xbc: An extensible MLIR compiler infrastructure for programming language and heterogeneous computing research

ANONYMOUS AUTHOR(S)

This paper presents the design and development of the ***eXtensible base compiler*** (**xbc**) infrastructure, an extensible MLIR infrastructure for programming language and heterogeneous computing research. **This extensible infrastructure permits testing novel programming languages and programming model concepts, such as support for quantum targets or alternative parallel programming models**. It also provides a platform for compiler practitioners to meet the demands of a rapidly changing hardware landscape. **xbc** targets traditional X86 processors, GPUs (NVIDIA and AMD), and quantum-classical heterogeneous systems. Our results demonstrate comparable performance to vendor compilers for GPUs.

Highlighting one of our results, we observe that NAS CG **xbc**'s GPU version on NVIDIA A100 is 2.26× faster than Clang's OpenMP offload and 1.3× faster than NVIDIA's OpenACC compilers. For AMD MI250x, **xbc** is 5× faster than Clang's OpenMP offload and 5.5× faster than AMD Clang's OpenMP offload.

CCS Concepts: • **Software and its engineering** → **Compilers**; **Dynamic compilers**; **Extensible languages**; • **Computer systems organization** → *Heterogeneous (hybrid) systems.*

Additional Key Words and Phrases: MLIR, compiler infrastructure, program representation, extensible languages, heterogeneous computing, parallel programming

1 Introduction

High-Performance Computing (HPC) has seen a paradigm shift from homogeneous to heterogeneous systems in the last couple of decades. The heterogeneous system era has seen the introduction of massively parallel architectures such as GPGPUs from vendors such as NVIDIA, AMD, and Intel, vector architectures such as the A64FX from Fujitsu/ARM, and also other types of devices such as FPGAs, ASICs, neural engines, and manycore processors.

The extent of heterogeneity in supercomputing architectures is growing and evolving, leading to the introduction of processor types beyond traditional von Neumann CPUs and GPUs. Over the past few years, it has become apparent that quantum computing approaches will likely play a role in future heterogeneous supercomputing efforts [18]. Mixed quantum-classical heterogeneous computing promises to continue computational scalability for currently intractable problems in drug design [24], climate modeling [36], and cryptography [9].

These technological shifts mean plenty of hardware features to be exposed at different hardware and software stack levels, thus leading to disruption in software development - a necessary evil. This disruption leads to significant challenges when legacy algorithms designed for conventional architectures are adapted to new ones. Requiring legacy algorithms to be revisited, sometimes improved further, or even creating newer ones to take the best advantage of the features in new architectures, e.g., adapting the classical GEMM algorithm to use tensor operations in GPU architectures. Consequently, programming languages and models must advance to represent these algorithms effectively in these architectures. This nontrivial process demands significant time and effort.

All of these factors mean compiler toolchains developed today must be flexible and adaptable regarding the underlying physical models that drive computational accelerators. Recent work has acknowledged this fact, and several approaches in programming [1, 4], including quantum programming and compilation [13], are building upon extensible frameworks such as the Multi-Level Intermediate Representation (MLIR).

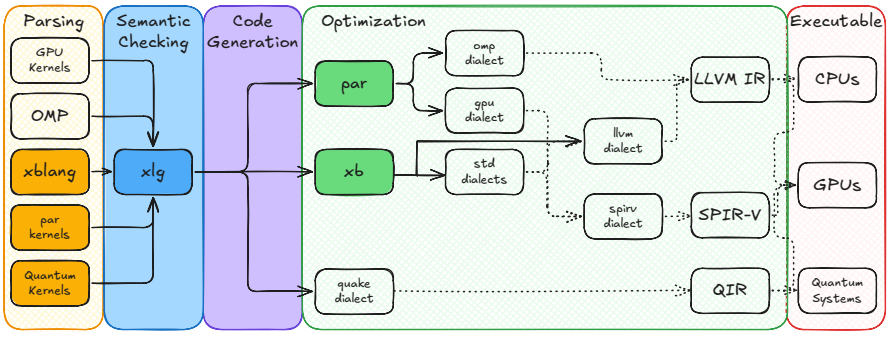

Fig. 1. Compilation workflow of the `xbc` compiler. Color-filled rectangles and solid arrow lines represent the contributions of this paper. Dashed lines indicate parts of the workflow that we did not change for this work.

Taking the dire need for a toolchain that helps expedite programming language and heterogeneous computing research and development as a motivation, we have designed and built the ***eXtensible base compiler*** (`xbc`), a high-level extensible compiler infrastructure for programming language and heterogeneous computing research that uses MLIR. Such an approach aims to help rapidly advance software to maximize the performance and efficiency of the evolving hardware.

2 *eXtensible base compiler* (`xbc`) Overview

In this section, we state the problem solved by our infrastructure, provide an overview of the paper, and state the contributions of this work. **The source code for this open-source project is available at [link removed for review].**

The `xbc` infrastructure tackles two challenges in research compilers. First, it **establishes an infrastructure for testing novel programming languages and programming model ideas**, such as support for quantum targets or alternative parallel programming models. Second, it **bridges the gap between high-level programming languages and MLIR**. Figure 1 shows an overview of the compilation workflow. The workflow starts with parsing high-level source code to generate `xlg`, an extensible MLIR replacement for traditional Abstract Syntax Trees (AST) (*see section 3*). The process proceeds with semantic checking on `xlg` and then a code generation stage *(see section 4)*. After the code generation stage, the IR is optimized and transformed into upstream MLIR dialects (*see Sections 5 and 6*). Then, the compiler translates the IR into representations like LLVM IR, which can be compiled into an executable.

The main contribution of this paper is the development of the `xbc` infrastructure. We breakdown this infrastructure into the following contributions:

- `xlg`, an extensible intermediate representation for a program structure intended to represent many programming language constructs.
- The `xb` dialect, an MLIR dialect for representing programming languages suitable for optimizations and transformations for lowering into, but not limited to, the LLVM and standard MLIR Dialects.
- The `par` MLIR dialect, along with its corresponding programming model, that can target heterogeneous systems.
- `xblang` an extensible research programming language for heterogeneous computation. This language targets both CPU and GPUs, as well as quantum simulators.

3 xlg Extensible Language Generator: an MLIR replacement for ASTs

We have designed a novel high-level IR called the extensible language generator, or xlg for short, to represent the abstract syntax of a programming language. This representation is similar to the traditional AST used in many compilers but with advantages over AST, such as interoperability with existing MLIR dialects. This section discusses the xlg representation, its benefits over ASTs, and its implementation on top of MLIR.

Prior to introducing xlg, we must introduce the notions of *Concepts* and *Constructs* as they are the basis of xlg. *Concepts* are abstract properties of programming language notions, such as having a type or being a symbol. They can be combined to create more specific *Concepts* like a typed symbol. **The set of *Concepts* is open to adding new *Concepts*, allowing the set to evolve and adapt to new notions**. Finally, *Concepts* define a type system that can enforce the structural properties of a language. For example, the `builtin_type` *Concept* can operate as a `type` *Concept* but not as a `decl` *Concept*.

Constructs, conversely, are concrete programming language notions, like a variable declaration or a built-in type. They utilize *Concepts* to define their public properties; in other terms, *Concepts* provide interfaces for interacting with *Constructs* generically. For instance, a variable declaration *Construct* is a typed symbol *Concept*. Similarly to *Concepts*, *Constructs* can also be expanded with new language features.

Using the notions of *Constructs* and *Concepts*, we build xlg, an IR for representing the structure of a program where *Constructs* generate SSA-like values, and *Concepts* are the type system over the values; see Listing 1 for an example of xlg. **Employing SSA values to define *Concepts* is one of the fundamental properties of xlg, as it makes the relations between the notions in the programming language and the IR explicit**, allowing the handling of programming language notions with traditional IR techniques.

```
1  %0 = xlg.func_context @mul {
2   %1 = xlg.builtin_type si32
3   %2 = xlg.var_decl <xlg::param_decl> @x type %1 : <xlg::builtin_type>
4   %3 = xlg.func_decl (%2: <xlg::param_decl>) -> (%1: <xlg::builtin_type>) {
5    %4 = xlg.ref_expr @x
6    %cst = arith.constant 1.500000e+00 : f32
7    %5 = xlg.to_xlg_expr %cst : f32
8    %6 = xlg.mul_expr %4, %5 : <xlg::ref_expr>, <xlg::to_xlg_expr>
9    xlg.return_stmt  %6 : <xlg::bin_op_expr>
10  }
11  xlg.return %3 : <xlg::func_decl>
12 }
```

Listing 1. xlg IR for a function multiplying an int by 1.5 and returning the result. Concepts are represented in between "<>"

One vital characteristic of xlg is its capability to compose with traditional MLIR dialects. We accomplish this by having casting operations in xlg that let converting xlg values to and from non-xlg values. In Listing 1, we show a simple example of this compatibility layer where a `arith.constant` operation is used in Line 6, and we convert it to xlg in Line 7. This example also highlights that not every *Construct* needs to be implemented in xlg, further increasing the composability of the xlg approach.

