universite PARIS-SACLAY

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

THESE DE DOCTORAT

NNT : 2020UPASA001

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

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 HPC

1 - 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 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 more accessible to non-specialized audiences. This goal is even more important as new computing architectures emerge over time. Developing high performance programs requires specialized implementations that can be achieved by either specializing implementations, or using libraries that provide specialized abstractions at various levels and for various domains.

2 - State of the art of metaprogramming

In this state of the art I will first give an overview of metaprogramming in historic and contemporary languages. Then I will focus on the state of the art of C++ metaprogramming, and notable high performance computing libraries as they are essential for the scope of my thesis.

2.1 . Metaprogramming styles and languages

2.1.1 . A short history of metaprogramming

2.1.2 . Metaprogramming in contemporary languages

Metaprogramming perpetuates itself in contemporary languages, with some being more widespread than others.

MetaOCaml MetaOCaml[26] 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 DLang more or less extends the metaprogramming model accidentally proposed by C++. It leverages templates and compile time function evaluation just like its predecessor, but with a much more permissive approach than its predecessor.

Rust Rust proposes metaprogramming through generics, macros and traits. Generics are similar to C++ templates, although

Braid

Terra Terra[9] implements a very explicit metaprogramming model. The language is based on LUA, and exploits the dynamic nature of the language together with LLVM 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 2.2, GPU computing libraries rely heavily on C++ metaprogramming to provide building blocks for portable high performance compute kernels.

2.2 . Metaprogramming and HPC in C++

2.2.1 . Core and application-specific 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[17], Brigand[19], and mp11[10].

Libraries for more specific uses were also implemented, such as Spirit[8] for writing parsers (not for compile time parsing), Compile Time Regular Expressions (CTRE)[14] for compiling regular expressions, and Compile Time Parser Generator(CTPG)[41] for generating LR1 parsers (also not for compile time parsing).

The benefits of metaprogrammed libraries are:

- 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.

2.2.2 . High performance computing libraries

Eigen
Blaze
NT2
EVE
HPX
Thrust, CUB, etc.

2.3 . Conclusion

- Metaprogramming isn't a new idea
- Some languages provide advanced metaprogramming capabilities
- C++ has solid metaprogramming constructs, and a complete HPC ecosystem (libraries, compilers, etc.)

2.3.1 . Language constructs of C++ metaprogramming

Metaprogramming in C++ relies on

Templates

Overloading

Parameter packs

3 - 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 paper, we propose to show how a modern C++ approach based on generative programming techniques such as vectorization and loop un- rolling 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.

3.1 . 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 [22], 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 gener- ally 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 [42]. Thus, it offers a framework to develop dense linear solvers using algorithmic skeletons [6] and an API which is more userfriendly than LAPACK, giving satisfactory performance results. A more generic approach is the one followed in recent years by C++ libraries built around expression templates [39] or other generative programming [**hpcs6**] principles. In this paper, 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 architec- tures bring further challenges for its design and optimization.

In this paper, we describe how we obtained optimized gen- erated C++ codes for the gemv routine to make it reach its peak performance on different target CPU architectures (Intel x86 and ARM in this paper). 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 paper 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.

3.2 . 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.

3.2.1 . Metaprogramming as code generation principles

Metaprogramming [15] 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 [7]. If metaprogramming is used in languages as different as C++ [1],D [**hpcs10**], OCaml [**hpcs11**] or Haskell [**hpcs12**], a subset of basic notions emerges:

- Code fragment generation: Any meta-programmable 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 paper. Such systems includes:

- SYCL [**hpcs13**]: 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 [**hpcs14**]: a framework for generating BLAS like oper- ations in ISO C99 from a small subset of kernels that can be retargeted for different back-end. Its performances are on par with open-source solutions like OpenBLAS and ATLAS.

- LGEN [**hpcs15**]: 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 paper, 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 meta-programs 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 meta-programs 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[**hpcs16**] 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 1 below demonstrates this particular feature.

Listing 3.1: Sample polymorphic lambda definition

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

Listing 1.

- 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 illustrates an example for this feature.

Listing 3.2: C++ 17 fold expressions

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

Listing 2.

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 3 gives an example about tuples.

Listing 3.3: Tuple and structured bindings

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

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

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

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

Listing 3.

3.2.2 . Application to HPC 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 [**hpcs17**] 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 95to 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 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 3.4: 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);
}
```

Listing 4.

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 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 an universal reference.

Listing 3.5: 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>{});
}
```

Listing 5.

A sample usage of the `for_constexpr` function is given in Listing 6 in a function printing every element from a `std::tuple`.

Listing 3.6: 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";
  });
}
```

Listing 6.

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 op- timizations 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.

3.3 . 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 [**hpcs18**]. 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% [**hpcs19**]. 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 [**hpcs20**].

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 paper, 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 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 3.7: 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 ...
}
```

Listing 7.

