Techniques avancées de génération de code pour le parallélisme

Advanced techniques for parallel code generation

Thèse de doctorat de l'université Paris-Saclay

École doctorale n° d'accréditation, dénomination et sigle
Spécialité de doctorat: voir annexe
Graduate School : voir annexe. Référent : voir annexe

Thèse préparée dans la (ou les) unité(s) de recherche **STIC** (voir annexe), sous la direction de **Joel FALCOU**, titre du directeur ou de la directrice de thèse

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

Jules PÉNUCHOT

Composition du jury

Membres du jury avec voix délibérative

Prénom NOM Titre, Affiliation	Président ou Présidente
Prénom NOM Titre, Affiliation	Rapporteur & Examinateur / trice
Prénom NOM Titre, Affiliation	Rapporteur & Examinateur / trice
Prénom NOM Titre, Affiliation	Examinateur ou Examinatrice
Prénom NOM Titre, Affiliation	Examinateur ou Examinatrice

THESE DE DOCTORAT

NNT : 2020UPASA001

Remerciements

université PARIS-SACLAY

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

Titre: Techniques avancées de génération de code pour le parallélisme
Mots clés: Métaprogrammation, compilation, C++

Résumé: Mettre le sum ici

Title: Advanced techniques for parallel code generation
Keywords: Metaprogramming, compilation, C++

Abstract: Mettre l'abstract ici

Contents

Part I

Current state of 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 doing alive and well: transistor count is still doubling every year despite people claiming it is not alive anymore. This is due to a common confusion between transistor count and computing power. While transistor count has been steadily rising, the throughput computing performance of single Central Processing Unit (CPU) cores has not.

The need for more parallel software is driven by the fact that processors have become increasingly parallel [52] in order to compensate for the lack of serial execution performance improvements.

Flynn's taxonomy describes four ways in which computers can perform operations concurrently or not:

- **Multiple Instruction Multiple Data stream (SIMD)**, which consists in having several processors or cores executing different proccesses. This category of parallelism itself has several sub-categories depending on the memory topology:
  - **Shared memory**, where processing units share a single pool of memory. Multi-threading is an example of such a topology as CPU cores share the same pool of RAM.
  - **Distributed memory**, where processing units have their own memories and cannot access other memory pools.
  - **Distributed shared memory**, where processing units have their own memory pools and can access other memory pools indirectly.
- **Multiple Instruction Single Data stream (SIMD)**, which is primarily used for fault tolerance by having execution redundancy.

- **Single Instruction Multiple Data stream (SIMD)**, which is a parallelism that can be implemented inside a single CPU core by having vector registers. These registers can hold several values, and single instructions can operate on the vectors themselves.
- **Single Instruction Single Data stream (SIMD)**, where a computer only has a single processor with no vector registers.

Note that these categories are not always mutually exclusive. Modern CPUs in personal computers nowadays have multiple cores, each one of them having vector registers. Therefore, modern CPUs leverage a mix of SIMD and MIMD computing to achieve higher throughput.

Graphical Processing Units (GPUs)

The objective of this thesis is to develop and assess new ways to write C++ metaprograms that enable better compile time performance, as well as better maintainability by using

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.

C and C++ preprocessor

Lisp macros

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.

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

Rust proposes metaprogramming through macros, generics, and traits.

Rust macros are more powerful than those proposed in C and C++. They have a

Braid [46] is language that implements metaprogramming through multi-staged programming (like MetaOCaml) for heterogeneous real-time 3D graphics. It is however unmaintained.

Julia [7] is a dynamic language that uses the LLVM

Terra Terra[12] 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.

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.

As we will see in section **??**, GPU computing libraries rely heavily on C++ metaprogramming to provide building blocks for portable high performance compute kernels.

1.2 . C++ language constructs for metaprogramming

Templates are an interesting technique for generative programming. As they are Turing-complete [55], one can design a set of **template metaprograms** [1] allowing the compiler to perform arbitrary computation at compile time and generating temporary C++ code fragment as an output. The resulting temporary source code is then merged with the rest of the source code and finally processed by the classic compilation process.

Listing 1.1 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 1.1: Example of compile time 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;
};

// Specialization 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;
```

Due to the fact that template code generation is performed at compile time, it is limited to the use of constants and supports pattern-matching and recursion thanks to template partial specialization. It can then be looked at as a pure functional language [28]. The execution of metaprograms by the compiler enables libraries to implement domain-specific optimizations.

Metaprograms can be seen as programs that take code as input and/or output. This distinction will be important to explain the work presented in this thesis, as it focuses a lot on methods to translate compile time compute results into C++ programs.

1.2.1 . 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.

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

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

```cpp
/// 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.

```cpp
/// 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>;
```

1.2.2 . 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 (TMP) in which types are used to represent everything from enumerations, arrays, or even functions.

- **Non-Type Template Parameters (NTTPs)** are complementary with type parameters. They can be used to wrap values within types, which can be particularly useful in the context of TMP. 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.

  For example a type template definition like `template<typename ... Ts > tuple_t{};` can be used to store an arbitrary list of types as a single type.

- **constexpr function parameters**, even though not all of them can be used as NTTPs, they can be used to produce NTTP compatible values.

1.2.3 . 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.

1.2.4 . 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

- **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 allow compile time programs to run much faster than pure TMP does.

1.3 . Metaprogramming libraries

As previously said C++ templates can be seen as a functional language. Over time a range of libraries emerged, aiming to provide functionalities similar to regular language such as containers and algorithms for use in template metaprograms. Notable examples of such libraries are MPL[26], Brigand[29], and mp11 [14].

Listing 1.2: boost.mp11 code example

```
template <int X> struct equals_to {
  template <typename Y>
  using apply = mp_bool<X == Y::type::value>;
};

using my_list = mp_list_c<int, 0, 2, 4, 6, 8, 10>;

using pos_of_6 =
    mp_find_if<my_list, equals_to<6>::apply>; // 3
```

Listing 1.3: 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
```

Listings 1.2 and 1.3 show how to use mp11 and Brigand to perform a basic task: finding the index of a value in a list at compile time.

Libraries for more specific uses were also introduced, such as Compile Time Regular Expression (CTRE) [19] for compiling regular expressions, and Compile Time Parser Generator (CTPG) [63] for generating LR1 parsers (also not for compile time parsing).

Listing 1.4: CTRE code example

```
auto input = "123,456,768"sv;

for (auto match : ctre::range<"([0-9]+),?">(input)) {
  std::cout << std::string_view{match.get<0>()}
            << "\n";
}
```

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

The C'est[35] is one such example, as it aims to be a constexpr compatible replacement for the C++ standard library. It implements the same containers and algorithms, although all of them usable in constant evaluations. For example, `std::deque` is not usable in constant evaluations whereas its C'est equivalent (`cest::deque`) is.

Listing 1.5: 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});
```

In listing 1.5 we can see an example compile time program equivalent to what was shown for Brigand and mp11. Thanks to new C++ features, metaprogramming can be partially done using C++ code. This makes compile time programs significantly easier to write and understand.

All these examples show that metaprogramming is not just a gimmick. There is enough interest in leveraging metaprogramming for generic programming that many frameworks, libraries, or languages have been developed around this paradigm and maintained sometimes for more than a decade. Its adoption in languages like Braid, Julia, Terra, Rust, or Dlang shows that it is not limited to C++.

However, most of generic libraries and frameworks that are developed and maintained today are based on C++, and C++ itself is likely to evolve to provide more ways to generate programs as shown by the recent standard proposal for Scalable Reflection in C++ [57].

1.4 . Applications of metaprogramming for HPC

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 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** [25] 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 3.1.

```
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 designed has since been updated.

```cpp
using blaze::DynamicVector;
using blaze::StaticVector;

int main() {
  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.

```cpp
using namespace nt2;

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

  x = 40.0 * y + 2.0;

  NT2_DISPLAY(x);
}
```

- **EVE** [22] provides generic abstractions over SIMD instructions as well as SIMD-optimized generic algorithms for the development of high performance and portable SIMD code [43].

```cpp
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** [34] 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.