3.1 Meta-programming in xlg

In addition to *Concepts* and *Constructs*, there is a third IR object in xlg: *Operators*. *Operators* are functors over the values produced by *Constructs*. Thus, they provide meta-programming capabilities.

In this subsection, we discuss several of the *Operators* that are part of `xlg`. Moreover, we discuss how classic IR techniques like function inlining and `xlg` *Operators* are used for handling templates.

There are two kinds of *Operators* in `xlg`: functional and procedural. As the name suggests, functional *Operators* are functions over `xlg` values. For example, the `xlg.template` *Operator* is a function where the parameters are `xlg` *Concepts* and the body is `xlg` code; see Listing 2 for an example of the `xlg.template` *Operator*. On the other hand, procedural *Operators* perform actions on the IR. For example, the `xlg.template_instance` *Operator* takes `xlg` values as inputs and calls an `xlg.template` functor; see Line 14 of Listing 2 for an example of a procedural operator.

```
1  // Template declaration
2  xlg.template @Template00(%T: !xlg.concept<xlg::template_type>)
3    -> !xlg.concept<xlg::object_decl> attributes { sym_id = "Template" } {
4   // Structure declaration
5   %sd = xlg.object_decl<xlg::object_decl> @Template {
6    // Variable using a template type
7    %vd = xlg.var_decl<xlg::member_decl> @x type %T : <xlg::template_type>
8   }
9   xlg.return %sd : !xlg.concept<xlg::object_decl>
10 }
11 // F64 type
12 %f64 = xlg.builtin_type f64
13 // Template instantiation
14 %si = xlg.template_instance @Template00(
15   %f64: !xlg.concept<xlg::builtin_type>) -> !xlg.concept<xlg::object_decl>
16 // Template after instantiation
17 %2 = xlg.object_decl @Template attributes {type = !xb.struct<@Template, f64>,
       usr = "Template"} {
18  %3 = xlg.var_decl <xlg::member_decl> @x type %0 : <xlg::builtin_type> {offset =
         0 : index, type = f64}
19 }
```

Listing 2. Example of the `xlg.template` functional *Operator*

Now that we have introduced `xlg` *Operators*, we will use Listing 2 to illustrate their role in meta-programming. In essence, this listing shows the usage of `xlg` *Operators* to create and handle template objects. We must observe several key things in this listing. First, the `xlg.template` *Operator* in Line 2 takes a `xlg::template_type` as an input parameter `%t`, and it returns a `xlg.object_decl` *Construct* in Line 9. Second, it is essential to observe that the `xlg.var_decl` *Construct* in Line 7 uses the `%t` parameter to define the type of the member. Third, it is vital to note that given the functional nature of the `xlg.template` *Operator*, we can use traditional function inlining to instantiate the template with a `xlg.template_instance` function call. Finally, Lines 17-19 show the result of instantiating the template. This example shows how `xlg` simplifies the often complex template instantiation process by reasoning at a different abstraction level using classic IR techniques.

3.2 `xlg` implementation

We implemented `xlg` as an MLIR dialect using our *Construct* and *Concept* infrastructure. *Constructs* are MLIR's operations augmented with *Concepts*, and *Concepts* use a mechanism similar to MLIR's interfaces to provide an API over *Constructs*. **Both are specified through a custom *LLVM TableGen* backend that generates the required C++ code for their functioning**; see Listing 3 for an example. Our `xlg` implementation contains a pre-defined minimal set of programming language notions that could represent a language like C, for instance, structure declarations, functions, arithmetic expressions, and statements like if and loops. Finally, since ***Concepts* and *Constructs* are open infrastructures, `xlg`-like dialects can be easily created**

to include missing programming language features, enabling the possibility of modeling the program abstract syntax generated by many programming languages.

To better exemplify the construction of **xlg**, this section and Subsection 4 show how to add support for a float multiplication expression. Listing 3 shows the creation of the **expr** *Concept* and the **mul_expr** *Construct*, and Line 8 in Listing 1 shows the IR representation of this *Construct*.

```
1 def ExprCep : Concept<"ExprConcept", "expr"> {
2   let args = (ins TypeAttr:$type);
3   let traits = [TypedOpInterface];
4 }
5 def MulExpr : XLG_Construct<"MulExprCep",
6  "mul_expr", /*parent concepts=*/ [ExprCep]> {
7   let args = (ins ExprCep:$lhs, ExprCep:$rhs);
8   let assembly = [{...}];
9 }
```

Listing 3. TableGen specification of the Expr Concept and the MulExpr Construct. In this listing, the Expr Concept has an internal type attribute. Meanwhile, the MulExpr Construct has two input operands; see Line 7

Now that we have presented **xlg** and how it can be extended with TableGen, Section 4 will discuss how to operate on it, including how to perform semantic checking and code generation on this IR.

4 Generic match-patterns, semantic checking, and code generation

While semantic checking and code generation have been around for decades, their design has often relied on the assumption of a fixed language specification. This assumption greatly simplifies the design of these components. For instance, a C compiler only needs to know how to verify C semantics. Thus, it is possible to tailor the checker to the language. However, this approach proves unsuitable for **xlg**, as it is a generic representation independent of the input language. As a result, we had to come up with a solution to this problem. This section discusses transformations with generic match patterns in the context of **xlg**. Furthermore, this section discusses semantic checking and code generation in the context of **xlg**, focusing on how to perform these tasks in an extensible manner.

To address this challenge, we revisit one of the classic techniques in compiler design: pattern-matching transformations. Pattern-matching is a well-established method used in modern compiler infrastructures like LLVM and MLIR to apply transformations. At its core, this method transforms an input if a matching predicate evaluates to true. Inspired by this approach, we introduce the idea of generic patterns to execute tasks like semantic checking and code generation within the context of **xlg**.

To achieve this objective, we extended the C++ MLIR pattern infrastructure to create what we refer to as generic patterns. At their core, generic patterns are match-only, require specialization to perform actions on the IR, and are created using our C++ API. As with traditional MLIR patterns, generic patterns also have a benefit, i.e., the driver will try to apply the most beneficial patterns first. The base class for all generic patterns is shown in Listing 4. This listing presents the `GenericPatternApplicator` class. Given a set of patterns, this class allows retrieving the pattern with the highest benefit that matches an input operation. These two classes are the basis of generic pattern matching in **xbc**.

```
1 // Base class for all patterns.
2 class GenericPattern : public mlir::Pattern {
3 public:
4   virtual mlir::LogicalResult match(mlir::Operation *op) const {
```

```
    return success();
  }
  // ...
};
// Class for retrieving patterns.
class GenericPatternApplicator {
public:
  const GenericPattern *getPattern(mlir::Operation *op);
  // ...
};
```

Listing 4. C++ classes for defining and retrieving generic patterns

The application of patterns depends on specialized drivers capable of handling the actions performed by the patterns; see Subsections 4.1 and 4.2 for examples. However, all the drivers share the C++ matching infrastructure. We built semantic and code generation drivers for this work. These drivers generally work by keeping a worklist of elements to visit and keep track of the changes introduced by the patterns. **One critical fact about these drivers is that they are not dependent on a fixed set of patterns.** Instead, these drivers take the patterns to apply as input. Thus, **these drivers solve the monolithic design of these compilation phases found in traditional compilers.**

Finally, as part of the **xbc** infrastructure, we provide a series of predefined semantic and code generation patterns for all the constructs available in **xlg**. These patterns allow compiler developers to start using **xlg** faster, as they can reuse these patterns.

4.1 Semantic checking

In this subsection, we discuss the process of performing semantic checking on **xlg** in an extensible manner. First, we discuss the creation and definition of semantic patterns. Finally, we explore the semantic checking state machine and driver.

4.1.1 ***Sema patterns:*** Semantic patterns, or *Sema* patterns, are the elemental units used to perform semantic checks in **xlg**. A *Sema* pattern is a generic pattern composed of two methods: `match` and `check`. The `match` method is inherited from `GenericPattern` class and is used to determine whether the pattern can be applied. The `check` method is invoked only if the pattern was matched and must return whether the operation is semantically valid. The `check` method returns one of four possible values:

- *success*, if the operation is semantically valid.
- *failure*, if the operation is invalid, and the semantic check will eventually fail. However, further checks can proceed.
- *success and reschedule*, if the operation is partially valid but requires further checks at a later point to be considered semantically valid.
- *fatal*, if the operation is invalid, and the semantic checks must be aborted.

The `check` method of a *Sema* pattern receives four arguments: the instance of an operation to be checked, a status field containing how many times the pattern has visited the operation, the symbol table enclosing the operation being checked, and an instance of the driver. Listing 5 provides an example of a semantic pattern to verify the **mul_expr** introduced in Listing 3. In this listing, Lines 3-5 check the input operands of the op and return if the check fails. Then, in Line 6, we try to promote the input operands to have a float type -i.e., we load or cast values. Finally, in Line 8, we set the type of the expression by extracting the type from the concept provided by the LHS expression.

```
struct MulChecker: SemaOpPattern<MulExpr> {
```

```
2  SemaResult check(MulExpr op, Status status, SymbolTable *symTable, SemaDriver &
       driver) const {
3   if (auto res = driver.checkOperands(op);
4       !res.isSuccess())
5    return res;
6   if (failed(promoteToFloat(op, op.getLhs(), op.getRhs(), driver)))
7    return op.emitError("incompatible operands");
8   op.setType(getConcept<Expr>(op.getLhs()).getType());
9   return success();
10  }
11 };
```

Listing 5. Semantic pattern for checking the `mul_expr` construct from Listing 3

Similarly to a `RewritePattern`, *Sema* patterns can be created on top of concrete operation instances, interfaces, or any operation. It is also essential to recall that *Concepts* are also MLIR interfaces; therefore, *Sema* patterns can also be created on top of *Concepts*. **The usage of *Concepts* as root for a pattern is key in `xlg`, as it allows the creation of reusable patterns**. For example, two function declaration *Constructs* with similar enough semantics but different code generation semantics could share the same pattern.

