs for measuring their total compilation time. This approach is compiler-agnostic, it allows to compare not just metaprograms but compilers as well.

However, Metabench was not updated since 2019, and its design makes it difficult to modify and reuse it for the assessment of metaprogramming techniques performance in a scientific context: non-CMake scripts are embedded in CMake strings making it difficult to modify and debug them, benchmarks are not standalone as they are embedded in the Metabench project itself, and web-based visualizations cannot be embedded in scientific publications.

3.1.2 . Templight

Templight [49] is the first effort to instrument Clang with a profiler. It aims to provide tools to measure resource usage (CPU time, memory usage, and file I/O) of template instantiations, and display the data in a GUI. It also provides debugging features, allowing to set breakpoints to interrupt template instantiations.

Templight is meant to be used as an interactive tool for template instantiation profiling and debugging. It does not provide a command line interface for the automated export of profiling data or graphs, nor can it be used to compare template instantiations.

3.1.3 . Clang's built-in profiler

Clang provides a built-in profiler [3] that generates in-depth time measurements of various compilation steps, which can be enabled by passing the `-ftime-trace` flag. Its output contains data that can be directly linked to symbols in the source code, making it easier to study the impact of specific symbols on various stages of compilation. The output format is a JSON file meant to be compatible with Chrome's flame graph visualizer, that contains a series of time events with optional metadata like the mangled C++ symbol or the file

related to an event. The profiling data can then be visualized using tools such as Google's Perfetto UI as shown in figure 3.1.

The JSON files have a rather simple structure. They contain a `traceEvents` field that holds an array of JSON objects, each one of them containing a `name`, a `dur` (duration), and `ts` (timestamp) field. It may also contain additional data in the field located at `/args/data`, as seen in listing 3.1. Duration and timestamps are expressed in microseconds.

Listing 3.1: Time trace event generated by Clang's internal profiler

```
{
  "pid": 8696,
  "tid": 8696,
  "ph": "X",
  "ts": 3238,
  "dur": 7,
  "name": "Source",
  "args": {
    "detail": "/usr/include/bits/wordsize.h"
  }
}
```

In this example the `/args/detail` field indicates the file that's being processed during this trace event. This field may also contain details related to symbols being processed for events like `InstantiateFunction`.

Clang's profiling data is both very exhaustive as it covers all compilation stages, but also very insightful as every timer event is annotated with metadata that relates to specific C++ symbols. It can be extracted automatically by adding a compilation flag, and there is a multitude of libraries that can parse its JSON output.

As such, the choice was made to reuse this data by making a tool to help automate the extraction, analysis, and comparison of time trace files for variable size compile time benchmarking.

3.2 . ctbench features

ctbench implements a new methodology for the analysis of compilation times: it allows users to define C++ sizable benchmarks to analyze the scaling performance of C++ metaprogramming techniques, and compare techniques against each other.

The project was designed to be an easy to install, and an easy to use alternative to current compile time benchmarking tools. The public API is based on CMake as the project is aimed at C++ software developers and researchers. Therefore, using familiar tools lowers barriers to entry to those who might want to contribute to it.

3.2.1 . CMake API for benchmark and graph target declarations

ctbench is meant to be used as a CMake package, and its API is meant to be as stable as possible. The API is relatively simple, and all of it is implemented in a single CMake script file which provides the following:

- Benchmark declaration functions, for the declaration and instantiation of user-defined sizable benchmark cases:
  - `ctbench_add_benchmark` takes a user-defined benchmark name, a C++ source file, as well as benchmark range parameters: iteration begin, end, and step parameters, as well as the number of samples per benchmark iteration. Benchmark iteration targets declared with this function are compiled with their sizes passed as the `BENCHMARK_SIZE` define (*i.e.* Clang will be invoked successively with parameters `-DBENCHMARK_SIZE=0` ... `-DBENCHMARK_SIZE=N`).
  - `ctbench_add_benchmark_for_range`, provides an interface similar to `ctbench_add_benchmark` except that benchmark range parameters are taken as a single list. This interface only exists to provide a more compact function call.
  - `ctbench_add_benchmark_for_size_list`, similar to the previous ones, provides a way to define benchmarks for a given size list instead of a range.
  - And finally, `ctbench_add_custom_benchmark` inherits the interface of `ctbench_add_benchmark` with an addition: a callback function name must be passed as a parameter. It will be called for each benchmark iteration target definition with the name and size of the target. This allows users to set compiler parameters other than ctbench's preprocessor directive.
- `ctbench_add_graph` allows the declaration of a benchmark. It takes a user-defined name, a path to a JSON configuration file, an output destination, and a list of benchmark names defined using the functions mentioned above. ctbench expects JSON files to include information relative to the graphs themselves: the plotter being used, predicates to filter out time trace events, output format, and so on. Each plotter has its own set of parameters.
- `ctbench_set_compiler` and `ctbench_unset_compiler` are a pair of commands that using different compilers for benchmarks. CMake does not allow using more than one compiler within a CMake build, but ctbench's compiler launcher can be used through these commands to work around that limitation and run benchmarks for compiler performance comparisons.

- The `ctbench-graph-all` target, which allows to build all the graphs declared with ctbench functions.

Benchmark data is laid out in folders under the following layout:
`<benchmark name>/<iteration size>/<repetition number>.json`.

3.3 . ctbench internal design

Besides the CMake scripts, ctbench is organized into several components: **compiler-launcher**, which is a simple executable that extends CMake's feature set, and **grapher** a C++ library and CLI tool that reads benchmark data and draws plots.

3.3.1 . compiler-launcher: working around CMake's limitations

CMake has its own limitations that would make the implementation of ctbench impractical, or even impossible without a compiler launcher. Thankfully, CMake does support compiler launchers through the target property `CMAKE_CXX_COMPILER_LAUNCHER` which allows us to work around those issues.

The main issue that compiler-launcher was meant to solve is the lack of a CMake interface to retrieve the path of the binaries it generates (which is needed to locate JSON trace files), and the lack of a Clang option to set the output path of time trace files until Clang 16.

The functionality of the wrapper was since extended to implement other features such as compilation time measurement for GCC or compiler permutation to enable the use of several C++ compilers inside a single CMake build.

3.3.2 . grapher: reading and plotting benchmark results

The grapher sub-project provides boilerplate code such as data structures for benchmark data representation, and data wrangling functions for reading benchmark data. It also serves as a framework for implementing and experimenting new data visualization techniques that might be relevant for compile time performance analysis.

- A set of data structures is provided to handle profiling data:

  `benchmark_instance_t` contains several data file paths to several repetitions of the same benchmark instantiated at the same size, as well as the instantiation size for all the repetition files:

```
struct benchmark_instance_t {
  unsigned size;
  std::vector<std::filesystem::path> repetitions;
};
```

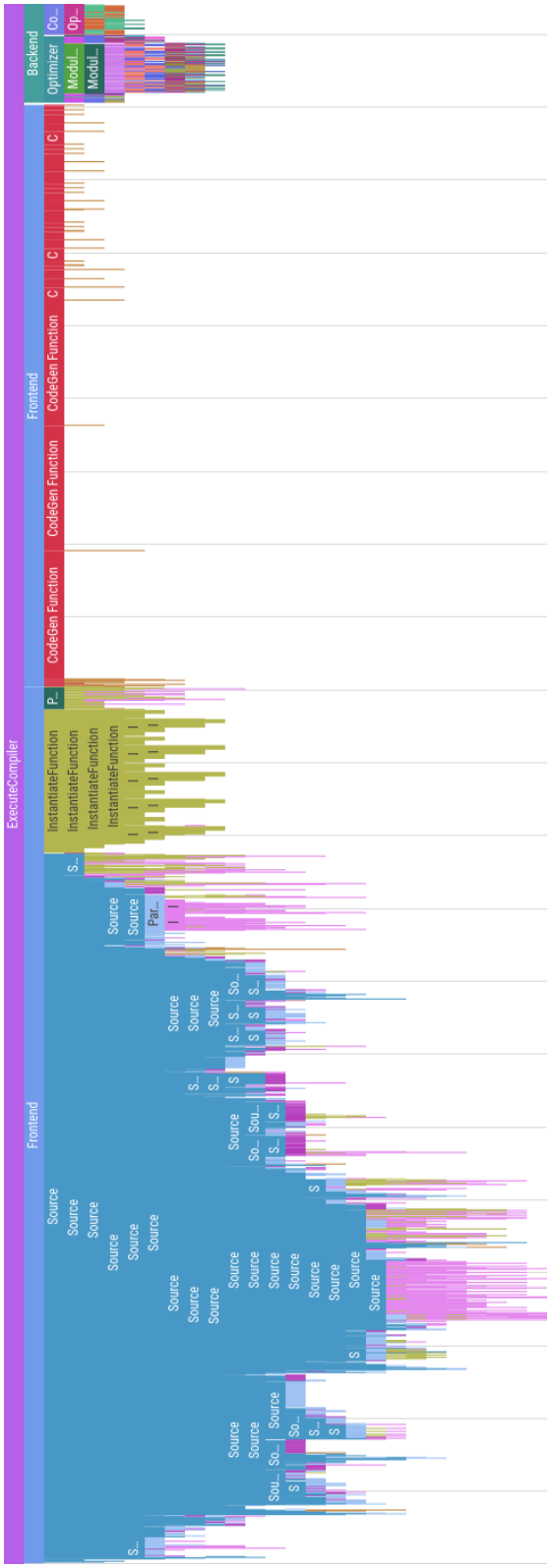


Figure 3.1: Clang time trace file in Perfetto UI

`benchmark_case_t` contains several `benchmark_instance_t` elements, as well as the name of the benchmark case:

```
struct benchmark_case_t {
  std::string name;
  std::vector<benchmark_instance_t> instances;
};
```

And finally, `benchmark_set_t` is used to gather different benchmark cases that will be compared together:

```
using benchmark_set_t =
  std::vector<benchmark_case_t>;
```

Note that JSON data is not stored directly. This is intentional since a profiling file for a single benchmark repetition can reach volumes up to several hundreds megabytes, therefore data loading is delayed to prevent RAM overcomsumption.

- Different plotter engines are provided, and they all follow the same abstraction pattern by inheriting `plotter_base_t`. Currently only few plotters are considered to be usable (`compare`, `compare_by`, and `stack`). Another one called `grouped_histogram` was implemented for experimenting but ended up being dropped.

  The `plotter_base_t` abstraction has a simple specification:

  ```
  struct plotter_base_t {
    virtual ~plotter_base_t() = default;
    virtual void plot(
      benchmark_set_t const &bset,
      std::filesystem::path const &dest,
      grapher::json_t const &config) const = 0;
    virtual grapher::json_t get_default_config() const =
        0;
  };
  ```

  To implement a new plotting strategy, one simply needs to create a structure that takes a `benchmark_set_t` and a JSON configuration structure as an input.

  This level of modularity is central to the design of ctbench as the goal is not just to provide a benchmarking tool that is easy to use, but also a platform for researchers to experiment new ways to visualize compilation profiling data.

- A JSON predicate engine is provided by the grapher library. It is able to convert JSON expressions into rich predicates that can be embedded in graph configuration files to target relevant compilation trace events.

- The CLI interface is part of the grapher project as a supplement to the CMake API. It can be used to generate graphs without using the CMake API, or for other things such as generating default configuration files.

3.4 . Sample use case

This example covers a short yet practical example of ctbench usage. We want to calculate the sum of a series of integers known at compile-time, using a type template to store unsigned integer values at compile-time.

We will be comparing the compile-time performance of two implementations: one based on a recursive function template, and another one based on C++ 11 parameter pack expansion.

First we need to include `utility` to instantiate our benchmark according to the size parameter using `std::make_index_sequence`, and define the compile-time container type for an unsigned integer:

```
#include <utility>

/// Compile-time std::size_t
template <std::size_t N> struct ct_uint_t {
  static constexpr std::size_t value = N;
};
```

The first version of the metaprogram in listing 3.2 is based on a recursive template function, which we believe to offer poor compile-time performance due to the many template instantiation it may trigger. The version in listing 3.3 relies on C++ 11 parameter pack expansion, which we believe to offer much better scaling.

Listing 3.2: Compile-time sum: recursive implementation

```
template <typename... Ts> constexpr auto sum();

template <> constexpr auto sum() {
  return ct_uint_t<0>{};
}
template <typename T> constexpr auto sum(T const &) {
  return T{};
}

template <typename T, typename... Ts>
constexpr auto sum(T const &, Ts const &...tl) {
  return ct_uint_t<T::value +
                   decltype(sum(tl...))::value>{};
}
```

Listing 3.3: Compile-time sum: parameter pack implementation

```
template <typename... Ts> constexpr auto sum();

template <> constexpr auto sum() {
  return ct_uint_t<0>{};
}

template <typename... Ts>
constexpr auto sum(Ts const &...) {
  return ct_uint_t<(Ts::value + ... + 0)>{};
}
```

Listing 3.4: Benchmark driver code

```
template <typename = void> constexpr auto foo() {
  return
      []<std::size_t... Is>(std::index_sequence<Is...>) {
        return sum(ct_uint_t<Is>{}...);
      }(std::make_index_sequence<BENCHMARK_SIZE>{});
}

constexpr std::size_t result = decltype(foo())::value;
```

Both versions share the same interface, and thus the same driver code in listing 3.4 is the same for both cases. It takes care of scaling the benchmark according to `BENCHMARK_SIZE`, which is defined by ctbench through the CMake API.

We may now declare the benchmark targets at the desired sizes and repetition number, as well as the graph target to draw and compare the results. This is done through the CMake code shown in listing 3.5.

The configuration file for the graph generator is stored in `compare-all.json`. It contains parameters such as the size of the benchmark, output formats, axis labels, and JSON pointers to designate the values to observe as shown in listing 3.6. In this case, it selects the `compare_by` plotter to generate one plot for each value pair designated by the JSON pointers in `key_ptrs`, namely `/name` and `/args/detail`. The first pointer designates the LLVM timer name, and the second *may* refer to metadata such a C++ symbol, or a C++ source filename. The `demangle` option may be used to demangle C++ symbols using LLVM.

Listing 3.5: ctbench benchmark and graph declarations

```
ctbench_add_benchmark(variadic_sum.expansion
  variadic_sum/expansion.cpp ${BENCH_RANGE} ${BENCH_REP})
ctbench_add_benchmark(variadic_sum.recursive
  variadic_sum/recursive.cpp ${BENCH_RANGE} ${BENCH_REP})

ctbench_add_graph(variadic_sum-graph compare-all.json
  variadic_sum.expansion variadic_sum.recursive)
```

Listing 3.6: ctbench graph configuration

```
{
  "plotter": "compare_by",
  "legend_title": "Timings",
  "x_label": "Benchmark size factor",
  "y_label": "Time (microsecond)",
  "draw_average": true,
  "demangle": false,
  "draw_points": false,
  "width": 800,
  "height": 400,
  "key_ptrs": ["/name", "/args/detail"],
  "value_ptr": "/dur",
  "plot_file_extensions": [".svg"]
}
```

The result is a series of graphs, each one designating a particular timer event, specific to a source or a symbol whenever it is possible (ie. whenever metadata is present in the `/args/detail` value of a timer event). Each graph compares the evolution of these timer events in function of the benchmark size.

The following graphs were generated on a Lenovo T470 with an Intel i5 6300U and 8GB of RAM. The compiler is Clang 14.0.6, and Pyperf [56] was used to turn off CPU frequency scaling to improve measurement precision.