```cpp
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** [6] implements GPU-accelerated equivalents of the Standard Library's algorithms, while CUB [40] 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.

```cpp
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.

In the next chapter, I will focus on the use of Boost.SIMD [20] for the generation of BLAS-compatible linear algebra kernels.

2 - Code generation at low level

BLAS-level functions 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 gemv exists, the interest for small-sized, highly-optimized versions of those routines emerged. In this section, we propose to show how a modern C++ approach based on generative programming techniques such as vectorization and loop unrolling in the framework of metaprogramming can deliver efficient automatically generated codes for such routines, that are competitive with existing, hand-tuned library kernels with a very low programming effort compared to writing assembly code. In particular, we analyze the performance of automatically generated small-sized gemv kernels for both Intel x86 and ARM processors. We show through a performance comparison with the OpenBLAS gemv kernel on small matrices of sizes ranging from 4 to 32 that our C++ kernels are very efficient and may have a performance that is up to 3 times better than that of OpenBLAS gemv.

2.1 . Introduction

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[65]. Thus, it offers a framework to develop dense linear solvers using algorithmic skeletons[9] 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[60] or other generative programming[11] 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.

In this section, we describe how we obtained optimized generated C++ codes for the gemv routine to make it reach its peak performance on different target CPU architectures (Intel x86 and ARM in this section). Our gemv C++ generated kernels achieve uniform performances that outperform in the most of the cases the peaks of the state-of-the-art OpenBLAS gemv kernel for small matrices of sizes ranging from 4 to 128. This chapter is organized as follows: in Section II, we describe various generative programming techniques including metaprogramming principles as well as the main strategies to get efficient small-scale BLAS functions. In Section III, we show how to apply these programming techniques and principles in order to develop an efficient small-scale gemv kernel that will be used to automatically generates efficient C++ codes for different target architectures. Then, we present in Section IV performance comparisons on two CPU architectures (Intel x86 and ARM) between our generated gemv kernels for small matrices and the OpenBLAS gemv. Finally, concluding remarks and perspectives are given in Section V.

2.2 . High-performance generative programming

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 proper C++ SIMD layer and a set of code generation techniques stemmed from Generic Programming.

2.2.1 . Metaprogramming as code generation principles

Metaprogramming[23] is about the design and the implementation of programs whose input and output are themselves programs. This term encompasses a large body of idioms, some language dependent, which can be used to define, manipulate or introspect arbitrary code fragment. One typical usage of metaprogramming is the design of libraries which expose a user API with an arbitrary high abstraction level while being able to get compiled to a very "close to the metal" implementation, often rivaling with a handwritten expert code on a given machine[10]. If metaprogramming is used in languages as different as C++ [2],D[8], OCaml[53] or Haskell[47], 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.

- Code expansion: Once the initial code fragments have been processed, the last step is to turn them into an actual code. This is often done in an explicit manner by using a function or a syntax construct provided by the metaprogramming layer to trigger the code generation. Note that this code generation can either lead to a code ready to be compiled – like in Haskell or C++ – or a code that can be run – like in OCaml– if the generation phase is done at runtime. Metaprogramming also includes other code generation techniques such as domain-specific languages and compilation infrastructures based on source-to-source compilers, which are actually able to perform the same techniques proposed by this section. Such systems includes:

- SYCL[37]: a single-source abstraction over OpenCL for heterogeneous systems. By using OpenCL, one can effectively write SYCL code for a large selection of architectures including SIMD capable CPU.

- BLIS[56]: a framework for generating BLAS like operations in ISO C99 from a small subset of kernels that can be retargeted for different backend. Its performances are on par with open-source solutions like OpenBLAS and ATLAS.

- LGEN[48]: 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.

In this section, we will 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. Classical design of metaprograms in C++ usually relies on complex template types that forced the compiler to follow intricate path during type deduction in order to take advantage of the Turing completeness of the template definition. By using template partial specialization and recursive definition, one could implement arbitrary transform on types in order to converge towards a code ready to be generated. Code fragments were usually static class member function which encapsulated the basic code block to be replicated and generated. If the efficiency of the code generated was as expected, the maintenance cost of the generating code was usually high. Template metaprograms were complex to write and read as the logic of the code generation was buried behind heaps of non-trivial syntax. Some progress was made by some infrastructure library like MPL[27] or Fusion, but the learning gap was still high.

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

```cpp
// 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;
```

2.2.2 . Application to High Performance Computing code generation

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 that is vectorization 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.

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);
}
```

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.

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 constant 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.

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.

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>{});
}
```

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.

2.3 . The matrix-vector multiplication routine: gemv

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%[54]. 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[18].

Here, we consider the matrix-vector multiplication routine for general dense matrices, gemv, which performs either $y := \alpha Ax + \beta y$ or $y := \alpha ATx + \beta y$, where A is an $m \times n$ 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 7 is made up of three main steps as detailed in Figure 1: Broadcast of each element of the vector in different registersSIMDSIMD Scalar Fig. 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 verisimilitude 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.

2.4 . 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[64]. We use GCC 7.2[50] 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.

2.4.1 . On X86 Intel processor

Fig. 2. GEMV performance on Intel i5-7200 processor using SIMD Extensions set (SSE-4.2)

In Figure 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.

Fig. 3. GEMV performance on Intel i5-7200 processor using Advanced Vector Extensions (AVX)

In Figure 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 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[62] 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.

2.4.2 . On ARM processor

The comparison between our automatically generated code and the ARM OpenBLAS gemv kernel is given in Figure 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 optimizations generates a better code than the hand-written assembly approach of the OpenBLAS library.

Fig. 4. GEMV performance on ARM Cortex A57 processor

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.

2.5 . 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 are highly optimized for small matrix sizes.

One of our next steps is to provide such metaprogrammed code generation process to tackle larger matrix sizes by precomputing an optimal tiling availability of prefetch and architecture, turning large-scale gemv into an optimized sequence of statically sized small-scale gemvs. Such a technique is also naturally applicable to more complex algorithms, such the matrix-matrix multiplication, gemm, where tiling is paramount, or LAPACK-level functions where the compile-time optimization may lead to an easier to parallelize solvers or decompositions.

Another objective will be to adapt such techniques to cooperate with tuning framework, hence providing the required level of performance with less input from the user.

3 - Code generation at expression level

3.1 . Expression level

In this chapter I will cover the use of TMP for higher levels of abstraction. Templates can be used to represent whole mathematical expressions at compile time by creating type-based arborescences. This type of representation is called an Expression Template (ET) [61].

Combined with compile-time mechanisms such as function overloading, specialization, and operator overloading, ETs can be used to implement expression level DSELs and convert complex mathematical expressions into high performance code.

There are two main libraries that are able to do just this: Eigen [25] and Blaze [30], which were cited in 1.3. In this section, I will introduce the basics of C++ TMP,

3.1.1 . Expression templates: a type-based representation for math formulas

Expression templates are template trees that represent math formulas. They are generated using operator and function overloading from expressions that must be known at compile time.

```
template<typename Left, typename Right>
struct add_t{};
```

3.1.2 . Optimized code generation

Enables a whole range of optimizations:

Eliminating temporaries with lazy evaluation

Leveraging BLAS libraries

Parallelism

SIMD

Multithreading

Graphical Processing Unit support efforts

Lots of rewrite to do

Potentially a job for source rewriting tools

- Eventually: GPU code generation, although Blaze needs a significant rewrite for that. Source rewriting tools might be a good fit for that job.

Expression templates can provide expression level APIs for HPC libraries. Still two limitations:

- Slow compilation times
- C++ syntax only

Expression templates are aging (pretty well but still). Newer C++ standards provide metaprogramming features that can fundamentally change the way we write metaprograms.

The next part of my thesis will focus on how to leverage these features to implement DSEL of arbitrary syntax, and the study of their impact on compilation times.

Part II

C++ metaprogramming beyond templates

4 - Establishing a methodology for compile time benchmarking

With TMP libraries like Eigen[25], Blaze[30], or CTRE[19] becoming larger and more common, we're seeing increasing computing needs at compile time. 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[16] or static reflection[58]. However, there is still no clear cut methodology to compare the performance impact of different metaprogramming strategies. But as new C++ features allow for new techniques with claimed better compile time performance, no proper methodology is provided to 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 aims to provide developer-friendly tools to declare and run benchmarks, then aggregate, filter out, and plot the data to analyze it. As such, ctbench is meant to become the first layer of a proper scientific methodology for analyzing compile time program behavior.

We'll first have a look at current tools for compile time profiling and benchmarking and establish the limits of current tooling, then I'll explain what ctbench brings to overcome these limits.

ctbench was first presented at the CPPP 2021 conference[41] which is the main C++ technical conference in France. It is being used to benchmark examples from the poacher[42] project, which was briefly presented at the Meeting C++ 2022[36] technical conference.

4.1 . Compile time benchmarking: state of the art

C++ TMP 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 frequent. Consequently developers now have to bear with longer compilation times, often without being able to explain them. 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.

4.1.1 . Metabench

Metabench[17] 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 is currently unmaintained, and its design choices make 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, they are embedded in the Metabench project itself,
- web-based visualizations cannot be embedded in LaTeX documents.

4.1.2 . Templight

Templight[44] 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.

4.1.3 . Clang's built-in profiler

Additionally, Clang has a built-in profiler[3] that provides 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 4.1.

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 4.1. Duration and timestamps are expressed in microseconds.

Listing 4.1: Time trace event generated by Clang's internal profiler