In this subsection, we discussed *Sema* patterns and how to implement them. In the following subsection, we will present the state machine used by the semantic checking driver to verify the validity of the code.

4.1.2 ***Sema State Machine:*** In this subsection, we discuss the cornerstone of the semantic checker in `xbc`, the *Sema* state machine. The *Sema* state machine is in charge of keeping track of the state of each operation, determining when the check failed or succeeded, and ensuring that there are no uncheckable patterns. We begin by discussing the possible states for an operation.

The state machine maintains an individual state for every operation in the IR. There are seven possible states for an operation: three states represent the internal states of the machine, and four states represent the outcome of a check. The three internal states are: *uninitialized*, *in checks*, and *none*. These states have the following meanings:

- The *uninitialized* state is the initial state of all operations. Moreover, it specifies that the checker has never visited the operation.
- The *in checks* state signifies that the driver is currently checking an operation.
- The *none* state is similar to *uninitialized*. However, unlike *uninitialized*, the checker has seen this operation before.

On the other hand, there are four possible outcome states: *succeeded*, *failed*, *rescheduled*, and *deferred*. The *succeeded* state indicates that the check was successful, and the pattern needs no further checks. The *failed* state alerts the checker that an operation is invalid and compilation will eventually fail. The *rescheduled* and *deferred* states imply that the checker must be reinvoked on the operation later. The difference between those states is that *rescheduled* means the check was partially successful but requires more information to verify the operation successfully. The *deferred* state is triggered when a dependent operation got *rescheduled*.

One of the challenges we needed to solve in our state machine was resolving dependencies between semantic constructs. For instance, when verifying a function definition, a recursive function might attempt to verify itself to determine the return type of a call expression, resulting in an infinite application of a pattern. **Our solution to this problem is allowing semantic patterns to perform multiple applications over a construct.** Hence, it is possible to verify the signature first and then verify the body of the declaration. The state machine also checks that there are no verification loops; if one is detected, the state machine aborts verification.

Now, we proceed to describe the algorithm behind the state machine. The state machine starts by inspecting the cache to check whether the operation has been visited before. If the state returned the cache is a *succeeded* or *failed*, then the state machine returns it, as there is no need to recompute these states. Additionally, if the state returned by the cache is *in checks*, a semantic checking cycle is detected, and the entire check is aborted. If the state is *uninitialized*, it is transformed to *none* and scheduled to get checked. Then, before applying the semantic pattern, the operation is switched to the *in checks* state. After applying the pattern, the state is updated with the result of the check. If the operation has to be *rescheduled*, it gets inserted into the back of the worklist. Finally, if the operation has been *deferred* or *deferred* too many times, the verification is aborted with an error.

In this subsection, we describe the state machine in charge of verifying the semantic state of the program. In the following subsection, we discuss the semantic checker driver.

4.1.3 ***Sema Driver:*** In this subsection, we describe the *Sema* checker driver. The *Sema* driver is responsible for assembling all the components of the semantic checking infrastructure together. It is composed of the state machine, a rewrite listener, and a set of API functions to perform semantic checking. We begin by discussing the rewrite listener.

The rewrite listener implements a set of listening methods that notify the driver of changes in the IR. For example, if an operation is inserted, the listener will notify the driver that an operation was added and that, as a consequence, the driver must check it. There are two kinds of notifications: an operation needs to be checked after insertion or modification, and an operation is erased, meaning that the driver must not check the operation. These notifications are used to manage the worklist of operations consumed by the *Sema* state machine.

The driver's API is composed of helper functions to check operations, operands, regions, blocks, and values. All of these functions in the end call the `checkOp` method. The `checkOp` invokes the state machine, checks the operation, and returns the status of the check. The driver also has an instance of the `TypeSystem` class, an extensible class to materialize a cast sequence between values; this allows patterns to introduce casts during semantic checks. This allows patterns to insert casts easily and to determine when values are incompatible and when an error must be produced.

Finally, the driver also provides a debug option that will print messages regarding the status of the semantic check. An example of these messages is presented in Listing 7. In this listing, we can observe when the driver starts checking a **`xlg.ref_expr`** operation corresponding to the symbol reference in Line 2 of Listing 6. This option is invoked using the flag `--debug-only=sema`.

```
1 fn add(x: i32) -> i32 {
2   return x + y;
3 }
```

Listing 6. Code sample with an invalid symbol reference

```
1 ** Begin checking : 'xlg.ref_expr' [state =
      uninitialized, count = 0]
2 * Pattern RefExprVerifier : 'xlg.ref_expr' {
3 sample.xb:2:14: error: symbol couldn't be found
4   return x + y;
5              ^
6 } -> pattern matched successfully : semantic
      result: `failed`
7 ** End checking : 'xlg.ref_expr' [state =
      failed, count = 1]
```

Listing 7. Debug messages produced by the *Sema* driver for the **`xlg.ref_expr`** operation obtained from the code in Listing 6

Finally, Listing 8 shows the result of applying the semantic checker to the code between Lines 5-9 in Listing 1. Here, we can observe how the referenced symbol from Listing 1 Line 5 got resolved into a **`value_ref_expr`** with a defined type. Furthermore, we can observe the results of the semantic pattern from Listing 5, where a **`xlg.load`** and **`xlg.cast`** expressions were inserted in Lines 4 and 5.

```
%4 = xlg.value_ref_expr %2 {type = !xb.memreg<si32>} : !xlg.concept<xlg::
    param_decl>
%cst = arith.constant 1.500000e+00 : f32
%5 = xlg.to_xlg_expr  %cst {type = f32} : f32
%6 = xlg.load_expr  %4 {type = si32} : <xlg::value_ref_expr>
%7 = xlg.cast_expr  %6 {type = f32} : <xlg::load_expr>
%8 = xlg.mul_expr  %7, %5 {type = f32} : <xlg::cast_expr>, <xlg::to_xlg_expr>
%9 = xlg.cast_expr  %8 {type = si32} : <xlg::mul_expr>
xlg.return_stmt  %9 : <xlg::cast_expr>
```

Listing 8. Result of applying the semantic checker pass to the code in Listing 1

In this subsection, we presented the *Sema* driver and how this can be used to check the semantic validity of `xlg`. Furthermore, **we showed how, by using generic patterns, our infrastructure allows us to perform this task in an extensible manner**. In the next subsection, we present the code generation infrastructure acting on `xlg`.

4.2 Code Generation

In this subsection, we present the `xbc` code generation infrastructure. However, we do not discuss it in as much detail as it shares many similarities with the *Sema* driver. First, we present the idea of *CodeGen* patterns as a mechanism to drive the code generation stage that takes `xlg` and produces regular MLIR. Second, we present the *CodeGen* driver. This driver orchestrates the code generation process.

CodeGen patterns are the elemental units used to generate code in `xlg`. *CodeGen* pattern are generic patterns with an additional `generate` method. The `generate` method transforms `xlg` operations into non-`xlg` operations and is invoked only when the pattern is matched. The `generate` method receives only two parameters: the operation to be transformed and the *CodeGen* driver and must return a `CGResult` value. `CGResult` values are a pointer union of an operation, a value, a type, or an attribute. The reason behind returning a pointer union is that some `xlg` *Constructs*, like types, must be transformed into MLIR types and not operations.

Similarly to *Sema* patterns, *CodeGen* patterns can be created on top of concrete operations, interfaces, *Concepts*, or opaque operations. An example of a *CodeGen* pattern is presented in Listing 9. This pattern takes the `xlg.mul_expr` introduced in Subsection 3.2 and generates the non-`xlg` `arith_op` operation.

```
struct MulCG : public OpCGPattern<MulExpr> {
  CGResult generate(MulExpr op, CGDriver &cg) const {
    return cg.replaceOpWithNewOp<ArithOp>(op, Operator::Multiply,
                                          cg.genValue(op.getLhs()),
                                          cg.genValue(op.getRhs()));
  }
};
```

Listing 9. Code Generation pattern for the `mul_expr` construct from Listing 3

The *CodeGen* driver orchestrates the application of patterns and works by traversing the `xlg` IR and maintaining mappings between `xlg` operations and the operations generated by the applied patterns. The driver has two main components: the IR mapper and a rewrite listener similar to the one found in the *Sema* driver.