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 program- ming 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.

3.4 . 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 [**hpcs21**]. We use gcc7.2 [**hpcs22**] 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.

3.4.1 . 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 dedi- cated 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 ker- nel. 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 [**hpcs23**] 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.

3.4.2 . 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.

3.5 . Conclusion

This paper 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 paper, that is for sizes ranging from 4 to 512. Therefore through this paper 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 meta-programmed code generation process to tackle larger matrix sizes by pre- computing an optimal tiling availability of prefetch and archi- tecture, turning large-scale gemv into an optimized sequence of statically sized small-scale gemvs. Such a technique is also nat- urally 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.

4 - Code generation at expression level

In this chapter I will cover the use of template metapramming 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[40], which will be referred to as *ET*.

Combined with compile-time mechanisms such as function overloading, specialization, and operator overloading, expression templates can be used to implement expression level *DSEL*s and convert large mathematical expressions into high performance code.

There are two main libraries that are able to do just this: Eigen[16] and Blaze[20].

How do they work?

Your math expressions aren't represented by structures, they're represented by types.

You don't use conditions at runtime but overloading at compile time.

Enables a whole range of optimizations:

- Lazy evaluation
- Fused operations using BLAS
- 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

Metaprogramming beyond expression templates

5 - Establishing a methodology for compile time benchmarking

With libraries like Eigen[16], Blaze[20], or CTRE[14] being developed with a large tempalte metaprogrammed implementation, 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[12] or static reflection[37]. 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[27] which is the main C++ technical conference in France. It is being used to benchmark examples from the poacher[28] project, which was briefly presented at the Meeting C++ 2022[25] technical conference.

5.1 . Compile time benchmarking: state of the art

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

As metaprogramming became easier with C++ 11 and C++ 17, it became more 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.

A first generation of tools aimed to tackle this issue with their own specific methodologies:

- Buildbench[35] measures compiler execution times for basic A-B compile time comparisons in a web browser,

- Metabench[13] instantiates variably sized benchmarks using embedded Ruby (ERB) templating and plots compiler execution time, allowing scaling analyses of metaprograms,

- Templight[29] adds Clang template instantiation inspection capabilities with debugging and profiling tools.

5.1.1 . 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 5.1.

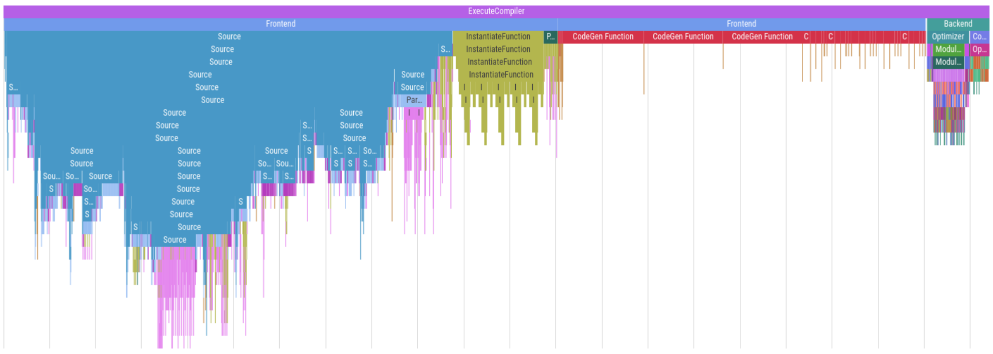

Figure 5.1: Clang time trace file in Perfetto UI

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 5.1. Duration and timestamps are expressed in microseconds.

Listing 5.1: Time trace event generated by Clang's internal profiler

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

In this example the `/args/detail` field indicates the file that's being processed 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.

5.2 . ctbench features

Originally inspired by Metabench[13], ctbench development was driven by the need for a similar tool that allows the observation of Clang's time-trace files to help get a more comprehensive view on the impact of metaprogramming techniques on compile times. A strong emphasis was put on developer friendliness, project integration, and component reusability.

ctbench provides:

- a well documented CMake API for benchmark declaration, which can be generated using the C++ pre-processor,
- a set of JSON-configurable plotters with customizable data aggregation features and boilerplate code for data handling, which can be reused as a C++ library.

In addition to ctbench's time-trace handling, it has a compatibility mode for compilers that do not support it like GCC. This mode works by measuring compiler execution time just like Metabench[13] and generating a time-trace file that contains compiler execution time. Moreover, the tooling allows setting different compilers per target within a same CMake build, allowing black-box compiler performance comparisons between GCC and Clang for example or comparisons between different versions of a compiler.

All these features make ctbench a very complete toolkit for compile time benchmarking, making comprehensive benchmark quick and easy, and the only compile time benchmarking tool that can use Clang profiling data for metaprogram scaling analysis.