```
{
  "pid": 8696,
```

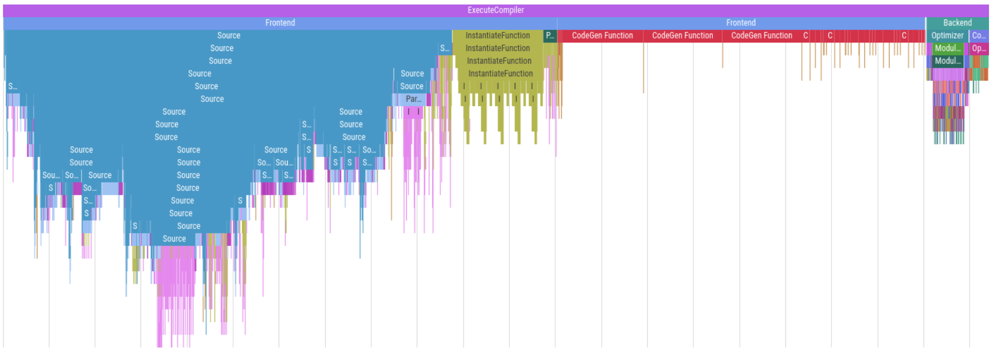

Figure 4.1: Clang time trace file in Perfetto UI

```
  "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 durin this trace event. This field may also contain details related to symbols being processed for events like `InstantiateFunction`.

Clang's profiler data is very exhaustive and insightful, however there is no tooling to make sense of it in the context of variable size compile time benchmarks. ctbench tries to bridge the gap by providing a tool to analyze this valuable data. It also improves upon existing tools by providing a solution that's easy to integrate into existing CMake projects, and generates graphs in various formats that are trivially embeddable in documents like research papers, web pages, or documentations. Additionally, relying on persistent configuration, benchmark declaration and description files provides strong guarantees for benchmark reproductibility, as opposed to web tools or interactive profilers.

4.1.4 . Conclusion

While Metabench could have been a good candidate for assessing the scalability of metaprogramming techniques, its design choices make it simpler to write a new tool that better fits the needs of scientific assessment of metaprogramming techniques.

ctbench's API and overall concept was largely inspired by this project for its simplicity and leverages Clang's profiling capabilities that are similar to those introduced by Templight.

4.2 . ctbench design and 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 a well maintained, easy to install, and easy to use alternative to current compile time benchmarking tools. It is developed with decent software quality in mind to offer a robust external API, and a well documented internal API to enable the reuse of its components.

All these goals are kept in mind to ensure that ctbench is not just a single use tool for this thesis, but a sustainable open-source project that enables researchers and developers to automatize the generation of reusable graphs and data artifacts.

4.2.1 . CMake API for benchmark and graph target declarations

The choice of using CMake as an entry point is to ensure that ctbench is not just a research tool made for a single set of benchmarks, but an easy-to-use solution for all C++ developers willing to analyze the compile time performance of their metaprograms.

The public CMake API of ctbench is meant to be as stable as reasonable. It is relatively simple, and all of it is implemented in a single CMake script file. It declares 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 using this function are compiled with their sized passed as the `BENCHMARK_SIZE` define.
  - `ctbench_add_benchmark_for_range`, similar to `ctbench_add_benchmark` except for the fact 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 `ctbench_add_benchmark` 's interface 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.

Now that we have a summary of ctbench's high level API, it is time to take a look at its internals.

4.2.2 . ctbench internals

Besides the CMake scripts, ctbench is organized into several components:

- **grapher**: a C++ library and CLI tool that reads benchmark data and draws plots. It

- **compiler-launcher**: a simple program that handles compiler flags in order to move JSON profiling output to the right place for grapher to retrieve it, measures compiler execution time if required, and invokes the desired compiler as part of the `ctbench_set_compiler` CMake command implementation.

grapher: reading and plotting benchmark results

The grapher sub-project provides data structures for reading

- benchmark data structures and algorithms `<benchmark target name >/<iteration size>/<iteration number>.json`

- plotter engines

- reusable svg
- predicate interface
- CLI
- sciplot

compiler-launcher: working around CMake's limitations

CMake has its own limitations that would make the implementation of ct-bench impractical, or even impossible without a compiler launcher. Thankfully, CMake does support compiler launchers through the `CMAKE_CXX_COMPILER_LAUNCHER` target property, allowing 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 support more use cases.

- find out where json files are located with `-ftime-trace`
- CLI option parsing
- compiler execution time measurement

4.2.3 . Software quality

An emphasis was put on software quality since the beginning of the project. As mentioned, the goal was not just to develop a plotting tool for a single compilation time analysis, but to provide a methodology along with a robust implementation to improve compilation time analysis as a whole.

- The use of common C++ development tools for the project facilitates makes it easier for users to contribute to the project.
- The use of a GitHub CI for building and testing the project guarantees that project remains functional at all times on all supported platforms. As of writing, the project is continuously built and tested for Ubuntu 23.04 and Arch Linux. The test environment is fully reproducible as well thanks to the use of Docker for its setup.
- compiler checks: clang-tidy and clang-format
- Packages are provided through the AUR and the vcpkg repository
- Accepted in JOSS, which puts an emphasis on software quality. the review itself is meant to improve the quality of the software thanks to the reviewers' feedback.

4.2.4 . Conclusion

ctbench is a well

4.3 . A ctbench 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 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:

```cpp
#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 is based on a recursive template function:

```cpp
/// Recursive compile-time sum 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>{};
}
```

And the other version relies on C++ 11 parameter pack expansion:

```cpp
/// Expansion compile-time sum implementation
template<typename ... Ts> constexpr auto sum();
```

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

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

Both versions share the same interface, and thus the same driver code as well. The driver code takes care of scaling the benchmark according to `BENCHMARK_SIZE`, which is defined by ctbench through the CMake API:

```cpp
// 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>{});
}

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

The CMake code needed to run the benchmarks is the following:

```cmake
ctbench_add_benchmark(
  variadic_sum.expansion variadic_sum/expansion.cpp ${
     BENCHMARK_START}
  ${BENCHMARK_STOP} ${BENCHMARK_STEP} ${
     BENCHMARK_ITERATIONS})

ctbench_add_benchmark(
  variadic_sum.recursive variadic_sum/recursive.cpp ${
     BENCHMARK_START}
  ${BENCHMARK_STOP} ${BENCHMARK_STEP} ${
     BENCHMARK_ITERATIONS})
```

Then a graph target can be declared:

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

with `compare-all.json` containing the following:

```json
{
  "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"]
}
```

This configuration file uses the `compare_by` plotter to generate one plot for each pair of elements 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.

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 graphs following were generated on a Lenovo T470 with an Intel i5 6300U and 8GB of RAM. The compiler is Clang 14.0.6, and Pyperf [51] was used to turn off CPU frequency scaling to improve measurement precision.

The first timer we want to look at is ExecuteCompiler, which is 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.

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.).

The Total Instantiate function timer is an interesting one as it explicitly targets function instanciation time. Note that timers that are prefixed with "Total" measure the total time spent in a timer section, regardless of the specific symbol or source associated to its individual timer events.