The IR mapper is in charge of keeping a mapping between `xlg` operations and the operations generated by the patterns. One major challenge that had to be addressed by this driver is that the number of results by an `xlg` operation might not agree with the number of results produced by

its non-`xlg` counterpart. For example, a function declaration in `xlg` produces a value, whereas the generated `xb.func` has no results. The IR mapper solves this issue, as patterns are expected to populate, and use this map to replace operations with the correct results.

Finally, in Listing 10, we present the result of applying the *CodeGen* driver to the IR generated by the *Sema* driver in Listing 8. As shown in the listing, all `xlg` operations in Listing 8 have been replaced by non-`xlg` operations. Furthermore, it is essential to note that the generated IR is simpler and appears easier to optimize. See Section 5 for details about the generated operations.

5 `xb` dialect IR

In Section 3, we demonstrated how `xlg` can represent a program's structure. However, `xlg` is not optimal for code optimization. Thus, `xlg` needs to be transformed into an IR suitable for transformations. **While upstream MLIR provides a wide range of dialects for building a compiler, there are known gaps for representing programming languages. For example, no operations exist to represent early-exit control flow, such as the ubiquitous break and continue statements**. Therefore, we created the `xb` IR dialect, an MLIR dialect that provides the basic building blocks for representing programming languages without the perceived limitations in upstream MLIR and suitable for transformations.

Before introducing the dialect, it is convenient to refer to Figure 1 and Listing 10 to understand the section better. Figure 1 shows a high-level overview of how the `xb` dialect fits with respect to upstream MLIR dialects; in particular, this figure shows the lowering paths to upstream MLIR dialects. Listing 10 presents a first code sample in `xb`. This sample shows the result of applying the code generation pass to Listing 8.

```
xb.func @mul(%arg0: si32) -> si32 {
 %x = var @x : !xb.memreg<si32>
 store %arg0, %x : si32, !xb.memreg<si32>
 %cst = arith.constant 1.500000e+00 : f32
 %0 = load %x : !xb.memreg<si32> -> si32
 %1 = arith_cast %0 : si32 -> f32
 %2 = arith_op "*" %1, %cst : f32
 %3 = arith_cast %2 : f32 -> si32
 return %3 : si32
}
```

Listing 10. Code generated by the *CodeGen* pass for the code in Listing 8

We classify the operations in the `xb` dialect into three types:

(1) General control flow operations
(2) `xb` type operations
(3) Generalizations of upstream MLIR operations

The general control flow operations include C-like for-loops, specialized range loops, if statements, early-exit control flow, and scoping operations. Together, these operations solve one of the biggest hurdles of upstream MLIR for representing programming languages. However, they come with a caveat: many upstream passes and analyses do not work on these operations. This limitation occurs because the MLIR language reference assumes no early exit control flow is present. As a result, we had to create passes and analyses to lower these control flow operations into legal constructs in MLIR.

While upstream MLIR can represent fundamental types like pointers and structs, there are no high-level dialects to interact with those types. The only MLIR dialects capable of handling these types are the LLVM and SPIR-V backend dialects. This limitation creates an obvious layering issue, as compilers must decide early in the pipeline whether to use LLVM or SPIR-V

operations for these types. As a solution to this problem, the `xb` dialect provides pointer and struct types and operations for interacting with them. These operations can then be converted to LLVM or SPIR-V later in the compilation pipeline, fixing the layering issue.

As part of the type-specific operations in MLIR, we also provide conversion operations to and from `memref`s to pointers. These operations enable reusing the vast infrastructure surrounding `memref`s in MLIR. However, it is critical to stress that since the `memref` type is not a pointer, these conversion operations lose type information and require careful usage. Finally, we added the `!xb.memreg` type to model references and static arrays. This type is similar to the MLIR `memref` type. However, it always has a static shape and can be converted losslessly to a pointer and back, whereas a `memref` is almost always converted to a struct with at least three members: two pointers and an offset.

The final group of operations in the `xb` dialect are generalizations of upstream MLIR operations, as they might add support for missing features or remove type restrictions. These operations include `xb.arith_op`, `xb.arith_cast`, `xb.func`, and `xb.call`, amongst others. For example, `xb.arith_op` covers many operations in the arith MLIR dialect but lifts arith's restriction on integer operands to be signless. Another good example are the `xb.func` and `xb.call` operations, as we add support for C-like variadic arguments. Finally, one critical aspect is that these operations can be lowered directly into standard MLIR operations in almost all cases.

In the upcoming subsection, we will explore the transformations the `xb` IR undergoes until it reaches `llvm`.

5.1 Converting `xb` to the `llvm` dialect

We begin the lowering process by canonicalizing the representation, ensuring, for example, that constants are unique, folding cast operations if trivial, and removing unnecessary extra scopes. Canonicalization is the first step, as a frontend could have made less than optimal choices when generating the IR.

After canonicalization, we can convert the `xb` control flow into MLIR's control flow. We begin this process early in the pipeline to reuse as many passes and analyses as possible from upstream MLIR later in the pipeline. The process starts by potentially collapsing loops, as the frontend might have requested explicit loop collapsing. We then lift loops and conditionals with trivial control flow to the SCF dialect, allowing optimizing these operations at a later stage. We then transform all remaining `xb` control flow operations into flat basic blocks and branches, arriving at a legal CFG in MLIR.

Once the CFG is valid, it is possible to convert many `xb` operations into operations in standard MLIR dialects, such as `func` and `memref`. For instance, we convert `xb.var` to `memref.alloca` operations, `xb.func` to `func.func`, etc. We also convert signed and unsigned integers to signless integers during this process. After the conversion, we canonicalize the IR again to remove intermediate operations generated by the conversion process. Finally, in a second conversion pass, we convert all remaining `xb` operations and types to the `llvm` dialect.

In this section, we presented an overview of the `xb` dialect and how it addresses shortcomings in upstream MLIR for expressing programming languages. The following section discusses the `par` dialect, an MLIR dialect for parallelism. Section 7 presents a case study using `xb` in a compiler to generate code for real-world benchmarks.

6 `par` MLIR dialect

In section 5, we introduced the `xb` dialect as an IR capable of representing programming languages and suitable for optimization. However, the `xb` dialect is not designed to express high-level semantics

for parallelism. We based this decision on the fact that parallel semantics deserve their dialect. Consequently, this section presents `par`, a dialect to express parallelism.

Before presenting `par`, it is essential to acknowledge that upstream MLIR already contains dialects for the directive-based programming models OpenMP [33] and OpenACC [32]. We still decided to create `par` to solve the gaps identified in these existing dialects. For example, the OpenMP dialect is directly translated to LLVM IR and optimized by LLVM. The OpenMP community made this design decision to reuse the OpenMP IRBuilder infrastructure created for Clang and Flang. Nevertheless, while it is a good decision from a software design point of view, this choice means that the benefits of using MLIR for optimizing code are currently lost.

In contrast, the OpenACC community plans to take full advantage of MLIR by progressively lowering the IR within MLIR instead of direct translation. However, this lowering implementation is not complete, and plenty of OpenACC MLIR code is locked in the source code of Flang, meaning it is currently impossible to reuse as it is tied to Flang's Fortran semantics. We envision `par` becoming a middle ground between OpenMP, OpenACC, and upstream MLIR.

The `par` dialect is designed to be intentionally close to OpenMP and OpenACC so new developers can quickly learn and use it. However, the `par` dialect adopts a more kernel-like programming model with the philosophy that users are responsible for constraining parallelism within the model, not the model constraining users. The complete set of currently available MLIR operations in the `par` dialect can be found in Zenodo [5].

In Listing 11, we present an example of the `par` dialect. In this example, we map a memory region into a device and create a parallel region to fill a vector with a value using a work-sharing loop. In the listing, it is essential to notice that work distribution is accomplished using attributes in the loop. This decision to use attributes was crucial to creating a generic pass for choosing the parallel schedule of the loops. This pass works by searching operations with the loop attribute and implementing the *ParLoopOp* interface. Once the pass identifies the loops, it transforms them to use an explicit parallel schedule determined by the attribute.

```
1  xb.func @parFill(%A: memref<?xf32>, %v: f32) {
2   %i = xb.var @i : !xb.memreg<index>
3   %c0 = constant 0 : index
4   %c1 = constant 1 : index
5   %n = memref.dim %A, %c0 : memref<?xf32>
6   par.data_region to_from(%A : memref<?xf32>) {
7    par.region private(%i : !xb.memreg<index>) {
8     xb.range_for %i in %c0 to %n step %c1 : !xb.memreg<index> attributes {
          semantics = #par.loop} {
9      %iv = load %i : !xb.memreg<index> -> index
10     %a = ptradd %A[%iv : index] : memref<?xf32> -> memref<f32, strided<[],
           offset: ?>>
11     store %v, %a : f32, memref<f32, strided<[], offset: ?>>
12    }
13   }
14  }
15  return
16 }
```

Listing 11. `par` IR for filling a vector with a constant value

The execution model of the `par` dialect is organized into parallelism levels, namely L3, L2, L1, and L0. Figure 2 shows the mapping of this hierarchy to hardware. Using CUDA terminology, `par`'s L3, L2, L1, and L0 levels are mapped to grid, block, warp, and threads. On the other hand, only the L3 level provides parallelism on CPUs, and all other levels execute sequentially.