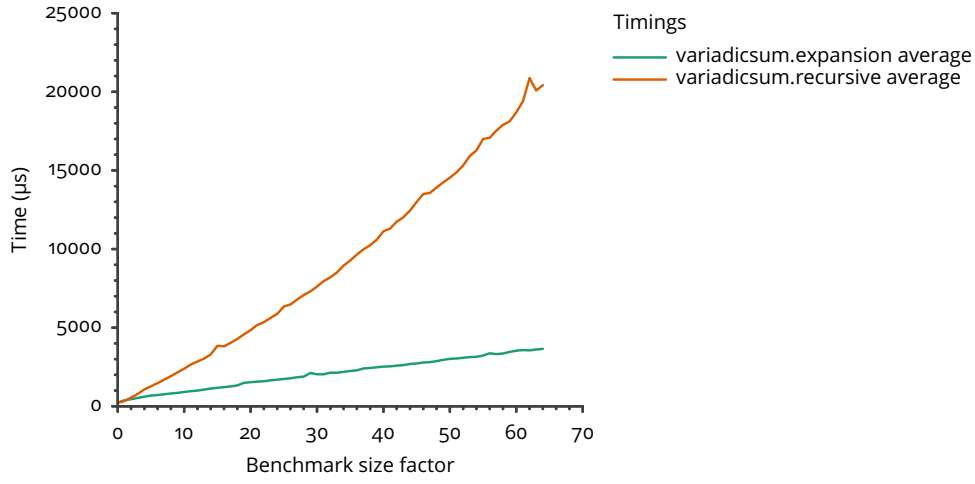


Figure 3.2: InstantiateFunction measurement for the foo<void> symbol

The first timer we want to look at in figure 3.3 is ExecuteCompiler, which measures the total compilation time. Starting from there we can go down in the timer event hierarchy to take a look at frontend and backend execution times, and finally InstantiateFunction as it makes up for most of the Frontend execution time.

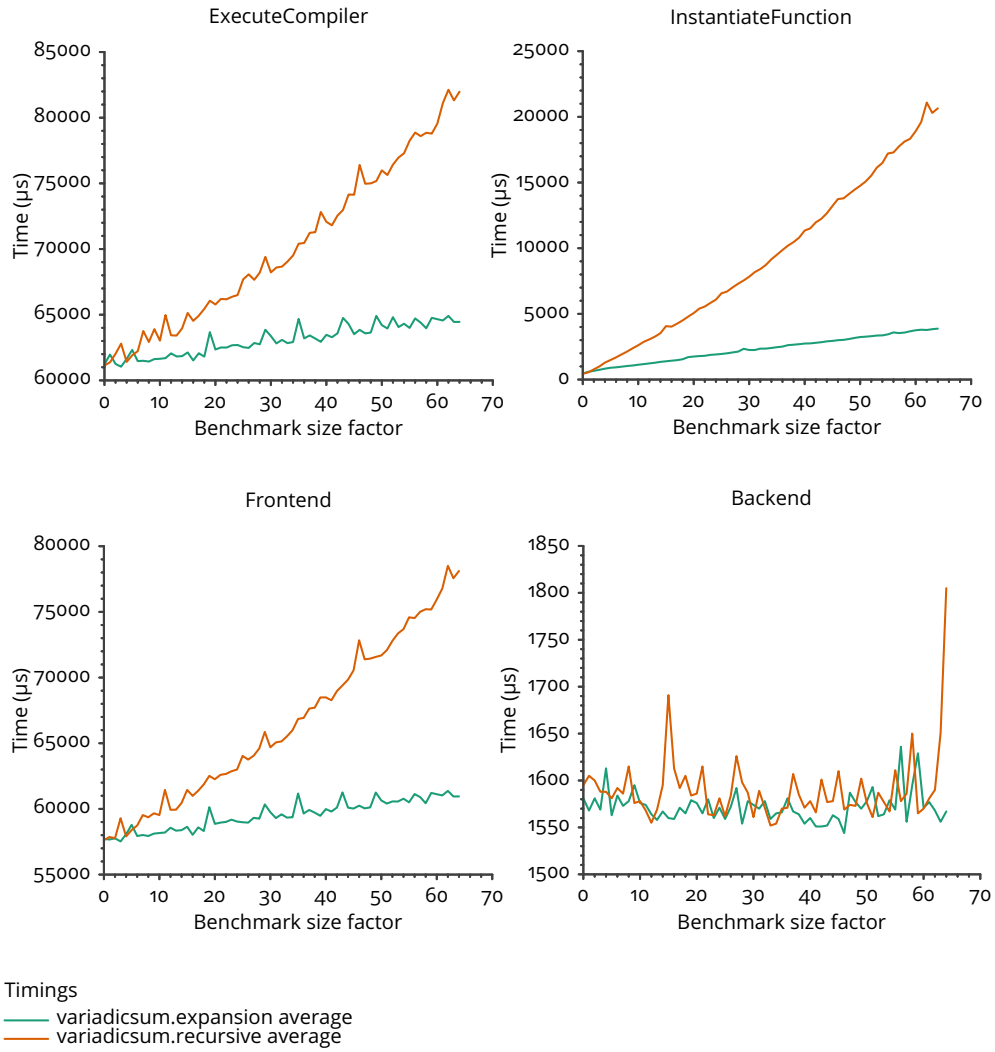


Figure 3.3: Timer aggregate measurements

The backend is not being impacted here, supposedly because this is purely a compile-time program and the output program is empty. However this might not be the case for all metaprograms, and metaprograms might have different impacts on the backend as they may generate programs in different ways (ie. generate more symbols, larger symbols, more data structures, etc.).

Finally, we can take a look at `InstantiateFunction/foovoid.svg`, shown in figure 3.2, which measures the InstantiateFunction event time specifically for the `foo<void>()` symbol, which is our driver template function. Using Perfetto UI to look at the timer event hierarchy, we determined that this event englobes all the events that grow with the instantiation size.

3.5 . Conclusion

Deep analysis of compile time scaling becomes necessary to understand compile time performance of metaprogramming techniques. Until now, no tool was really able to combine the depth of profiling data analysis with variable size compile time benchmarking. Moreover, there was no tool that could really fit in scientific work where reproductibility is a necessity.

ctbench answers that need as a simple, powerful, and extensible open-source solution that is peer-reviewed and distributed as a reusable software package.

Such level of detail and granularity in the analysis of compile-time benchmarks was never reached before, and may help us set better practices to improve the compile-time performance of metaprograms, and give better insight on compiler performance as well.

4 - Constexpr parsing for HPC

4.1 .Introduction

C++ is often touted as a *Zero-Cost Abstraction* langage due to some of its design philosophy and its ability to compile high level abstractions into very efficient binary code. Some more radical techniques can be used to force the compiler to interpret C++ code as a DSEL.

In the field of High Performance Computing, C++ users are often driven to use libraries built on top of those idioms like Eigen[26] or Blaze[30, 31]. They all suffer from a major limitation: by being tied to the natural C++ syntax, they can't express nor embed arbitrary languages.

In this chapter, we try to demonstrate that the new features of C++ 23 related to compile time programming are able to help developers designing DSELs with arbitrary syntax by leveraging constexpr computations, compile time dynamic objects and lazy evaluation through lambda functions. After contextualizing our contribution in the general DSELs domain, this chapter will explain the core new techniques enabled by C++ 23 and how we can apply to build two different DSELs with their own non-C++ syntax. We'll also explore the performances of those DSELs in term of compile time to assess their usability in parallel programming.

4.2 . Advanced constexpr programming

As mentioned in 1.2, the constexpr allows variables and functions to be used in constant evaluation, making a whole subset of the C++ language itself usable for compile-time logic.

As the language evolves, this subset becomes larger as more C++ become usable in constexpr functions. Some of the most notable evolutions are allowing constexpr memory allocations and virtual function calls in constant evaluations since C++ 20. These two additions make dynamic memory and heritage-based polymorphism possible in constexpr functions. Therefore more regular C++ code can be executed at compile time, including parsers for arbitrary languages.

This section will cover advanced technicalities of constexpr programming, including constexpr memory allocation and the restrictions that make constexpr allocated memory difficult to use for C++ code generation.

4.2.1 . constexpr functions and constexpr variables

Functions that are constexpr qualified can still be used in other contexts than constant evaluation happening at compile time. In non-constant evaluation, constexpr functions can still call non-constexpr functions. But in constant evaluations, constexpr functions must only call other constexpr functions. This applies to methods as well. In order to make a C++ class or structure fully usable in constant evaluations, its methods –including the constructors and destructor– must be constexpr.

Variables that are constexpr qualified can be used in constant expressions. Note that they are different from non-constexpr variables used in constexpr functions. There are more requirements on constexpr variables. Their values must be literal, meaning that memory allocated in constexpr function bodies cannot be stored in constexpr variables.

4.2.2 . constexpr memory allocation and constexpr virtual function calls

With C++ 20, `std::allocate` and `std::deallocate` became constexpr functions, allowing memory allocations to happen in constant evaluations.

However constexpr allocated memory is not transient, *i.e.* memory allocated in constant expressions cannot be stored in constexpr variables, and NTTPs cannot hold pointers to constexpr allocated memory either.

Note that this restriction does not mean that data stored in constexpr memory cannot be passed through. There are techniques to use data in constexpr allocated memory

Listing 4.1: Illustration of constraints on constexpr allocated memory

```
// Constexpr function generate returns a non-literal value
constexpr std::vector<int> generate() { return
    {1,2,3,4,5}; }

// Function template foo takes a polymorphic NTTP
template<auto bar> constexpr int foo() { return 1; }

// generate's return value cannot be stored in a constexpr
     variable
// or used as a NTTP, but it can be used to produce other
    literal

// constexpr auto a = generate();          // ERROR
constexpr auto b = generate().size();      // OK
// constexpr auto c = foo<generate()>();   // ERROR
constexpr auto d = foo<&generate>();       // OK
```

Let's have a closer look at the four assignment cases:

- Case `a`: `generate`'s return value is non-literal and therefore cannot be stored in a constexpr variable.
- Case `b`: `generate`'s return value is used in a constant expression to produce a literal value. Therefore the expression's result can be stored in a constexpr variable.
- Case `c`: similarly to case `a`, `generate`'s return value cannot be used as an NTTP because it is not a literal value.
- Case `d`: function references are allowed as NTTPs.

Notice how the last example works around restrictions of constexpr allocations by using a generator function instead of passing the non-empty `std::vector<int>` value directly. This technique along with the definition of lambdas can be used to explore more complex structures returned by constexpr functions such as pointer trees.

Moreover, constexpr allocated memory being non transient does not mean that its content cannot be transferred to NTTP compatible data structures.

Listing 4.2: Transformation from a dynamic vector to a constexpr static array

```
constexpr auto generate_as_array() {
  constexpr std::size_t array_size =
      generate().size();
  std::array<int, array_size> res{};
  std::ranges::copy(generate(), res.begin());
  return res;
}

constexpr auto e = generate_as_array();          // OK
constexpr auto f = foo<generate_as_array()>(); // OK
```

Listing 4.2 shows how `generate`'s result can be evaluated into a static array. Static arrays are literal as long as the values they hold are literal. Therefore the result of `generate_as_array` can be stored in a constexpr variable or used directly as an NTTP for code generation.

The addition of constexpr memory allocation goes hand in hand with the ability to use virtual functions in constant evaluations. This feature allows calls to virtual functions in constant expressions [16]. This allows heritage-based polymorphism in constexpr programming when used with constexpr allocation of polymorphic types.

Listing 4.3: `tree_t` type definition

```
/// Enum type to differentiate nodes
enum node_kind_t {
  constant_v,
  add_v,
};

/// Node base type
struct node_base_t {
  node_kind_t kind;

  constexpr node_base_t(node_kind_t kind_) : kind(kind_)
     {}
  constexpr virtual ~node_base_t() = default;
};

/// Tree pointer type
using tree_ptr_t = std::unique_ptr<node_base_t>;

/// Checks that an object is a tree generator
template <typename T>
concept tree_generator = requires(T fun) {
  { fun() } -> std::same_as<tree_ptr_t>;
};

/// Constant node type
struct constant_t : node_base_t {
  int value;
  constexpr constant_t(int value_)
      : node_base_t(constant_v), value(value_) {}
};

/// Addition node type
struct add_t : node_base_t {
  tree_ptr_t left;
  tree_ptr_t right;
  constexpr add_t(tree_ptr_t left_, tree_ptr_t right_)
      : node_base_t(add_v), left(std::move(left_)),
        right(std::move(right_)) {}
};

/// Generates an arbitrary tree
constexpr tree_ptr_t gen_tree() {
  return std::make_unique<add_t>(
      std::make_unique<add_t>(std::make_unique<constant_t
         >(1),
                              std::make_unique<constant_t
                                 >(2)),
      std::make_unique<constant_t>(3));
}
```

4.3 . Constexpr abstract syntax trees and code generation

The addition of constexpr memory and virtual function calls allows constexpr implementation of ASTs, akin to those found in modern C++ compiler frontends like Clang. However, these ASTs cannot be used directly as NTTPs for generating structures and functions.

This section will cover the implementation of constexpr ASTs, and techniques to work around the limitations that prevent their direct use as NTTPs either through functional wrapping techniques, or through their convertion of into values that satisfy NTTP requirements.

4.3.1 . Code generation from pointer tree data structures

In this subsection, we introduce three techniques that will allow us to use a pointer tree generated from a constexpr function as a template parameter for code generation.

To illustrate them, we use a minimalistic use case. A generator function returns a pointer tree representation of addition and constant nodes, which we will use to generate functions that evaluate the tree itself.

Listing 4.3 shows the type definitions, a concept to match tree generator functions, as well as the generator function itself. This is a common way to represent trees in C++, but the limits mentioned in 4.2 make it impossible to use the result of `gen_tree` directly as a template parameter to generate code.

The techniques we will use to pass the result as a template parameter are:

- passing functions that return nodes as NTTPs,
- converting the tree into an expression template representation,
- serializing the tree into a dynamically allocated array, then converting the dynamic array into a static array that can be used as a NTTP,

The compilation performance measurements in 5.1.3 will rely on the same data passing techniques, but with more complex examples such as embedded compilation of Brainfuck programs, and of LaTeXmath formulae into high performance math computation kernels.

Pass By Generator (PBG)

One way to use dynamically allocated data structures as template parameters is to pass their generator functions instead of their values. You may have noted in listing 4.1 that despite `generate`'s return value being non-literal, the function itself can be passed as a NTTP.

Its result can be used to produce literal constexpr results, and the function itself can be used in generator lambda functions defined at compile-time.

Listing 4.4: Code generation with the pass by generator technique

```
/// Accumulates the value from a tree returned by
    TreeGenerator.
/// TreeGenerator() is expected to return a std::
    unique_ptr<tree_t>.
template <tree_generator auto Fun> constexpr auto codegen
    () {
  static_assert(Fun() != nullptr, "Ill-formed tree");

  constexpr node_kind_t Kind = Fun()->kind;

  if constexpr (Kind == add_v) {
    // Recursive codegen for left and right children:
    // for each child, a generator function is generated
    // and passed to codegen.
    auto eval_left = codegen<[]() {
      return static_cast<add_t &&>(*Fun()).left;
    }>();
    auto eval_right = codegen<[]() {
      return static_cast<add_t &&>(*Fun()).right;
    }>();

    return [=]() { return eval_left() + eval_right(); };
  }

  else if constexpr (Kind == constant_v) {
    constexpr auto Constant =
        static_cast<constant_t const &>(*Fun()).value;
    return [=]() { return Constant; };
  }
}
```

In listing 4.4, we show how a constexpr pointer tree result can be visited recusively using generator lambdas to pass the subnodes' values, and used to generate code.

This technique is fairly simple to implement as it does not require any transformation into an ad hoc data structure to pass the tree as a type or NTTP.

The downside of using this value passing technique is that the number of calls of the generator function is proportional to the number of nodes. Experiments in 5.1.3 highlight the scaling issues induced by this code generation method. And while it is very quick to implement, there are still difficulties related to constexpr memory constraints and compiler or library support.

Note that GCC 13.2.1 is still unable to compile such code. And while Clang 17.0.6 supports it, getting this technique to compile can be difficult as constexpr memory rules are enforced in an erratic way, and often with poorly explained compiler errors.

For example, if the expression in an `if constexpr` condition involves constexpr allocated memory, the result of the condition must be stored in an intermediate constexpr variable to avoid a compiler error:

```
constexpr std::vector<int> foo() {
  return {{1, 2, 3, 4, 5}};
}

// Compilation error
if constexpr (foo().size() == 5);

// OK
if constexpr (constexpr bool val =
                  foo().size() == 5;
              val);
```

With no constexpr intermediate variable, Clang tells that the condition expression is not a constant expression, despite the condition expression being usable to initialize a constexpr variable.