Finally, we can take a look at `InstantiateFunction/foovoid.png` which measures the InstantiateFunction event time specifically for `foo<void>()`, which

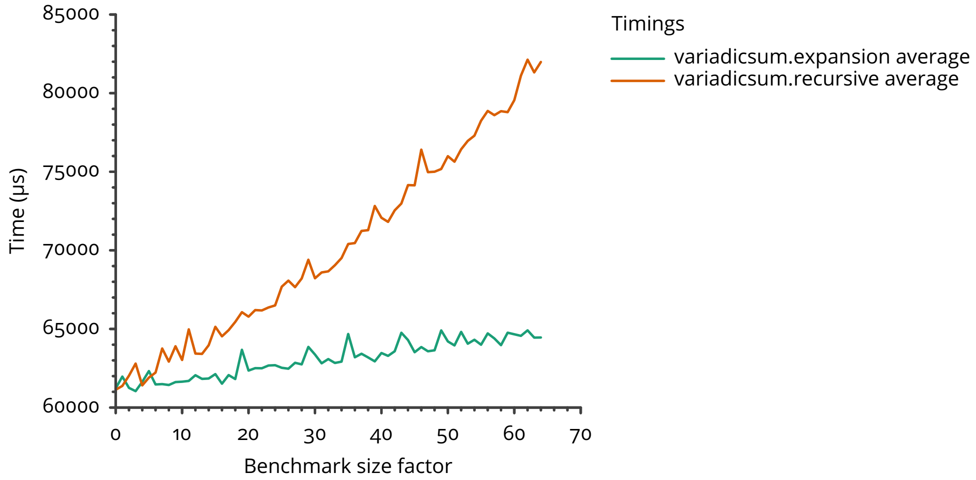

Figure 4.2: ExecuteCompiler

is our driver template function. Using Perfetto UI to look at the timer event hierarchy, we can validate that the timer event for this specific symbol includes the InstantiateFunction time for all the symbols that may be instantiated within this function.

This level of detail and granularity in the analysis of compile-time benchmarks was never reached before, and may help us set good practices to improve the compile-time performance of metaprograms.

4.4 .Related projects

- Poacher (https://github.com/jpenuchot/poacher): Experimental constexpr parsing and code generation for the integration of arbitrary syntax DSL in C++ 20
- Rule of Cheese (https://github.com/jpenuchot/rule-of-cheese): A collection of compile time microbenchmarks to help set better C++ metaprogramming guidelines to improve compile time performance

4.5 .Acknowledgements

We acknowledge contributions from Philippe Virouleau and Paul Keir for their insightful suggestions.

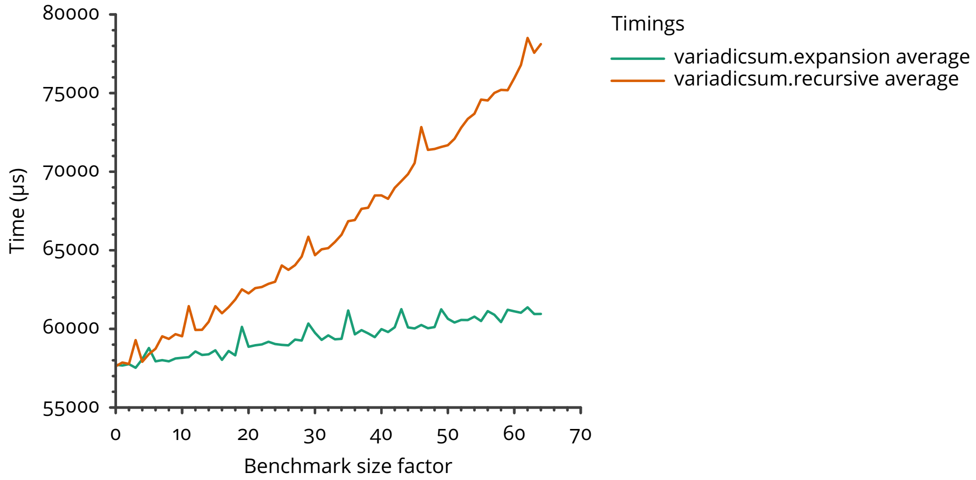

Figure 4.3: Total Frontend

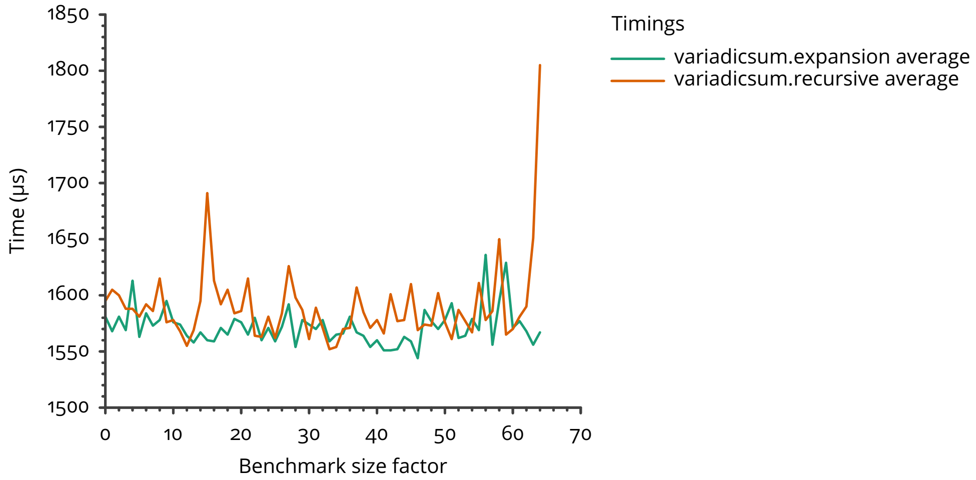

Figure 4.4: Total Backend

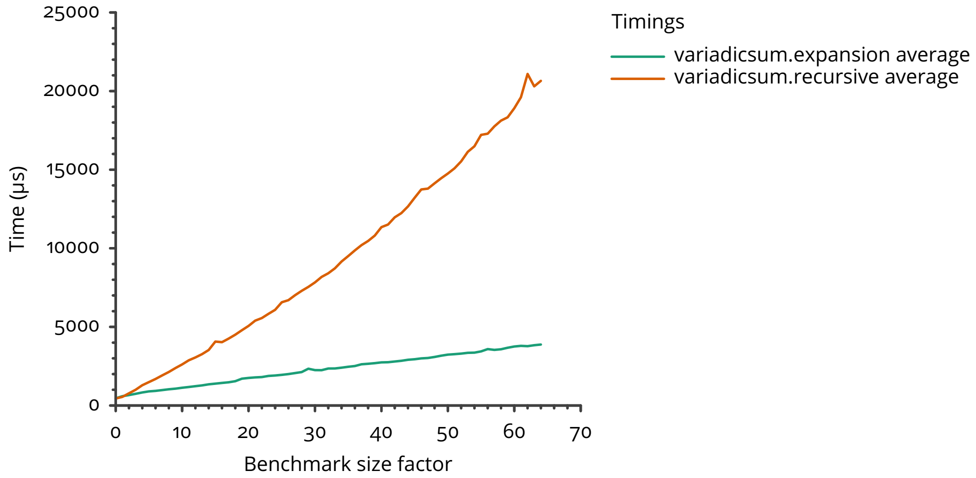

Figure 4.5: Total InstantiateFunction

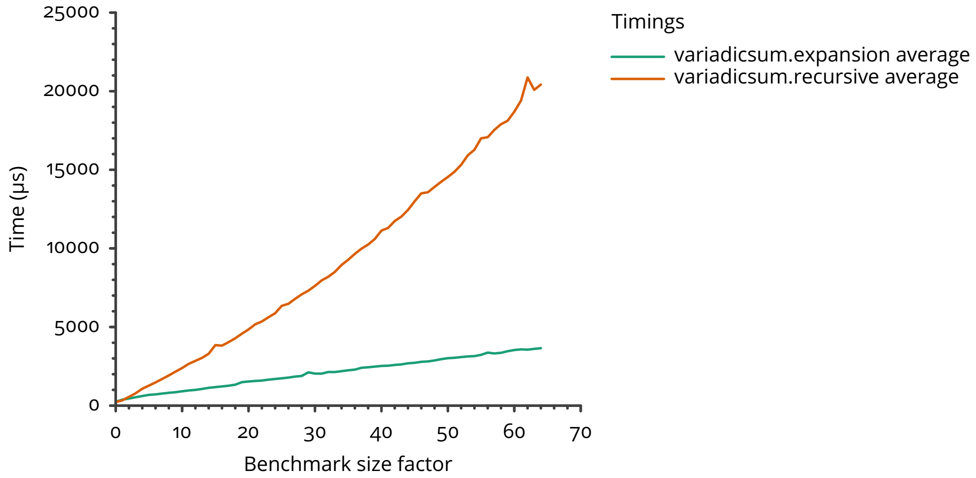

Figure 4.6: InstantiateFunction foovoid

5 - Constexpr parsing for high performance computing

5.1 .Introduction

C++ is often touted as a *Zero-Cost Abstraction* langage due to some of its design philosophy and its ability to compile abstraction to a very efficient binary code. Some more radical techniques can be used to force the compiler to interpret C++ code as a DSEL. TMP is such a technique and it spawned a large corpus of related idioms from compile time function evaluation to lazy evaluation via ETs.

