MLIR compiler for *XBLang*: An extensible programming language

***Abstract*—The MLIR compiler infrastructure and its capability to represent intermediate representations proved a significant addition to the compiler developer toolset. MLIR paved way for optimizing performance beyond the capabilities of lower-level representations such as LLVM IR. As a result, the compiler developer community is showing increased interest in this technology, with major compiler projects like Flang now using the MLIR infrastructure as their IR. Within MLIR are dialects for representing OpenMP and OpenACC constructs; the MLIR community and vendors are actively developing these dialects. However, there are a few high-level front-ends for interacting with these dialects, slowing development as there is less room for testing and benchmarking.**

This paper presents the design and development of XBLang, an extensible compiler front-end infrastructure for interacting with MLIR, (which unlike projects like Flang that are bound to a standard and a fixed set of features), aims to allow open access to MLIR as a whole. XBLang comprises of an extensible high-level front-end -suitable for creating and testing new language constructs and a middle-end based entirely on MLIR. XBLang can already target multicore CPUs and GPU parallelism and successfully runs on targets like NVIDIA, AMD GPUS, and CPUs like A64FX, showing speedups or comparable performance to vendor compilers for GPUs in preliminary testing. The XBLang compiler can already interact with the OpenMP MLIR dialect for generating parallel code. Highlighting one of our results from evaluation, we observe that for NAS CG *XBLang*'s GPU version on NVIDIA A100 is $74\times$ faster than Clang-serial, $2.23\times$ faster than Clang's OpenMP offload, and $1.17\times$ faster than NVIDIA's OpenACC compilers; for AMD MI250x, *XBLang* is $36\times$ faster than Clang sequential, $5\times$ faster than Clang's OpenMP offload, and $6\times$ faster than AMD Clang's OpenMP offload.

***Index Terms*—parallel languages, parallel programming, compiler infrastructure**

I. Introduction

In the last couple of decades, there has been a paradigm shift from homogeneous systems to heterogeneous systems. The heterogeneous system era has seen the introduction of massively parallel architectures such as GPGPUs from vendors such as NVIDIA and AMD, vector architectures such as the A64FX from Fujitsu/ARM, and also other types of devices such as FPGAs, ASICs, neural engines, and many-core processors. This shift means plenty of parallelism to be exposed and expressed at the different hardware and software stack levels.

Architectures are evolving so rapidly that it has led to a perpetual disruption in software. Software developers are facing non-trivial challenges porting legacy algorithms as they often need to revisit the algorithms that work well for conventional architectures but cannot necessarily reap the benefits of the rich features of the novel architectures. To that end, they need to create newer parallel algorithms where need be, create suitable abstractions, create parallel-friendly data structures, and adapt these changes to evolving programming models; these steps are not trivial and are prone to introduce bugs.

Hardware vendors use different programming models for their accelerators, exacerbating this disruption. Hence, unlike CPUs and traditional programming languages like C/C++/Fortran, there is no unified and accessible way to program these devices. Parallel programming models like OpenMP [1], OpenACC [2], and Kokkos [3], among others, have been addressing the portability issue for several years now. However, these models do not provide access to all hardware features that native languages like CUDA and HIP do, creating a gap detrimental to end users.

To further complicate the already challenging landscape, compiler developers face a complex, never-ending race to implement host language compiler features and a myriad of programming models. For example, while the OpenMP specification continues to grow, they are still developing offloading implementations for the widely popular OpenMP directive-based parallel programming model; see SOLLVE V&V [4] for reference on the status of some of these implementations. An OpenMP 6.0 release is scheduled for later this year, while most compiler developers are still implementing features from OpenMP 5.0, ratified in 2018. To that end, most OpenMP offloading stories [5]–[12] almost only use OpenMP 4.5 features ratified in 2015.

C++ is a prominent language for HPC [13]. However, its syntax, semantics, and lengthy ISO specification still present a challenge. Rust, Python, and lately Julia [14] are becoming languages of interest to HPC. The constant addition of new features and C++'s syntax and semantics to the language has made it difficult for researchers to maintain or develop parallel tools targeting C++. It is relatively easy to introduce new parallel programming models or features into C; for example, the CETUS compiler [15] and derived projects [16] do not support C++, only C. However, the syntax and semantics of C have restrictions that languages like Rust [17] and Python [18] have removed, and setting up rules that are more amicable to developers.

Given the challenges mentioned above and open research opportunities, we make the following contributions with this paper:

- The ***XBLang*** language, an extensible programming language conceived to target the evolving hardware land-

Identify applicable funding agency here. If none, delete this.

scape, constantly demanding newer software techniques for maximizing performance.

- The `xbc` compiler, an MLIR-based extensible compiler for ***XBLang***. This compiler introduces a new MLIR dialect to express the semantics of a high-level programming language capable of interacting with existing MLIR dialects, thus leveraging existing MLIR features and capabilities.
- The **`par`** dialect, a parallel programming model capable of targeting CPUs and GPUs, **created to demonstrate the extensibility properties of the language and the compiler**. The model's performance matches or surpasses vendor performance compilers, outperforming other open-source compilers and programming models for our test cases. We envision our model to evolve to meet the demands of the rich hardware features that expose multiple levels of parallelism.

II. Motivation

Developers and researchers often create new HPC programming model abstractions for various reasons, such as accommodating new programming paradigms, introducing optimizations, addressing known gaps in existing models, and coping with the evolution of hardware architectures.

Clang-LLVM offers the environment to address many of the aforementioned issues for enhancing the models within C++. However, despite offering several options, such as modifying Clang's source, Libtooling tools, Clang plugins, and LLVM passes, tools developed within the environment also have drawbacks and limitations.

Clang plugins and Libtooling tools can only perform source-to-source changes or static analysis on C/C++/Obj-C code, as these methods revolve around Clang's immutable AST. Thus, by design, the source code needs to be rewritten and reparsed to manipulate it -which presents other limitations, or the tool must limit itself to perform static analysis.

LLVM passes can modify and manipulate the IR produced by Clang, thus overcoming one of the critical limitations of plugins and tools. However, LLVM-IR has no high-level information about the language and its semantics, limiting the type of possible manipulations.

One challenge remains: none of the above techniques can introduce new language constructs. Modifying Clang's source code allows the introduction of new language constructs; however, this modification process could be more convenient and less error-prone for newcomers. Compiling these source code modifications might take significant time, depending on the system, reducing productivity. Furthermore, none of the modifications can be easily shared, as it requires patching and rebuilding the source code, creating another fatal drawback in the approach.

Finally, one of the significant drawbacks of all the mentioned options is that none offer effective high-level manipulations of the IR. An MLIR front-end would solve this last problem; however, the struggle to introduce new constructs remains.

All the issues mentioned above strongly motivated us to create ***XBLang***, a programming language and compiler designed for extensibility based on MLIR, capable of addressing architecture evolution thanks to the power of the MLIR infrastructure.

III. Related work

The CETUS compiler infrastructure [15] is a source-to-source compiler targeting the C language, typically used for research on compiler optimizations targeting multi-core architectures. It uses ANTLR as its parser, permits transformation passes, and can auto-parallelize certain constructs. In their paper, the authors mention that one of the reasons for not targeting C++ is its difficulty to parse.