5.2.1 . CMake API for benchmark declaration

The decision of using CMake as a scripting language for ctbench was rather straightforward as it is the most widely used build system generator for C++ projects.

In this subsection I'll explain the CMake API. It provides functions for generating benchmark targets at several sizes, and with a specific directory layout that can be read by the grapher tools. It also provides functions for generating graph targets to compare benchmark cases.

Listing 5.2: ctbench CMake API use example

```
# Fine ctbench package
find_package(ctbench REQUIRED)

# Set benchmark parameters:
# range and number of samples

# From 1 to 10 with a step of 1
set(
  BENCH_RANGE 1 10 1
  CACHE STRING "Bench range"
)

# 10 samples per benchmark iteration
set(
  BENCH_SAMPLES 10
  CACHE STRING "Samples per iteration"
)

# Add time trace options to enable Clang's
# profiler output, and set the granularity
# to the tiniest value for accurate data
add_compile_options(
  -ftime-trace -ftime-trace-granularity=1
)

# Declare benchmark targets
# with previously defined parameters

ctbench_add_benchmark_for_range(
  hello_world-flat
  hello_world/flat.cpp
  BENCH_RANGE ${BENCH_SAMPLES}
)

ctbench_add_benchmark_for_range(
  hello_world-pass_by_generator
  hello_world/pass_by_generator.cpp
  BENCH_RANGE ${BENCH_SAMPLES}
)

# Declare graph target
ctbench_add_graph(
  bfbench-consecutive_loops
  graph-config.json
  hello_world-flat
  hello_world-pass_by_generator
)
```

Listing 5.2 shows how to use the CMake API to generate a graph comparing two fictional benchmark cases. Each benchmark case is instantiated at sizes going from 1 to 10 with a step of 1, with 10 samples for each size. These are defined in the `BENCH_RANGE` and `BENCH_SAMPLES` variables, respectively.

The `add_compile_options` function is a CMake primitive that allows setting compile options for the target declarations that follow it. We are using it to enable Clang's profiler, and set the granularity to its minimum (*i.e.* 1 microsecond) for very fine data gathering.

Then we use the `ctbench_add_benchmark_for_range` to declare two benchmarks targets named `hello_world-flat` and `hello_world-pass_by_generator` . The function takes a target name, a C++ source file, a range variable name, and a sample number as parameters. A similar function called `ctbench_add_benchmark_for_size_lis` allows defining benchmark targets using a list of sizes instead of an incremental range.

Both commands will declare CMake targets that generate the time trace files. The files will be located at the following path: `<benchmark target name >/<iteration size>/<iteration number>.json`. This Moreover, the compiler is called through a compiler launcher to work around CMake's lack for access to compile commands.

At this point we can declare a graph target using `ctbench_add_graph`. It takes a target name and a path to a JSON configuration file followed by an arbitrary number of benchmark targets as arguments.

5.2.2 . JSON API for plotter configuration

JSON configuration files are used to pass parameters such as the plot generator name, some generic plotting parameters such as an export format list or axis names, and generator specific parameters like predicates for filtering or group key specifiers for aggregating time trace events.

Listing 5.3: ctbench configuration file example

```
{
  "plotter": "compare_by",
  "legend_title": "Timings",
  "x_label": "Benchmark size factor",
  "y_label": "Time (microsecond)",
  "draw_average": true,
  "average_error_bars": false,
  "draw_median": false,
  "demangle": false,
  "draw_points": false,
  "width": 1000,
  "height": 400,
  "key_ptrs": ["/name", "/args/detail"],
  "value_ptr": "/dur",
  "plot_file_extensions": [".svg", ".png"],
  "filters": [
    {
      "type": "match",
      "regex_match": false,
      "matcher":
      {
        "name": "ExecuteCompiler"
      }
    }
  ]
}
```

Listing 5.3 shows a configuration file for the `compare_by` plotter. The `key_ptrs` contains a list of JSON pointers pointing to the time trace event properties that constitute the keys to aggregate them. Therefore events will be aggregated by the contents of the fields at the `/name` and `/args/detail` fields. Only the events that match the predicates in the `filters` field will be considered, therefore only the `ExecuteCompiler` event will be observed in that case.

This configuration is a good start for compiler execution benchmarking. The `filters` field can be removed to process all time trace events, but there can be a lot of events making graph generation very long.

Other options are worth mentioning:

- Graph customization options such as the width and height of the output file, legend title, or axis labels.
- Plot file extensions, allowing several export formats to be used at once.
- Allowing median and average to be drawn, as well as single data points, or even average error bars.

- C++ symbol demangling using LLVM's demangler.

Time trace event filtering is done through a predicate engine that will be further explained in subsection 5.2.4.