In the field of High Performance Computing, C++ users are often driven to use libraries built on top of those idioms like Eigen[25] 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 thesis, 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 thesis 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 said DSELs in term of compile time to assess their usability in realistic code.

5.2 . A technical background of C++ DSELs

By definition, a Domain Specific Language (DSL) is a computer language specialized to a particular application domain, contrary to a general-purpose language, which is broadly applicable across domains, and lacks specialized features for a particular domain. DSELs are a subclass of DSL that rely on an existing general-purpose language to host it. DSELs then reuse the host language syntax and tool ecosystem to be compiled or interpreted.

In C++, the compile time process of generating new code is TMP to ensure performance and correctness.

5.2.1 . Constexpr programming

Expressions that generate results for use at compile time are called constant expressions. In C++ 11, the constexpr keyword was introduced to qualify functions or variables that may be used in constant expressions. Consequently, regular functions and variables became usable for the evaluation of NTTPs.

However not all functions and variables can be qualified as constexpr, and not all constant expression results can be used as NTTPs.

Since the introduction of the constexpr specifier, the requirements on functions for being constexpr specifiable have constantly been relaxed as new C++ standards were adopted. In C++ 20, notable changes made dynamic memory allocations[45] and virtual constexpr function calls[15] allowed in compile time constexpr function executions.

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.

constexpr functions 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.

constexpr variables 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.

constexpr memory allocation Starting from C++ 20, `std::allocate` and `std::deallocate` are 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 5.1: 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}; }
```

```cpp
// 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 5.2: Transformation from a dynamic vector to a constexpr static array

```cpp
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 5.2 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.

constexpr virtual functions This feature allows calls to virtual functions in constant expressions [15]. This allows heritage-based polymorphism in constexpr programming when used with constexpr allocation of polymorphic types.

Cest: standard-like containers for constexpr programming

With dynamic memory and virtual functions making their way to constexpr programming, the next logical step is to make standard C++ containers available in constant evaluations. Modifying the C++ standard to accomodate that need takes time. However, implementing standard-like containers that are good enough for prototyping and experimenting with constexpr programs can be done fast enough.

The C'est [35] library was created with this goal in mind, filling the gap in the C++ standard, and accommodating the need for 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 a useful library for compatibility with platforms that do not have C++ 23 compatible compilers and standard libraries such as old versions of Debian.

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 and compilers started implementing it.

I contributed to its development myself by implementing a constexpr version of `std::unique_ptr`. This collaboration led Joel Falcou and myself to meet with its initiator, Paul Keir, in Glasgow and Paris. Our collaboration also led to speak together at Meeting C++ 2022 [36] where we talked about C'est and its use cases for constexpr programming research.

5.2.2 . C++ Domain Specific Embedded Languages

DSELs in C++ use TMP via the *Expression Template* idiom. **Expression Templates** [61, 59] is a technique implementing a form of **delayed evaluation** in C++ [49]. Expression Templates are built around the *recursive type composition* idiom [33] that allows the construction, at compile time, of a type representing the Abstract Syntax Tree (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++ EDSLs rely on *Expression Templates* and therefore are limited to the C++ syntax. New techniques are becoming more popular through the use of constexpr strings to embed arbitray DSELs. One major example is the CTRE [19] that implements most of the Perl Compatible Regular Expression (PCRE) syntax. However, CTRE still relies on type-based TMP to parse regular expressions and transform them into regular expression evaluators.

5.3 . Code generation from constexpr allocated structures

In this section, we will present different solutions to generate code from non-literal data structures generated with constexpr functions.

Non-literal data structures allow more flexibility through the use of dynamic memory, and allow simpler code as they do not require contraptions to get around the lack of dynamic allocations.

However, keep in mind that there are still limitations. The solutions evaluated in this thesis are still workarounds. What they allow is the use of dynamically-sized structures for code generation.

The first subsection covers a case where we need to convert a pointer tree returned by a constexpr function into code.

The second one will cover a case where the tree is returned in a serialized representation.

5.3.1 . 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 5.3: `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));
}
```

Listing 5.3 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 5.2.1 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 **??** 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

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 5.1 that depsite `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 5.4: Pass-by-generator

```
/// 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 5.4, 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 **??** 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. GCC 13.2.1 is still unable to compile such code

Pass-by-generator + ET

Why through? Interoperability with type-based metaprogramming libraries.

Types:

```
/// 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 {};
```

Codegen:

```
/// 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 to_expression_template() {
  static_assert(Fun() != nullptr, "Ill-formed tree");

  constexpr node_kind_t Kind = Fun()->kind;

  if constexpr (Kind == add_v) {
    // Recursive type generation using the pass-by-
       generator technique
    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{});
}
```

FLAT

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.

```
/// 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;
```

This requires the tree to be serialized first. For the sake of demonstration, we will serialize the tree into an ad hoc data representation that is identical to the original one, except pointers are replaced with `std::size_t` indexes, and `nullptr` is replaced with an arbitrary value called `null_index` as shown in listing 5.3.1.

Heritage polymorphism is also replaced by `std::variant`

These nodes will be stored in `std::vector` containers, and the indexes will refer to the position of other nodes within the container. Note that our tree nodes are not polymorphic. If needed, `std::variant` could have been used to have polymorphic nodes in the serialized representation.

Listing 5.5: constexpr tree serialization implementation

```
/// Serializes the current subtree and returns
/// the top node's index to the caller.
```

```
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;
}

/// Returns a serialized representation of a pointer
   tree.
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 5.5 shows the implementation of the serialization step. The implementation itself has nothing particular, except for the functions being constexpr.

Once the data is serialized, it can be converted into a static array container. 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.

Listing 5.6: 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 5.6 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.

Static arrays are not dynamically allocated, therefore they can be used as NTTPs if their values do not hold pointers to constexpr allocated memory either. In our case, the elements only hold integers, so a `std::array` of `flat_node_t` elements can be used as a NTTPs.

Listing 5.7: Code generation implementation for the flat backend

```
/// Generates code from a tree serialized into a
   static array,
/// with CurrentIndex being the index of the starting
   node which
/// defaults to 0, ie. the top node of the whole tree.
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>();

/// Takes a tree generator function as non-type
   template parameter
/// and generates the lambda associated to it.
template <tree_generator auto Fun> constexpr auto
   codegen() {
  return codegen_aux<tree_as_array<Fun>, 0>();
}
```

To complete the implementation, we must implement a code generation function that accepts a serialized tree as an input as shown in listing 5.7. Note that this function is almost identical to the one shown in listing 5.4. 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 **??**. 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.

5.3.2 . 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 5.1: RPN formula reading example

Parsing algorithms may output serialized data. In this case, the serialization step described in 5.3.1 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 **??** where we will be using a Shunting Yard parser [13] to parse math formulae to a Reverse Polish Notation (RPN), which is its postfix notation.

Once converted into its postfix notation, a formula can be read using the following method:

- read symbols in order,
- put constants and variables on the top of a stack,
- when a function f or operator of arity N is being read, unstack N values and stack the result of f applied to the N operands.