The OpenARC compiler [16], [19] is a source-to-source research compiler built using the CETUS compiler. It implements the full OpenACC 1.0 specification and a subset of the 2.0 specification, thus capable of targeting accelerators such as GPUs via the OpenACC programming model.

The ROSE compiler infrastructure [20] is another source-to-source compiler capable of targeting C and C++ -up to C++11, to create program transformation and analysis tools. It provides a unified AST as its intermediate representation for applying compiler analyses, optimizations, and transformations.

As mentioned in Section II, Clang can act as a source-to-source compiler through Libtooling tools or Clang plugins. For example, several projects [21]–[24] use it for such purposes. However, they all suffer from the drawbacks we have mentioned. Other source-to-source compilers include Mercurium [25] and Insieme [26].

Furthermore, in the survey article [27], the authors seek to answer why and in what contexts HPC practitioners avoided source-to-source transformation tools. Their approach consisted of a survey of papers that intended to use a source-to-source tool but argued against using them. According to the survey, some of the main reasons why practitioners avoided using these tools include the tools needing to be easier to extend to support new programming models; also, there are still challenges associated with effectively parsing the source language.

The COMET compiler [28], a Domain Specific Language, is built using MLIR, for sparse and dense tensor algebra computations. It is leveraging MLIR at multiple stages to perform high and low-level transformations, such as transforming Tensor contractions into Transpose-Transpose-GEMM-Transpose operations. It also supports execution in multiple targets, including CPUs and GPUs.

Polygeist [29], [30] is an MLIR front-end for a subset of C/C++ designed to leverage MLIR's representation power and existing dialects for performing polyhedral optimization and parallel code transpilation, achieving excellent performance in both cases. It uses Clang Libtooling API to process the C++ input. However, its functionality is limited to a specific function specified as a command line argument.

Halide [31], [32] is a programming language used for high-performance image and array processing. Programs are written

in C++ using Halide's API rather than being a standalone language. The API then creates an internal representation, with the compiler transforming this IR to increase performance and then lowering it to LLVM IR. It supports multiple CPU and GPU architectures.

Exo [33] is a domain-specific language that uses the principle of exocompilation: externalizing target-specific code generation support and optimization policies to user-level code, i.e., the developer decides which optimizations to perform and when. An LLVM/MLIR is currently under construction.

Mojo [34] is a programming language introduced by the company Modular. Mojo is a Python superset capable of delivering up to 35000x speedup over Python. It uses MLIR for its internal representation, enabling it to perform optimizations at a higher level. However, GPU support is still under development, with no performance results [35].

We have also been closely following other up-and-coming models such as Carbon [36], an experimental programming language and an experimental replacement to C++, introduced by Google in 2022, and Triton [37], a language for facilitating the writing of efficient Deep-Learning primitives, being developed by OpenAI and using MLIR for its IR.

The crucial difference between ***XBLang*** and other programming languages is ***XBLang***'s extensibility, powered by MLIR. This feature pretends to simplify the programming language while allowing extensions to enrich it. We demonstrate the power of extensibility with the **par** extension, a general-purpose portable parallel programming model.

IV. Language

This section presents ***XBLang*** and its extensibility properties. Furthermore, we present **par**, a language extension for writing portable parallel programs capable of yielding high performance.

The syntax of ***XBLang*** 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. Declarations such as structs, functions, and variable definitions follow Rust's syntax style. The semantics of ***XBLang*** are those of C, plus some extensions such as type inference and a module system instead of C includes.

In ***XBLang***, we use the term "Language Context" to describe a set of semantic and syntactic rules that govern how a particular language construct works. Each construct of ***XBLang*** has a language context, which can have multiple parent and child contexts, with child contexts following the rules set by their parent contexts. For example, typed and named declarations govern function declarations, forcing functions to have a well-defined type and a name.

XBLang uses language dialects to extend its functionality. Each dialect encapsulates a specific set of language constructs and provides the rules for creating the language context for each construct. Dialects can introduce any elements necessary for their semantic and syntactic purposes and determine what is legal inside a context owned by the dialect.

The only overall restriction imposed on dialects is that any side effects produced by a dialect construct must always have a valid semantic meaning that the parent context can understand. Failing this restriction is considered an error, e.g., inside an ***XBLang*** statement, all child contexts must follow statement semantics, i.e., at the top level, they are ordered and structured.

Syntactically, language contexts can be created by specifying the dialect's keyword or by invoking the desired dialect construct directly, provided the dialect registers the construct within the parent context and there are no syntactic conflicts. Listing 1 shows this mechanism; in this listing, the **par** keyword establishes the intent of creating a **par** construct, like `map` or `region`. In contrast, the `loop` construct does not need to be surrounded by the **par** keyword, as it is registered globally in ***XBLang***.

As mentioned before, dialects are free to define their syntax and semantics inside a particular language context; Listing 2 shows the usage of the `id` and `dim` constructs, as they do not have a valid semantic or syntactic meaning outside a **par** region context. However, inside a valid context, they provide thread ID information.

```
1 fn saxpy_xblang(x: f64*, y: f64*, a: f64, n: i32)
2 {
3   par map(toFrom: x[0 : n]) map(to: y[0 : n])
4   par region firstprivate(a, n)
5   loop(let i: i32 in 0 : n)
6     x[i] = a * y[i] + x[i];
7 }
```

Listing 1: ***XBLang*** high level version of the SAXPY kernel.

```
1 fn saxpy_xblang(x: f64*, y: f64*, a: f64, n: i32)
2 {
3   par map(toFrom: x[0 : n]) map(to: y[0 : n])
4   par region firstprivate(a, n) {
5     let i: i32 = id<L2:> + id<L2> * dim<L2:>;
6     if (i < n)
7       x[i] = a * y[i] + x[i];
8   }
9 }
```

Listing 2: ***XBLang*** low level version of the SAXPY kernel.

A. **par** Programming Model

We designed the ***XBLang*** **par** dialect to be similar to commonly used programming models, such as OpenMP and OpenACC, so new developers could quickly learn and use it.

However, unlike directive-based programming models such as OpenMP and OpenACC created to convert a sequential program into a parallel program, the **par** dialect adopts the philosophy that users are responsible for constraining parallelism within the model, not the model constraining users.

The complete set of currently available clauses in the **par** dialect is presented in Listing 3. Their definition is close to the definitions of similarly named OpenMP clauses. In many cases, code written using the clauses in Listing 3 will result in portable code across CPUs and GPUs, demonstrated in Section VII.

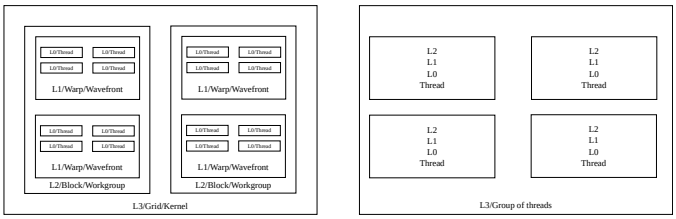

Figure 1: Mapping of the thread hierarchy system in **par** to hardware. The mapping for GPU offloading is on the left, while the diagram on the right shows the mapping for CPU architectures.

```
1 map[[queue]]?([[clause [, clause...]]])
2 region[[queue]]?[[clause [, clause...]]]
3 loop[[clause [, clause...]]]
4 atomic(op: lvalue, rvalue)
5 reduce(op: [lvalue, [, lvalue...]])
6 sync
7 lead
8 wait
```

Listing 3: A complete list of clauses available in **par**.