Pass by generator and expression templates

Another method consists in transforming constexpr ASTs into expression templates as an intermediate representation. This serves as a proof that making constexpr programming interoperable with older metaprogramming paradigms is feasible, but it will also help us assess the viability of using the pass by generator method in coordination with expression templates.

The first step consists in creating the intermediate representation itself, which is a type-based representation of the AST.

Listing 4.5: cap

```
/// Type representation of a constant
template <int Value> struct et_constant_t {};
/// Type representation of an addition
template <typename Left, typename Right> struct et_add_t
    {};
```

Listing 4.5 shows the type-based intermediate representation of the AST representation from listing 4.3. It is meant to be a one to one reimplementation of its value-based equivalent.

From there, we can implement a metaprogram that transforms a constexpr AST into this new type-based representation.

Listing 4.6: AST to Expression Template transformation

```
template <tree_generator auto Fun>
constexpr auto to_expression_template() {
  static_assert(Fun() != nullptr, "Ill-formed tree");

  constexpr node_kind_t Kind = Fun()->kind;

  if constexpr (Kind == add_v) {
    using TypeLeft =
        decltype(to_expression_template<[]() {
          return static_cast<add_t &&>(*Fun()).left;
        }>());
    using TypeRight =
        decltype(to_expression_template<[]() {
          return static_cast<add_t &&>(*Fun()).right;
        }>());

    return et_add_t<TypeLeft, TypeRight>{};
  }

  else if constexpr (Kind == constant_v) {
    constexpr auto Value =
        static_cast<constant_t &>(*Fun()).value;
    return et_constant_t<Value>{};
  }
}

template <int Value>
constexpr auto
codegen_impl(et_constant_t<Value> /*unused*/) {
  return []() { return Value; };
}

template <typename ExpressionLeft,
          typename ExpressionRight>
constexpr auto
codegen_impl(et_add_t<ExpressionLeft,
                      ExpressionRight> /*unused*/) {
  auto eval_left = codegen_impl(ExpressionLeft{});
  auto eval_right = codegen_impl(ExpressionRight{});
  return [=]() { return eval_left() + eval_right(); };
}

template <tree_generator auto Fun>
constexpr auto codegen() {
  using ExpressionTemplate =
      decltype(to_expression_template<Fun>());
  return codegen_impl(ExpressionTemplate{});
```

As shown in listing 4.6, it is very similar to the transformation code in 4.3.1 since it relies on the same technique to traverse the AST. However, its output is not a C++ lambda but an expression template, and as such proves that constexpr programming is interoperable with existing type-based paradigms, even when constexpr allocated memory is involved.

It is worth mentioning that both this technique and the previous one induce very high compilation times, as we will see in 5.1.

FLAT - AST serialization

To overcome the performance issues of the previously introduced techniques, we can try a different approach. Instead of passing trees as they are, they can be transformed into static arrays which can be used as NTTPs.

This transformation might sound odd at first: the generated ASTs are dynamic structures that need dynamic memory allocation, and static arrays have a fixed size. However, because the size of the output is known at compile time, it is possible to determine the size of the serialized AST in a first step to determine the size of the array, and then transfer the serialized content into the array.

In short: "static" array sizes are static at run-time but not at compile-time. This can be exploited to generate values that meet the requirements of NTTPs but still contain the same data as the ASTs they were generated from.

```
/// Serialized representation of a constant
struct flat_constant_t {
  int value;
};

/// Serialized representation of an addition
struct flat_add_t {
  std::size_t left;
  std::size_t right;
};

/// Serialized representation of a node
using flat_node_t = std::variant<flat_add_t,
    flat_constant_t>;

/// Defining max std::size_t value as an equivalent to std
    ::nullptr.
constexpr std::size_t null_index = std::size_t(0) - 1;
```

The first step consists in implementing the intermediate representation nodes which are similar to the AST nodes in which pointers are replaced with `std::size_t` indexes, and `nullptr` is replaced with an arbitrary value called `null_index`, as shown in listing 4.3.1. This intermediate representation also

replaces heritage polymorphism with the use of `std::variant`. These nodes are then stored in `std::vector` containers, and the indexes refer to the positions of other nodes within the same container.

Listing 4.7: constexpr AST serialization

```
constexpr std::size_t
serialize_impl(tree_ptr_t const &top,
               std::vector<flat_node_t> &out) {
  // nullptr translates directly to null_index
  if (top == nullptr) {
    return null_index;
  }

  // Allocating space for the destination node
  std::size_t dst_index = out.size();
  out.emplace_back();

  if (top->kind == add_v) {
    auto const &typed_top =
        static_cast<add_t const &>(*top);

    // Serializing left and right subtrees,
    // initializing the new node
    out[dst_index] = {flat_add_t{
        .left = serialize_impl(typed_top.left, out),
        .right =
            serialize_impl(typed_top.right, out)}};
  }

  if (top->kind == constant_v) {
    auto const &typed_top =
        static_cast<constant_t const &>(*top);
    out[dst_index] = {
        flat_constant_t{.value = typed_top.value}};
  }

  return dst_index;
}

constexpr std::vector<flat_node_t>
serialize(tree_ptr_t const &tree) {
  std::vector<flat_node_t> result;
  serialize_impl(tree, result);
  return result;
}
```

Listing 4.7 shows the implementation of the serialization step. Note that all the code that was presented so far is ordinary C++ code, except for the functions being constexpr. From there we can implement the last transformation, which consists in transferring the serialized data into a static array.

Listing 4.8: Definition of `serialize_as_array`

```
/// Evaluates a tree generator function into a serialized
    array.
template <tree_generator auto Fun>
constexpr auto serialize_as_array() {
  constexpr std::size_t Size = serialize(Fun()).size();
  std::array<flat_node_t, Size> result;
  std::ranges::copy(serialize(Fun()), result.begin());
  return result;
}
```

Listing 4.8 shows the implementation of a helper function that takes a constexpr tree generator function as an input, serializes the result, and returns it as a static array. This can be done because the generator function is constexpr, therefore it can be used to produce constexpr values. The size of the resulting `std::vector` can be stored at compile time to set the size of a static array. The elements contained in the resulting array are literal values since they do not hold pointers to dynamic memory, therefore the array itself is a literal value as well.

To complete the implementation, we must implement a code generation function that accepts a serialized tree as an input as shown in listing 4.9. Note that this function is almost identical to the one shown in listing 4.4. The major difference is that `TreeGenerator` is called only twice regardless of the size of the tree. This allows much better scaling as we will see in 5.1.3. The downside is that it requires the implementation of an ad hoc data structure and a serialization function, which might be more or less complex depending on the complexity of the original tree structure.

4.3.2 . Using algorithms with serialized outputs

Current token	Action	Stack
2	Stack 2	2
3	Stack 3	2, 3
2	Stack 2	2, 3, 2
*	Multiply 2 and 3	2, 6
+	Add 2 and 6	8

Figure 4.1: Reverse Polish Notation (RPN) formula reading example

Parsing algorithms may output serialized data. In this case, the serialization step described in 4.3.1 is not needed, and the result can be converted into a static array. This makes the code generation process rather straightforward as no complicated transformation is needed, while still scaling decently as we will see in 5.1.3 where we will be using a Shunting Yard parser [14] to parse math formulae to a RPN, which is its postfix notation.

Once converted into its postfix notation, a formula can be read using the following method: reading symbols in order, putting constants and variables on the top of a stack, and when a function f or operator of arity N is being read, unstacking N values and stack the result of f applied to the N operands.

Figure 4.1 shows a formula reading example with the formula `2 + 3 * 2`, or `2 3 2 * +` in reverse polish notation. Starting from there, we will see how code can be generated using RPN representations of addition trees in C++.

Listing 4.9: Code generation implementation for the flat backend

```
template <auto const &Tree, std::size_t Index>
constexpr auto codegen_aux() {
  static_assert(Index != null_index, "Ill-formed tree (
     null index)");

  if constexpr (std::holds_alternative<flat_add_t>(Tree[
     Index])) {
    constexpr auto top = std::get<flat_add_t>(Tree[Index])
       ;

    // Recursive code generation for left and right
       children
    auto eval_left = codegen_aux<Tree, top.left>();
    auto eval_right = codegen_aux<Tree, top.right>();

    // Code generation for current node
    return [=]() { return eval_left() + eval_right(); };
  }

  else if constexpr (std::holds_alternative<
     flat_constant_t>(
                          Tree[Index])) {
    constexpr auto top = std::get<flat_constant_t>(Tree[
       Index]);
    constexpr int Value = top.value;
    return []() { return Value; };
  }
}

/// Stores serialized representations of tree generators'
    results.
template <tree_generator auto Fun>
static constexpr auto tree_as_array = serialize_as_array<
    Fun>();

template <tree_generator auto Fun> constexpr auto codegen
    () {
  return codegen_aux<tree_as_array<Fun>, 0>();
}
```

In listing 4.10, we have the type definitions for an RPN representation of an addition tree as well as `gen_rpn_tree` which returns an RPN equivalent of `gen_tree`'s result.

Similar to the flat backend, an `eval_as_array` takes care of evaluating the `std::vector` result into a statically allocated array.

Listing 4.10: RPN example base type and function definitions

```
/// Type for RPN representation of a constant
struct rpn_constant_t {
  int value;
};

/// Type for RPN representation of an addition
struct rpn_add_t {};

/// Type for RPN representation of an arbitrary symbol
using rpn_node_t =
    std::variant<rpn_constant_t, rpn_add_t>;

/// RPN equivalent of gen_tree
constexpr std::vector<rpn_node_t> gen_rpn_tree() {
  return {rpn_constant_t{1}, rpn_constant_t{2},
          rpn_add_t{}, rpn_constant_t{3}, rpn_add_t{}};
}
```

In listing 4.11, we have function definitions for the implementation of `codegen` which takes an RPN generator function, and generates a function that evaluates the tree.

The code generation process happens by updating an operand stack represented by a `kumi::tuple`. It is a standard-like tuple type with additional element access, extraction, and modification functions. We are using it to store functions that evaluate parts of the subtree.

The algorithm reads **symbols** (*i.e.* functions, operators, constants, and variables) one by one from a **symbol queue**, and generates **operands** (*i.e.* C++ lambdas that return a result) that are stored on an **operand stack**. For each **symbol** being read from the queue:

- If a **constant** or **variable** is read, a lambda that evaluates its value is added to the operand stack. Since the operands are C++ lambdas, they can return anything users may want them to return, such as a C++ variable or even another function's result.
- If a **function** or **operator** of arity N is read, N operands will be consumed from the top of the *operand stack* to generate a new one. This new operand evaluates the consumed operands passed to the function corresponding to the symbol being read from the queue.

Once all symbols are read, there should be only one **operand** remaining on the operand stack: a function that evaluates the whole formula.

Listing 4.11: Codegen implementation for RPN formulas

```
template <auto const &RPNTree, std::size_t Index = 0>
constexpr auto
codegen_impl(kumi::product_type auto stack) {
  // The end result is the last top stack operand
  if constexpr (Index == RPNTree.size()) {
    static_assert(stack.size() == 1, "Invalid tree");
    return kumi::back(stack);
  }

  else if constexpr (std::holds_alternative<
                          rpn_constant_t>(
                          RPNTree[Index])) {
    // Append the constant operand
    auto new_operand = [=]() {
      return get<rpn_constant_t>(RPNTree[Index]).value;
    };
    return codegen_impl<RPNTree, Index + 1>(
        kumi::push_back(stack, new_operand));
  }

  else if constexpr (std::holds_alternative<rpn_add_t>(
                          RPNTree[Index])) {
    // Fetching 2 top elements
    // and popping them off the stack
    auto left = kumi::get<stack.size() - 1>(stack);
    auto right = kumi::get<stack.size() - 2>(stack);
    auto stack_remainder =
        kumi::pop_back(kumi::pop_back(stack));

    // Append new operand and process next element
    auto new_operand = [=]() { return left() + right(); };
    return codegen_impl<RPNTree, Index + 1>(
        kumi::push_back(stack_remainder, new_operand));
  }
}

template <auto Fun>
static constexpr auto rpn_as_array = eval_as_array<Fun>();

template <auto Fun> constexpr auto codegen() {
  return codegen_impl<rpn_as_array<Fun>, 0>(
      kumi::tuple{});
}
```

4.3.3 . Conclusion

We now have a set of methods that allows us to convert constexpr ASTs into C++ code, each with their flaws and advantages with regard to compilation times, implementation difficulty, or simply final code size. So far, all of them require advanced C++ knowledge and tinkering to get Clang to compile them. But as difficult as their implementation may be, it is still possible to design toolchains that minimize the amount of code related to AST transformation, and keep most of the computing in constexpr functions that are easier to write and maintain for most C++ developers.

We can alreaady notice a few differences between those techniques: pass by generator-based backends seem to have much higher compilation times, and they induce errors that are harder to diagnose despite the implementation being seemingly straightforward. The Flat backend however, requires more work to serialize the AST, but serializing a pointer tree does not require much advanced C++ knowledge. Moreover, the serialization code can be executed, debugged, and tested in a regular run-time environment.

So far, it seems that the pass by generator method is more suitable for prototyping, whereas the serialization method is more suitable in larger metaprograms that might benefit from having better readability and performance at scale.

5 - Applied constexpr parsing

Now that the idea of using constexpr functions to generate code was proven to work, it is time to put it into practice and observe the viability of its integration in a high performance computing development cycle.

I will present two examples in this chapter: the first one is a constexpr Brainfuck parser that serves as a simple use case for experimenting different methods to generate code from constexpr ASTs.

These observations guided the implementation of the second example, which is a constexpr parser for simple math languages. If features a code generation backend that supports high performance code generation via Blaze.

5.1 . Brainfuck parsing and code generation

Now that we introduced the various techniques to generate programs from pointer trees generated by constexpr functions, we will use them in the context of compile time parsing and code generation for the Brainfuck language. Therefore use data structures and code generation techniques introduced in subsection 4.3.1.

5.1.1 . Constexpr Brainfuck parser and AST

The Brainfuck AST is defined in the header shown in appendix .1.1. The header file also contains helper function definitions to handle AST nodes safely, such as `visit` which will be used in one of the code generation backends.

Here are the main data types:

- `node_interface_t`, which is a common base type for all AST nodes.
- `ast_token_t`, which represents a single AST token.
- `ast_block_t`, which represents an AST block, which simply is a `std::vector` of `std::unique_ptr<node_interface_t>`.
- `ast_while_t`, which represents a while conditional block. The instruction block itself is contained in an `ast_block_t` value.

The implementation of the AST are available in appendix .1.1 where you can observe that all the types are implemented as they would be for a regular Brainfuck parser, except all their methods are constexpr.

Listing 5.1: Definition of the AST visitor function

```
template <typename F>
constexpr auto visit(F f, ast_node_ptr_t const &p) {
  switch (p->get_kind()) {
  case ast_token_v:
    return f(static_cast<ast_token_t const &>(*p));
  case ast_block_v:
    return f(static_cast<ast_block_t const &>(*p));
  case ast_while_v:
    return f(static_cast<ast_while_t const &>(*p));
  }
}
```

The `visit` function implementation also looks like a regular C++ function as shown in listing 5.1. It is a higher order function that allows recursive operations on the AST to be carried in a type-safe manner.

The Brainfuck parser, again, looks like nothing special. For that reason I will not get into the implementation details. The function definition is available in appendix .1.2.

On the surface: the parser takes a pair of begin and end iterators as an input. It parses Brainfuck tokens until it reaches the end iterator or a while end token, and returns a pair containing an iterator pointing after the last parsed token and the parsing result.

When a while begin token is reached, it calls itself recursively and resumes parsing at the position of the iterator returned by the callee, which is right after the while block.

The main parsing function implementation (including the function prototype) is very compact: it fits in 40 lines of code with a max line width set to 84. It is no different from a regular Brainfuck parsing function except for it being constexpr, and it can actually be used as a regular C++ program.