Figure 5.1 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 5.8: 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 5.8, 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 5.9: Codegen implementation for RPN formulae

```
/// Codegen implementation.
/// Reads tokens one by one, updates the stack
   consequently
/// by consuming and/or stacking operands.
/// Operands are functions that evaluate parts of the
   subtree.
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))
           ;
  }
}

/// Stores static array representations of RPN
   generators' results.
template <auto Fun>
static constexpr auto rpn_as_array = eval_as_array<Fun
   >();

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

In listing 5.9, 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.

Symbols are read in order, and the stack is updated depending on the symbol:

- When a terminal is read, a function that evaluates its value is stacked. In this case terminals are simply constants, but they can be anything a C++ lambda can return such as a variable or another function's result.
- When a function or operator of arity N is read, N operands will be consumed from the top of the stack and a new operand will be stacked. The new operand evaluates the consumed operands passed to the function corresponding to the symbol being read.

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

5.3.3 . Preliminary observations

The presented code snippets already allow us to draw a few observations.

- Code generation from constexpr dynamic structures is not a trivial process. It still requires advanced C++ knowledge to understand the limits of what is or isn't possible to do with NTTPs and constexpr functions.

  While the examples show working examples for code generation from constexpr function results with constexpr allocated memory, they do not show the time it takes to comply with C++ constexpr restrictions.

  For example, condition results in `if constexpr` statements must be stored in a constexpr variable if the result is evaluated from an temporary object that contains constexpr allocated memory.

- The easiest way to get around NTTP constraints, *i.e.* the pass-by-generator (PBG) technique, is very costly in terms of compilation times.

  Section **??** covers that scaling issue in more detail. The overall assesment is that this technique can be used for small metaprograms, but it fails to scale properly as larger ones are being considered due to its quadratic compilation time complexity.

  In our Brainfuck metacompilation examples, we were able to trigger Clang's timeout using this technique when compiling a Mandelbrot visualizer whereas the so-called "flat" backend was able to generate the program in less than a minute.

- Compiler support for constexpr programming and constant evaluation in general is still very inconsistent across compilers. GCC still has issues with `if constexpr` where templates are instantiated even when they are in a discarded statement.

- Library support is also limited. Most of the containers from the C++ standard library are not usable in constexpr functions simply because the C++ standard lacks constexpr qualfications for their methods (and constexpr qualification is not implicit).

  As of today, the only exceptions are the main containers such as `std::vector`, `std::string` (since C++20), or `std::unique_ptr` (since C++23).

  The C'est [35] library however provides more standard-like containers that are usable in constexpr functions in C++20, which is enough to make the previous examples run on Clang in C++20.

So far we can conclude that while being doable, code generation from constexpr function results with dynamic memory is still not accessible to all C++ programmers.

An effort on the language itself would be needed to allow easier data passing from constexpr allocated memory to NTTP.

Alternatively, the language could allow code generation

5.4 . 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 5.3.1.

5.4.1 . 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 5.10: 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 5.10. 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 iteretor 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 condensed: 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.

5.4.2 . A variety of techniques to generate code from a dynamic AST

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 5.2.1, 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.

Pass-by-generator + ET

The first technique I implemented was passing AST nodes through lambdas to convert them into expression templates.

actually very hard to implement because `std::unique_pointer` lifetime management gets more difficult when combined with constexpr constraints.

Serializing the AST into a literal value

The last technique I will discuss, which is by far the most efficient in regard to compilation time, consists in transforming the AST into a literal value that can be used as a NTTP.

As I've discussed in 5.2.1, a constexpr `std::vector` return value can be transformed into a static array with a trivial transformation. As long as a static array holds literal values, the array itself remains a literal value as well, and can therefore be used as a NTTP.

In order to do this, an intermediate serialized representation of the AST which must only contain literal values must be implemented. It is a fairly trivial task:

- Polymorphism through the inheritance of `node_interface_t` can be replaced with the use of `std::variant`, which is literal as long as the value it holds is literal as well.

- Pointers contained in AST nodes can be replaced by integer indexes pointing to internal values of the array (static or dynamic) in which serialized AST nodes are stored.

Listing 5.11: PBG intermediate representation type definitions

```cpp
/// 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>;
```

In listing 5.11, I show how this was implemented in the backend that will be used for the benchmarks. Note that the serialized AST can be stored either as a `flat_ast_t` for convenience in constant evaluations, which can then be transformed into a `fixed_flat_ast_t<N>` to be passed as a NTTP.

To serialize the AST, we will rely on a function called `block_gen` which visits the AST recursively and generates serializes the AST progressively.

Listing 5.12: `block_gen_state_t` definition

```
/// Support structure for generate_blocks function
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 5.12 we begin by defining a structure called `block_gen_state_t` that contains the intermediate result.

Listing 5.13: 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 5.13 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. The generic version relies on the `visit` function, and for that reason overloads for derivatives of `node_interface_t` must be declared first. To make it simpler, I chose to define them as well to avoid unnecessary forward declarations.

Listing 5.14: `block_gen` implementation

```
/// block_gen for a single AST token. Basically just a
/// push_back.
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 5.14 shows the implementation of `block_gen` overloads for all types derived from `node_interface_t`.

The most important one is the implementation for `ast_block_t`. This is the overload that generates new blocks in the `blocks` vector of the `block_gen_state_t` structure.

It works by simply allocating a new block in the `block_gen_state_t` structure, and traverse each node by calling `block_gen` recursively. This ensures the elements contained in AST block nodes remain contiguous which is instrumental to make code generation easier later on.

The elements contained in these blocks have their indexes pointing to the elements of the `blocks` vector of the `block_gen_state_t` structure, but the expected result is a single `std::vector` containing all AST nodes.

The `flatten` function is responsible for transforming the contents of `block_gen_state_t` into such a representation.

Listing 5.15: `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());
```

```cpp
  // 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;
}
```

Listing 5.15 shows the implementation of `flatten`, which simply 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. Note that derializing the blocks into an infix representation would essentially get us back to the unparsed representation of Brainfuck programs.

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 5.16: `parse_to_fixed_flat_ast` implementation

```cpp
/// 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;
```

```cpp
  std::ranges::copy(
      flatten(parser::parse_ast(ProgramString)),
      arr.begin());

  return arr;
}
```

Listing 5.16 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 TMP begins.

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 5.17: `program_state_t` definition

```cpp
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 5.17.

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 5.18: Overloaded `codegen` entry point function

```cpp
// 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 5.18 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 though function overloading.

Listing 5.19: `codegen` specialization for `flat_token_t` elements

```
/// Code generation implementation
/// for a single instruction
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 5.19 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 a series of specializations.

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 5.20: `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 5.20. 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 5.21: `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 5.21 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 show how code generation from a NTTP can be implemented for a serialized AST.

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

Listing 5.22: `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...
  }
}
```

Listing 5.22 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.

Difficulties

- Embedding text from the original string
- `if constexpr` requiring the definition of a constexpr variable

Implementation complexity

Small synthetic variable-size 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.

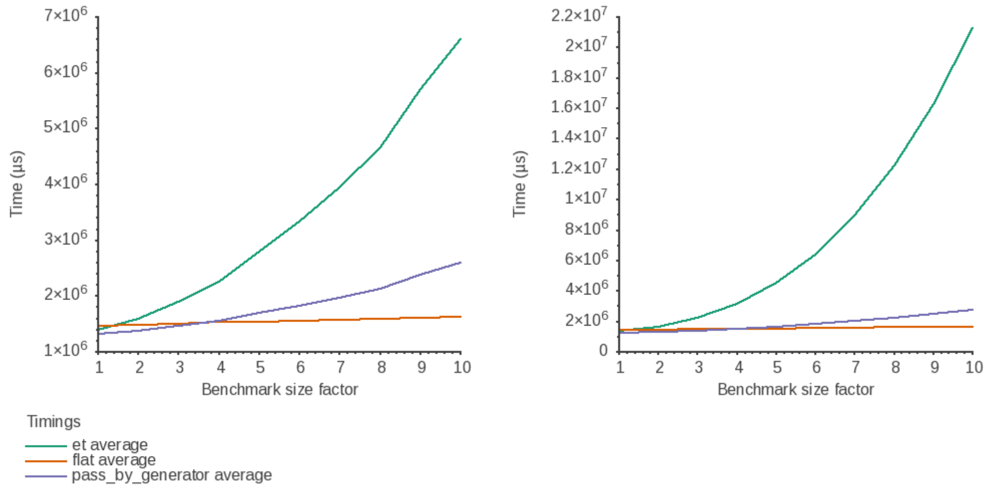

Figure 5.2: Compiler execution time measurements for consecutive loops (left) and nested loops (right)

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 so comparisons can be made properly.