The execution model of the **par** dialect is organized into parallelism levels, namely L0, L1, L2, and L3. Higher levels have more parallel resources, while lower levels have less or no parallel resources. Figure 1 shows the mapping of this hierarchy to hardware; it also demonstrates that only the L3 level provides parallelism in the CPU model, with L1 and L2 levels considered *degenerate* providing no parallelism.

The model also defines that constructs executing in *degenerate levels* execute sequentially and always have `id` 0 and `dim` 1. This definition is the key to allowing portability across CPU and GPU and enabling the creation of sequential code.

The `region` construct is the primary method for exposing parallelism in the dialect. These regions always launch an L3 level of threads, with all threads executing the same region. In a CPU, this mechanism is similar to calling `pthread_create`, 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 `loop` construct. 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; this is shown in Listing 4, where the top `loop` runs at the L3 (`<L3>`) level. In contrast, the nested `loop` expands over L2, L1 and L0 (`<L2:>`).

```
1 fn smpv(q: f64*, A: f64*, y: f64*, rowPtr: i32*,
      colIndx: i32*, n: i32) {
2  par region firstprivate(n)
3  loop<L3>(let i: i32 in 0 : n) {
4   let sum: f64 = 0.;
5   loop<L2:>[reduce(sum)](let col: i32 in rowPtr[i]
        : rowPtr[i + 1])
6     sum += A[colIndx[col]] * y[col];
7   lead q[i] = sum;
8  }
9 }
```

Listing 4: Work distribution for an SPMV kernel in ***XBLang***.

In Listings 1 - 5, we compare programming models and languages using the SAXPY kernel as reference code. Listings 1 & 2 represent two possible implementations within **par** for the kernel. The first one represents a high-level version similar to the OpenMP version in listing 5, with the `loop` construct distributing the work automatically into all available parallelism levels. The second version expresses the same idea but uses low-level constructs like `id` and `dim`; this version is akin to a typical CUDA kernel. Both listings show the memory mapping mechanism in **par**; internally, a custom memory manager calling CUDA or HIP handles this mapping.

```
1 void saxpy_omp(double *x, double *y, double a, int
      n) {
2 #pragma omp target loop map(tofrom: x[0 : n]) \
3                         map(to: y[0 : n])
4   for (int i = 0; i < n; ++i)
5     x[i] = a * y[i] + x[i];
6 }
```

Listing 5: OpenMP version of the SAXPY kernel.

V. Multi-Level Intermediate Representation (MLIR)

This section presents a general overview of MLIR, going through some of the key concepts required to understand Section VI-B, as we chose MLIR to create the compiler's middle-end due to its extensibility properties and vast existing infrastructure.

A. Overview

The Multi-Level IR (MLIR) project [38] is an extensible compiler infrastructure capable of representing arbitrary *graph-like* IRs, with *graph-like* meaning that IR operations and IR values form a graph structure. MLIR employs a hierarchical structure to represent IRs, allowing for extensibility and modularity in the representation. The key IR concepts in MLIR are:

- *Regions* are ordered list of *Blocks*, allowing modularity, e.g., the body of a function can be represented by a *Region*.
- A *Block* is an ordered list of operations, and in SSACFG *Regions*, they represent compiler basic blocks in an SSA style. One crucial distinction to traditional basic blocks is that in MLIR *Blocks* can have arguments, allowing to

perform control flow between *Blocks* without the nuances of *Phi* nodes [39].

- A *Value* is a unique typed result produced by an *Operation*, being the edges communicating between operations.

Additionally, MLIR defines an open type system used by *Values*, with the semantics of the types being "application-specific" [39]. MLIR also has the concept of *Attributes* and *Properties* for storing additional information in operations and types. Some examples are integer constants or symbol names. Unlike other intermediate representations like LLVM IR, MLIR allows for symbols, providing a non-SSA procedure to refer to operations, e.g., a function or a global variable.

To allow for extensibility and modularity, MLIR introduces the concept of dialects where *Operations*, *Types*, *Attributes*, and other concepts with a common purpose are grouped to provide semantics and a programming interface. From the implementation standpoint, the creation of dialects also simplifies parsing and printing the IR as it encapsulates the information on how to perform the actions.

MLIR dialects MLIR provides a set of core dialects as a starting point for interacting with the infrastructure. These dialects are maintained and developed by the MLIR community; all are under active development and have ample infrastructure built around them, e.g., optimization passes and IR transformations. The `builtin` dialect defines basic infrastructure, like integer and floating point types, and higher-level types, like `memref`, intended to describe general memory references, like ranked or unranked tensors in memory. Other relevant dialects are: `arith` & `math` for arithmetic and math operations, `func` for defining and interacting with functions, `cf` & `scf` for basic and structured control flow, and `affine` for affine operations like loops.

VI. Compiler

This section explores each of the stages of our compiler alongside relevant implementation details. Figure 2 presents a high-level representation of the compilation workflow. This figure shows the compilation process from an input program to LLVM IR, with Clang compiling the IR into an executable. We chose to leave the executable generation to Clang as it natively handles LLVM IR, can call vendor tools for device code compilation, and can handle more complicated compilation tasks such as Link Time Optimization (LTO).

In the following subsections, we explore the front-end and MLIR middle-end of the compiler and explain how to extend the compiler by introducing new language constructs.

A. Front-end

We use LLVM TableGen to specify almost every aspect of XBlang language dialects front-end, with custom TableGen backends generating the necessary code and hooks to interact with the rest of the compiler. This design decision allows the use of all the existing machinery inside the TableGen language, further facilitating the extension of the compiler.

To make the compiler dependent only on the LLVM and MLIR projects and as self-contained as possible, we opted against using existing Lex & Parser generators, such as Flex [40], Bison [41], or ANTLR [42], for specifying the grammar. Instead, we use self-made generators capable of producing classical DFA lexers and Packrat [43] parsers. Any dialect can use the lexer generator to generate a complete DFA or specific routines to lex tokens, simplifying the introduction of intricate tokens, as shown in Listing 6.

```
1  class FloatLiteral<string suffix> {
2   list<TokenRule> toks = [
3    TokenRule<"FloatLiteral", [], [{(
4      (digit_sequence [eE] [+\-]? digit_sequence)
5    | (digit_sequence '.' ([eE] [+\-]?
         digit_sequence)?)
6    | (digit_sequence? '.' digit+ ([eE] [+\-]?
         digit_sequence)?))}]>
7    ];
8    list<TokenRule> tokens = !foreach(tok, toks,
9      TokenRule<tok.identifier # suffix, [],
10       tok.rule # ruleSuffix>);
11 }
12 def FloatLiteral<"f32">; // 0.f32, .32f32
13 def FloatLiteral<"f64">; // 1.e-10f64, 1.32f64
```

Listing 6: Tablegen class for recognizing float literals with a suffix. A `TokenRule` specifies the creation of a lexing rule associated with a named token.

