The syntax dialect: creating a parser generator with MLIR

Fabian Mora-Cordero (U. of Delaware, USA)
fmorac@udel.edu

Dr. Sunita Chandrasekaran (U. of Delaware, USA)

Thursday 24th October, 2024

Outline

Motivation

- My PhD dissertation explored how to create more modular compilers and languages
- Leading the work to explore dynamic parsing combinators for easily extending language syntax
- None of the existing tools provided me the required flexibility
- So I built one

```
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
   ''';
  }
 }
}
```

Program for printing thread ID information from a GPU

Motivation

- My PhD dissertation explored how to create more modular compilers and languages
- Leading the work to explore dynamic parsing combinators for easily extending language syntax
- None of the existing tools provided me the required flexibility
- So I built one

```
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
   ''';
  }
 }
}
```

Program for printing thread ID information from a GPU

Motivation

- My PhD dissertation explored how to create more modular compilers and languages
- Leading the work to explore dynamic parsing combinators for easily extending language syntax
- None of the existing tools provided me the required flexibility
- So I built one

```
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
   ''';
  }
 }
}
```

Program for printing thread ID information from a GPU

Motivation

- My PhD dissertation explored how to create more modular compilers and languages
- Leading the work to explore dynamic parsing combinators for easily extending language syntax
- None of the existing tools provided me the required flexibility
- So I built one

```
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], [gszl]>> {
   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
   ’’’;
  }
 }
}
```

Program for printing thread ID information from a GPU

The syntax dialect: IR

- Provides a high-level description of formal grammars
- Incorporates useful high-level constructs like functions over syntax expressions
- Has dedicated operations for lexical and syntax analysis

```
syntax.macro @Interleave(%arg0: !syntax.expr,
    %arg1: !syntax.expr) -> !syntax.expr {
  %and = and %arg1, %arg0
  %zom = zero_or_more %and
  %and_0 = and %arg0, %zom
  return %and_0
}
// a(,a)*
syntax.rule @rule {
  %a = terminal #syntax.literal<"a">
  %comma = terminal #syntax.literal<",">
  %ret = call @Interleave(%a, %comma)
  return %ret
}
// mlir-opt --inline
syntax.rule @rule{
  %a = terminal #syntax.literal<"a">
  %comma = terminal #syntax.literal<",">
  %and = and %a, %comma
  %zom = zero_or_more %and
  %and_1 = and %a, %zom
  return %and_1
}
```

Interleave macro example

The syntax dialect: IR

- Provides a high-level description of formal grammars
- Incorporates useful high-level constructs like functions over syntax expressions
- Has dedicated operations for lexical and syntax analysis

```
syntax.macro @Interleave(%arg0: !syntax.expr,
    %arg1: !syntax.expr) -> !syntax.expr {
  %and = and %arg1, %arg0
  %zom = zero_or_more %and
  %and_0 = and %arg0, %zom
  return %and_0
}
// a(,a)*
syntax.rule @rule {
  %a = terminal #syntax.literal<"a">
  %comma = terminal #syntax.literal<",">
  %ret = call @Interleave(%a, %comma)
  return %ret
}
// mlir-opt --inline
syntax.rule @rule{
  %a = terminal #syntax.literal<"a">
  %comma = terminal #syntax.literal<",">
  %and = and %a, %comma
  %zom = zero_or_more %and
  %and_1 = and %a, %zom
  return %and_1
}
```

Interleave macro example

The syntax dialect: IR

- Provides a high-level description of formal grammars
- Incorporates useful high-level constructs like functions over syntax expressions
- Has dedicated operations for lexical and syntax analysis

```
syntax.macro @Interleave(%arg0: !syntax.expr,
    %arg1: !syntax.expr) -> !syntax.expr {
  %and = and %arg1, %arg0
  %zom = zero_or_more %and
  %and_0 = and %arg0, %zom
  return %and_0
}
// a(,a)*
syntax.rule @rule {
  %a = terminal #syntax.literal<"a">
  %comma = terminal #syntax.literal<",">
  %ret = call @Interleave(%a, %comma)
  return %ret
}
// mlir-opt --inline
syntax.rule @rule{
  %a = terminal #syntax.literal<"a">
  %comma = terminal #syntax.literal<",">
  %and = and %a, %comma
  %zom = zero_or_more %and
  %and_1 = and %a, %zom
  return %and_1
}
```

Interleave macro example

The syntax dialect: IR

- Most operations are Pure or constant-like
- We can use CSE to optimize redundancy

```
// Rule for (ab | abc)
syntax.rule @rule {
  %a = terminal #syntax.literal<"a">
  %b = terminal #syntax.literal<"b">
  %c = terminal #syntax.literal<"c">
  %ab = and %a, %b
  %tmp = and %a, %b
  %abc = and %tmp, %c
  %ret = or %ab, %abc
  return %ret
}
// mlir-opt --cse
syntax.rule @rule{
  %a = terminal #syntax.literal<"a">
  %b = terminal #syntax.literal<"b">
  %c = terminal #syntax.literal<"c">
  %and = and %a, %b
  %and_2 = and %and, %c
  %or = or %and, %and_2
  return %or
}
```