5.2.3 . ctbench-ttw: ctbench's compiler launcher

CMake abstracts most of the compilation process and makes it impossible to retrieve derivative files in a reliable way. The object files themselves are compiled into temporary directories before being relocated to their final locations.

Because time trace result files are located in the same directory as the object files and are not moved to the output build directory by CMake, a compiler launcher named `ctbench-ttw` was added to the project. It can be used to compile benchmark targets by setting the `CXX_COMPILER_LAUNCHER` CMake target property.

The launcher catches the compilation flags to then locate and copy time trace files generated by Clang which are located in the same directory as the object file. The name of a time trace file is the same as its binary with its extension replaced by `.json`. An object named `main.o` would have its time trace file generated as `main.json` in the same directory.

To use the compiler launcher, one simply adds it to the beginning of a command along with the desired destination for the time trace file. For example, when compiling a C++ file using the command `clang++ main.cpp -o main.o -ftime-trace`, you can add `ctbench-ttw /tmp/dest.json` as a prefix: `ctbench-ttw /tmp/dest.json clang++ main.cpp -o main.o -ftime-trace`. Doing so will launch the compiler, locate `main.json` and copy it to `/tmp/dest.json`.

`ctbench-ttw` was improved over time to can provide other features that couldn't be implemented using pure CMake, such as compiler execution time measurement and allowing the use of different compilers for targets inside a single CMake project. These two additional features make compiler performance comparison possible for variable sized compile time benchmarks.

5.2.4 . grapher

The grapher component of ctbench was developed as a tookit that can be used through executable tools, and as a C++ library as well.

It contains all the code needed to aggregate Clang's time-trace data generated by ctbench benchmark targets, read JSON configuration files for plotters, and generate plots with different plot generators.

Library for handling benchmark directory structure

The most important part of ctbench's grapher library is its core types. They define a hierarchy of elements:

- Benchmark set, which constitutes a set of different benchmark cases to compare.
- Benchmark case, which represents a benchmark case that will be instantiated at several sizes.
- Benchmark instance, a benchmark case instantiated at a given size.
- Repetition, a single repetition, run, or sample of a benchmark instance.

The type definitions are shown in listing 5.2.4.

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

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

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

The grapher library also provides type aliases for containers or even basic values to enforce the use of coherent types across the whole codebase. The most important are `json_t` and `value_t`, which represent JSON objects and scalar values extracted from JSON objects respectively (see listing 5.2.4).

```
using json_t = nlohmann::basic_json<boost::container::
    flat_map>;
using value_t = unsigned long;
```

Boost's `flat_map` container is used instead of `std::map` for JSON objects instead as overall performance was measurably better with it.

Most importantly, the grapher library provides a function to extract benchmark data from a folder structure as laid out by the benchmark declaration functions described in 5.2.1.

Homogeneous plotter engines and CLI boilerplate

Besides boilerplate code, grapher also provides an interface for the addition of new plotters. This and command line interface boilerplate code make the addition and implementation of new visualization strategies rather straightforward.

The plotter interface The plotter interface

Listing 5.4: `plotter_base_t` type definition

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

  virtual grapher::json_t get_default_config() const =
      0;
};

using plotter_t = std::unique_ptr<plotter_base_t>;
```

Command line interface The grapher project provides a command line interface for interaction with users through shell commands, and with the CMake API as well. It provides two executables that serve as drivers for grapher plotters:

- `ctbench-grapher-plot`: Main program that generates plots. It takes a configuration file and an output folder as options followed by several benchmark result folders. It is essentially a command line interface over plotters. It builds a benchmark set, and passes it to a plotter along with a JSON config object and a destination folder.

- `ctbench-grapher-utils`: Auxiliary program currently used to obtain default JSON configurations for every plotter. Its functionality may be extended in the future.

The command line interface is implemented using LLVM's CommandLine 2.0 library. Both programs can be used without ctbench's CMake API for reuse in any other scripting environment as CMake may not always be the best way to generate C++ compile time benchmark cases.

For example, Metabench[13] uses Embedded Ruby as a source code generator to instantiate benchmarks at arbitrary sizes. Benchmarks could very well be generated using other methods such as Python scripting, or even compiling C++ sources that have already been pre-processed. Note that a staged pre-processing method could be implemented using the compiler launcher described in 5.2.3.

Time trace event predicate engine

grapher provides a powerful predicate engine for event filtering. The predicate engine is an interpreter for a JSON-based predicate language for integration in JSON configuration files, as we seen in 5.2.2.

The language implements a wide variety of predicates:

- `regex` and `streq`: true if the value at the given pointer matches the given regex or string.
- `match`: true if a time trace event matches a given JSON object, ie. if the event has all the values present in the given JSON object. Regex matching can also be enabled for value matching.
- `op_or` and `op_and`: binary operators to combine other predicates.
- `val_true` and `val_false`: arbitrary true/false values, mostly used for debugging and testing.

These are powerful enough to express most conditions on time trace events. They can be used to target very specific

Other miscellaneous stuff

5.3 . 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:

```
/// 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:

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

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

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

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:

```
// 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:

```
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:

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

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

```
{
  "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[34] was used to turn off CPU frequency scaling to improve measurement precision.

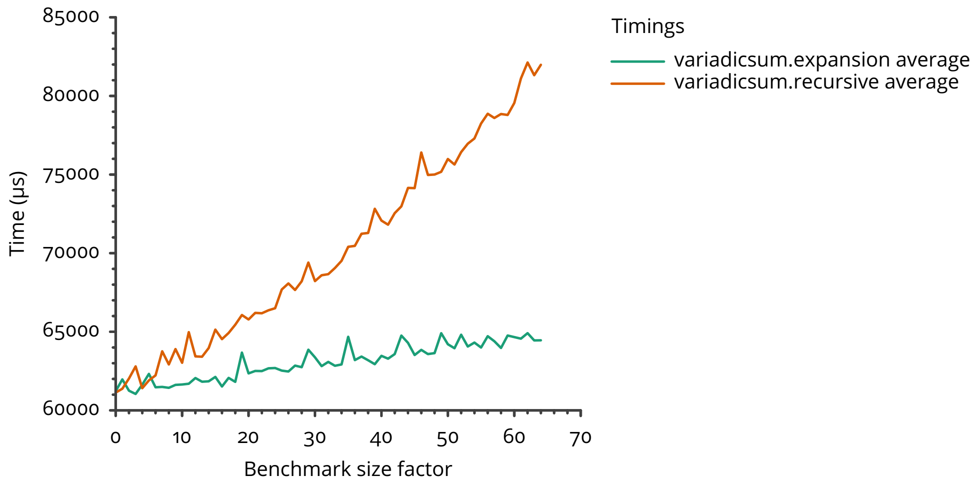

Figure 5.2: ExecuteCompiler

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.

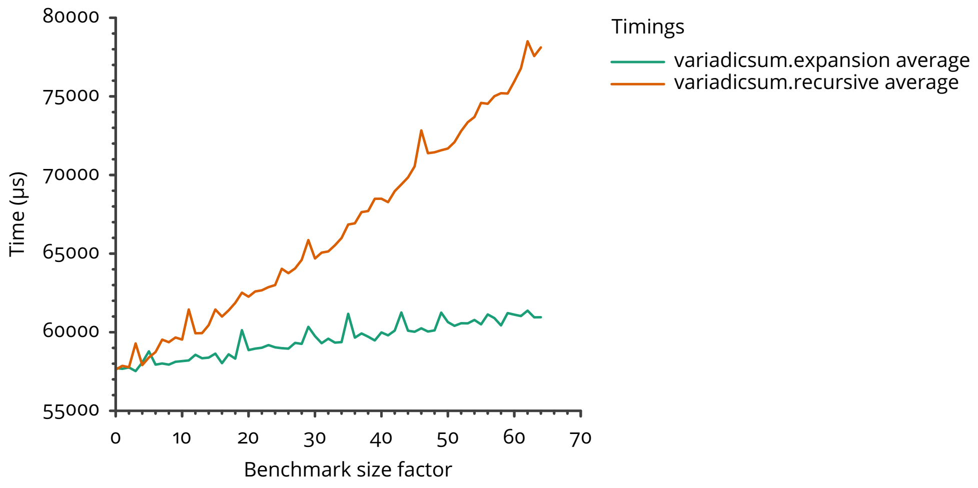

Figure 5.3: Total Frontend

The backend is not being impacted here, supposedly because this is purely a compile-time program, and the output program is empty. However this

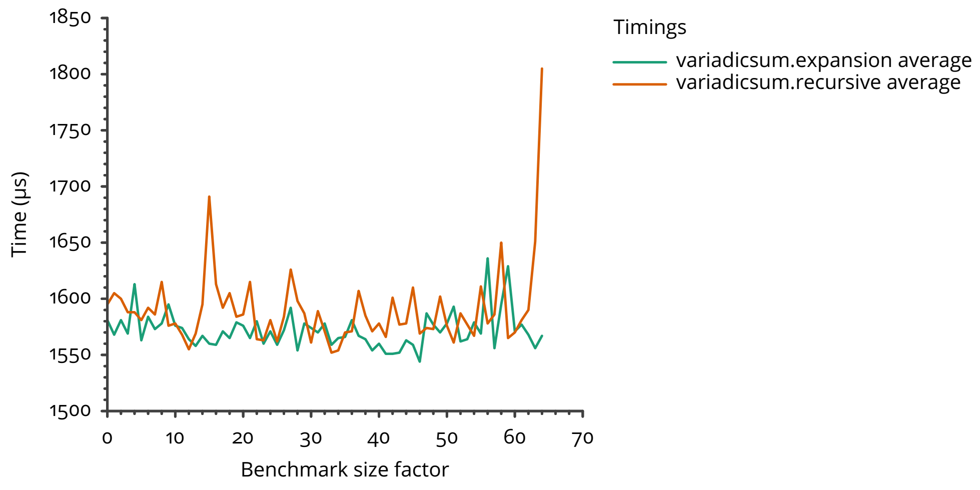

Figure 5.4: Total Backend

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.).

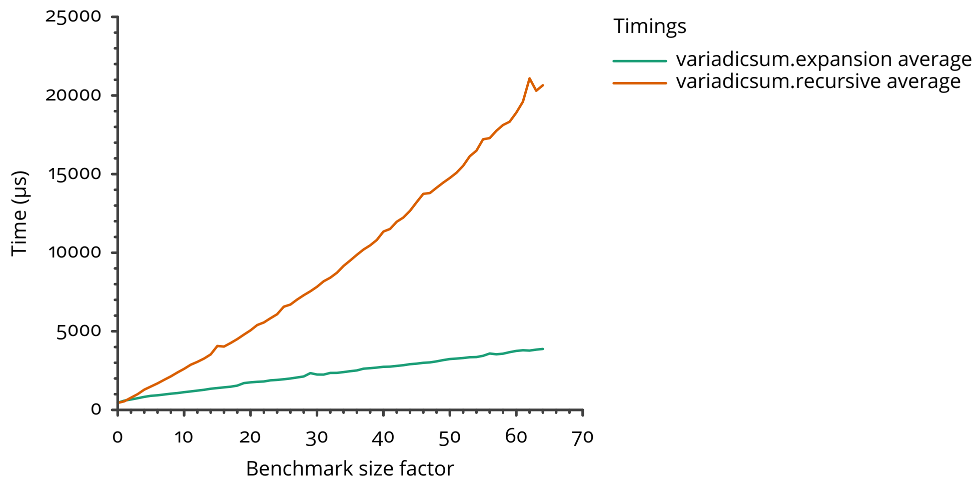

Figure 5.5: Total InstantiateFunction

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 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 in-

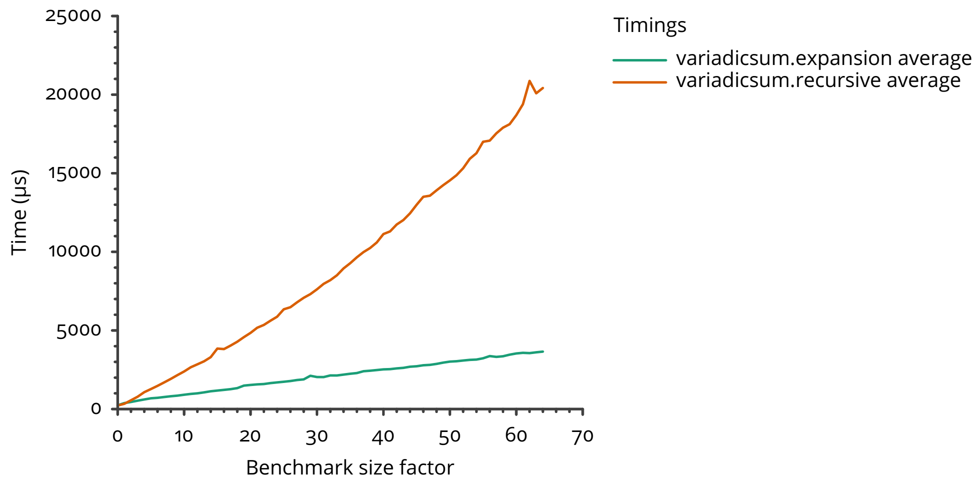

Figure 5.6: InstantiateFunction foovoid

cludes 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.

5.4 .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

5.5 .Acknowledgements

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

6 - Constexpr parsing for high performance computing

6.1 .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*. Template metaprogramming is such a technique and it spawned a large corpus of related idioms from compile time function evaluation to lazy evaluation via *Expression Templates*.