Dialect grammars are specified in a custom Parsing Expression Grammar (PEG) format, allowing the specification of complex grammars with little effort. This format permits the introduction of C++ code actions and includes macro expressions. Parse macros enable describing a common concept once and employing it multiple times later, diminishing code bloating, e.g., the `SepList(expr, sep)` macro expands into a list of zero or more `expr` separated by `sep`.

Dialects are also free to provide their lexer, parser, or specific lexing or parsing routines as long as they comply with the compiler interface. For example, the ***XBLang*** dialect provides a custom method for parsing binary expressions, using operator precedence parsing to speed up the parsing of expressions.

Dialects specify AST nodes through TableGen, providing an easy mechanism for introducing new constructs. Listing 7 presents a simplified version of such a node specification for the **`par`** region construct; in this listing, the `RegionStmt` belongs to the **`par`** dialect; it inherits the properties found in the `Stmt` node, defines a symbol table and has multiple children, like a statement body. The syntax for the construct is specified through the format field, allowing a unified front-end specification; for parsing a `RegionStmt`, the construct has an optional `LaunchStmt` and a required body statement.

The structure of the Abstract Syntax Tree (AST) used by the front-end mirrors, to some extent, that of Clang's, grouping classes of related semantic objects and forming a class hierarchy, providing a helpful interface for interacting with it. This hierarchy uses type safeness as the first semantic checker mechanism, as inserting a node of type `Decl` into a node that expects a `Stmt` for a particular field is impossible.

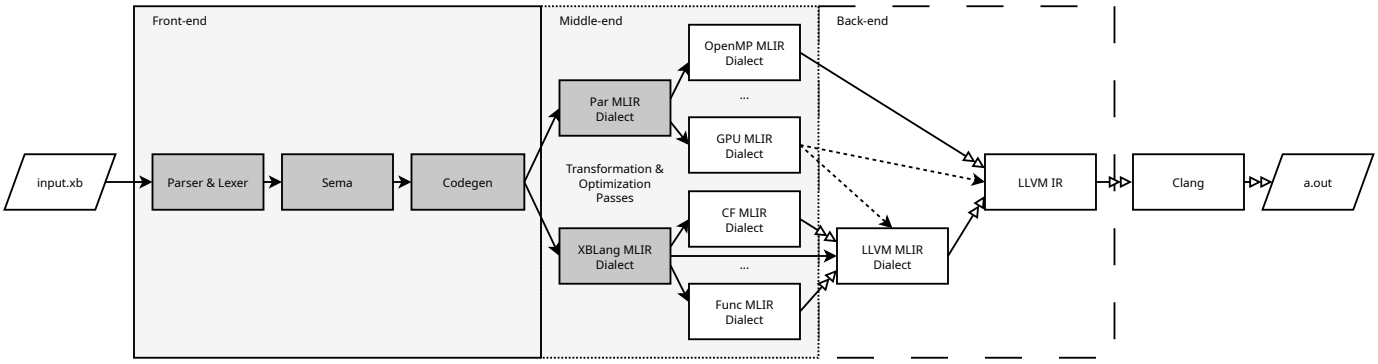

Figure 2: Compilation workflow of the ***XBLang*** compiler. Rectangles colored in shades of gray and black arrowheads represent the contributions of this paper. Dashed arrows indicate contributions to the MLIR community resulting from this work. Double arrowheads indicate parts of the workflow that we did not change for this work. Please refer to Section V-A for an MLIR overview.

```
def ParRegionStmt: ParNode<"RegionStmt", Stmt, [
    SymbolTable]> {
 let leaves = (ins "LaunchStmt":$launchParameters,
                   ..., "Stmt":$body);
 let format = [{ 'region' EE
  (LaunchStmt:stmt { result.setLaunchParameters(
      stmt); })? ...
  Stmt:body { result.setBody(body); }
 }];
}
```

Listing 7: Tablegen record for the `par region` construct, showing its children and syntax. Blocks enclosed in curly braces inside the `format` field denote C++ code actions; the remainder of the field represents syntax.

The *Sema* stage will perform semantic checks on the AST, name resolution, and type inference for certain constructs. It works by traversing the AST recursively and verifying each of the nodes in the AST in the case of a semantic error, which aborts further checks and compilation. Finally, the code generation stage handles the creation of the initial ***XBLang*** & **`par`** MLIR IR from the AST representation; for more details see Section VI-B. Dialects can easily accommodate new constructs by providing the necessary semantic or code-generation hooks.

In this subsection, we demonstrated how to extend the front-end to accommodate new language constructs. In the following subsection, we present the MLIR middle-end.

B. MLIR middle-end

We built the middle-end of the ***XBLang*** compiler using the MLIR compiler infrastructure; see Figure 2 for a general overview of the workflow and contributions. The middle-end relies on the **`xb`** and **`par`** MLIR dialects; these are the direct contributions of this work. It is organized into 6 compilation stages, discussed later in this section.

The **`xb`** MLIR dialect is one of the main contributions of this work. This dialect provides operations required to represent ***XBLang***'s source code as a high-level IR, particularly suitable for applying high-level transformations. Many of the operations of this dialect represent generalized versions of operations in trunk MLIR; this is particularly true for the control flow operations in ***XBLang***. For example, there are no operations modeling loops with **`break`** conditions in trunk MLIR; however, they are available in **`xb`**. Another great example is the **`xb.`**`bop` operation, as it allows the model of general binary operations, even between operands of different types.

We also created compatibility layers so the **`xb`** dialect is interoperable with trunk MLIR dialects. For example, the cast operation allows converting **`xb`** values with pointer or reference types to MLIR `memref`s. All trunk control flow operations can operate inside **`xb`** control flow operations. However, the opposite is not true; **`xb`** operations containing return operations might not work inside `scf` or `affine` operations.

The only passes and patterns from trunk MLIR used by the middle-end stages are the *canonicalize* and *cse* passes and patterns used for conversion, like *convert-gpu-to-nvvm* or *convert-cf-to-llvm*.

Another critical contribution of this work is the **`par`** MLIR dialect, capable of representing both portable and non-portable parallel programs generated by the front-end. Depending on the compilation target, the middle-end converts operations from this dialect to operations in either ***XBLang***, `omp`, or `gpu` dialects.

To further explain the compilation stages used by the middle-end, the code appearing in Listing 8 will be used as a reference. This code represents a simple parallel vector fill function, with subsequent listing showing the parallelization for GPUs; an ellipsis in a listing indicates elided code.