These make it much easier to debug as it can be ran through a C++ debugger like GDB or LLDB, and also more maintainable as it does not require any template metaprogramming experience to understand the implementation. Additionally, constexpr execution enforces checks on memory allocations and deallocations as well as memory bound checking. Therefore testing functions in constexpr contexts can help finding memory safety issues.

Once the constexpr parser is implemented, the next step consists in figuring out how to transform its result, which contains dynamic memory, into C++ code.

As you may remember from subsection 4.2, there is no direct way to use values holding pointers to dynamic memory directly as NTTPs. Therefore it must be conveyed by other means or transformed into literal values to be used as template parameters for C++ code generation.

I implemented several of these workarounds to compare them. This will give us a clearer idea of their implementation difficulty, and they will enable us to run compilation time benchmarks to compare their compilation time performance.

5.1.2 . Pass-by-generator and Expression Template backends

The first backend implemented in the poacher project was the expression template backend, where the AST is transformed into a type-based intermediate representation as described in 4.3.1. It was later simplified to remove the intermediate representation transformation step, which gave the pass by generator backend.

Listing 5.2: expression template backend intermediate representation

```
template <typename... Nodes> struct et_block_t {
  constexpr et_block_t() {}
  constexpr et_block_t(Nodes const &...) {}
};

template <typename... Nodes> struct et_while_t {
  constexpr et_while_t() {}
  constexpr et_while_t(Nodes const &...) {}
};

template <token_t Token> struct et_token_t {};
```

The implementations of these two backends do not differ significantly from the ones described in 4.3.1 and 4.3.1: the generators that evaluate each node are passed as template parameters, only to work around the fact that pointers to constexpr allocated memory cannot be used in a NTTP.

Listing 5.2 shows the type-based intermediate representation used for the expression template backend. `et_block_t` and `et_while_t` structures contain a pack of arbitrary types that may be `et_token_t` elements for single tokens, or `et_while_t` elements for nested while loops.

From there, the code generation occurs in the same way as it did in 4.3.1: the expression template is traversed recursively using overloaded functions to generate the C++ code that corresponds to every while block, and down to every instruction in the expression template. The complete implementations of these backends are available in appendices .1.3 and .1.4.

5.1.3 . Serializing the AST into a literal value

The last remaining backend to implement is the one that transforms the AST into a serialized, NTTP compatible intermediate representation. The case of Brainfuck introduces a notable difference compared to the simple use case seen in 4.3.1: while AST nodes in Brainfuck can have an arbitrary number of children nodes, as opposed to add nodes in the simple use case I presented

earlier. This introduces a few technical differences with regard to serialization and code generation, which I will cover in detail in this subsection.

Listing 5.3: pass by generator intermediate representation type definitions

```
/// Represents a single instruction token
struct flat_token_t {
  token_t token;
};

/// Block descriptor at the beginning of every block
/// of adjacent instructions.
struct flat_block_descriptor_t {
  size_t size;
};

/// Represents a while instruction, pointing to
/// another instruction block
struct flat_while_t {
  size_t block_begin;
};

/// Polymorphic representation of a node
using flat_node_t =
    std::variant<flat_token_t,
                 flat_block_descriptor_t,
                 flat_while_t>;

/// AST container type
using flat_ast_t = std::vector<flat_node_t>;

/// NTTP-compatible AST container type
template <size_t N>
using fixed_flat_ast_t = std::array<flat_node_t, N>;
```

Listing 5.3 shows the definition of the serialized Brainfuck intermediate representation. It is similar to the intermediate representation in 4.3.1, except that it can represent ranges of instructions through `flat_block_descriptor_t`. This representation of node blocks implies that nodes within the same block must be stored contiguously after the serialization process.

To guarantee that nodes remain contiguous, the serialization process was split in two parts: a first phase that generates a vector of blocks where references are materialized by indexes to the blocks within that vector of vectors, and a second phase that merges these blocks together into a single vector, and replaces the block indexes by the beginning position of their subrange in the result of the merging process.

Listing 5.4: `block_gen_state_t` definition

```
struct block_gen_state_t {
  /// Flat AST blocks, result of block_gen
  std::vector<flat_ast_t> blocks;

  /// Keeping track of which block is being generated
  size_t block_pos = 0;

  /// Keeping track of the size of the AST
  size_t total_size = 0;
};
```

In listing 5.4 we begin by defining the structure called `block_gen_state_t` that contains the result of the first serialization phase, and other useful values for the serialization process.

Listing 5.5: Generic `block_gen` implementation

```
/// Extracts an AST into a vector
/// of blocks of contiguous operations
constexpr void block_gen(ast_node_ptr_t const &,
                         block_gen_state_t &);

// Node-specific overloads...

constexpr void block_gen(ast_node_ptr_t const &p,
                         block_gen_state_t &s) {
  visit([&s](auto const &v) { block_gen(v, s); }, p);
}
```

Listing 5.5 shows the implementation of the generic version of `block_gen`. It consists in a forward declaration followed by node-specific implementations which call the generic version recursively thanks to the `visit` function. Specialized overloads of the `block_gen` function follow the same interface: they are invoked with a node and the state of the block generation process as a mutable reference.

Listing 5.6 shows the implementation of `block_gen` overloads for all types derived from `node_interface_t`.

The most important one is the overload for `ast_block_t`. It transfers each block in the `blocks` vector of the `block_gen_state_t` structure by allocating them in the vector, and calling `block_gen` recursively for each node in the source block to effectively transfer them into the destination block.

Once the vector of blocks is generated, the `flatten` function is responsible for transforming the contents of `block_gen_state_t` into such a representation.

Listing 5.6: Specialized `block_gen` overloads

```
/// block_gen for a single AST token.
constexpr void block_gen(ast_token_t const &tok,
                         block_gen_state_t &s) {
  s.blocks[s.block_pos].push_back(
      flat_token_t{tok.token});
}

/// block_gen for an AST block.
constexpr void block_gen(ast_block_t const &blo,
                         block_gen_state_t &s) {
  // Save & update block pos before adding a block
  size_t const previous_pos = s.block_pos;
  s.block_pos = s.blocks.size();
  s.blocks.emplace_back();

  // Preallocating
  s.blocks[s.block_pos].reserve(blo.content.size() + 1);
  s.total_size += blo.content.size() + 1;

  // Adding block descriptor as a prefix
  s.blocks[s.block_pos].push_back(
      flat_block_descriptor_t{blo.content.size()});

  // Flattening instructions
  for (ast_node_ptr_t const &node : blo.content) {
    block_gen(node, s);
  }

  // Restoring block pos after recursive block
  // processing
  s.block_pos = previous_pos;
}

/// block_gen for a while instruction.
constexpr void block_gen(ast_while_t const &whi,
                         block_gen_state_t &s) {
  s.blocks[s.block_pos].push_back(
      flat_while_t{s.blocks.size()});
  block_gen(whi.block, s);
}
```

Listing 5.7 shows the implementation of `flatten` which calls `block_gen`, chains the blocks together, and translates block indexes to account for the new layout.

The postfix serialization layout (ie. the fact that blocks are laid out one after the other, and not one inside the other) ensures that traversing the AST remains trivial.

Listing 5.7: `flatten` implementation

```
constexpr flat_ast_t
flatten(ast_node_ptr_t const &parser_input) {
  flat_ast_t serialized_ast;

  // Extracting as vector of blocks
  block_gen_state_t bg_result;
  block_gen(parser_input, bg_result);

  // Small optimization to avoid reallcations
  serialized_ast.reserve(bg_result.total_size);

  // block_map[i] gives the index of
  // bg_result.blocks[i] in the serialized
  // representation
  std::vector<size_t> block_map;
  block_map.reserve(bg_result.blocks.size());

  // Step 1: flattening
  for (flat_ast_t const &block : bg_result.blocks) {
    // Updating block_map
    block_map.push_back(serialized_ast.size());

    // Appending instructions
    std::ranges::copy(
        block, std::back_inserter(serialized_ast));
  }

  // Step 2: linking
  for (flat_node_t &node : serialized_ast) {
    if (std::holds_alternative<flat_while_t>(node)) {
      flat_while_t &w_ref =
          std::get<flat_while_t>(node);
      w_ref.block_begin =
          block_map[w_ref.block_begin];
    }
  }

  return serialized_ast;
}
```

From there, the only thing that needs to be done is to evaluate this dynamic array into a static one to make it usable as an NTTP.

Listing 5.8 shows the implementation of the final function that takes a string as a program and transforms it into a `fixed_flat_ast_t`.

So far, this is the only function that takes a template parameter as an input, and thus where the distinction between constexpr programming and template metaprogramming begins.

Listing 5.8: `parse_to_fixed_flat_ast` implementation

```
/// Parses a BF program into a fixed_flat_ast_t value.
template <auto const &ProgramString>
constexpr auto parse_to_fixed_flat_ast() {
  // Getting AST vector size into a constexpr variable
  constexpr size_t AstArraySize =
      flatten(parser::parse_ast(ProgramString))
          .size();

  // Initializing static size array
  fixed_flat_ast_t<AstArraySize> arr;
  std::ranges::copy(
      flatten(parser::parse_ast(ProgramString)),
      arr.begin());

  return arr;
}
```

All the remaining work consists in implementing a function that takes the serialized AST as a template parameter and generates a program from it.

Listing 5.9: `program_state_t` definition

```
struct program_state_t {
  constexpr program_state_t() : i(0) {
    for (auto &c : data) {
      c = 0;
    }
  }

  std::array<char, 30000> data;
  std::size_t i;
};
```

The functions generated by the codegen functions must take a `program_state_t` as a reference to execute Brainfuck code. The program state structure definition is shown in listing 5.9.

The implementation principle for the code generation functions is to visit nodes recusively to compose and generate lambdas. We will explore two ways to implement the AST traversal: one based on a monolithic implementation based on function overloading to differentiate each type of node, and another one based on `if constexpr`.

The **overloaded** version consists in a code generation entry point function that selects instruction-specific code generation functions depending on the type of token stored in the current instruction.

Listing 5.10: Overloaded `codegen` entry point function

```
// Necessary forward declaration
template <auto const &Ast,
          size_t InstructionPos = 0>
constexpr auto codegen();

// Specialization implementations...

/// Generic code generation entrypoint
template <auto const &Ast, size_t InstructionPos>
constexpr auto codegen() {
  constexpr flat_node_t Instr =
      Ast[InstructionPos];

  // Calling specialized codegen versions
  // using function overloading
  return codegen<Ast, InstructionPos>(
      decltype(get<Instr.index()>(Instr)){});
}
```

Listing 5.10 shows the implementation of the entry point function whose only purpose is to unwrap the type of the current element and use it to call the appropriate code generation function depending on the instruction type. The dispatch is done automatically through function overloading.

Listing 5.11: `codegen` specialization for `flat_token_t` elements

```
template <auto const &Ast,
          size_t InstructionPos = 0>
constexpr auto codegen(flat_token_t) {
  // Extracting token value
  constexpr flat_token_t Token =
      get<flat_token_t>(Ast[InstructionPos]);

  // Returning code for a single Brainfuck
  // instruction

  // >
  if constexpr (Token.token ==
                pointer_increase_v) {
    return [](program_state_t &s) { ++s.i; };
  }
  // <
  else if constexpr (Token.token ==
                     pointer_decrease_v) {
    return [](program_state_t &s) { --s.i; };
  }

  // More instructions...
}
```

In listing 5.11 we can see the implementation of a code generation function for simple instructions, *i.e.* any instruction represented by a token that isn't a while block delimiter.

The dispatch over the tokens is done using `if constexpr`, but it could have been done using a variable template with template specializations for each case.

Note that the token passed as a regular parameter is only used for the overload selection. The token value used to evaluate conditions in the `if constexpr` statements is sourced from the AST passed as a NTTP, therefore it is still constexpr.

Listing 5.12: `codegen` specialization for `flat_block_descriptor_t` elements

```
/// Code generation implementation
/// for a code block
template <auto const &Ast,
          size_t InstructionPos = 0>
constexpr auto codegen(flat_block_descriptor_t) {
  return [](program_state_t &s) {
    // Generating an index sequence type
    // with a size equal to the code block size.
    // It will be passed to the template lambda
    // to expand its indexes.
    auto index_sequence =
        std::make_index_sequence<
            get<flat_block_descriptor_t>(
                Ast[InstructionPos])
                .size>{};

    // Static unrolling on the block's
    // instructions, made possible by the
    // contiguity of its elements
    [&]<size_t... Indexes>(
        std::index_sequence<Indexes...>) {
      // Expansion on the index to generate code
      // for each node and invoke it with the
      // program state
      (..., codegen<Ast, 1 + InstructionPos +
                             Indexes>()(s));
    }(index_sequence);
  };
}
```

The trickiest part of code generation is generating code blocks, as shown in listing 5.12. Doing so requires the use of a compile time unrolling technique based on C++ parameter packs. Iteration over the elements must be done that way to keep the index constexpr and use it as a NTTP to generate code.

The result of this metafunction is an anonymous function that evaluates all the Brainfuck code within a block. At this point the only function that remains

is the `flat_while_t` overload.

Listing 5.13: `codegen` specialization for `flat_while_t` elements

```
/// Code generation implementation for a while block
template <auto const &Ast, size_t InstructionPos = 0>
constexpr auto codegen(flat_while_t) {
  return [](program_state_t &s) {
    while (s.data[s.i]) {
      codegen<Ast, get<flat_while_t>(Ast[InstructionPos])
                       .block_begin>()(s);
    }
  };
}
```

The `codegen` implementation for a while block 5.13 is trivial: it returns a function that runs a while loop as defined by the Brainfuck language specification, and the body itself is the code generation result for the block element it points to. This implementation is a good way to illustrate code generation from a NTTP for a serialized AST.

Listing 5.14: `if constexpr` based `codegen` implementation

```
template <auto const &Ast,
          size_t InstructionPos = 0>
constexpr auto codegen() {
  constexpr flat_node_t Instr =
      Ast[InstructionPos];

  if constexpr (holds_alternative<flat_token_t>(
                    Instr)) {
    constexpr flat_token_t Token =
        get<flat_token_t>(Instr);
    // Single token code generation...
  }

  else if constexpr (holds_alternative<
                         flat_block_descriptor_t>(
                         Instr)) {
    constexpr flat_block_descriptor_t
        BlockDescriptor =
            get<flat_block_descriptor_t>(Instr);
    // Block code generation...
  }

  else if constexpr (holds_alternative<
                         flat_while_t>(Instr)) {
    constexpr flat_while_t While =
        get<flat_while_t>(Instr);
    // While loop code generation...
  }
}
```

A **monolithic** implementation of `codegen` can be implemented by replacing overloading with `if constexpr` and `std::holds_alternative`.

Listing 5.14 shows the `if constexpr` based code generation implementation. Note that bits of code were cut to make the listing shorter, but they are the same as the overloaded `codegen` overload function bodies. This `codegen` implementation remains functionally identical to the previous one.

Variable-sized benchmarks

We first begin by running two variable-sized benchmarks, consisting in measuring compiler execution time as the AST widens, and as the AST deepens.

The first variable-sized benchmark consists in generating a valid BF AST by concatenating strings to generate a succession of BF while loops in a constexpr string. This benchmark was instantiated with sizes going from 1 to 10 with a step of 1, with 10 timing iterations for each size. The second benchmark generates a string with nested loops, making the AST deeper as the benchmark size increases instead of making it wider as in the previous case. Both benchmarks generate programs of the same size.