In the field of High Performance Computing, C++ users are often driven to use libraries built on top of those idioms like Eigen[16] or Blaze[20, 21]. 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 paper, we try to demonstrate that the new features of C++ 23 related to compile time programming are able to help developers designing *DSEL*s with arbitrary syntax by leveraging `constexpr` computations, compile time dynamic objects and lazy evaluation through lambda functions. After contextualizing our contribution in the general *DSEL*s domain, this paper will explain the core new techniques enabled by C++ 23 and how we can apply to build two different *DSEL*s with their own non-C++ syntax. We'll also explore the performances of said *DSEL*s in term of compile time to assess their usability in realistic code.

6.2 . A technical background of C++ metaprogramming and 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. Domain Specific Embedded Languages (*DSEL*s) are a subclass of *DSL* that rely on an existing general-purpose language to host it. *DSEL*s 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 **Template Metaprogramming** to ensure performance and correctness.

6.2.1 . Template Metaprogramming

C++ template metaprogramming [2] is a technique based on the use of the template type system of C++ to perform arbitrary computation at compile time. This property of C++ templates is due to the fact they define a Turing-complete sub-language manipulating types and integral constants at compile time [36]. Due to the fact that template code generation is performed at compile time, uses constants and supports pattern-matching and recursion thanks to template partial specialization, it can also be looked at as a pure functional language [18].

Templates are an interesting technique for generative programming. As they are Turing-complete, one can design a set of template metaprograms acting as a *DSL* compiler run 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. Through this technique, compile time constants, data structures and complete functions can be manipulated. The execution of metaprograms by the compiler enables the library to implement domain-specific optimizations that lead to a complete domain oriented code generation. Such a technique can be hosted by several languages featuring metaprogramming features (incidental or by design) like D [5], Haskell [32] and OCaml [31].

Listing 6.1: Example of compile time computation using C++ templates