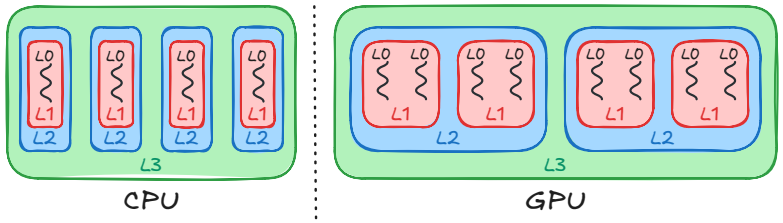

Fig. 2. Mapping of the thread hierarchy system in `par` to hardware. The mapping for CPU architectures is on the left, while the diagram on the right shows the mapping for GPU offloading.

The `region` construct is the primary method for exposing parallelism in the dialect; see Listing 11 Line 7. These regions start with all available threads in the hierarchy and execute the same region. In a CPU, this mechanism is similar to `omp parallel`, while for accelerators such as GPUs, opening a parallel region would be identical to calling a kernel in CUDA or HIP. Within parallel regions, there are no distinctions between threads, and the model assumes that potentially all threads could be working simultaneously, meaning all threads are active and ready to distribute work to specific classes of threads.

In `par`, work can be distributed across all parallel hierarchy levels using the `#par.loop` attribute. For instance, distributing a loop at the top level is equivalent to distributing a `loop` at the grid level, using CUDA terminology —loops default to running on all available levels if no nested loops or other clauses exist. If there is nested parallelism, then the compiler decides how to map the loops onto the available levels; this equates to the outer `loop` scheduled at the top level and the nested `loop` expanded into lower levels. The dialect also allows for specifying how to distribute the work across levels by adding scheduling parameters to the `#par.loop` attribute. For example, `#par.loop<L3:L2>` specifies that the loop must be distributed using the L3 and L2 parallel levels.

Finally, we created lowerings from `par` to upstream MLIR dialects. These lowerings differ depending on whether the parallel target is sequential, multi-core, or GPU. For sequential targets, all parallel operations are removed and substituted by `xb` operations. If the parallel target is multi-core, then the `par` operations are converted into OpenMP operations. We use the OpenMP dialect for this conversion as the CPU implementation in LLVM performs exceptionally well. For GPU targets, we transform most `par` operations into the `gpu`, `scf`, and `memref` upstream dialects. However, depending on the GPU target, we also transform some operations to target specific dialects like NVVM and ROCDL.

7 `xblang`: an extensible programming language

In this section, we will discuss creating a new extensible programming language called `xblang` using `xlg`, `xb`, and `par`. We created `xblang` to showcase the possibilities provided by the extensibility properties of our infrastructure.

Before discussing `xblang`, we must clarify what we mean by an extensible programming language. **We define extensibility as the ability to introduce new programming constructs to a language easily at compilation time.** To further explain this notion, we use Listing 12 as an example of a program in `xblang` that prints a string from a GPU. Without any extra information, this program is invalid in `xblang`, as the base language does not understand how to interpret `omp parallel`, `gpu::region` or `mlir::inline` constructs. However, the program can become valid by inserting the `omp`, `gpu`, and `mlir` extensions into `xblang` at compile time, thus extending the set of valid input programs. One final thing to observe from this example is the `mlir::inline` construct. This construct allows programmers to use MLIR natively in the language, similar to ASM inline in other languages, adding a beneficial tool for prototyping.

```
fn [extern] omp_get_thread_num() -> i32;
fn main() {
 let bsz: i32 = 4;  // Block size
 let gsz: i32 = 2;  // Grid size
 omp parallel firstprivate(bsz, gsz) {
  let ompId = omp_get_thread_num();
  gpu::region<<[bsz], [gsz]>> {
   let tid : i32 = threadIdx.x;
   let bid : i32 = blockIdx.x;
   mlir::inline(ompId: 'i32', tid: 'i32', bid: 'i32') '''
   gpu.printf "Host Thread ID: %d, Block ID: %d, Thread ID: %d\n" %ompId, %bid,
       %tid : i32, i32, i32
   ''';
  }
 }
}
```

Listing 12. **xblang**'s program for printing thread ID information from a GPU.

xblang's syntax is similar to that of Rust and C, with blocks of statements delimited by curly braces, expressions delimited by semicolons, and the usual control flow statements. The semantics of **xblang** are close to those of C, with some extensions such as type inference and a module system instead of the C include system.

xblang utilizes language extensions to extend its functionality. Each extension encapsulates specific language constructs and provides rules for creating and interpreting each construct. **Extensions are also free to introduce any elements necessary for their semantic and syntactic purposes**. The only requirement imposed on extensions is to ensure that any side effects produced by a construct have a valid semantic meaning in the surrounding context. For example, it is not valid to create a statement if the context expects a declaration. We use **xlg**'s *Concepts* and *Constructs* to model and enforce these rules. Syntactically, extensions are invoked by specifying the extension's keyword or by invoking the desired construct directly, provided the dialect registers the construct within the parent context, and there are no syntactic conflicts.

Listing 13 shows a SAXPY kernel written using the **par** extension. This extension directly models the semantics of the operations in the **par** dialect. To ease the porting of programs, the syntax of the extension closely resembles that of OpenMP. This listing also shows the memory mapping mechanism in **par**; internally, a custom memory manager calling CUDA or HIP handles this mapping.

```
fn saxpy_xb(x: f64*, y: f64*, a: f64, n: i32) {
  par map(toFrom: x[0 : n]) map(to: y[0 : n])
  par region firstprivate(a, n)
  loop(let i: i32 in 0 : n)
    x[i] = a * y[i] + x[i];
}
```

Listing 13. **xblang**'s high level version of the SAXPY kernel.

Finally, **extensions are implemented as compiler plugins** and are loaded through the command **xbc** `--load-extension=OMPExtension.so`. These plugins are implemented by providing the plugin registration hook and all the appropriate interfaces to interact with **xlg**. This further elucidates the sheer extensibility power of our infrastructure, as **developers can create these plugins in isolation and share them, breaking the complexity of an all-powerful compiler into modular components.**

In this section, we presented a high-level overview of **xblang** and its extensibility properties. The following section evaluates the **xbc** infrastructure by creating a quantum extension and evaluating parallel benchmarks.

8 Evaluation

In this section, we evaluate the **xbc** infrastructure from two angles: extensibility and performance. In Subsection 8.1, we evaluate the extensibility properties of the infrastructure by creating a Quantum language extension for **xblang**. In Subsection 8.2, we benchmark the code generated by the **xbc** infrastructure against several popular benchmarks.

8.1 Quantum extension and the Quake Dialect

In this subsection, we will introduce our quantum extension for **xblang**. Instead of creating a quantum dialect, we decided to use the existing Quake MLIR dialect to embolden our claims of extensibility and demonstrate composability as an infrastructure. We will begin this subsection by providing an overview of the Quake MLIR dialect [13] and conclude it by discussing how we leverage our infrastructure to target Quake.

CUDA-Q [13] is an open-source platform created by NVIDIA for quantum programming research. It is envisioned as a platform for researchers to conduct quantum simulations using GPUs, CPUs, or Quantum Processing Units (QPUs). As a result, CUDA-Q offers a valuable platform for experimentation while waiting for quantum computers to become more accessible. As part of the CUDA-Q platform, NVIDIA developed the MLIR dialect: Quantum Kernel Execution (Quake). NVIDIA conceived this dialect to represent and optimize quantum circuits using the MLIR infrastructure.

Before fully introducing Quake, it is essential to discuss the execution model of CUDA-Q. CUDA-Q takes inspiration from CUDA and sets forth a model where quantum kernels are insulated from host code. In simpler terms, this means that the host in CUDA-Q views QPUs as black boxes with classical inputs and outputs that run quantum algorithms.

The quake dialect puts forth two semantic models: value and memory. The memory model treats operations on a qubit as alterations to the qubit's reference through time. Meanwhile, the value model treats the outcome of a quantum operation as new qubits. In this dissertation, we focus on the memory model, as it is easier to express its semantics using a programming language. Listing 14 shows an example of Quake using memory semantics.

```
1 func.func @bell() {
2   %0 = quake.alloca !quake.ref
3   %1 = quake.alloca !quake.ref
4   quake.h %0 : (!quake.ref) -> ()
5   quake.x [%0] %1 : (!quake.ref, !quake.ref) -> ()
6   return
7 }
```

Listing 14. Quake memory-semantic model

Quake models quantum gates as MLIR operations that receive the input and control qubits. The result of a Quake gate depends on the semantic model; in the memory model, there are no results, as the input qubits were modified in place. It is essential to mention that Quake provides standard quantum gates such as Hadamard, Pauli operations, and general rotations, allowing the expression of most quantum circuits [13].

Finally, as part of CUDA-Q, Quake provides translations to LLVM and QIR [34]. Which can then be simulated with the CUDA-Q QIR runtime library [13]. Therefore, Quake provides an end-to-end environment to express and simulate quantum circuits.