Listing 9 shows the MLIR generated by the *CodeGen* stage. This first representation uses mostly operations from the **`xb`** and **`par`** dialects. This listing shows that the **`xb`** MLIR dialect provides a close representation of the source code in Listing 8, with easily identifiable variable declarations (**`xb.`**`var`) as well

```
1 fn fill(x: f64*, v: f64, n: i32) {
2  par map(toFrom: x[0 : n])
3  par region([1], [1]) firstprivate(v, n)
4  loop (let i: i32 in 0 : n)
5   x[i] = v;
6 }
```

Listing 8: Parallel fill vector function written in ***XBLang***, MLIR representations of this code appear in listings 9 & 10.

as parallel regions (**par.**region) and loops (**par.**loop).

```
1  xb.func @fill(%arg0: !xb.ptr<f64>, %arg1: f64, %
       arg2: si32) {
2   %0 = xb.constant(1 : si32) : si32
3   %1 = xb.constant(0 : si32) : si32
4   %2 = par.default_queue !xb.address
5   %x = xb.var[param] @x : !xb.ptr<f64> = %arg0
6   %v = xb.var[param] @v : f64 = %arg1
7   %n = xb.var[param] @n : si32 = %arg2
8   %3 = xb.range %1 "<" %n !xb.ref<si32> : !xb.
       range_t<si32>
9   %4 = xb.array_view %x !xb.ref<!xb.ptr<f64>> [%3 !
       xb.range_t<si32>] : tensor<?xf64>
10  par.data_region map(toFrom: %4 : tensor<?xf64> ->
        %x : !xb.ref<!xb.ptr<f64>> [%2:!xb.address])
        {
11   par.region firstprivate(%v : !xb.ref<f64>, %n :
       !xb.ref<si32>) {
12    %5 = xb.cast %n : !xb.ref<si32> -> si32
13    par.loop (%i : si32 in %1 : %0 : %5) {
14     %6 = xb.array %x !xb.ref<!xb.ptr<f64>> [%i :
         si32] : !xb.ref<f64>
15     %7 = xb.bop "=" %6 !xb.ref<f64>, %v !xb.ref<
         f64> : !xb.ref<f64>
16    }
17   }
18  }
19  xb.return
20 }
```

Listing 9: MLIR code generated by the *CodeGen* stage for the code in Listing 8.

We now present the middle-end stages. Each stage is composed of multiple passes grouped to accomplish a given goal; for example, the lowering stages transform higher-level constructs into lower-level ones.

1) High-level transformations: The output from *CodeGen* is the input for the first middle-end stage, known internally as high-level transformations. In this stage, the **par** dialect applies its first set of transformations, explicitly placing memory mappings in the IR, transforming L3 reduction clauses into an L2 reduction and an atomic operation, and privatizing variables, amongst others.

2) Concretization: The concretization stage is one of the main stages in the pipeline, as it concretizes the IR generated by the high-level stage into an IR suitable for lowering to core MLIR dialects. Listing 10 shows the output of this stage. One of the fundamental roles of this stage is the introduction of implicit casting operations, like loading memory - see line 14 in the listing. Another critical transformation performed at this stage is type promotion, ensuring that binary operations have the same operand type after this stage.

In this stage, the **par** dialect will make explicit any runtime calls, collapse nested loops, and distribute loops across the parallel hierarchy. During this stage, the compiler will replace mapped memory values inside parallel regions with the appropriate value.

```
1  xb.func @fill(%arg0: !xb.ptr<f64>, %arg1: f64, %
       arg2: si32) {
2   ...
3   %x = xb.var[param] @x : !xb.ptr<f64> = %arg0
4   ...
5   %10 = xb.call @__xblangMapData(%1, %6, %0, %9,
        %4) : (ui32, !xb.address, index, index, !xb.
        address) -> !xb.address<#gpu.address_space<
        global>>
6   %11 = xb.cast %10 : !xb.address<#gpu.
        address_space<global>> -> !xb.ptr<f64, #gpu.
        address_space<global>>
7   %12 = xb.cast %v : !xb.ref<f64> -> f64
8   %13 = xb.cast %n : !xb.ref<si32> -> si32
9   par.region {
10   %x_0 = xb.var[local] @x : !xb.ptr<f64, #gpu.
         address_space<global>> = %11
11   %v_1 = xb.var[local] @v : f64 = %12
12   %n_2 = xb.var[local] @n : si32 = %13
13   xb.scope {
14    %20 = xb.cast %n_2 : !xb.ref<si32> -> si32
15    %21 = par.id l3_l0 0
16    %22 = par.dim l3_l0 0
17    %23 = xb.bop "+" %21 si32, %3 si32 : si32
18    %i = xb.var[local] @i : si32
19    %24 = xb.bop "=" %i !xb.ref<si32>, %23 si32 : !
          xb.ref<si32>
20    xb.loop condition: {
21     %25 = xb.cast %i : !xb.ref<si32> -> si32
22     %26 = xb.bop "<" %25 si32, %20 si32 : i1
23     xb.yield Fallthrough %26 i1
24    } body : {
25     %25 = xb.cast %x_0 : !xb.ref<!xb.ptr<f64, #gpu
           .address_space<global>>> -> !xb.ptr<f64, #
           gpu.address_space<global>>
26     ...
27     %28 = xb.array %25 !xb.ptr<f64, #gpu.
           address_space<global>> [%27 index] : !xb.
           ref<f64, #gpu.address_space<global>>
28     %29 = xb.cast %v_1 : !xb.ref<f64> -> f64
29     %30 = xb.bop "=" %28 !xb.ref<f64, #gpu.
           address_space<global>>, %29 f64 : !xb.ref<
           f64, #gpu.address_space<global>>
30    } iteration : {
31     %25 = xb.cast %i : !xb.ref<si32> -> si32
32     %26 = xb.bop "+" %25 si32, %22 si32 : si32
33     %27 = xb.bop "=" %i !xb.ref<si32>, %26 si32 :
           !xb.ref<si32>
34    }
35   }
36  }
37  ...
38  %19 = xb.call @__xblangMapData(%2, %15, %0, %18,
        %4) : (ui32, !xb.address, index, index, !xb.
        address) -> !xb.address<#gpu.address_space<
        global>>
39 }
```

Listing 10: IR representation obtained after applying the concretization stage.

3) High lowering: This stage performs the first lowering pass of the compiler, converting many of **xb**'s operations into operations in standard MLIR dialects, such as **xb.**`var` to `memref.alloca`; all these lowering transformations are direct contributions of this work. Listing 11 shows a partial output of this stage. An important thing to observe from the listing is that not all operations get lowered at this stage; for example, casting, scope, loop operations, and types from the ***XBLang*** dialect remain.

There are multiple reasons for performing a partial instead of a complete lowering; the first reason is that not all operations in ***XBLang*** have an adequate equivalent operation in trunk high-level dialects, like **xb.**`loop`. The second reason is that the `memref` dialect models well-defined references to memory regions, not raw pointers. Thus, we overcome this restriction by providing casts between ***XBLang*** pointers and `memrefs` (see line 15 in the listing) and model raw pointers inside the `memref` dialect as nearly infinite `memrefs` (see line 16).

At this stage, the **par** dialect lowers into the `gpu` for GPU parallelism or the `omp` dialect for CPU parallelism.

```
1  func.func @fill(%arg0: !xb.ptr<f64>, %arg1: f64, %
       arg2: i32) {
2   ...
3   %alloca = memref.alloca() : memref<!xb.ptr<f64>>
4   memref.store %arg0, %alloca[] : memref<!xb.ptr<
       f64>>
5   ...
6   %11 = gpu.launch async blocks ... threads ... {
7    ...
8    xb.scope {
9     %17 = memref.load %alloca_4[] : memref<i32>
10    %18 = gpu.global_id  x
11    %19 = index.casts %18 : index to i32
12    %20 = gpu.grid_dim  x
13    %21 = gpu.block_dim  x
14    %22 = index.mul %20, %21
15    %23 = index.casts %22 : index to i32
16    %alloca_5 = memref.alloca() : memref<i32>
17    memref.store %19, %alloca_5[] : memref<i32>
18    xb.loop condition : {
19     %24 = memref.load %alloca_5[] : memref<i32>
20     %25 = arith.cmpi slt, %24, %17 : i32
21     xb.yield Fallthrough %25 i1
22    } body ...
23   }
24   gpu.terminator
25  }
26  ...
27  %16 = func.call @__xblangMapData(...)
28  xb.return
29 }
```

Listing 11: Lowered version of the IR from the Listing 10.