```cpp
// Template type accepting an integer
// as a non-type template parameter
template <unsigned N> struct fibonacci_t;

// General definition
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;
```

Listing 6.1 shows basic principles of C++ template metaprogramming. The `fibonacci_t` type template accepts a *Non-Type Template Parameter* (*NTTP*, *i.e.* a template parameter that is a regular value instead of a type) 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.

Note that the `value` field is specified as `constexpr`. It indicates that a variable or function can be used in compile time expressions. We will talk about `constexpr` in detail in 6.2.2.

6.2.2 . 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 non-type template parameters.

However not all functions and variables can be qualified as `constexpr`, and not all constant expression results can be used as *NTTP*s.

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[30] and virtual `constexpr` function calls[11] 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 evaluation.

However `constexpr` allocated memory is not transient, *i.e.* memory allocated in constant expressions cannot be stored in `constexpr` variables, and non-type template parameters 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 6.2: Illustration of constraints on `constexpr` allocated memory

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

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

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

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

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

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

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 6.3: Transfer of data from a dynamic `std::vector` to a constexpr static `std::array`

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

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

Listing 6.3 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 [11]. This allows heritage-based polymorphism in `constexpr` programming when used with `constexpr` allocation of polymorphic types.

6.2.3 . C++ Domain Specific Embedded Languages

*DSEL*s in C++ use template metaprogramming via the *Expression Template* idiom. **Expression Templates** [40, 38] is a technique implementing a form of **delayed evaluation** in C++ [33]. Expression Templates are built around the *recursive type composition* idiom [23] that allows the construction, at compile time, of a type representing the abstract syntax tree 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 Abstract Syntax Tree (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 *DSEL*s. One major example is the Compile-Time Regular Expressions library (CTRE) [14] that implements most of the Perl Compatible Regular Expression (PCRE) syntax. However, CTRE still relies on type-based template metaprogramming to parse regular expressions and transform them into regular expression evaluators.

6.3 . 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 paper 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.

6.3.1 . 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 6.4: `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 6.4 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 6.2.2 make it impossible to use the result of `gen_tree` directly as a template parameter to generate code.

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

- passing functions that return nodes as non-type template parameters,
- 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 non-type template parameter,

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 6.2 that depsite `generate`'s return value being non-literal, the function itself can be passed as a non-type template argument.

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 6.5: 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 6.5, 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 non-type template parameter.

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 non-type template parameters.

```
/// 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 6.3.1.

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 6.6: `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 6.6 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 6.7: 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 6.7 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 non-type template parameters 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 non-type template parameters.

Listing 6.8: 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 6.8. Note that this function is almost identical to the one shown in listing 6.5. 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.

6.3.2 . Using algorithms with serialized outputs

Parsing algorithms may output serialized data. In this case, the serialization step described in 6.3.1 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[**shunting-yard**] to parse math formulae to a Reverse Polish notation (RPN), which is its postfix

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 6.1: RPN formula reading example

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 6.1 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 6.9: 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 6.9, 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 6.10: 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 6.10, 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.

6.3.3 . 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 *NTTP*s 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 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[24] 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 non-type template parameters.