8.1.1 ***Quantum extension:*** In order to create our quantum extension and target the Quake dialect, we leverage the Semantic and Code generation infrastructure introduced in Section 4. Before continuing, it is essential to state that for this extension, no *Concepts* or *Constructs* were needed, as existing `xlg` ones proved to be enough. However, when we created this extension, we added the `xlg.generic_expr` *Constructs* to `xlg` to model quantum operators. It is also helpful to briefly introduce Listing 15. This listing shows the usage of `xblang` and the quantum extension to express the Ansatz operator used in the Max-cut quantum algorithm. We chose the Max-cut algorithm due to its significance for quantum supremacy [17], as Max-cut is an NP-hard problem.

```
fn [target = qpu] ansatz_operator(theta: f64*, nq: i32, nl: i32) {
  let q: qubit[nq];
  qu::h(q);
  for (let i: i32 in 0 : nl) {
   for (let j: i32 in 0 : nq) {
    qu::x  ctrl[q[j]] (q[(j + 1) % nq]);
    qu::rz [2. * theta[i]] (q[(j + 1) % nq]);
    qu::x  ctrl[q[j]] (q[(j + 1) % nq]);
   }
   for (let j: i32 in 0 : nq)
    qu::rx [2. * theta[i + nl]] (q[(j + 1) % nq]);
  }
 }
```

Listing 15. Code in `xblang` for the Ansatz operator for solving the Max-cut problem

To do semantic checking on quantum programs, we added semantic patterns that match a *Constructs* upon detecting quantum types, thus allowing the pattern to infer the appropriate Quake types. We also added patterns that targeted the `xlg.generic_expr` *Construct* to promote floats into doubles, among other things, as the Quake dialect requires these for correctly functioning. **Similarly, we added code generation patterns that override the default behavior of `xlg` patterns**. Thus, they generate Quake operations instead of the usual `xb` operations.

We evaluated our extension on several code samples from the CUDA-Q [13] GitHub repository like the one in Listing 15. However, we do not present these results in Section 8.2 as in all cases, we obtained the same performance results as CUDA-Q, as all the performance gains are provided by Quake and not `xb`. Nevertheless, **this extension exhibits the extensibility properties of our infrastructure and its ability to compose with external projects**. It also highlights how `xblang` could be used to further compiler research with new programming abstractions.

8.2 `xbc` benchmarking

To evaluate the efficacy of `xbc` and the parallel extension, we have chosen applications representing some of Berkeley's seven dwarfs [6] that represent common HPC application patterns. As part of the selection criteria, we choose applications that utilize OpenMP offloading capabilities, can run on GPU-based systems, and are maintained regularly. The benchmarks used for our evaluation are CG, representing Sparse Linear Algebra; Miniweather, representing Structured Grids; EP, XSBench, and RSBench, representing Monte Carlo simulations. These applications are part of state-of-the-art benchmark suites like SPEC-ACCEL [19], SPEChpc2021 [23], and ECP proxy applications [2].

The raw files of the results presented have been uploaded in Zenodo [5]. The specific test cases used to demonstrate the efficacy of `xbc` are the NAS parallel benchmark codes EP (Embarrassingly Parallel) & CG (Conjugate Gradient), both using Class=C and three mini-apps: MiniWeather (200x100 grid size, 5000s being the length of the simulation), RSBench (large simulation) and XSBench (large simulation). We use Perlmutter [25] at LBNL and Frontier [31] at ORNL for evaluation

purposes. All tests were run ten times and averaged; time measurements were made using C++ 'high_resolution_clock' timers. In the case of GPU runs, we also included device-wide synchronization calls before making the measurement.

We use a single GPU for all our runs: NVIDIA A100 on Perlmutter and AMD MI250x on Frontier. We verified the output correctness of the programs generated by **xbc** using the built-in validation mechanism for validating their output for all benchmarks and platforms. All benchmarks were compiled using *-O3* optimization flags.

Fig. 3 shows results using Perlmutter. The compilers used on Perlmutter include Clang OpenMP offloading 19.0.0 (82c5d350d2), nvc OpenMP offloading 24.5, Cray OpenMP offloading 17.0.0 (b59b7a8e91), nvc OpenACC 24.5 and our **xbc**. We observe that **xbc** performs the best for EP and CG even compared to vendor compilers; for XSBench **xbc** performs close to the other compilers targeting GPU with *clang-omp* performing the best; for mini-weather **xbc** performs almost close to the rest besides *clang-omp* that performs the worst. In general, **xbc** performs close to the best for each benchmark.

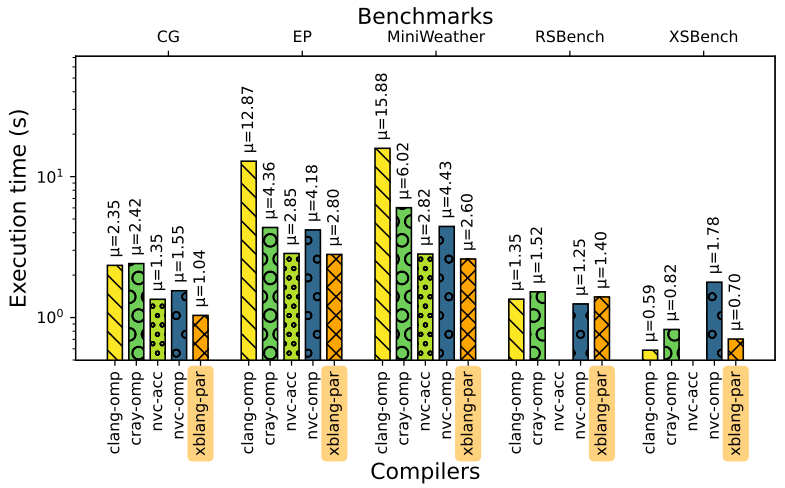

Fig. 3. **Lower the better:** Execution time for five benchmarks on Perlmutter NVIDIA A100 GPUs comparing **xbc** with other compilers; RSBench and XSBench do not have an OpenACC equivalent hence *nvc-acc* bar does not exist. μ represents the average execution time. All benchmarks were compiled using *-O3* flags.

Fig. 4 presents results using Frontier. The compilers used on Frontier include Clang OpenMP offloading 19.0.0 (82c5d350d2), ROCM's 6.0.0 AMD Clang (7208e8d15f), Cray OpenMP offloading 17.0.0 (b59b7a8e91) and our **xbc** compiler. We observe that **xbc** performs better than all other compilers in 3 benchmarks: EP, CG, and Mini-Weather; for RSBench **xbc** performs close to the best-performing compiler AMD-Clang; for XSBench, the figure shows there is room for improvement in **xbc**. After profiling RSBench and XSBench using AMD's *rocprof*, we became aware that **xbc** 's kernels appear to have the same performance as the other compilers, with the performance downgrade coming from **xbc**'s runtime for mapping memory to and back from the device.

On Perlmutter, we investigated why **xbc** performs better in specific programs like CG. For this, we profiled both Clang and **xbc** versions of the benchmark using the NVIDIA profiling tool Nsys on Perlmutter's A100. From these profiles, we noticed that we overperform in kernels that perform reductions. For example, in kernels with reduction having a reduction, **xbc**'s version of the kernel is 2.9 times faster than Clang's version. After looking at the LLVM IR generated by Clang and

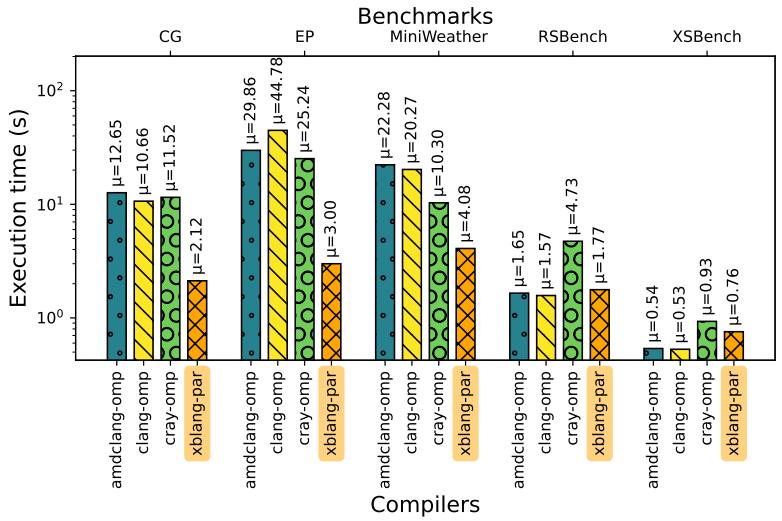

Fig. 4. **Lower the better:** Execution time for five benchmarks on Frontier AMD MI250x GPUs comparing `xbc` with other compilers. μ represents the average execution time. All benchmarks were compiled using only *-O3* flags.

speaking to LLVM OpenMP developers, we can confirm that `xbc`'s implementation of reductions is the primary source for overperformance.

In the case of Frontier, we can see that EP `xbc` outperforms other compilers between 8.3× to 14.5×. Similarly, CG `xbc` outperforms the other compilers by about 5×. In this case, the difference is the ability of the compilers to generate the correct parallelism for the hardware. For example, a simple change in the dominant kernel loop: remove the *parallel for* in the *omp pragma* of the outer loop and add it to the *omp pragma* of the inner loop (see Zenodo [5]), improves time of CG for Clang and AMD-clang from 11s to 3.4s and 4.98s respectively. This shows that the large performance gap of `xbc` on Frontier is due to compiler optimization missed opportunities for the particular GPU device.