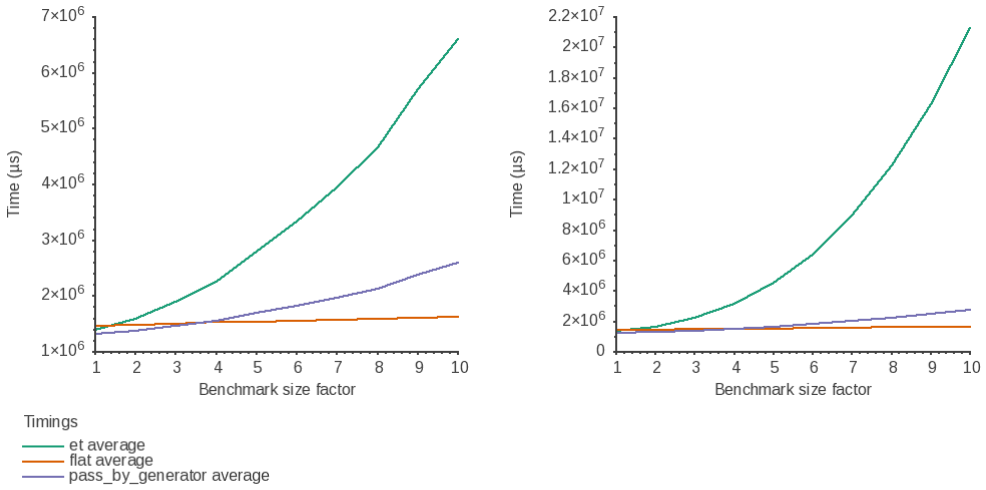


Figure 5.1: Compiler execution time measurements for consecutive loops (left) and nested loops (right)

Figure 5.1 highlights considerably higher compiler execution times for the expression template based backend, high enough to suggest that the use of expression templates induces an overhead higher than parsing and generating Brainfuck programs using the pass by generator backend. However the pass by generator backend still has a compile time overhead much higher than the flat backend, which shows near constant compiler execution times on these small scale benchmarks.

Finally, AST deepening has a much higher impact on compile times than AST widening with the expression template backends, whereas the other backends seem to scale similarly as the AST grows wider or deeper.

Large Brainfuck programs

The following benchmarks consist in measuring compiler execution times for compiling Brainfuck code examples. These example programs are also used to validate the metacompiler's backend implementations by compiling them and verifying their output.

- A Hello World program (106 tokens).
- The same Hello World program, ran twice (212 tokens).
- A Mandelbrot set fractal viewer (11672 tokens).

Backend	Hello World	Hello World x2	Mandelbrot
Flat (Monolithic)	0.63	0.80	18.16
Flat (Overloaded)	0.66	0.90	28.51
pass by generator	3.55	12.73	Failure (timeout)
expression template	19.18	74.51	Failure (timeout)

Figure 5.2: Brainfuck compile time measurements in seconds

The measurements in figure 5.2 help us better assess how various metacompiling techniques behave at scale. The Hello World program is meant to represent a simple embedded expression, while the Mandelbrot visualization is a much larger program meant to represent an upper bound of what a DSEL would be reasonably used for as it is more than 200 lines of code.

However, proposals such as `std::embed` [42] could dramatically increase the size of embedded programs as whole source files could be used directly as embedded expressions or programs in C++.

Conclusion

Here is what can be said about each backend:

- **Flat backends**: they are not the easiest to implement due to the additional intermediate representation and serialization step that need to be implemented.

  However, they present clear benefits when it comes to compilation times. So far they are the only ones that can be used at scale. The Mandelbrot example is supposed to illustrate an extreme case where DSELs

are used to integrate large programs (approximately 11'000 AST nodes), and yet both "flat" implementations manage to keep compilation times well under a minute.

- **pass by generator backend**: The pass by generator backend has the shortest implementation. However, it might not be the least difficult to implement.

  The code examples do not reflect the time spent debugging constexpr allocated memory errors. While the pass by generator may be a decent route for rapid prototyping to compile simple embedded expressions or programs, its poor performance scaling might be an issue.

  Judging from the implementation, the initial hypothesis was that its compilation time complexity as a function of program size would be quadratic and was confirmed by a compilation time analysis. This hypothesis was confirmed by compilation time benchmarks on small and large scale programs.

- **expression template backend**: This backend was originally meant to be a demonstrator for the interoperability of constexpr memory and expression templates. It shows no particular advantage compared to the two other backends: it is slower than the pass by generator backend while requiring additional effort to implement the expression template intermediate representation.

As a broad conclusion for compile time Brainfuck code generation, it can be said that using NTTP-based techniques to generate code is preferable in order to avoid compilation times to increase dramatically.

5.2 . Math parsing and high performance code generation

Thanks to the previous experiments, we know that using serialized structures as NTTPs is preferable to keep compilation times lower. As such, we will be generating code from mathematical expressions using a parsing algorithm that transforms infix formulas into their RPN representation.

In 4.3.2, we already demonstrated that generating code from RPN formulas is a rather easy task, therefore this section will only cover the Shunting Yard algorithm, and the use of RPN code generation applied to high performance computing.

5.2.1 . The Shunting-Yard algorithm

As parsing algorithms and constexpr dynamic data representations were already covered in 5.1, the implementation of the Shunting Yard algorithm

will not be covered in detail here. A thoroughly commented constexpr implementation is available in appendix .1.5. It features the algorithm itself, as well as data structures it relies on, and higher order helper functions for code generation.

The working principle of the algorithm is rather simple: an operator and an operand stack are read in order, and moved from the input list to the output queue and the operator stack. If an operator of lower precedence than the operator on top of the operator stack is read, then the operator stack is dumped in the output queue. The worst case time and memory complexity of the algorithm is $O(N)$.

Once again, code generation from postfix notation formulas was already covered in 4.3.2, so we will skip straight to the use of Blaze to generate high performance code from constexpr formulas.

5.2.2 . Using Blaze for high performance code generation

All the technical aspects of high performance code generation from constexpr DSELs in C++ 23 have been covered and implemented individually so far:

- a constexpr Shunting Yard parser for math formulas,
- a code generator for RPN formulas,
- and an optimizing library for math computation (the Blaze library).

I will now demonstrate that all these layers can work together as a complete high performance compilation chain for a math DSELs.

I will start by introducing a simple demonstrator language: Tiny Math Language (TML). It is a very basic language for simple math that can read math formulas containing integers, `x` and `y` variables, `sin` and `max` functions, a small set of infix operators (`+`, `-`, `*`, `/`, `^`), as well as `{}` and `()` as parenthesis pairs.

Listing 5.15: TML parser implementation

```
/// Parses a TML formula to RPN.
constexpr shunting_yard::parse_result_t
parse(std::string_view const &formula) {
  namespace sy = shunting_yard;

  // Defining various tokens
  sy::token_specification_t tiny_math_language_spec{
      .variables =
          {
              sy::variable_t("x"),
              sy::variable_t("y"),
          },
      .functions =
```

```
            {
                sy::function_t("sin"),
                sy::function_t("max"),
            },
        .operators =
            {
                sy::operator_t("+", sy::left_v, 10),
                sy::operator_t("-", sy::left_v, 10),
                sy::operator_t("*", sy::left_v, 20),
                sy::operator_t("/", sy::left_v, 20),
                sy::operator_t("^", sy::right_v, 30),
            },
        .lparens = {sy::lparen_t("("),
                    sy::lparen_t("{")},
        .rparens = {sy::rparen_t(")"),
                    sy::rparen_t("}")}};

  // Running the Shunting yard algorithm
  // with the TML language specification
  return parse_to_rpn(formula,
                      tiny_math_language_spec);
}
```

Using the Sunting Yard implementation from appendix .1.5, we can specify variable identifiers, function identifiers, infix operators (with precedence), and parenthesis as shown in listing 5.15 to define a simple math language that can be parsed by a constexpr function.

We will now oversee the code generation implementation. As a quick reminder, generating code from a RPN formula consists in reading tokens one by one, stacking operands, and consuming them as needed. The stack is implemented with a `kumi::tuple` object, which is an enhanced version of `std::tuple`.

Listing 5.16: TML operand examples

```
/// Operand representing the 42 constant
auto forty_two = [](auto const &, auto const &) {
  return 42;
};

/// Operand representing the x variable
auto x = [](auto const &input_x,
            auto const &) -> auto const & {
  return input_x;
};

// Note: We assume operand_a and operand_b
// are operands we consumed from the stack.

/// Operand representing the plus operator
```

```
auto plus = [operand_a,
             operand_b](auto const &input_x,
                        auto const &input_y) {
  return operand_a(input_x, input_y) +
         operand_b(input_x, input_y);
};
```

Each operand in the tuple is a function that takes `x` and `y` parameters and returns an arbitrary value. As such, operands have a form similar to the function objects shown in listing 5.16. The parameters are not used by the `forty_two` operand, the `x` operand simply forwards the first parameter, and the `plus` operand forwards the parameters to its sub-operands.

The result of the code generation is a function that takes two elements of arbitrary type as an input, and performs the operations on them.

Listing 5.17: TML usage example

```
int main() {
  namespace tml = tiny_math_language;

  static constexpr auto formula =
      "sin((x + 3) / 3 * y ^ 2)";

  // Runtime parsing prints parsing steps
  // for debugging
  tml::parse(formula);

  // Input vectors
  constexpr std::size_t vec_size = 16;
  blaze::DynamicVector<double> vector_x(vec_size, 1.),
      vector_y(vec_size, 12.);

  // Generating code from the formula
  auto function = tml::codegen<formula>();

  // Running the generated code
  blaze::DynamicVector<double> result =
      function(vector_x, vector_y);

  // Printing and verifying the result
  double const expected_value =
      std::sin((1. + 3.) / 3. * std::pow(12., 2.));
  for (double const &element : result) {
    fmt::println("{}", element);
    if (std::abs(element - expected_value) > 0.01) {
      throw;
    }
  }
}
```

Listing 5.17 shows how to generate code from a formula and use the generated function to perform operations on Blaze vectors. We use this code to verify the validity of the result.

Listing 5.18: Result type checking for generated function output

```
using namespace blaze;

using expected_type = DVecMapExpr<
    DVecScalarMultExpr<
        DVecDVecMultExpr<
            DVecMapExpr<
                DynamicVector<
                    float, false,
                    AlignedAllocator<float>,
                    GroupTag<0>>,
                Bind2nd<Add, float>, false>,
            DVecMapExpr<
                DynamicVector<
                    float, false,
                    AlignedAllocator<float>,
                    GroupTag<0>>,
                Bind2nd<Pow, float>, false>,
            false>,
        float, false>,
    Sin, false>;

static_assert(std::is_same<
              expected_type,
              decltype(function(
                  vector_x, vector_y))>::value);
```

We also verify that the expression `function(vector_x, vector_y)` generates the right Blaze expression template by compiling the code in listing 5.18. The `static_assert` expression will not compile if the generated type is not the same as `expected_type`.

5.2.3 . Studying the compilation time overhead of parsing for high performance code generation

A question remains about the use of compile time math parsing for high performance code generation: what is the compilation time impact? To answer that question, I will again measure compilation times for benchmarks representing what would be a "realistic" use case in terms of size, and simpler synthetic benchmarks to study the scaling of the performance overhead as math formula sizes increase.

For the "realistic" use case, we will simply consider the formula from listing 5.17 and measure the compiler execution time with and without compile time parsing to generate a Blaze expression. In one measurement where we compile a TML formula into its evaluation via Blaze, we observe a compilation

time of 5.94 seconds. In another one where we compile the same formula using the Blaze API directly and without including the TML header, the measured compilation time is 5.62 seconds.

These initial measurements show that a 6% compile time overhead when using the TML parser on top of Blaze. However, to better understand the impact of this method at scale, we need to run variable size benchmarks to study more than a single use case.

Listing 5.19: Benchmark case: Blaze with TML parsing

```
#include <boost/preprocessor/repetition/repeat.hpp>
#include <blaze/Blaze.h>
#include <tiny_math_language.hpp>

// Generating a series of additions
#define REPEAT_STRING(z, n, str) str
static constexpr const char *program_string =
    BOOST_PP_REPEAT(BENCHMARK_SIZE, REPEAT_STRING,
                    "x + y + ") "x";

/// Benchmark entry point.
template <typename T = void> inline auto bench_me() {
  blaze::DynamicVector<unsigned int> vector_a(32, 12);
  blaze::DynamicVector<unsigned int> vector_b(32, 12);

  blaze::DynamicVector<unsigned int> res =
      tiny_math_language::codegen<program_string>()(
          vector_a, vector_b);

  return res;
}

void foo() { bench_me(); }
```

Listing 5.19 introduces a variable size benchmark where a TML formula is generated from a size factor N. The formula consists in adding `x` and `y` terms N times to `x`. The formula is then parsed to generate a C++ function.

`vector_a` and `vector_b` are then fed to the generated function to generate a Blaze expression, which is assigned to a Blaze vector.

Listing 5.20: Benchmark case: Blaze without TML parsing

```
template <typename T = void> inline auto bench_me() {
  blaze::DynamicVector<unsigned int> vector_a(32, 12);
  blaze::DynamicVector<unsigned int> vector_b(32, 12);

  blaze::DynamicVector<unsigned int> res =
      BOOST_PP_REPEAT(BENCHMARK_SIZE, REPEAT_STRING,
                      vector_a + vector_b +) vector_a;

  return res;
```

```
}
```

Listing 5.20 shows an equivalent code where the parsing part is skipped. This will serve as a baseline to study the compile time overhead of TML parsing.

Finally, a third benchmark case is added that consists in parsing the same TML formulas but without using Blaze for code generation. Instead, the Blaze vectors are replaced with simple integers.

ctbench is used to instantiate, compile, and plot the compilation times of these three benchmark cases with N going from 0 to 20, ie. with formulas having from one final term (`vector_a` alone), to 41 final terms (the subexpression `vector_a + vector_b` contains 2 final terms, and it is being added up to 20 times to `vector_a`).

The benchmark cases are named as follows:

- `formula-blaze` designates the case where TML is parsed to generate high performance code with Blaze,
- `no-formula-blaze` designates the case where Blaze is being used to generate high performance directly (*i.e.* without TML parsing),
- `formula-no-blaze` designates the case where TML is used to generate code without Blaze but with integers instead.

Looking at the results of the benchmarks shown in figure 5.3, we can make the following observations:

- The `formula-no-blaze` case seems to have a quadratic complexity, although it has the shortest compilation time of all three cases until N reaches 16 (*i.e.* when the formula contains 33 final terms). After this point, the compile time offset of TML parsing becomes larger than the Blaze code generation itself.
- The `no-formula-blaze` case scales almost linearly with pretty low scaling factor, but it starts at almost 5 seconds. This is to be expected since Blaze has a very large code base. To put Blaze's code base in perspective, its package in the Arch Linux repositories weighs more than 34MiB, and it only contains C++ header files.
- The `formula-blaze` case looks pretty close to what we would expect: it starts slightly above `no-formula-blaze` and follows a scaling pattern similar to `formula-no-blaze`.

The compilation times on these benchmark cases remain under 20 seconds within the range of our measurements, which covers cases assumably

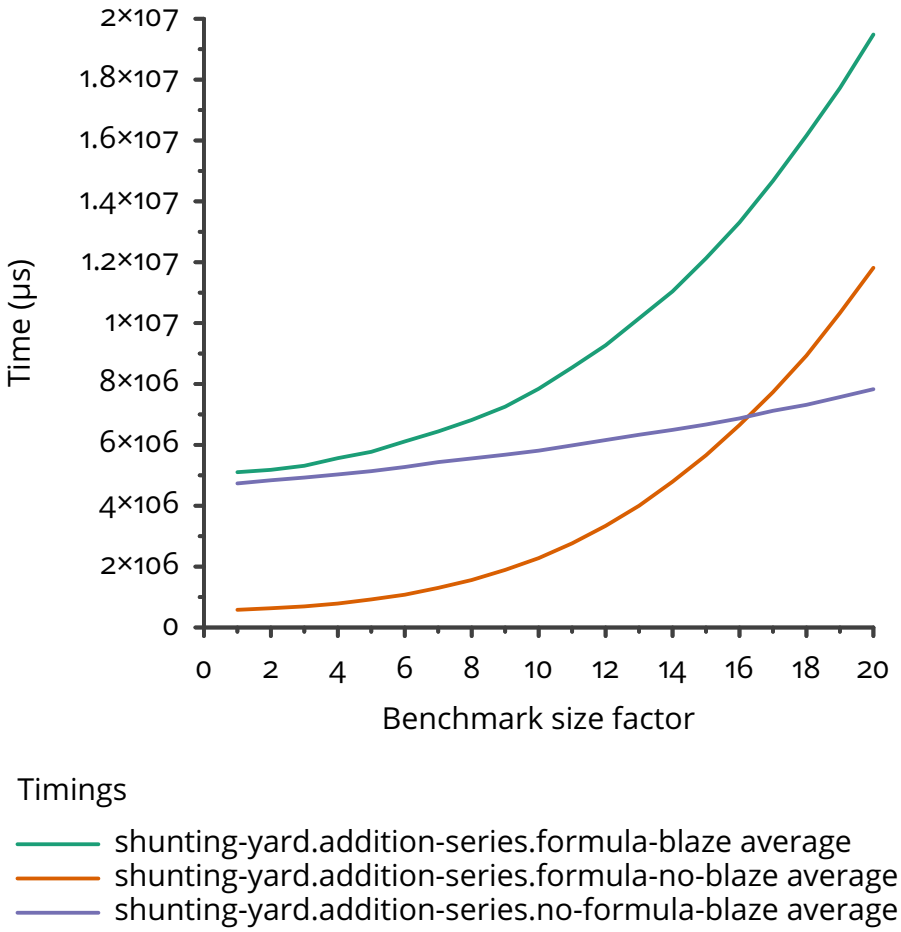


Figure 5.3: Code generation parsing benchmarks

larger than most use cases for the Blaze library. Moreover, math formulas can be divided in subexpressions in an effort to make a program clearer, or simply to mitigate the quadratic complexity of TML code generation.