Figure 5.2 both highlight 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 PBG backend. However the PBG 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).

Backend	Hello World	Hello World x2	Mandelbrot
Flat (Monolithic)	0.63	0.80	18.16
Flat (Overloaded)	0.66	0.90	28.51
PBG	3.55	12.73	Failure (timeout)
ET	19.18	74.51	Failure (timeout)

Figure 5.3: Brainfuck compile time measurements in seconds

- A Mandelbrot set fractal viewer (11672 tokens).

The measurements in figure 5.3 help us better assess how various meta-compiling 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` [39] could dramatically increase the size of embedded programs as whole source files could be used directly as embedded expressions or programs in C++.

Conclusions

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.

- **PBG backend**: The PBG 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 PBG may be a decent route for rapid prototyping to compile simple embedded expressions or programs, its poor performance scaling might be problematic.

  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.

- **ET backend**: This backend was originally meant to be a demonstrator for the interoperability of constexpr memory and ETs. It shows no particular advantage compared to the two other backends: it is slower than the PBG backend while requiring additional effort to implement the ET 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 to avoid compilation times increasing dramatically.

5.5 . Math parsing and high performance code generation

Now that we have a good overview of what to

5.5.1 . The Shunting-Yard algorithm

As seen in 5.3.2, generating code from RPN formulas is a rather easy task, and converting formulas from an infix notation to a postfix notation can be done using the Shunting Yard algorithm [13].

As parsing algorithms and constexpr dynamic data representations were already covered in 5.4, the implementation of the Shunting Yard algorithm will not be covered in detail here. A thoroughly commented constexpr implementation is available in appendix .1.3. 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.

Once again, code generation from postfix notation formulas was already covered in 5.3.2, so we will skip straight to the use of Blaze to generate high performance code from constexpr formulas.

5.5.2 . 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 (+, -, *, /, and^), as well as `{}` and `()` as parenthesis pairs.

Listing 5.23: 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 .1.3, we can specify variable identifiers, function identifiers, infix operators (with precendence), and parenthesis as shown in listing 5.23 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 5.24: 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 5.24. 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 5.25: 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 5.25 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 5.26: 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 5.26. The `static_assert` expression will not compile if the generated type is not the same as `expected_type`.

5.5.3 . 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 5.25 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 5.27: Benchmark case: Blaze with TML parsing

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

```cpp
// 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 5.27 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 5.28: Benchmark case: Blaze without TML parsing

```cpp
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 5.28 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.

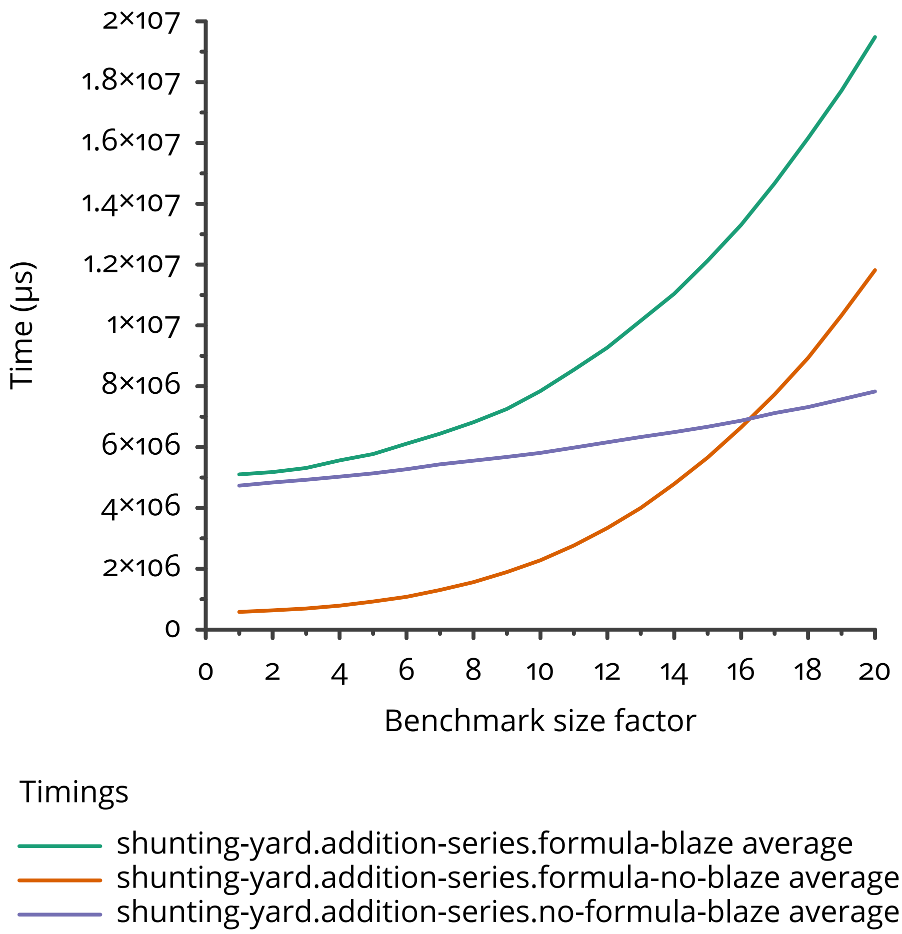

Figure 5.4: Code generation parsing benchmarks

Looking the results of the benchmarks shown in figure 5.4, 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 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 TMP 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 5.4.2. A hypothesis for this higher complexity is the cost induced by the tuple operations during the code generation step.

Conclusion

5.6 .Discussion

We wanted to demonstrate that using constexpr code to implement parsers for DSEL of arbitrary syntax in C++ 23 is possible despite limitations on constexpr memory allocation, and that doing so is possible with reasonable impact on compilation times.

We achieved that by implementing a constexpr parser for the Brainfuck language, with code generation backends implementing three different strategies to transform constexpr program representations into code using function generators, ETs, and non-type template parameters. We also demonstrated the interoperability of these constexpr parsers by implementing a parser for mathematical languages that can be used as a frontend for existing high performance C++ computation libraries.

Our benchmarks highlight compilation time scaling issues with pass-by-generator and ET code generation strategies for large programs, and excellent scaling capabilities for non-type template parameter based code generation strategies. These results can be used to decide which strategy to adopt for the implementation of future DSEL based on constexpr parsers based on considerations for compilation times or implementation complexity.

Going forward, constexpr parser generators could help reduce DSEL implementation time and help embed more languages into C++ 23. Further research has to be made to determine the impact of such generators on DSEL implementation complexity and compilation times.

Glossary

AST Abstract Syntax Tree. 54, 55, 68–71, 73–75, 77, 78, 80, 81

constant evaluation The evaluation of an expression that is performed at compile time.. 15, 71

constexpr A value or function that can be used in a constant expression.. 15, 51–55, 57, 58, 60, 62, 67–70, 75, 77–80, 82–84, 87

CTPG Compile Time Parser Generator. 16

CTRE Compile Time Regular Expression. 16, 55

DSEL Domain Specific Embedded Language. 9, 35, 36, 51, 54, 55, 81, 83, 87

DSL Domain Specific Language. 51

ET Expression Template. 35, 51, 81, 82, 87

GPU Graphical Processing Unit. 12, 19, 35

HPC High Performance Computing. 5, 9, 11, 12, 17, 18, 25, 36

JIT Just-In-Time. 12

literal value A value that does not hold any pointer to dynamic memory.. 69, 70

NTTP Non-Type Template Parameter. 14, 15, 52–54, 57, 58, 60, 62, 67–71, 74, 77, 78, 82

PBG pass-by-generator. 67, 70, 80–82

PCRE Perl Compatible Regular Expression. 55

RPN Reverse Polish Notation. 64, 82–84

SFINAE Substitution Failure Is Not An Error. 15

SIMD Single Instruction Multiple Data stream. 22, 23, 25–27, 29–31

TML Tiny Math Language. 83–85

TMP Template Metaprogramming. 14, 15, 35, 39, 51, 54, 55, 75

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] 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.

[7] 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.

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

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

[10] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. USA: ACM Press/Addison-Wesley Publishing Co., 2000. isbn: 0201309777.

[11] 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.

[12] 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.