Alternatively, the language could allow code generation

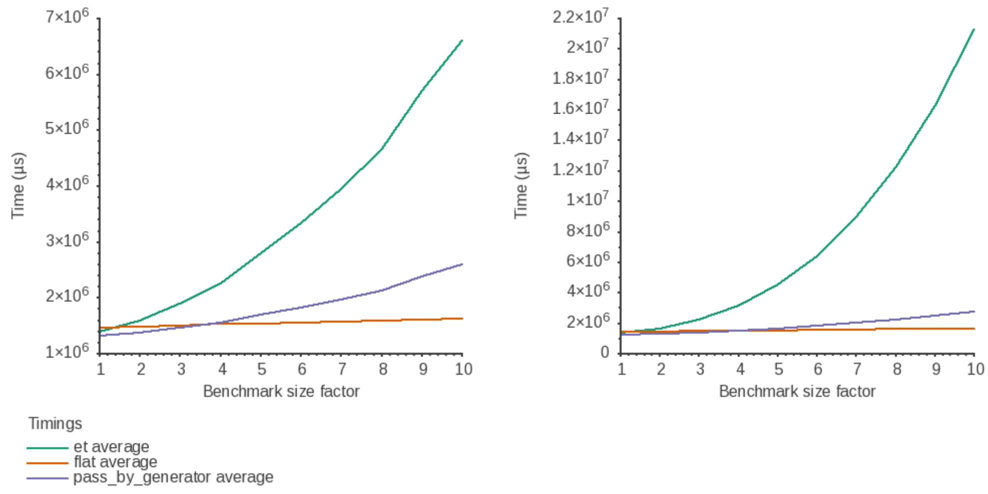

Figure 6.2: Compiler execution time measurements for consecutive loops (left) and nested loops (right)

6.4 . Brainfuck parsing and code generation

6.4.1 . Constexpr Brainfuck parser and AST

6.4.2 . Assessing the usability of different code generation techniques for high performance code development

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.

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 6.2 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 pass-by-generator backend. However the pass-by-generator backend still has a compile time overhead much higher than the flat backend, which shows near constant compiler execution times

Backend	Hello World	Hello World x2	Mandelbrot
Flat	1.81	2.25	49.68
Pass-by-generator	9.77	34.37	Failure (timeout)
Expression template	50.60	192.73	Failure (timeout)

Figure 6.3: BF compile time measurements in seconds

on these small scale benchmarks.

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

Large Brainfuck programs

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

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

The measurements in figure 6.3 help us better understand how various metacompiling techniques behave at scale. The "Flat" backend shows very good performance on all examples, including the Mandelbrot example that is about 100 times larger than the Hello World example. However the other cases highlight severe scaling issues and tend to confirm our previous hypothesis being that using generator functions to pass values makes the code generation quadratic. Finally, the "Expression template" backend performance highlights heavy performance impact when expression templates are being used, which is likely due to the complexity of the mechanisms expression templates involve like SFINAE and overload resolution.

Conclusions

6.5 . Math parsing and high performance code generation

6.5.1 . The Shunting-Yard algorithm

6.5.2 . Code generation from a postfix math notation

6.6 .Conclusion

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, expression templates, 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 expression template 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.

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] Walter Bright. *Templates Revisited*. 2014. url: http://dlang.org/templates-revisited.html.

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

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

[8] Joel de Guzman and Hartmut Kaiser. *Boost.org spirit module*. 2001. url: https://github.com/boostorg/spirit.

[9] 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.

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

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

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

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

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

[15] 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.

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

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

[18] 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.

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

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

[21] 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. doi: 10.1109/hpcsim.2012.6266939.

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

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

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

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

[26] 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.

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

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

[29] 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.

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

[31] Jocelyn Serot and Joël Falcou. “Functional meta-programming for parallel skeletons”. In: *Computational Science–ICCS 2008*. Springer Berlin Heidelberg, 2008, pp. 154–163.

[32] Tim Sheard and Simon Peyton Jones. “Template meta-programming for Haskell”. In: *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*. ACM. 2002, pp. 1–16.

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

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

[35] Fred Tingaud. *Build-Bench*. 2017. url: https://build-bench.com/.

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

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

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

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

[40] 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.

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

[42] 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`.

7 -Appendix

.1 . Poacher

.1.1 . BR AST implementation

```
//-----------------------------------------------------
// 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;
};

/// Helper class
template <typename Child> struct make_visitable_t {
  template <typename F>
  constexpr auto visit(F &&f) const {
    return f(*static_cast<Child const *>(this));
  }
};

// Helpers

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

```cpp
/// 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,
                     make_visitable_t<ast_token_t> {
  token_t token_;

public:
  constexpr ast_token_t(token_t t)
      : node_interface_t(ast_token_v), token_(t) {}

  /// Returns the token's value.
  constexpr token_t get_token() const {
    return token_;
  }
};

/// AST node type for Brainfuck code blocks
struct ast_block_t : node_interface_t,
                     make_visitable_t<ast_block_t> {
public:
  using node_ptr_t = ast_node_ptr_t;

private:
  ast_node_vec_t content_;

public:
  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;

/// Returns a const reference to its content.
constexpr ast_node_vec_t const &
get_content() const {
  return content_;
}
```

.1.2 . BF parser implementation

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

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

  ast_node_vec_t block_content;

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

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

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

      parse_begin = while_block_end;
    }

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

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

} // namespace impl
```