Overall, this prototype DSEL proves the viability of compile time parsing for high performance code generation in C++ 23 using as little template metaprogramming as possible. However TML code generation does not scale as well as Brainfuck code generation using the Flat backend, which scaled linearly as shown in 5.1.3. A hypothesis for this higher complexity is the cost induced by the tuple operations during the code generation step.

5.3 .Conclusion

The Brainfuck example shows that using constexpr programming to embed large programs written in foreign languages in C++ can be achieved. AST

serialization makes it possible to store parsing results and avoid running into quadratic compilation time issues, as long as the code generation methods being .

The case of TML also shows that AST serialization by itself is not a magic bullet to avoid running into compilation time scaling issues. The compilation time offset of TML parsing, despite being reasonable within the bounds of expected use cases for TML, was quadratic.

These results show that more effort is needed to determine which metaprogramming techniques should be avoided to avoid runaway compilation times, and how C++ library code or modifications to the C++ language itself could facilitate scalable code generation from constexpr function results.

Conclusion & perspectives

In the first part we have shown that metaprogramming is a widespread practice, and that it is still evolving across many languages with C++ continuously evolving on that front, while being the most comprehensive platform for parallel computing. We then demonstrated that code generation is a viable option to greatly reduce the implementation complexity of high performance linear algebra kernels, while still providing similar, or even higher level of performance. These results reinforce the hypothesis that metaprogramming can be used to tackle the challenge of portability in the context of High Performance Computing, even for very low level tasks where the reduction of abstraction overhead is critical to achieve decent performance.

Before moving on to constexpr programming, we adressed the lack of tools for the scientific study of compilation times with ctbench. We proposed a new compile-time study methodology based on Clang's built-in profiler that help us understand the impacts of metaprograms on each step of the compilation process. While it offers limited benchmarking capabilities when used with GCC as the latter does not output any detailed profiling data, conclusions drawn on Clang might be relevant for GCC, even if they are not directly transposable.

Then we demonstrated that using constexpr code to implement embedded language parsers for in C++ 23 is possible despite limitations on constexpr memory allocation, and that doing so is possible with reasonable impact on compilation times.

The Brainfuck example shows that code generation backends are a determining factor for embedded compiler performance, and if implemented correctly, a code generator can provide adequate scaling for embedding large programs. More specifically, we have shown that code generation backends must rely on value-based metaprogramming in favor of type-based metaprogramming to achieve decent compile time performance scaling. However, using value-based metaprogramming for code generation might require additional implementation efforts, and backends based on the pass by generator technique might be a better fit for quick DSEL prototyping.

On the other hand, the TML example demonstrated the interoperability between constexpr programming and existing high performance libraries relying on expression templates. It also showed that using value-based representations for code generation is not always trivial as the use of tuples was necessary to store partial results during the code generation phase.

Allowing the direct use of constexpr allocated memory as Non-Type Template Parameters could help greatly reduce the implementation complexity

of compile-time code generators, and improve their performance as well by removing the requirement for additional serialization steps or compile time heavy workarounds such as pass by generator. Another way forward could also be the development of reflection and slicing capabilities for C++, or an imperative-style code generation API.

In the meantime, developing constexpr serialization libraries could greatly help with the development of performant code generation backends as it would reduce the development effort to use more performant code generation methods, and could be achievable without waiting for a new C++ release.

Glossary

AST Abstract Syntax Tree. 17, 69–72, 74–78, 80–83

BLAS Basic Linear Algebra Subprograms. 18, 24–26, 28, 31, 32, 35

constant evaluation The evaluation of an expression that is performed at compile time.. 16, 20, 72

constexpr A value or function that can be used in a constant expression.. 15–17, 19, 53–57, 59, 60, 62–64, 68–71, 76, 78, 79, 81, 83–85, 93

CPU Central Processing Unit. 9, 10

CTPG Compile Time Parser Generator. 20, 21

CTRE Compile Time Regular Expression. 17, 20, 21

DSEL Domain-Specific Embedded Language. 9, 17, 18, 53, 82–84, 91, 93

DSL Domain-Specific Language. 53

ERB Embedded Ruby. 40

expression template empty. 17, 53, 82, 83, 93

GPU Graphical Processing Unit. 10, 18, 23

HPC High Performance Computing. 9, 11, 13, 18, 21, 22

JIT Just-In-Time. 13

literal value A value that does not hold any pointer to dynamic memory.. 71

NTTP Non-Type Template Parameter. 15, 16, 54–56, 58, 59, 61, 64, 68, 69, 71, 72, 76, 78–80, 83

pass by generator empty. 68, 72, 82, 83

PCRE Perl Compatible Regular Expression. 17

RPN Reverse Polish Notation. 65, 66, 83–85

SFINAE Substitution Failure Is Not An Error. 16

Bibliography

[1] David Abrahams and Aleksey Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*. Addison-Wesley Professional, 2004. isbn: 0321227255.

[2] David Abrahams and Aleksey Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*. Addison-Wesley Professional, 2004. isbn: 0321227255.

[3] Anton Afanasyev. *Adds '-ftime-trace' option to clang that produces Chrome 'chrome://tracing' compatible JSON profiling output dumps*. 2019. url: https://reviews.llvm.org/D58675.

[4] AMD. *Amd core math library (acml).* url: https://developer.amd.com/amd-cpu-libraries/amd-math-library-libm/.

[5] Edward Anderson et al. *LAPACK users' guide*. SIAM, 1999.

[6] Benjamin A. Antunes and David R. C. Hill. *Reproducibility, Replicability, and Repeatability: A survey of reproducible research with a focus on high performance computing*. 2024. arXiv: 2402.07530 [cs.SE].

[7] Nathan Bell and Jared Hoberock. "Chapter 26 - Thrust: A Productivity-Oriented Library for CUDA". In: *GPU Computing Gems Jade Edition*. Ed. by Wen-mei W. Hwu. Applications of GPU Computing Series. Boston: Morgan Kaufmann, 2012, pp. 359–371. isbn: 978-0-12-385963-1. doi: https://doi.org/10.1016/B978-0-12-385963-1.00026-5. url: https://www.sciencedirect.com/science/article/pii/B9780123859631000265.

[8] Jeff Bezanson et al. "Julia: Dynamism and Performance Reconciled by Design". In: *Proc. ACM Program. Lang.* 2.OOPSLA (Nov. 2018). doi: 10.1145/3276490. url: https://doi.org/10.1145/3276490.

[9] L Susan Blackford et al. "An updated set of basic linear algebra subprograms (BLAS)". In: *ACM Transactions on Mathematical Software* 28.2 (2002), pp. 135–151.

[10] Walter Bright. *Templates revisited - d programming language*. url: https://dlang.org/articles/templates-revisited.html.

[11] Murray Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Cambridge, MA, USA: MIT Press, 1991. isbn: 0262530864.

[12] Krzysztof Czarnecki, Kasper Østerbye, and Markus Völter. “Generative Programming”. In: *Object-Oriented Technology ECOOP 2002 Workshop Reader*. Ed. by Juan Hernández and Ana Moreira. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 15–29.

[13] Zachary DeVito et al. “Terra: a multi-stage language for high-performance computing”. In: *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*. 2013, pp. 105–116.

[14] Edsger Wybe Dijkstra. In: *ALGOL-60 Translation* (1961). url: http://www.cs.utexas.edu/~EWD/MCReps/MR35.PDF.

[15] Peter Dimov. *C++11 metaprogramming library*. 2015. url: https://boost.org/libs/mp11.

[16] Peter Dimov and Vassil Vassilev. *P1064R0: Allowing Virtual Function Calls in Constant Expressions*. https://wg21.link/p1064r0. May 2018.

[17] Peter Dimov et al. “More constexpr containers”. In: (2019). url: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0784r7.html.

[18] Louis Dionne, Bruno Dutra, Odin Holmes, et al. *Metabench: A simple framework for compile-time microbenchmarks*. url: https://github.com/ldionne/metabench/.

[19] Tingxing Dong et al. “Optimizing the SVD Bidiagonalization Process for a Batch of Small Matrices”. In: *Procedia Computer Science* 108 (2017). International Conference on Computational Science, ICCS 2017, 12-14 June 2017, Zurich, Switzerland, pp. 1008–1018. issn: 1877-0509. doi: https://doi.org/10.1016/j.procs.2017.05.237. url: https://www.sciencedirect.com/science/article/pii/S1877050917308645.

[20] Hana Dusíková. *Compile Time Regular Expression in C++*. 2018. url: https://github.com/hanickadot/compile-time-regular-expressions.

[21] Pierre Estérie et al. “Boost.SIMD: Generic programming for portable SIMDization”. In: *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2012, pp. 431–432.

[22] Joel Falcou. “Software Abstractions for Parallel Architectures”. Habilitation à diriger des recherches. Universite de Paris 11, Dec. 2014. url: https://inria.hal.science/tel-01111708.

[23] Joel Falcou and Jocelyn Sérot. “EVE, an Object Oriented SIMD Library”. In: *Computational Science - ICCS 2004*. Ed. by Marian Bubak et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 314–321. isbn: 978-3-540-24688-6.

[24] Joël Falcou et al. “NT2: Une bibliothèque haute-performance pour la vision artificielle NT2: A High-performance library for computer vision.” In: (). url: https://www.lrde.epita.fr/dload/seminar/2008-09-14/falcou-3.pdf.

[25] Yoshihiko Futamura. “Partial Evaluation of Computation Process—AnApproach to a Compiler-Compiler”. In: *Higher Order Symbol. Comput.* 12.4 (Dec. 1999), pp. 381–391. issn: 1388-3690. doi: 10.1023/A:1010095604496. url: https://doi.org/10.1023/A:1010095604496.

[26] Gaël Guennebaud, Benoit Jacob, et al. “Eigen”. In: 3 (2010). url: https://eigen.tuxfamily.org.

[27] Aleksey Gurtovoy and David Abrahams. *Boost.org mpl module*. 2002. url: https://boost.org/libs/mpl.

[28] Seyed Hossein Haeri, Sibylle Schupp, and Jonathan Hüser. “Using functional languages to facilitate C++ metaprogramming”. In: *Proceedings of the 8th ACM SIGPLAN Workshop on Generic Programming*. WGP ’12. Copenhagen, Denmark: Association for Computing Machinery, 2012, pp. 33–44. isbn: 9781450315760. doi: 10.1145/2364394.2364400. url: https://doi.org/10.1145/2364394.2364400.

[29] Odin Holmes et al. *Instant compile time C++ 11 metaprogramming library*. 2015. url: https://github.com/edouarda/brigand.

[30] Klaus Iglberger. *Blaze C++ Linear Algebra Library*. 2012. url: https://bitbucket.org/blaze-lib.

[31] Klaus Iglberger et al. “High Performance Smart Expression Template Math Libraries”. In: *Proceedings of the 2nd International Workshop on New Algorithms and Programming Models for the Manycore Era (APMM 2012) at HPCS 2012*. 2012.

[32] Intel. *Math kernel library (mkl).* url: https://www.intel.com/software/products/mkl/.

[33] Jaakko Jarvi. “Compile time recursive objects in C++”. In: *Technology of Object-Oriented Languages, 1998. TOOLS 27. Proceedings*. IEEE. 1998, pp. 66–77.

[34] Neil D. Jones. “An introduction to partial evaluation”. In: *ACM Comput. Surv.* 28.3 (Sept. 1996), pp. 480–503. issn: 0360-0300. doi: 10.1145/243439.243447. url: https://doi.org/10.1145/243439.243447.

[35] Hartmut Kaiser et al. “HPX - The C++ Standard Library for Parallelism and Concurrency”. In: *Journal of Open Source Software* 5.53 (2020), p. 2352. doi: 10.21105/joss.02352. url: https://doi.org/10.21105/joss.02352.

[36] Paul Keir. *Towards a constexpr version of the C++ standard library*. 2020. url: https://github.com/pkeir/cest.

[37] Paul Keir et al. *Meeting C++ - A totally constexpr standard library*. 2022. url: https://www.youtube.com/watch?v=ekFPm7e__vI.

[38] Oleg Kiselyov. “The Design and Implementation of BER MetaOCaml”. In: *Functional and Logic Programming*. Ed. by Michael Codish and Eijiro Sumii. Cham: Springer International Publishing, 2014, pp. 86–102. isbn: 978-3-319-07151-0.

[39] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. url: https://doc.rust-lang.org/stable/book/.

[40] Yannis Lilis and Anthony Savidis. “A Survey of Metaprogramming Languages”. In: *ACM Comput. Surv.* 52.6 (Oct. 2019). issn: 0360-0300. doi: 10.1145/3354584. url: https://doi.org/10.1145/3354584.

[41] LYDIA MAIGNE et al. “PARALLELIZATION OF MONTE CARLO SIMULATIONS AND SUBMISSION TO A GRID ENVIRONMENT”. In: *Parallel Processing Letters* 14.02 (2004), pp. 177–196. doi: 10.1142/S0129626404001829. eprint: https://doi.org/10.1142/S0129626404001829. url: https://doi.org/10.1142/S0129626404001829.

[42] JeanHeyd Meneide. *P1040R6: std::embed and #depend*. Feb. 2020. url: https://wg21.link/p1040r6.

[43] Duane Merrill. *CUB*. 2010. url: https://nvlabs.github.io/cub/.

[44] Eric Niebler. *Boost.org proto module*. 2008. url: https://github.com/boostorg/proto.

[45] Jules Penuchot. *ctbench: compile time benchmarking for Clang*. 2021. url: https://www.youtube.com/watch?v=1RZY6skM0Rc.

[46] Jules Penuchot and Joel Falcou. “ctbench - compile-time benchmarking and analysis”. In: *Journal of Open Source Software* 8.88 (2023), p. 5165. doi: 10.21105/joss.05165. url: https://doi.org/10.21105/joss.05165.

[47] Jules Penuchot, Joel Falcou, and Amal Khabou. “Modern Generative Programming for Optimizing Small Matrix-Vector Multiplication”. In: *2018 International Conference on High Performance Computing Simulation (HPCS)*. 2018, pp. 508–514. doi: 10.1109/HPCS.2018.00086.

[48] Jules Pénuchot. *Blaze CUDA - CUDA compatibility for Blaze*. 2019. url: https://github.com/STEllAR-GROUP/blaze_cuda.

[49] Zoltán Porkoláb, József Mihalicza, and Norbert Pataki. “Measuring Compilation Time of C++ Template Metaprograms”. In: (2009). url: http://aszt.inf.elte.hu/~gsd/s/cikkek/abel/comptime.pdf.