4) Low transforms: Low transforms involve transformations in lower-level dialects and translating the `gpu` dialect into LLVM IR. One example of such transformation is transforming `gpu.allreduce` operations into a combination of shuffle operations and `gpu.barrier` operations, MLIR already has an existing transformation for this operation. However, it does not apply to AMDGPU targets, and it is computationally expensive as it has little information on the semantic nature of the reduction; for example, a reduction clause attached to the `loop` might not need broadcast its result to the neighboring threads.

5) LLVM lowering: This stage is the final MLIR stage and the second lowering stage. It is in this stage that all ***XBLang*** operations are fully lowered into the `llvm` MLIR dialect, making explicit all the casts, flattening scopes, and fully expanding **xb.**`loops` into explicit control flow operations.

6) LLVM Translation: After the code gets lowered to the `llvm` MLIR dialect, the final stage is the invocation of the translation infrastructure in MLIR, translating the `llvm` MLIR dialect into LLVM IR, with LLVM IR being the final output of the compiler.

VII. Results

The raw files of the results presented have been uploaded in Zenodo [44]. To demonstrate the efficacy of ***XBLang***, we use a number of case studies. Among them are two 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 at LBNL and Frontier at ORNL for evaluation purposes. All tests were run 10 times and averaged out; 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 100 and AMD MI250x on Perlmutter and Frontier. For all benchmarks and platforms, we verified the output correctness of the programs generated by ***XBLang*** using the built-in validation mechanism for validating their output.

Fig. 3 shows results using Perlmutter. The compilers used on Perlmutter include Clang 18.0.0 (8823e961), GCC OpenMP offloading 12.1.1, Clang OpenMP offloading 18.0.0 (8823e961), nvc OpenMP offloading 23.5.0, Cray OpenMP offloading 15.0.1, nvc OpenACC 23.5.0 and our ***XBLang*** compiler. We observe that ***XBLang*** performs the best for EP and CG even compared to vendor compilers; for XSBench ***XBLang*** performs close to the other compilers targeting GPU with *clang-omp* performing the best; for mini-weather ***XBLang*** performs almost close to the rest besides *clang-omp* that performs the worst; for RSBench & XSBench ***XBLang*** shows there is room for improvement.

Fig. 4 presents results using Frontier. The compilers used on Frontier include Clang 18.0.0, (e816c89c) Clang OpenMP offloading 18.0.0 (e816c89c), ROCM's 5.7.0 AMD Clang, Cray OpenMP offloading 16.0.1 and our ***XBLang*** compiler. We observe that ***XBLang*** performs better than all other compilers in 3 benchmarks: EP, CG, and Mini-Weather; for RSBench ***XBLang*** performs close to the best-performing compiler AMD-Clang; for XSBench, the figure shows there is room for improvement in ***XBLang***. After profiling RSBench and XSBench using *rocprof*, we became aware that ***XBLang***'s kernels appear to be faster than the other compilers, with the

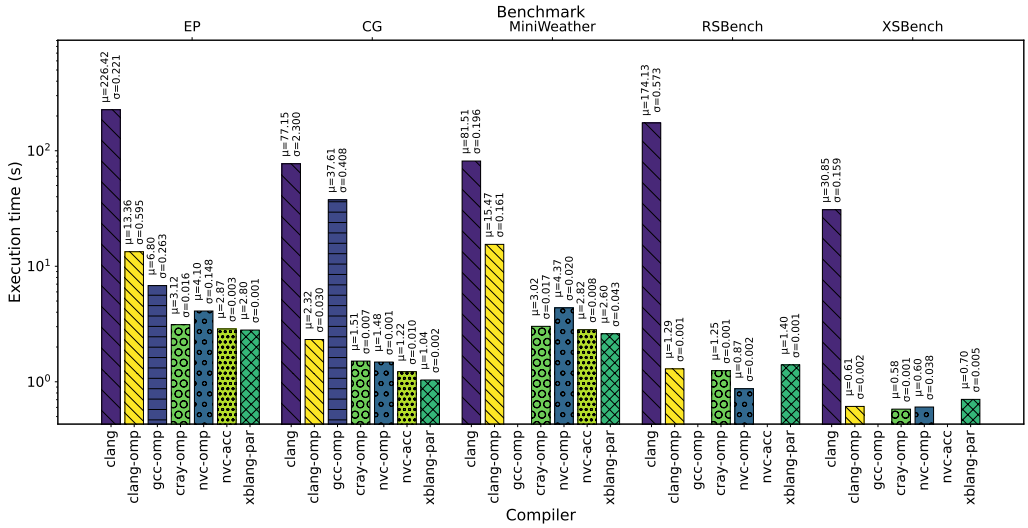

Figure 3: **Lower the better:** Execution time for five benchmarks on Perlmutter NVIDIA A100 GPUs comparing **XBLang** with other compilers; clang (first bar) represents the sequential run while the rest represent GPU runs; *gcc-omp* offloading for RSBench failed to compile; *gcc-omp* offloading for MiniWeather and XSBench 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, while σ is the standard deviation

performance downgrade coming from ***XBLang***'s runtime for mapping memory to and back from the device.

We investigated why ***XBLang*** performs so much better in specific programs like CG. For this, we profiled both Clang and ***XBLang*** 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, ***XBLang***'s version of the kernel is 2.9 times faster than Clang's version. After looking at the LLVM IR generated by Clang and speaking to LLVM OpenMP developers, we can confirm that ***XBLang***'s implementation of reductions is the primary source for overperformance.

Fig. 5 presents results using Perlmutter CPU-only nodes. The compilers used on Perlmutter include Clang 18.0.0 (8823e961), GCC OpenMP offloading 12.2.0, Clang OpenMP offloading 18.0.0 (8823e961), nvc OpenMP offloading 23.5.0, Cray OpenMP offloading 15.0.1, nvc OpenACC 23.5.0 and our ***XBLang*** compiler. We observe that ***XBLang*** performs similarly to Clang in 3 of the 5 benchmarks, while ***XBLang***'s CG and MiniWeather performance is not as good. After further analysis, we concluded that the reason for ***XBLang***'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.

XBLang **on 64-bit ARM architecture:** We evaluated ***XBLang*** on a Fujitsu A64fx processor; this processor was used to build the Fugaku Supercomputer in Japan. For full specs of the test machine, please refer to [45]. Since the difference in results between a traditional X86 and an A64fx was not significant enough, we have not discussed them in detail here, but for further reference on Wombat results, please refer to [44].

In summary, results show that ***XBLang*** 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 ***XBLang*** 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.

VIII. Conclusions and Future Work

We introduce a new programming language ***XBLang***, an extensible compiler infrastructure, and a parallel programming model targeting compiler developers developing new programming language abstractions. Promising results show potential for this novel direction of compiler research. As immediate future steps, we will also create a translator to convert a subset of conventional C codes to ***XBLang*** so that any programmer can seamlessly port their codes to our language without needing to go through the learning curve of understanding a new language. We will also introduce meta-programming capabilities, templates, and other high-level constructs to ***XBLang*** programming language and increase the number of constructs available in the `par` dialect. We will evaluate the updated language and compiler with more tests and mini-apps.