Fig. 5 presents results using Perlmutter CPU-only nodes. The compilers used on Perlmutter include Clang OpenMP offloading 19.0.0 (82c5d350d2), nvc OpenMP offloading 24.5, Cray OpenMP offloading 17.0.0 (b59b7a8e91), nvc OpenACC 24.5 and our `xbc` compiler. We observe that `xbc` performs similarly to Clang in three of the five benchmarks, while `xbc`'s CG and MiniWeather performance is not as good. After further analysis, we concluded that the reason for `xbc`'s performance degradation is a bug where certain variables are not being privatized correctly; if said variables are manually privatized, then performance is again comparable to Clang's.

Finally, **while we introduced some optimizations in our pipeline**, for example, we fused the synchronization points from multiple consecutive reduction operations into one, reducing synchronization overhead. **There are also implicit performance gains due to the differences in the underlying programming models**. We designed `par`'s model with GPU-like kernels in mind; therefore, maintaining an internal state is unnecessary. In contrast, OpenMP's host-centric programming model requires maintaining an internal state, which compilers cannot always optimize, thus incurring the extra overhead seen in the results for the OpenMP compilers. **It is also important to note that the optimization pipeline in `xbc` is still in its early stages and out of the scope of this paper.**

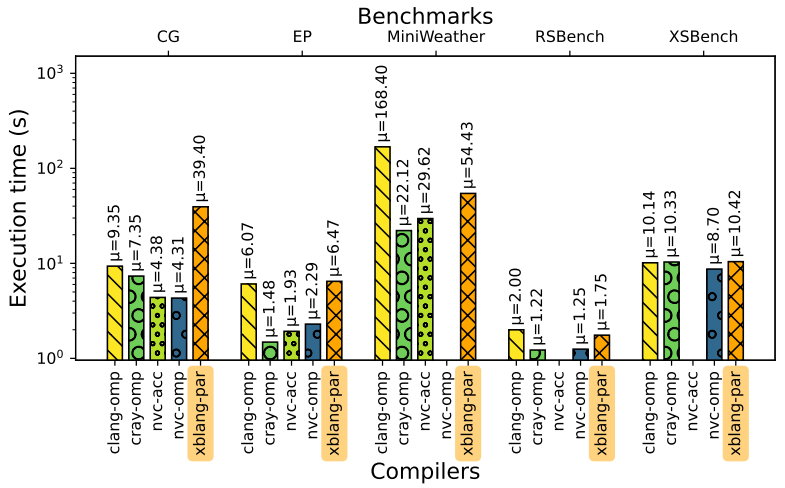

Fig. 5. **Lower the better:** Execution time for five benchmarks on Perlmutter's AMD EPYC 7763 CPUs comparing **xbc** with other compilers; *nvc-omp* for MiniWeather failed to execute; RSBench and XSBench do not have an OpenACC equivalent hence *nvc-acc* bar does not exist. μ represents the average execution time. All benchmarks were compiled using only *-O3* flags

In summary, results show that **xbc** for GPUs, with its simplified syntax and semantics, has the potential to be either at par or better than directive-based compilers for GPUs and comparable performance for CPUs. While we have more experiments to perform, we hypothesize that **xbc** offers the programmer and compiler researchers a lightweight language and compiler. It is quite appealing that our language is at par with vendor compilers and other compilers such as GNU and Clang under multiple situations.

9 Related work

This section gives an overview of the state of the art and related work, providing a motivation for our work. In software development, compiler practitioners often create source-to-source tools, extend existing languages and compilers, or create new programming languages. Here is a summary and the identified gaps:

Source-to-source (S2S) tools are classic yet popular options for conducting research; see CETUS [11] and derived projects [21, 22], the ROSE compiler [35], and other S2S tools [7, 8, 12, 15, 16, 38]. However, as the survey article [26] presents, these tools have many perceived shortcomings and detractors. Practitioners avoided using these tools because they are difficult to extend to support new programming models, leading to complex and fragile workflows, may interfere with compiler optimizations, and are always limited to being within the compounds of the target language.

Clang-LLVM [20] enables the creation of new HPC programming model abstractions within C++. LLVM passes can easily modify Clang's intermediate representation (IR), thus providing a robust workflow for introducing optimizations and support for new hardware architectures. However, LLVM IR has no high-level information about the language and its semantics, limiting the type of possible manipulations.

Modifying Clang's frontend source code allows the introduction of new language constructs; however, compiling source code modifications might take significant time, depending on the system, reducing productivity. Furthermore, **language extension modifications on Clang would require patching Clang's source and, therefore, are not easy to combine with other language**

extensions or with the permanent Clang development, as it requires acceptance from the Clang community.

An MLIR frontend would overcome some of the shortcomings of LLVM passes, thanks to MLIR's capability to represent high-level information. Two notable projects leveraging MLIR are ClangIR [1], a high-level IR for Clang, and LLVM-Flang [4], a Fortran compiler. However, both are still under active development, with ClangIR in early development and Flang not fully in production but getting close to [3]. Nonetheless, the struggle to introduce new constructs into these existing frontends persists, as they are not general MLIR frontends.

Other related work includes the COMET compiler [37], a DSL for sparse and dense tensor algebra computations based on MLIR; it is not a generic programming language frontend for MLIR.

Polygeist [29] performs polyhedral optimization and parallel code transpilation, achieving excellent performance in both cases. It is an MLIR frontend for a subset of C/C++; hence, by design, it is not an extensible frontend for MLIR. Instead, it is a high-level C/C++ code optimizer. Mojo [28] is a programming language introduced by the company Modular. It is a Python superset capable of delivering up to 68,000x speedup over Python [27]. It uses MLIR for its internal representation, enabling it to perform optimizations at a higher level. However, their compiler is not open-source. It is also important to mention that porting HPC codes to a Pythonic language might pose additional challenges compared to other options due to the high-level essence of Pythonic languages and the traditional low-level nature of HPC codes.

xDSL [10] is a Python-based toolbox for easily expressing MLIR dialects in Python without the need to write C++ code. The toolbox is designed to convert Python to MLIR C++ for an easier workflow when developing a DSL. They accomplish this with IRDL [14], an MLIR dialect used to define dialects, operations, types, and attributes within MLIR. VAST [30] is an MLIR-based library for programs written in C/C++ to provide an analysis at various stages of the compilation process and transform parsed C/C++ code into a high-level MLIR dialect.

Our approach sets us apart from the above-related efforts by providing an extensible infrastructure for representing and compiling programming languages, offering the essential toolchain for software development, and enabling compiler developers to adapt to the rapid changes in hardware evolution in heterogeneous computing.

10 Conclusions and Future Work

We introduced the eXtensible Base Compiler `xbc`, a high-level extensible compiler infrastructure that uses MLIR. We can target heterogeneous systems consisting of CPUs, GPUs, and Quantum systems. `xbc` consists of a new intermediate representation (IR), `xlg` that represents programming language constructs. We also introduced two other MLIR dialects: `par`, an IR for expressing parallelism with GPU and CPU lowering to LLVM, and the `xb` MLIR dialect, suitable for optimizations and transformations, demonstrating its capabilities in lowering to both LLVM and standard MLIR dialects. Lastly, we introduce an extensible programming language `xblang` to express the semantics of a high-level programming language and interact with existing MLIR dialects, such as the Quake dialect, showcasing the extensibility and potential of our approach. We demonstrate our compelling results at par with vendor compilers under multiple circumstances. Furthermore, we will create language extensions for other programming models in `xblang` for other researchers to expand and improve them with new parallel programming abstractions. We will evaluate with more tests and mini-apps. Finally, we will make `xblang` a fully self-extensible programming language. This is possible given the close integration of `xbc` and MLIR, and dialects like IRDL and PDL. This work would allow developers to optimize programs to their preference without leaving the language. Moreover, this would allow more expert domain knowledge to be involved in optimizations.

References

[1] LLVM 2024. *ClangIR · A new high-level IR for clang*. LLVM. https://llvm.github.io/clangir/

[2] Exascale Computing Project 2024. *ECP Proxy Applications*. Exascale Computing Project. https://proxyapps.exascaleproject.org/app/

[3] LLVM Discourse (Ed.). 2024. *[PROPOSAL] Rename 'flang-new' to 'flang'*. https://discourse.llvm.org/t/proposal-rename-flang-new-to-flang/69462/67

[4] LLVM 2024. *The Flang Compiler*. LLVM. https://flang.llvm.org/docs/

[5] Anonymous. 2024. xbc: An extensible MLIR compiler infrastructure for programming language and heterogeneous computing research. https://doi.org/10.5281/zenodo.14166659

[6] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. 2009. A View of the Parallel Computing Landscape. *Commun. ACM* 52, 10 (10 2009), 56–67. https://doi.org/10.1145/1562764.1562783

[7] Jairo Balart, Alejandro Duran, Marc Gonzàlez, Xavier Martorell, Eduard Ayguadé, and Jesús Labarta. 2004. Nanos mercurium: a research compiler for openmp. In *Proceedings of the European Workshop on OpenMP*, Vol. 8. 56.