[50] Michaël Roynard, Edwin Carlinet, and Thierry Géraud. “A Modern C++ Point of View of Programming in Image Processing”. In: *Proceedings of the 21st ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. GPCE 2022. Auckland, New Zealand: Association for Computing Machinery, 2022, pp. 164–171. isbn: 9781450399203. doi: 10.1145/3564719.3568692. url: https://doi.org/10.1145/3564719.3568692.

[51] Adrian Sampson, Kathryn S. McKinley, and Todd Mytkowicz. “Static Stages for Heterogeneous Programming”. In: *Proc. ACM Program. Lang.* 1.OOPSLA (Oct. 2017). doi: 10.1145/3133895. url: https://doi.org/10.1145/3133895.

[52] Tim Sheard and Simon Peyton Jones. “Template Meta-Programming for Haskell”. In: *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*. Haskell ’02. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2002, pp. 1–16. isbn: 1581136056. doi: 10.1145/581690.581691. url: https://doi.org/10.1145/581690.581691.

[53] Daniele G. Spampinato and Markus Püschel. “A Basic Linear Algebra Compiler for Structured Matrices”. In: *Proceedings of the 2016 International Symposium on Code Generation and Optimization*. CGO ’16. Barcelona, Spain: Association for Computing Machinery, 2016, pp. 117–127. isbn: 9781450337786. doi: 10.1145/2854038.2854060. url: https://doi.org/10.1145/2854038.2854060.

[54] Diomidis Spinellis. “Notable design patterns for domain-specific languages”. In: *Journal of Systems and Software* 56.1 (2001), pp. 91–99. issn: 0164-1212.

[55] Richard M Stallman. *Using the gnu compiler collection: a gnu manual for gcc version 4.3. 3*. CreateSpace, 2009.

[56] Victor Stinner. *Pyperf*. 2016. url: https://pyperf.readthedocs.io/en/latest/.

[57] Herb Sutter and James Larus. "Software and the Concurrency Revolution: Leveraging the full power of multicore processors demands new tools and new thinking from the software industry." In: *Queue* 3.7 (Sept. 2005), pp. 54–62. issn: 1542-7730. doi: 10.1145/1095408.1095421. url: https://doi.org/10.1145/1095408.1095421.

[58] Walid Taha. "A Gentle Introduction to Multi-stage Programming". In: *Domain-Specific Program Generation: International Seminar, Dagstuhl Castle, Germany, March 23-28, 2003. Revised Papers*. Ed. by Christian Lengauer et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 30–50. isbn: 978-3-540-25935-0. doi: 10.1007/978-3-540-25935-0_3. url: https://doi.org/10.1007/978-3-540-25935-0_3.

[59] Stanimire Tomov, Rajib Nath, and Jack Dongarra. "Accelerating the Reduction to Upper Hessenberg, Tridiagonal, and Bidiagonal Forms through Hybrid GPU-Based Computing". In: *Parallel Comput.* 36.12 (Dec. 2010), pp. 645–654. issn: 0167-8191. doi: 10.1016/j.parco.2010.06.001. url: https://doi.org/10.1016/j.parco.2010.06.001.

[60] Erwin Unruh. *Prime number computation*. ANSI X3J16-94-0075/ISO WG21-462. 1994.

[61] Daveed Vandevoorde et al. *P1240R2: Scalable Reflection*. https://wg21.link/p1240r2. Jan. 2022.

[62] David Vandevoorde and Nicolai M. Josuttis. *C++ Templates*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002. isbn: 0201734842.

[63] Todd Veldhuizen. "Expression templates". In: *C++ Report* 7.5 (1995), pp. 26–31.

[64] Todd L. Veldhuizen. "Expression templates". In: *C++ Report* 7.5 (June 1995). Reprinted in C++ Gems, ed. Stanley Lippman, pp. 26–31. issn: 1040-6042.

[65] Qian Wang et al. "AUGEM: Automatically generate high performance Dense Linear Algebra kernels on x86 CPUs". In: *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 2013, pp. 1–12. doi: 10.1145/2503210.2503219.

[66] Zhang Xianyi and Martin Kroeker. *OpenBLAS - An optimized BLAS library*. url: https://www.openblas.net.

[67] H. Ye et al. "High level tranforms toreduce energy consumption of signal and image processing operators". In: *2013 23rd International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS)*. 2013, pp. 247–254. doi: 10.1109/PATMOS.2013.6662183.

[68] Field G. Van Zee et al. "The libflame Library for Dense Matrix Computations". In: *Computing in Science and Engineering* 11.6 (2009), pp. 56–63. doi: 10.1109/MCSE.2009.207.

A -Appendix

.1 . Poacher

.1.1 . Brainfuck AST definition header

Listing 1: `brainfuck/ast.hpp`

```
#pragma once

#include <cstdint>
#include <memory>
#include <vector>

namespace brainfuck {

//-------------------------------------------------------
// TOKEN TYPE

/// Represents a Brainfuck token.
enum token_t : char {
  pointer_increase_v = '>', // ++ptr;
  pointer_decrease_v = '<', // --ptr;
  pointee_increase_v = '+', // ++*ptr;
  pointee_decrease_v = '-', // --*ptr;
  put_v = '.',              // putchar(*ptr);
  get_v = ',',              // *ptr=getchar();
  while_begin_v = '[',      // while (*ptr) {
  while_end_v = ']',        // }
  nop_v,                    // nop
};

/// Converts a char into its corresponding
/// Brainfuck token_t value.
constexpr enum token_t to_token(char c) {
  switch (c) {
  case pointer_increase_v:
    return pointer_increase_v;
  case pointer_decrease_v:
    return pointer_decrease_v;
  case pointee_increase_v:
    return pointee_increase_v;
  case pointee_decrease_v:
    return pointee_decrease_v;
  case put_v:
    return put_v;
  case get_v:
    return get_v;
  case while_begin_v:
```

```
    return while_begin_v;
  case while_end_v:
    return while_end_v;
  }
  return nop_v;
}

//------------------------------------------------------
// AST

/// Holds the underlaying node type
enum ast_node_kind_t : std::uint8_t {
  /// AST token node
  ast_token_v,
  /// AST block node
  ast_block_v,
  /// AST while node
  ast_while_v,
};

/// Parent class for any AST node type,
/// holds its type as an ast_node_kind_t
struct node_interface_t {
private:
  ast_node_kind_t kind_;

protected:
  constexpr node_interface_t(ast_node_kind_t kind)
      : kind_(kind){};

public:
  /// Returns the node kind tag.
  constexpr ast_node_kind_t get_kind() const {
    return kind_;
  }
  constexpr virtual ~node_interface_t() = default;
};
// Helpers

/// Token vector helper
using token_vec_t = std::vector<token_t>;

/// AST node pointer helper type
using ast_node_ptr_t =
    std::unique_ptr<node_interface_t>;

/// AST node vector helper type
using ast_node_vec_t = std::vector<ast_node_ptr_t>;

// !Helpers
```

```
/// AST node type for single Brainfuck tokens
struct ast_token_t : node_interface_t {
  token_t token;

  constexpr ast_token_t(token_t token_)
      : node_interface_t(ast_token_v), token(token_) {}
};

/// AST node type for Brainfuck code blocks
struct ast_block_t : node_interface_t {
  using node_ptr_t = ast_node_ptr_t;

  ast_node_vec_t content;

  constexpr ast_block_t(ast_node_vec_t &&content_)
      : node_interface_t(ast_block_v),
        content(std::move(content_)) {}

  constexpr ast_block_t(ast_block_t &&v) = default;
  constexpr ast_block_t &
  operator=(ast_block_t &&v) = default;

  constexpr ast_block_t(ast_block_t const &v) =
      delete;
  constexpr ast_block_t &
  operator=(ast_block_t const &v) = delete;
};

/// AST node type for Brainfuck while loop
struct ast_while_t : node_interface_t {
  ast_block_t block;

  constexpr ast_while_t(ast_block_t &&block_)
      : node_interface_t(ast_while_v),
        block(std::move(block_)) {}
};

template <typename F>
constexpr auto visit(F f, ast_node_ptr_t const &p) {
  switch (p->get_kind()) {
  case ast_token_v:
    return f(static_cast<ast_token_t const &>(*p));
  case ast_block_v:
    return f(static_cast<ast_block_t const &>(*p));
  case ast_while_v:
    return f(static_cast<ast_while_t const &>(*p));
  }
}

} // namespace brainfuck
```

.1.2 . Brainfuck parser implementation header

Listing 2: `brainfuck/parser.hpp`

```
#pragma once

#include <algorithm>

#include <iterator>
#include <memory>
#include <string>

#include <brainfuck/ast.hpp>

namespace brainfuck::parser {

/// Parser implementation
namespace impl {

/// Converts a string into a list of BF tokens
constexpr token_vec_t
lex_tokens(std::string const &input) {
  token_vec_t result;
  result.reserve(input.size());
  std::transform(input.begin(), input.end(),
                 std::back_inserter(result),
                 [](auto current_character) {
                   return to_token(current_character);
                 });
  return result;
}

/// Parses BF code until the end of the block (or the
/// end of the formula, ie. parse_end), then returns
/// an iterator to the last parsed token or parse_end
/// if the parser has parsed all the tokens.
constexpr std::tuple<ast_block_t,
                     token_vec_t::const_iterator>
parse_block(token_vec_t::const_iterator parse_begin,
            token_vec_t::const_iterator parse_end) {
  using input_it_t = token_vec_t::const_iterator;

  ast_node_vec_t block_content;

  for (; parse_begin != parse_end; parse_begin++) {
    // While end bracket: return block content and
    // while block end position
    if (*parse_begin == while_end_v) {
      return {std::move(block_content), parse_begin};
    }

    // While begin bracket: recurse,
```

```
    // then continue parsing from the end of the block
    else if (*parse_begin == while_begin_v) {
      // Parse while body
      auto [while_block_content, while_block_end] =
          parse_block(parse_begin + 1, parse_end);

      block_content.push_back(
          std::make_unique<ast_while_t>(
              std::move(while_block_content)));

      parse_begin = while_block_end;
    }

    // Any other token that is not a nop instruction:
    // add it to the AST
    else if (*parse_begin != nop_v) {
      block_content.push_back(ast_node_ptr_t(
          std::make_unique<ast_token_t>(
              *parse_begin)));
    }
  }

  return {ast_block_t(std::move(block_content)),
          parse_end};
}

} // namespace impl

/// Driver function for the token parser
constexpr ast_node_ptr_t
parse_ast(std::string const &input) {
  token_vec_t const tok = impl::lex_tokens(input);
  ast_block_t parse_result = get<ast_block_t>(
      impl::parse_block(tok.begin(), tok.end()));
  return std::make_unique<ast_block_t>(
      std::move(parse_result));
}

} // namespace brainfuck::parser
```

.1.3 . Brainfuck "ET" code generation backend

Listing 3: `brainfuck/backends/expression_template.hpp`

```
#pragma once

// "expression_template" backend: C++20-compatible
// backend that turns the AST into an expression
// template using constexpr lambda construction to get
// around AST not being constexpr due to constexpr
// memory allocation.

// Issue: *terrible* performance as programs get a bit
// too big.

#include <cstdio>

#include <brainfuck/ast.hpp>
#include <brainfuck/program.hpp>

namespace brainfuck::expression_template {

template <typename... Nodes> struct et_block_t {
  constexpr et_block_t() {}
  constexpr et_block_t(Nodes const &...) {}
};

template <typename... Nodes> struct et_while_t {
  constexpr et_while_t() {}
  constexpr et_while_t(Nodes const &...) {}
};

template <token_t Token> struct et_token_t {};

// f being a cosntexpr function with return type
// ast_node_t
template <typename f> constexpr auto to_et(f);

// f being a cosntexpr function with return type
// ast_token_t
template <typename f>
constexpr auto
to_et(f, std::integral_constant<ast_node_kind_t,
                                ast_token_v>) {
  constexpr token_t T =
      static_cast<ast_token_t const &>(*f{}()).token;
  return et_token_t<T>{};
}

// f being a cosntexpr function with return type
// ast_block_t
template <typename f>
```

```
constexpr auto
to_et(f, std::integral_constant<ast_node_kind_t,
                                ast_block_v>) {
  constexpr std::size_t S =
      static_cast<ast_block_t const &>(*f{}())
          .content.size();
  return []<std::size_t... Is>(
              std::index_sequence<Is...>) {
    return et_block_t(to_et([]() constexpr {
      return std::move(
          static_cast<ast_block_t &>(*f{}())
              .content[Is]);
    })...);
  }(std::make_index_sequence<S>{});
}

// f being a cosntexpr function with return type
// ast_while_t
template <typename f>
constexpr auto
to_et(f, std::integral_constant<ast_node_kind_t,
                                ast_while_v>) {
  constexpr std::size_t S =
      static_cast<ast_while_t const &>(*f{}())
          .block.content.size();
  return []<std::size_t... Is>(
              std::index_sequence<Is...>) {
    return et_while_t(to_et([]() constexpr {
      return std::move(
          static_cast<ast_while_t &>(*f{}())
              .block.content[Is]);
    })...);
  }(std::make_index_sequence<S>{});
}

template <typename f> constexpr auto to_et(f) {
  constexpr ast_node_kind_t Kind = f{}()->get_kind();
  return to_et(f{},
               std::integral_constant<ast_node_kind_t,
                                      Kind>{});
}

// codegen meta-function overloads

template <typename... Ts>
inline auto codegen(et_block_t<Ts...>) {
  return [](program_state_t &state) {
    (codegen(Ts{})(state), ...);
  };
}
```

```
template <typename... Ts>
inline auto codegen(et_while_t<Ts...>) {
  return [](program_state_t &state) {
    while (state.data[state.i])
      (codegen(Ts{})(state), ...);
  };
}

inline auto codegen(et_token_t<pointer_increase_v>) {
  return [](program_state_t &state) { ++state.i; };
}

inline auto codegen(et_token_t<pointer_decrease_v>) {
  return [](program_state_t &state) { --state.i; };
}

inline auto codegen(et_token_t<pointee_increase_v>) {
  return [](program_state_t &state) {
    state.data[state.i]++;
  };
}

inline auto codegen(et_token_t<pointee_decrease_v>) {
  return [](program_state_t &state) {
    state.data[state.i]--;
  };
}

inline auto codegen(et_token_t<put_v>) {
  return [](program_state_t &state) {
    std::putchar(state.data[state.i]);
  };
}

inline auto codegen(et_token_t<get_v>) {
  return [](program_state_t &state) {
    state.data[state.i] = std::getchar();
  };
}

inline auto codegen(et_token_t<nop_v>) {
  return [](program_state_t &s) {};
}

} // namespace brainfuck::expression_template
```