References

[1] OpenMP ARB, "OpenMP 5.2," 2020. [Online]. Available: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf

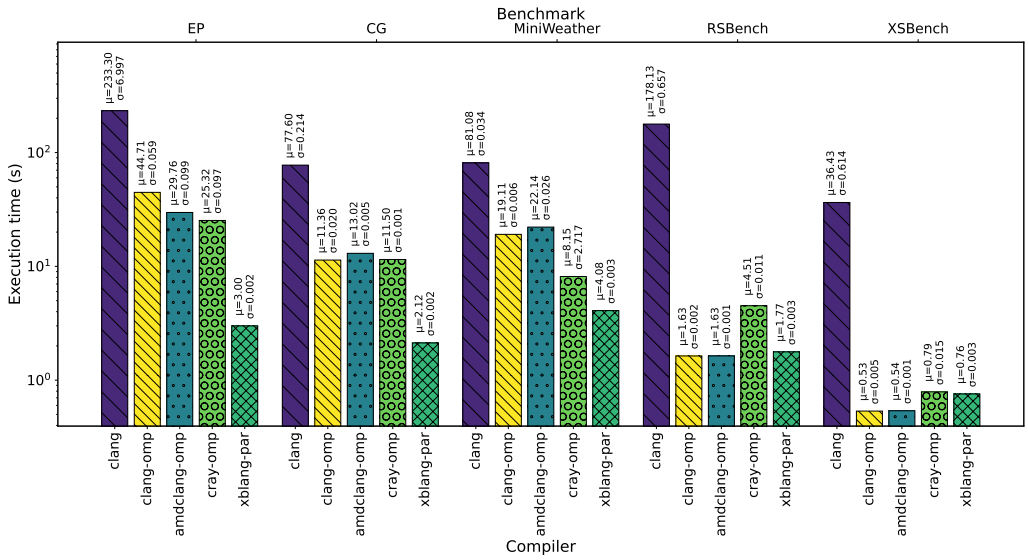

Figure 4: **Lower the better:** Execution time for five benchmarks on Frontier AMD MI250x GPUs comparing **XBLang** with other compilers; clang (first bar) represents the sequential run while the rest represent GPU runs. μ represents the average execution time, while σ is the standard deviation

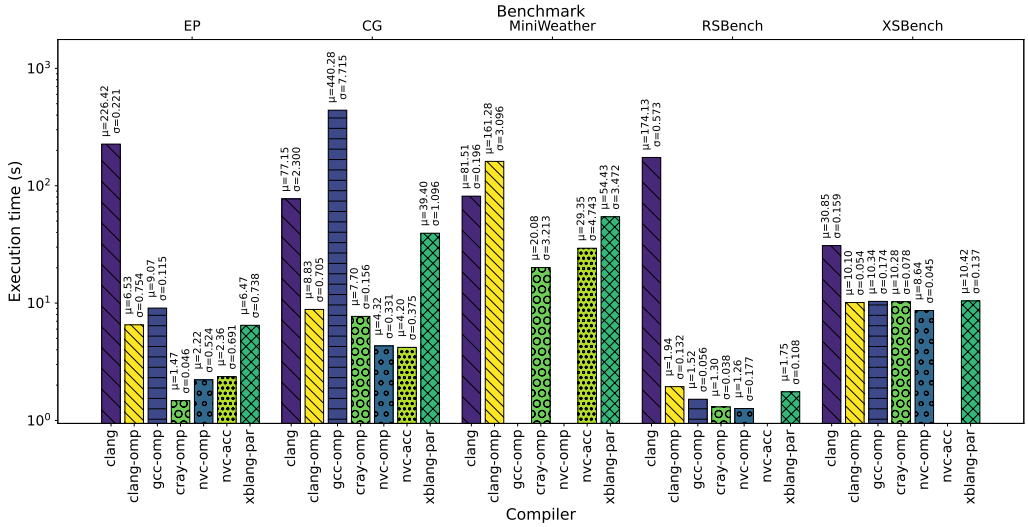

Figure 5: **Lower the better:** Execution time for five benchmarks on Perlmutter's AMD EPYC 7763 CPUs comparing **XBLang** with other compilers; clang (first bar) represents the sequential run while the rest represent multi-core runs; *gcc-omp* and *nvc-omp* for MiniWeather failed to execute; RSBench and XSBench do not have an OpenACC equivalent hence *nvc-acc* bar does not exist. μ represents the average execution time, while σ is the standard deviation

[2] OpenACC, "OpenACC 3.2," 2020. [Online]. Available: https://www.openacc.org/sites/default/files/inline-images/Specification/OpenACC-3.1-final.pdf

[3] H. C. Edwards and C. R. Trott, "Kokkos: Enabling performance portability across manycore architectures," in *2013 Extreme Scaling Workshop (xsw 2013)*. IEEE, 2013, pp. 18–24.

[4] T. Huber, S. Pophale, N. Baker, M. Carr, N. Rao, J. Reap, K. Holsapple, J. H. Davis, T. Burnus, S. Lee, D. E. Bernholdt, and S. Chandrasekaran, "Ecp sollve: Validation and verification testsuite status update and compiler insight for openmp," in *2022 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2022, pp. 123–135.

[5] I. Karlin, T. Scogland, A. C. Jacob, S. F. Antao, G.-T. Bercea, C. Bertolli, B. R. d. Supinski, E. W. Draeger, A. E. Eichenberger, J. Glosli *et al.*, "Early experiences porting three applications to openmp 4.5," in *International Workshop on OpenMP*. Springer, 2016, pp. 281–292.

[6] G. Juckeland, O. Hernandez, A. C. Jacob, D. Neilson, V. G. V. Larrea, S. Wienke, A. Bobyr, W. C. Brantley, S. Chandrasekaran, M. Colgrove *et al.*, "From describing to prescribing parallelism: Translating the spec accel openacc suite to openmp target directives," in *International Conference on High Performance Computing*. Springer, 2016, pp. 470–488.

[7] R. Gayatri, C. Yang, T. Kurth, and J. Deslippe, "A case study for performance portability using openmp 4.5," in *International Workshop on Accelerator Programming Using Directives*. Springer, 2018, pp. 75–95.

[8] J. M. Diaz, S. Pophale, K. Friedline, O. Hernandez, D. E. Bernholdt, and S. Chandrasekaran, "Evaluating support for openmp offload features," in *Proceedings of the 47th International Conference on Parallel Processing Companion*, 2018, pp. 1–10.

[9] J. H. Davis, C. Daley, S. Pophale, T. Huber, S. Chandrasekaran, and N. J. Wright, "Performance assessment of openmp compilers targeting nvidia v100 gpus," in *International Workshop on Accelerator Programming Using Directives*. Springer, 2020, pp. 25–44.

[10] C. Daley, H. Ahmed, S. Williams, and N. Wright, "A case study of porting hpgmg from cuda to openmp target offload," in *International Workshop on OpenMP*. Springer, 2020, pp. 37–51.