[8] Gábor Dániel Balogh, Gihan R Mudalige, István Zoltán Reguly, SF Antao, and C Bertolli. 2018. Op2-clang: A source-to-source translator using clang/llvm libtooling. In *2018 IEEE/ACM 5th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*. IEEE, 59–70.

[9] Charles H. Bennett and Gilles Brassard. 2014. Quantum cryptography: Public key distribution and coin tossing. *Theoretical Computer Science* 560 (Dec. 2014), 7–11. https://doi.org/10.1016/j.tcs.2014.05.025

[10] Nick Brown, Tobias Grosser, Mathieu Fehr, Michel Steuwer, and Paul Kelly. 2022. xDSL: A common compiler ecosystem for domain specific languages. https://sc23.supercomputing.org/ Supercomputing 2023, SC23 ; Conference date: 12-11-2023 Through 17-11-2023.

[11] Chirag Dave, Hansang Bae, Seung-Jai Min, Seyong Lee, Rudolf Eigenmann, and Samuel Midkiff. 2009. Cetus: A source-to-source compiler infrastructure for multicores. *Computer* 42, 12 (2009), 36–42.

[12] Joel E Denny, Seyong Lee, and Jeffrey S Vetter. 2018. Clacc: Translating openacc to openmp in clang. In *2018 IEEE/ACM 5th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*. IEEE, 18–29.

[13] The CUDA Quantum development team. 2024. *CUDA Quantum*. https://doi.org/10.5281/zenodo.10835935

[14] Mathieu Fehr, Jeff Niu, River Riddle, Mehdi Amini, Zhendong Su, and Tobias Grosser. 2022. IRDL: an IR definition language for SSA compilers. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 199–212. https://doi.org/10.1145/3519939.3523700

[15] Mikhail R Gadelha, Jeremy Morse, Lucas Cordeiro, and Denis Nicole. 2017. Using clang as a Frontend on a Formal Verification Tool.

[16] Philipp Gschwandtner, Juan J Durillo, and Thomas Fahringer. 2014. Multi-objective auto-tuning with insieme: Optimization and trade-off analysis for time, energy and resource usage. In *European Conference on Parallel Processing*. Springer, 87–98.

[17] G. G. Guerreschi and A. Y. Matsuura. 2019. QAOA for Max-Cut requires hundreds of qubits for quantum speed-up. *Scientific Reports* 9, 1 (May 2019), 6903. https://doi.org/10.1038/s41598-019-43176-9

[18] Travis S. Humble, Alexander McCaskey, Dmitry I. Lyakh, Meenambika Gowrishankar, Albert Frisch, and Thomas Monz. 2021. Quantum Computers for High-Performance Computing. *IEEE Micro* 41, 5 (2021), 15–23. https://doi.org/10.1109/MM.2021.3099140

[19] Guido Juckeland, William Brantley, Sunita Chandrasekaran, Barbara Chapman, Shuai Che, Mathew Colgrove, Huiyu Feng, Alexander Grund, Robert Henschel, Wen-Mei W. Hwu, Huian Li, Matthias S. Müller, Wolfgang E. Nagel, Maxim Perminov, Pavel Shelepugin, Kevin Skadron, John Stratton, Alexey Titov, Ke Wang, Matthijs van Waveren, Brian Whitney, Sandra Wienke, Rengan Xu, and Kalyan Kumaran. 2015. SPEC ACCEL: A Standard Application Suite for Measuring Hardware Accelerator Performance. In *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation*, Stephen A. Jarvis, Steven A. Wright, and Simon D. Hammond (Eds.). Springer International Publishing, Cham, 46–67.

[20] C. Lattner and V. Adve. 2004. LLVM: a compilation framework for lifelong program analysis amp; transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* 75–86. https://doi.org/10.1109/CGO.2004.1281665

[21] Seyong Lee and Jeffrey S. Vetter. 2014. OpenARC: Extensible OpenACC Compiler Framework for Directive-Based Accelerator Programming Study. In *2014 First Workshop on Accelerator Programming using Directives*. 1–11. https://doi.org/10.1109/WACCPD.2014.7

[22] Seyong Lee and Jeffrey S. Vetter. 2014. OpenARC: open accelerator research compiler for directive-based, efficient heterogeneous computing. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing (HPDC '14)*. Association for Computing Machinery, New York, NY, USA, 115–120. https:

//doi.org/10.1145/2600212.2600704

[23] Junjie Li, Alexander Bobyr, Swen Boehm, William Brantley, Holger Brunst, Aurelien Cavelan, Sunita Chandrasekaran, Jimmy Cheng, Florina M. Ciorba, Mathew Colgrove, Tony Curtis, Christopher Daley, Mauricio Ferrato, Mayara Gimenes de Souza, Nick Hagerty, Robert Henschel, Guido Juckeland, Jeffrey Kelling, Kelvin Li, Ron Lieberman, Kevin McMahon, Egor Melnichenko, Mohamed Ayoub Neggaz, Hiroshi Ono, Carl Ponder, Dave Raddatz, Severin Schueller, Robert Searles, Fedor Vasilev, Veronica Melesse Vergara, Bo Wang, Bert Wesarg, Sandra Wienke, and Miguel Zavala. 2022. SPEChpc 2021 Benchmark Suites for Modern HPC Systems. In *Companion of the 2022 ACM/SPEC International Conference on Performance Engineering* (Bejing, China) *(ICPE '22)*. Association for Computing Machinery, New York, NY, USA, 15–16. https://doi.org/10.1145/3491204.3527498

[24] Weitang Li, Zhi Yin, Xiaoran Li, Dongqiang Ma, Shuang Yi, Zhenxing Zhang, Chenji Zou, Kunliang Bu, Maochun Dai, Jie Yue, Yuzong Chen, Xiaojin Zhang, and Shengyu Zhang. 2024. A Quantum Computing Pipeline for Real World Drug Discovery: From Algorithm to Quantum Hardware. arXiv:2401.03759 [physics.chem-ph]

[25] MERSC. 2024. *Perlmutter Architecture*. https://docs.nersc.gov/systems/perlmutter/architecture/

[26] Reed Milewicz, Peter Pirkelbauer, Prema Soundararajan, Hadia Ahmed, and Tony Skjellum. 2021. Negative Perceptions About the Applicability of Source-to-Source Compilers in HPC: A Literature Review. In *International Conference on High Performance Computing*. Springer, 233–246.

[27] Modular Inc. 2023. A journey to 68,000x speedup over Python - Part 3. https://www.modular.com/blog/mojo-a-journey-to-68-000x-speedup-over-python-part-3

[28] Modular Inc. 2023. Mojo programming manual. https://docs.modular.com/mojo/programming-manual.html

[29] William S. Moses, Ivan R. Ivanov, Jens Domke, Toshio Endo, Johannes Doerfert, and Oleksandr Zinenko. 2023. High-Performance GPU-to-CPU Transpilation and Optimization via High-Level Parallel Constructs. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming* (Montreal, QC, Canada) *(PPoPP '23)*. Association for Computing Machinery, New York, NY, USA, 119–134. https://doi.org/10.1145/3572848.3577475

[30] Trail of Bits. 2024. *VAST*. https://github.com/trailofbits/vast

[31] OLCF. 2024. *Frontier user guide*. https://docs.olcf.ornl.gov/systems/frontier_user_guide.html

[32] OpenACC. 2024. OpenACC, Directives for Accelerators. http://www.openacc.org/.

[33] OpenMP Architecture Review Board. 2022. OpenMP Technical Report 11: Version 6.0 Preview 1. https://www.openmp.org/wp-content/uploads/openmp-TR11.pdf

[34] QIR Alliance. 2021. *QIR Specification*. https://github.com/qir-alliance/qir-spec Also see https://qir-alliance.org.

[35] Dan Quinlan and Chunhua Liao. 2011. The ROSE source-to-source compiler infrastructure. In *Cetus users and compiler infrastructure workshop, in conjunction with PACT*, Vol. 2011. Citeseer, 1.

[36] F. Tennie and T. N. Palmer. 2023. Quantum Computers for Weather and Climate Prediction: The Good, the Bad, and the Noisy. *Bulletin of the American Meteorological Society* 104, 2 (2023), E488 – E500. https://doi.org/10.1175/BAMS-D-22-0031.1

[37] Ruiqin Tian, Luanzheng Guo, Jiajia Li, Bin Ren, and Gokcen Kestor. 2021. A High Performance Sparse Tensor Algebra Compiler in MLIR. In *2021 IEEE/ACM 7th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*. 27–38. https://doi.org/10.1109/LLVMHPC54804.2021.00009

[38] Xingfu Wu, Michael Kruse, Prasanna Balaprakash, Hal Finkel, Paul Hovland, Valerie Taylor, and Mary Hall. 2021. Autotuning PolyBench Benchmarks with LLVM Clang/Polly Loop Optimization Pragmas Using Bayesian Optimization (extended version). *arXiv preprint arXiv:2104.13242* (2021).

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009