.1.4 . Brainfuck "pass-by-generator"" code generation backend

Listing 4: `brainfuck/backends/pass_by_generator.hpp`

```hpp
#pragma once

#include <cstdio>
#include <type_traits>
#include <utility>

#include <brainfuck/ast.hpp>
#include <brainfuck/program.hpp>

namespace brainfuck::pass_by_generator {

/// Declaration of the `generates` concept which
/// is satisfied if Generator has an operator()
/// function that returns a value of type ReturnType.
template <typename Generator, typename ReturnType>
concept generates = requires(Generator gen) {
  { gen() } -> std::same_as<ReturnType>;
};

/// Code generation function.
/// It accepts a generator of ast_node_ptr_t
/// as a template parameter, and generates
/// the corresponding code for it.
template <generates<ast_node_ptr_t> auto>
constexpr auto codegen();

namespace detail {

/// Accepts a generator function that returns a vector
/// of ast_node_ptr_t, and generates its code.
template <generates<ast_node_vec_t> auto Generator>
constexpr auto vector_codegen() {
  // Getting size as a constexpr value
  constexpr std::size_t VectorSize =
      Generator().size();

  // Performing a static unrolling
  // on the vector's elements
  return [&](program_state_t &state) {
    [&]<std::size_t... IndexPack>(
        std::index_sequence<IndexPack...>) {
      (...,
       // A lambda has to be used here
       // for the program to compile
       [&]() {
         codegen<[]() -> ast_node_ptr_t {
           return std::move(Generator()[IndexPack]);
         }>()(state);
```

```
        }());
    }(std::make_index_sequence<VectorSize>{});
  };
}

/// This variable template is used
/// as a token to instruction map
template <token_t Token>
inline constexpr auto instruction_from_token =
    [](program_state_t &) {};

template <>
inline constexpr auto
    instruction_from_token<pointer_increase_v> =
        [](program_state_t &state) { state.i++; };
template <>
inline constexpr auto
    instruction_from_token<pointer_decrease_v> =
        [](program_state_t &state) { state.i--; };
template <>
inline constexpr auto
    instruction_from_token<pointee_increase_v> =
        [](program_state_t &state) {
          state.data[state.i]++;
        };
template <>
inline constexpr auto
    instruction_from_token<pointee_decrease_v> =
        [](program_state_t &state) {
          state.data[state.i]--;
        };
template <>
inline constexpr auto instruction_from_token<put_v> =
    [](program_state_t &state) {
      std::putchar(state.data[state.i]);
    };
template <>
inline constexpr auto instruction_from_token<get_v> =
    [](program_state_t &state) {
      state.data[state.i] = std::getchar();
    };
} // namespace detail

template <generates<ast_node_ptr_t> auto Generator>
constexpr auto codegen() {
  // Getting kind as a constexpr variable
  constexpr ast_node_kind_t Kind =
      Generator()->get_kind();

  // Node is a single token
  if constexpr (Kind == ast_token_v) {
```

```
    // Getting token as a constexpr variable
    constexpr token_t Token =
        static_cast<ast_token_t const &>(*Generator())
            .token;

    return detail::instruction_from_token<Token>;
  }

  // Node is a block
  else if constexpr (Kind == ast_block_v) {
    return detail::vector_codegen<
        []() -> ast_node_vec_t {
          return std::move(
              static_cast<ast_block_t &>(*Generator())
                  .content);
        }>();
  }

  // Node is a while loop
  else if constexpr (Kind == ast_while_v) {
    // Extracting the while body
    auto while_body = detail::vector_codegen<
        []() -> ast_node_vec_t {
          return std::move(
              static_cast<ast_while_t &>(*Generator())
                  .block.content);
        }>();

    // Encapsulating it inside a while loop
    return [while_body](program_state_t &state) {
      while (state.data[state.i]) {
        while_body(state);
      }
    };
  }
}

} // namespace brainfuck::pass_by_generator
```

.1.5 . Constexpr Shunting Yard implementation

Listing 5: `shunting_yard.hpp`

```
#pragma once

// Implementation based on:
// https://en.wikipedia.org/wiki/Shunting_yard_algorithm

#include <algorithm>
#include <iterator>
#include <string_view>
#include <variant>
#include <vector>

#include <kumi/tuple.hpp>

#include <fmt/core.h>

namespace shunting_yard {

/// Operator associativity
enum operator_associativity_t { left_v, right_v };

/// Literal version for token base type (see token_t)
struct token_base_t {
  std::string_view text;
};

/// Literal version for empty token for parsing
/// failure management
struct failure_t : token_base_t {
  constexpr failure_t() : token_base_t{{}} {}
};

/// Literal version for variable spec type
struct variable_t : token_base_t {
  constexpr variable_t(std::string_view identifier)
      : token_base_t{identifier} {}
};

/// Literal version for function spec type
struct function_t : token_base_t {
  constexpr function_t(std::string_view identifier)
      : token_base_t{identifier} {}
};

/// Literal version for operator spec type
struct operator_t : token_base_t {
  operator_associativity_t associativity;
  unsigned precedence;
```

```
  constexpr operator_t(
      std::string_view identifier,
      operator_associativity_t associativity_,
      unsigned precedence_)
      : token_base_t{identifier},
        associativity(associativity_),
        precedence(precedence_) {}
};

/// Literal version for left parenthesis spec type
struct lparen_t : token_base_t {
  constexpr lparen_t(std::string_view identifier)
      : token_base_t{identifier} {}
};

/// Literal version for right parenthesis spec type
struct rparen_t : token_base_t {
  constexpr rparen_t(std::string_view identifier)
      : token_base_t{identifier} {}
};

/// Constant (unsigned integer)
struct constant_t : token_base_t {
  unsigned value;
  constexpr constant_t(unsigned value_,
                       std::string_view number)
      : token_base_t{number}, value(value_) {}
};

/// Literal generic type for a token.
using token_variant_t =
    std::variant<failure_t, variable_t, function_t,
                 operator_t, lparen_t, rparen_t,
                 constant_t>;

// Sanity check
namespace _test {
constexpr token_variant_t
    test_literal_token(constant_t(1, "one"));
}

/// Token kind to match with token types
enum token_kind_t {
  failure_v,
  variable_v,
  function_v,
  operator_v,
  lparen_v,
  rparen_v,
  constant_v,
};
```

```
/// Helper kind getter for literal_failure_t.
constexpr token_kind_t get_kind(failure_t const &) {
  return failure_v;
}
/// Helper kind getter for literal_variable_t.
constexpr token_kind_t get_kind(variable_t const &) {
  return variable_v;
}
/// Helper kind getter for literal_function_t.
constexpr token_kind_t get_kind(function_t const &) {
  return function_v;
}
/// Helper kind getter for literal_operator_t.
constexpr token_kind_t get_kind(operator_t const &) {
  return operator_v;
}
/// Helper kind getter for literal_lparen_t.
constexpr token_kind_t get_kind(lparen_t const &) {
  return lparen_v;
}
/// Helper kind getter for literal_rparen_t.
constexpr token_kind_t get_kind(rparen_t const &) {
  return rparen_v;
}
/// Helper kind getter for literal_constant_t.
constexpr token_kind_t get_kind(constant_t const &) {
  return constant_v;
}

/// Definition of the various tokens for an algebra:
/// variable identifiers, function identifiers, infix
/// operators, and parenthesis.
struct token_specification_t {
  std::vector<variable_t> variables;
  std::vector<function_t> functions;
  std::vector<operator_t> operators;
  std::vector<lparen_t> lparens;
  std::vector<rparen_t> rparens;
};

/// Represents the parsing result of parse_formula.
using parse_result_t = std::vector<token_variant_t>;

/// Tries to parse a token from the token list and
/// returns an iterator to it.
/// - If found, it will be removed from the beginning
/// of formula.
/// - If not, formula remains unchanged and the
/// iterator will be the end of the
///   range.
```

```
/// Whitespaces are not trimmed by the function either
/// before or after the parsing.
constexpr auto parse_token_from_spec_list(
    std::string_view &formula,
    auto const &token_list_begin,
    auto const &token_list_end) {
  // Try to find the token from the list
  auto token_iterator = std::find_if(
      token_list_begin, token_list_end,
      [&](token_variant_t const &token) {
        return std::visit(
            [&](auto const &visited_token) -> bool {
              return formula.starts_with(
                  visited_token.text);
            },
            token);
      });

  // If found, remove it from the beginning
  if (token_iterator != token_list_end) {
    formula.remove_prefix(
        token_iterator->text.size());
  }

  return token_iterator;
}

/// Tries to parse a number.
/// - If found, it will return a constant_t object
/// holding its value, and remove
///   the number from the beginning of the formula.
/// - If not, formula remains unchanged and it will
/// return a failure_t object. Whitespaces are not
/// trimmed by the function either before or after the
/// parsing.
constexpr token_variant_t
parse_number(std::string_view &text) {
  // Checking for presence of a digit
  std::size_t find_result =
      text.find_first_not_of("0123456789");

  if (find_result == 0) {
    return failure_t();
  }

  // No character other than a digit means it's all
  // digit
  std::size_t number_end_pos =
      find_result == std::string_view::npos
          ? text.size()
          : find_result;
```

```
  // Accumulate digits
  unsigned result = 0;
  for (std::size_t digit_index = 0;
       digit_index < number_end_pos; digit_index++) {
    result += result * 10 + (text[digit_index] - '0');
  }

  std::string_view number_view =
      text.substr(0, number_end_pos);
  text.remove_prefix(number_end_pos);
  return constant_t(result, number_view);
}

/// Trims characters from ignore_list at the beginning
/// of the formula.
constexpr void trim_formula(
    std::string_view &formula,
    std::string_view ignore_list = ", \t\n") {
  if (std::size_t n =
          formula.find_first_not_of(ignore_list);
      n != std::string_view::npos) {
    formula.remove_prefix(n);
  }
}

/// Parses a formula. The result is a vector of
/// pointers to token_spec_t elements contained in the
/// various vectors of spec.
parse_result_t constexpr parse_to_rpn(
    std::string_view formula,
    token_specification_t const &spec) {
  // The functions referred to in this algorithm are
  // simple single argument functions such as sine,
  // inverse or factorial.

  // This implementation does not implement composite
  // functions, functions with a variable number of
  // arguments, or unary operators.

  parse_result_t output_queue;
  std::vector<token_variant_t> operator_stack;

  if !consteval {
    fmt::print("Starting formula: \"{}\"\n", formula);
  }

  // There are tokens to be read
  while (trim_formula(formula), !formula.empty()) {
    // Debug logs
    if !consteval {
```

```
    fmt::println("- Remaining: \"{}\"", formula);

    fmt::print("  Output queue: ");
    for (token_variant_t const &current_token :
         output_queue) {
      fmt::print("{} ",
                 std::visit(
                     [](auto const &visited_token) {
                       return visited_token.text;
                     },
                     current_token));
    }
    fmt::println("");

    fmt::print("  Operator stack: ");
    for (token_variant_t const &current_token :
         operator_stack) {
      fmt::print("{} ",
                 std::visit(
                     [](auto const &visited_token) {
                       return visited_token.text;
                     },
                     current_token));
    }
    fmt::println("");
  }
  // read a token

  // Token is a number constant
  if (token_variant_t parsed_token =
          parse_number(formula);
      std::holds_alternative<constant_t>(
          parsed_token)) {
    if !consteval {
      fmt::println("Reading number");
    }
    // Put it into the output queue
    output_queue.push_back(parsed_token);
  }

  // Token is a variable
  else if (auto variable_spec_iterator =
               parse_token_from_spec_list(
                   formula, spec.variables.begin(),
                   spec.variables.end());
           variable_spec_iterator !=
           spec.variables.end()) {
    if !consteval {
      fmt::println("Reading variable");
    }
    // Put it into the output queue
```

```
  output_queue.push_back(*variable_spec_iterator);
}

// Token is a function
else if (auto function_spec_iterator =
             parse_token_from_spec_list(
                 formula, spec.functions.begin(),
                 spec.functions.end());
         function_spec_iterator !=
         spec.functions.end()) {
  if !consteval {
    fmt::println("Reading function");
  }
  // Push it onto the operator stack
  operator_stack.push_back(
      *function_spec_iterator);
}

// Token is an operator 'a'
else if (auto operator_a_spec_iterator =
             parse_token_from_spec_list(
                 formula, spec.operators.begin(),
                 spec.operators.end());
         operator_a_spec_iterator !=
         spec.operators.end()) {
  if !consteval {
    fmt::println("Reading operator");
  }

  operator_t const &operator_a =
      *operator_a_spec_iterator;

  // while there is an operator 'b' at the top of
  // the operator stack which is not a left
  // parenthesis,
  while (!operator_stack.empty() &&
         std::visit(
             [&]<typename BType>(
                 BType const &operator_b_as_auto)
                 -> bool {
               if constexpr (std::is_same_v<
                                 BType,
                                 operator_t>) {
                 operator_t const &operator_b =
                     operator_b_as_auto;
                 // if ('b' has greater precedence
                 // than 'a' or
                 // ('a' and 'b' have the same
                 // precedence and 'a' is
                 // left-associative))
                 if (operator_b.precedence >
```

```
                        operator_a.precedence ||
                    (operator_a.precedence ==
                         operator_b.precedence &&
                     operator_a.associativity ==
                         left_v)) {
                  return true;
                }
              } else if constexpr (
                  std::is_same_v<BType,
                                 function_t>) {
                // or 'b' is a function
                return true;
              }
              // left parenthesis or lower
              // precedence operator
              return false;
            },
            operator_stack.back())) {
    // pop 'b' from the operator stack into the
    // output queue
    output_queue.push_back(operator_stack.back());
    operator_stack.pop_back();
  }
  // push 'a' onto the operator stack
  operator_stack.push_back(
      *operator_a_spec_iterator);
}

// Token is a left parenthesis (i.e. "(")
else if (auto lparen_token_iterator =
             parse_token_from_spec_list(
                 formula, spec.lparens.begin(),
                 spec.lparens.end());
         lparen_token_iterator !=
         spec.lparens.end()) {
  if !consteval {
    fmt::println("Reading lparen");
  }
  // push it onto the operator stack
  operator_stack.push_back(
      *lparen_token_iterator);
}

// Token is a right parenthesis (i.e. ")")
else if (auto rparen_token_iterator =
             parse_token_from_spec_list(
                 formula, spec.rparens.begin(),
                 spec.rparens.end());
         rparen_token_iterator !=
         spec.rparens.end()) {
  if !consteval {
```

```
        fmt::println("Reading rparen");
      }
      // the operator at the top of the operator stack
      // is not a left parenthesis
      while (operator_stack.empty() ||
             !std::holds_alternative<lparen_t>(
                 operator_stack.back())) {

        // {assert the operator stack is not empty}
        if (operator_stack.empty()) {
          // If the stack runs out without finding a
          // left parenthesis, then there are
          // mismatched parentheses.
          fmt::println("Parenthesis mismatch.");
          throw;
        }
        // pop the operator from the operator stack
        // into the output queue
        output_queue.push_back(operator_stack.back());
        operator_stack.pop_back();
      }

      // {assert there is a left parenthesis at the
      // top of the operator stack}
      if (operator_stack.empty() ||
          !std::holds_alternative<lparen_t>(
              operator_stack.back())) {
        throw;
      }

      // pop the left parenthesis from the operator
      // stack and discard it
      operator_stack.pop_back();

      // there is a function token at the top of the
      // operator stack
      if (!operator_stack.empty() &&
          std::holds_alternative<function_t>(
              operator_stack.back())) {
        // pop the function from the operator stack
        // into the output queue
        output_queue.push_back(operator_stack.back());
        operator_stack.pop_back();
      }
    }
  }
  /* After the while loop, pop the remaining items
   * from the operator stack into the output queue. */

  // there are tokens on the operator stack
  while (!operator_stack.empty()) {
```

```
    // If the operator token on the top of the stack
    // is a parenthesis, then there are mismatched
    // parentheses.

    // {assert the operator on top of the stack is not
    // a (left) parenthesis}
    if (std::holds_alternative<lparen_t>(
            operator_stack.back())) {
      throw;
    }

    // pop the operator from the operator stack onto
    // the output queue
    output_queue.push_back(operator_stack.back());
    operator_stack.pop_back();
  }

  return output_queue;
}

/// Asuming Fun is a constexpr function that returns a
/// std::vector value, eval_as_array will store its
/// contents into an std::array.
template <auto Fun> constexpr auto eval_as_array() {
  constexpr std::size_t Size = Fun().size();
  std::array<typename decltype(Fun())::value_type,
             Size>
      res;
  std::ranges::copy(Fun(), res.begin());
  return res;
}

/// For each token in RPNStackAsArray, consume_tokens
/// will call the
template <auto const &RPNStackAsArray,
          std::size_t RPNStackIndex = 0>
constexpr auto consume_tokens(auto consumer,
                              auto state) {
  // If no token is left to handle, return the value
  // stack
  if constexpr (constexpr std::size_t RPNStackSize =
                    kumi::size_v<std::remove_cvref_t<
                        decltype(RPNStackAsArray)>>;
                RPNStackIndex == RPNStackSize) {
    return state;
  }
  // Otherwise, apply stack for given token and
  // recurse on next token
  else if constexpr (RPNStackIndex < RPNStackSize) {
    // Apply current stack and pass front token as a
    // template parameter
```

```
    return consume_tokens<RPNStackAsArray,
                          RPNStackIndex + 1>(
        consumer,
        consumer.template
        operator()<RPNStackAsArray, RPNStackIndex>(
            state));
  }
}

} // namespace shunting_yard
```