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

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

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

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

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

[18] 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.

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

[20] 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.

[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 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.

[23] Joel Falcou et al. “Meta-Programming Applied to Automatic SMP Parallelization of Linear Algebra Code”. In: *Proceedings of the 14th International Euro-Par Conference on Parallel Processing*. Euro-Par ’08. Las Palmas de Gran Canaria, Spain: Springer-Verlag, 2008, pp. 729–738. isbn: 9783540854500. doi: 10.1007/978-3-540-85451-7_78. url: https://doi.org/10.1007/978-3-540-85451-7_78.

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

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

[27] Aleksey Gurtovoy and David Abrahams. “The Boost C++ metaprogramming library”. In: *cit. on* (2002), p. 22.

[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: ACM, 2012, pp. 33–44. isbn: 978-1-4503-1576-0.

[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] 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.

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

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

[37] Ronan Keryell, Ruyman Reyes, and Lee Howes. "Khronos SYCL for OpenCL: A Tutorial". In: *Proceedings of the 3rd International Workshop on OpenCL*. IWOCL '15. Palo Alto, California: Association for Computing Machinery, 2015. isbn: 9781450334846. doi: 10.1145/2791321.2791345. url: https://doi.org/10.1145/2791321.2791345.

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

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

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

[42] Jules Penuchot. *poacher: C++ compile-time compiling experiments*. 2020. url: https://github.com/jpenuchot/poacher/.

[43] 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.

[44] 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.

[45] Barry Revzin. *P2670R0: Non-transient constexpr allocation*. https://wg21.link/p2670r0. Oct. 2022.

[46] 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.

[47] 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.

[48] 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.

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

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

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

[52] 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.

[53] 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.

[54] 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.

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

[56] Field G. Van Zee and Robert A. van de Geijn. "BLIS: A Framework for Rapidly Instantiating BLAS Functionality". In: *ACM Trans. Math. Softw.* 41.3 (June 2015). issn: 0098-3500. doi: 10.1145/2764454. url: https://doi.org/10.1145/2764454.

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

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

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

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

[61] 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.

[62] 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.

[63] Piotr Winter. *C++ Compile Time Parser Generator*. 2021. url: https://github.com/peter-winter/ctpg.

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

[65] 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.

6 -Appendix

.1 . Poacher

.1.1 . Brainfuck AST definition header

Listing 1: `brainfuck/include/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
```

```cpp
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/include/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
```

```cpp
    // 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 . Constexpr Shunting Yard implementation

Listing 3: shunting-yard/include/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} {}
```

```cpp
};

/// 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
```

```cpp
/// 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;
}
```

```cpp
/// 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
```

```cpp
        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
```

.1.4 . Constexpr Shunting Yard implementation

Listing 4: shunting-yard/include/tiny_math_language.hpp

```
#pragma once

#include <shunting_yard.hpp>

#include <kumi/tuple.hpp>

#include <fmt/core.h>

#include <vector>

namespace tiny_math_language {

/// Parses a given formula to Reverse Polish Notation
/// (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 our
  // token specification
  sy::parse_result_t parsing_result =
      parse_to_rpn(formula, tiny_math_language_spec);

  // Debug output, only at runtime
  if !consteval {
    fmt::print("Parsed output: ");
    for (sy::token_variant_t const &current_token :
         parsing_result) {
      fmt::print("{} ",
                 std::visit(
                     [](auto const &visited_token) {
                       return visited_token.text;
                     },
                     current_token));
    }
    fmt::print("\n");
  }

  return parsing_result;
}

/// Parses a given formula and generates
/// the corresponsing code
template <auto const &Formula>
constexpr auto codegen() {
  namespace sy = shunting_yard;

  // First transformation into an array
  constexpr static auto rpn_result_array =
      sy::eval_as_array<[]() constexpr {
        return parse(Formula);
      }>();

  /// Updates the stack depending on
  /// the token being read.
  auto process_token =
      [&]<auto const & RPNStackAsArray,
          std::size_t RPNStackIndex>(
          auto operand_stack_tuple) constexpr {
        constexpr sy::token_variant_t TokenVariant =
            RPNStackAsArray[RPNStackIndex];
        constexpr std::size_t TypeId =
```

```
    TokenVariant.index();

// Extracting token variant
// into its underlying type
constexpr auto Token =
    std::get<TypeId>(TokenVariant);

// Constant handling
if constexpr (constexpr sy::token_kind_t
                  TokenKind =
                      sy::get_kind(Token);
              TokenKind == sy::constant_v) {
  constexpr sy::constant_t Constant = Token;
  // Push the constant operand at the front
  return kumi::push_front(
      operand_stack_tuple,
      [constant = Constant.value](
          auto const &, auto const &) {
        return constant;
      });
}

// Variable handling
else if constexpr (TokenKind ==
                   sy::variable_v) {
  constexpr sy::variable_t Variable = Token;

  if constexpr (Variable.text == "x") {
    return kumi::push_front(
        operand_stack_tuple,
        [](auto const &input_x,
           auto const &) -> auto const & {
          return input_x;
        });
  } else if constexpr (Variable.text == "y") {
    return kumi::push_front(
        operand_stack_tuple,
        [](auto const &, auto const &input_y)
            -> auto const & {
          return input_y;
        });
  }
}

// Function handling
```

```cpp
else if constexpr (TokenKind ==
                     sy::function_v) {
  constexpr sy::function_t Function = Token;

  if constexpr (Function.text == "sin") {
    auto const &operand =
        kumi::get<0>(operand_stack_tuple);
    auto head =
        kumi::pop_front(operand_stack_tuple);

    return kumi::push_front(
        head, [operand](auto const &input_x,
                        auto const &input_y) {
          return sin(
              operand(input_x, input_y));
        });
  }

  else if constexpr (Function.text == "max") {
    auto const &operand_a =
        kumi::get<1>(operand_stack_tuple);
    auto const &operand_b =
        kumi::get<0>(operand_stack_tuple);
    auto head = kumi::pop_front(
        kumi::pop_front(operand_stack_tuple));

    return kumi::push_front(
        head, [operand_a, operand_b](
                  auto const &input_x,
                  auto const &input_y) {
          return max(
              operand_a(input_x, input_y),
              operand_b(input_x, input_y));
        });
  }
}

// Operator handling
else if constexpr (TokenKind ==
                     sy::operator_v) {
  constexpr sy::operator_t Operator = Token;

  // Pop left and right operands
  // from the top of the stack
  auto const &operand_l =
```

```
    kumi::get<1>(operand_stack_tuple);
auto const &operand_r =
    kumi::get<0>(operand_stack_tuple);
auto tail = kumi::pop_front(
    kumi::pop_front(operand_stack_tuple));

if constexpr (Operator.text == "+") {
  // Generate and push the plus operand
  // at the front of the stack
  return kumi::push_front(
      tail, [operand_l, operand_r](
                auto const &input_x,
                auto const &input_y) {
        return operand_l(input_x, input_y) +
               operand_r(input_x, input_y);
      });
} else if constexpr (Operator.text == "-") {
  return kumi::push_front(
      tail, [operand_l, operand_r](
                auto const &input_x,
                auto const &input_y) {
        return operand_l(input_x, input_y) -
               operand_r(input_x, input_y);
      });
} else if constexpr (Operator.text == "*") {
  return kumi::push_front(
      tail, [operand_l, operand_r](
                auto const &input_x,
                auto const &input_y) {
        return operand_l(input_x, input_y) *
               operand_r(input_x, input_y);
      });
} else if constexpr (Operator.text == "/") {
  return kumi::push_front(
      tail, [operand_l, operand_r](
                auto const &input_x,
                auto const &input_y) {
        return operand_l(input_x, input_y) /
               operand_r(input_x, input_y);
      });
} else if constexpr (Operator.text == "^") {
  return kumi::push_front(
      tail, [operand_l, operand_r](
                auto const &input_x,
                auto const &input_y) {
```

```
                  return pow(
                      operand_l(input_x, input_y),
                      operand_r(input_x, input_y));
                });
            }
          }
        };

  // Processing the RPN representation of the formula.
  // The result should be a single lambda in a tuple.
  auto [result] =
      sy::consume_tokens<rpn_result_array>(
          process_token, kumi::make_tuple());
  return result;
}

} // namespace tiny_math_language
```