[11] B. Chapman, B. Pham, C. Yang, C. Daley, C. Bertoni, D. Kulkarni, D. Oryspayev, E. D'Azevedo, J. Doerfert, K. Zhou *et al.*, "Outcomes of openmp hackathon: Openmp application experiences with the offloading model (part ii)," in *OpenMP: Enabling Massive Node-Level Parallelism: 17th International Workshop on OpenMP, IWOMP 2021, Bristol, UK, September 14–16, 2021, Proceedings 17*. Springer, 2021, pp. 81–95.

[12] S. Bak, C. Bertoni, S. Boehm, R. Budiardja, B. M. Chapman, J. Doerfert, M. Eisenbach, H. Finkel, O. Hernandez, J. Huber *et al.*, "Openmp application experiences: porting to accelerated nodes," *Parallel Computing*, vol. 109, p. 102856, 2022.

[13] T. M. Evans, A. Siegel, E. W. Draeger, J. Deslippe, M. M. Francois, T. C. Germann, W. E. Hart, and D. F. Martin, "A survey of software implementations used by application codes in the exascale computing project," *The International Journal of High Performance Computing Applications*, vol. 36, no. 1, pp. 5–12, 2022. [Online]. Available: https://doi.org/10.1177/10943420211028940

[14] W.-C. Lin and S. McIntosh-Smith, "Comparing julia to performance portable parallel programming models for hpc," in *2021 International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. IEEE, 2021, pp. 94–105.

[15] C. Dave, H. Bae, S.-J. Min, S. Lee, R. Eigenmann, and S. Midkiff, "Cetus: A source-to-source compiler infrastructure for multicores," *Computer*, vol. 42, no. 12, pp. 36–42, 2009.

[16] S. Lee and J. S. Vetter, "Openarc: Extensible openacc compiler framework for directive-based accelerator programming study," in *2014 First Workshop on Accelerator Programming Using Directives*. IEEE, 2014, pp. 1–11.

[17] N. D. Matsakis and F. S. Klock, "The rust language," *ACM SIGAda Ada Letters*, vol. 34, no. 3, pp. 103–104, 2014.

[18] G. VanRossum and F. L. Drake, *The python language reference*. Python Software Foundation Amsterdam, Netherlands, 2010.

[19] S. Lee and J. S. Vetter, "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*, 2014, pp. 115–120.

[20] D. Quinlan and C. Liao, "The ROSE source-to-source compiler infrastructure," in *Cetus users and compiler infrastructure workshop, in conjunction with PACT*, vol. 2011. Citeseer, 2011, p. 1.

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

[22] X. Wu, M. Kruse, P. Balaprakash, H. Finkel, P. Hovland, V. Taylor, and M. Hall, "Autotuning polybench benchmarks with llvm clang/polly loop optimization pragmas using bayesian optimization (extended version)," *arXiv preprint arXiv:2104.13242*, 2021.

[23] G. D. Balogh, G. R. Mudalige, I. Z. Reguly, S. Antao, and C. Bertolli, "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, 2018, pp. 59–70.

[24] M. R. Gadelha, J. Morse, L. Cordeiro, and D. Nicole, "Using clang as a frontend on a formal verification tool," 2017.

[25] J. Balart, A. Duran, M. Gonzàlez, X. Martorell, E. Ayguadé, and J. Labarta, "Nanos mercurium: a research compiler for openmp," in *Proceedings of the European Workshop on OpenMP*, vol. 8, 2004, p. 56.

[26] P. Gschwandtner, J. J. Durillo, and T. Fahringer, "Multi-objective autotuning with insieme: Optimization and trade-off analysis for time, energy and resource usage," in *European Conference on Parallel Processing*. Springer, 2014, pp. 87–98.

[27] R. Milewicz, P. Pirkelbauer, P. Soundararajan, H. Ahmed, and T. Skjellum, "Negative perceptions about the applicability of source-to-source compilers in hpc: A literature review," in *International Conference on High Performance Computing*. Springer, 2021, pp. 233–246.

[28] R. Tian, L. Guo, J. Li, B. Ren, and G. Kestor, "A high performance sparse tensor algebra compiler in mlir," in *2021 IEEE/ACM 7th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*, 2021, pp. 27–38.

[29] W. S. Moses, L. Chelini, R. Zhao, and O. Zinenko, "Polygeist: Raising c to polyhedral mlir," in *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2021, pp. 45–59.

[30] W. S. Moses, I. R. Ivanov, J. Domke, T. Endo, J. Doerfert, and O. Zinenko, "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*, ser. PPoPP '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 119–134. [Online]. Available: https://doi.org/10.1145/3572848.3577475

[31] J. M. Ragan-Kelley, "Decoupling algorithms from the organization of computation for high performance image processing," Ph.D. dissertation, Massachusetts Institute of Technology, 2014.

[32] J. Ragan-Kelley, A. Adams, D. Sharlet, C. Barnes, S. Paris, M. Levoy, S. Amarasinghe, and F. Durand, "Halide: Decoupling algorithms from schedules for high-performance image processing," *Commun. ACM*, vol. 61, no. 1, p. 106–115, dec 2017. [Online]. Available: https://doi.org/10.1145/3150211

[33] Y. Ikarashi, G. L. Bernstein, A. Reinking, H. Genc, and J. Ragan-Kelley, "Exocompilation for productive programming of hardware accelerators," in *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ser. PLDI 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 703–718. [Online]. Available: https://doi.org/10.1145/3519939.3523446

[34] Modular Inc, "Mojo programming manual," 2023. [Online]. Available: https://docs.modular.com/mojo/programming-manual.html

[35] ——, "Modular docs - ai engine faq," 2023. [Online]. Available: https://docs.modular.com/engine/faq.html#performance

[36] Google, "Google brands carbon language as "experimental successor to c++"," 2022. [Online]. Available: https://devclass.com/2022/07/20/google-brands-carbon-language-as-experimental-successor-to-c/

[37] P. Tillet, H. T. Kung, and D. Cox, "Triton: An intermediate language and compiler for tiled neural network computations," in *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, ser. MAPL 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 10–19. [Online]. Available: https://doi.org/10.1145/3315508.3329973

[38] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "Mlir: Scaling compiler infrastructure for domain specific computation," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2021, pp. 2–14.

[39] "Mlir language reference," 2023. [Online]. Available: https://mlir.llvm.org/docs/LangRef

[40] W. Estes, "Lexical analysis with flex, for flex 2.6.2: Top," 2017. [Online]. Available: https://westes.github.io/flex/manual/

[41] F. S. Foundation, "Bison - gnu project - free software foundation," 2021. [Online]. Available: https://www.gnu.org/software/bison/

[42] T. J. Parr and R. W. Quong, "Antlr: A predicated-ll(k) parser generator," *Software: Practice and Experience*, vol. 25, no. 7, pp. 789–810, 1995. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380250705

[43] B. Ford, "Packet parsing: a practical linear-time algorithm with backtracking," Ph.D. dissertation, Massachusetts Institute of Technology, 2002.

[44] Anonymous, "MLIR compiler for XBLang: An extensible programming language," Oct. 2023. [Online]. Available: https://doi.org/10.5281/zenodo.8412122

[45] Oak Ridge National Lab, "Wombat." [Online]. Available: https://www.olcf.ornl.gov/olcf-resources/compute-systems/wombat/