Employing CSE for grammar simplification

The syntax dialect: IR

- Most operations are Pure or constant-like
- We can use CSE to optimize redundancy

```
// Rule for (ab | abc)
syntax.rule @rule {
  %a = terminal #syntax.literal<"a">
  %b = terminal #syntax.literal<"b">
  %c = terminal #syntax.literal<"c">
  %ab = and %a, %b
  %tmp = and %a, %b
  %abc = and %tmp, %c
  %ret = or %ab, %abc
  return %ret
}
// mlir-opt --cse
syntax.rule @rule{
  %a = terminal #syntax.literal<"a">
  %b = terminal #syntax.literal<"b">
  %c = terminal #syntax.literal<"c">
  %and = and %a, %b
  %and_2 = and %and, %c
  %or = or %and, %and_2
  return %or
}
```

Employing CSE for grammar simplification

Lexical analysis with syntax

Generating lexers: TableGen backend

- We created a TableGen backend to construct lexical analyzers
- The MLIR code then gets analyzed and optimized and then translated to C++
- The lexer generator is based on traditional DFA's

```
// Rule in TableGen
def IdentifierStx :
  Rule<"Identifier", "[a-zA-Z_] [a-zA-Z_0-9]*">;
// MLIR generated by the tablegen backend
syntax.dfa @Lexer {
 rule @Identifier  {
  %t0 = terminal #syntax.char_class<"A-Z_a-z">
  %t1 = terminal #syntax.char_class<"0-9A-Z_a-z">
  %zom = zero_or_more %t1
  %a = and %t0, %zom
  return %a
 }
}
```

Lexer for an identifier

Generating lexers: TableGen backend

- We created a TableGen backend to construct lexical analyzers
- The MLIR code then gets analyzed and optimized and then translated to C++
- The lexer generator is based on traditional DFA's

```
// Rule in TableGen
def IdentifierStx :
  Rule<"Identifier", "[a-zA-Z_] [a-zA-Z_0-9]*">;
// MLIR generated by the tablegen backend
syntax.dfa @Lexer {
 rule @Identifier  {
  %t0 = terminal #syntax.char_class<"A-Z_a-z">
  %t1 = terminal #syntax.char_class<"0-9A-Z_a-z">
  %zom = zero_or_more %t1
  %a = and %t0, %zom
  return %a
 }
}
```

Lexer for an identifier

Generating lexers: TableGen backend

- We created a TableGen backend to construct lexical analyzers
- The MLIR code then gets analyzed and optimized and then translated to C++
- The lexer generator is based on traditional DFA's

```
// Rule in TableGen
def IdentifierStx :
  Rule<"Identifier", "[a-zA-Z_] [a-zA-Z_0-9]*">;
// MLIR generated by the tablegen backend
syntax.dfa @Lexer {
 rule @Identifier  {
  %t0 = terminal #syntax.char_class<"A-Z_a-z">
  %t1 = terminal #syntax.char_class<"0-9A-Z_a-z">
  %zom = zero_or_more %t1
  %a = and %t0, %zom
  return %a
 }
}
```

Lexer for an identifier

Lexical transformations

- We begin with inlining, canonicalizing, and applying CSE
- We transform the IR into a DFA
- We perform traditional DFA minimization
- We generate C++ for the IR

```
syntax.dfa @Main {
 rule @Identifier  {
  %t0 = terminal #syntax.char_class<"A-Z_a-z">
  %t1 = terminal #syntax.char_class<"0-9A-Z_a-z">
  %zom = zero_or_more %t1
  %a = and %t0, %zom
  return %a
 }
}
```

Lexer for an identifier

Lexical transformations

- We begin with inlining, canonicalizing, and applying CSE
- We transform the IR into a DFA
- We perform traditional DFA minimization
- We generate C++ for the IR

```
dfa @Main {
  %eps = eps
  %terminal = terminal #syntax.char_class<"A-Z_a-z">
  %terminal_0 = terminal #syntax.char_class<"0-9A-Z_a-z">
  lex_state @StateMain0 {
    lex_transition %terminal -> @StateMain1
  }
  lex_state @StateMain1 {
    lex_transition %terminal_0 -> @StateMain1
    lex_transition %eps -> @StateMain2
  }
  lex_state final @StateMain2 id = @Identifier
}
```

Code after minimizing the DFA

Lexical transformations

- We begin with inlining, canonicalizing, and applying CSE
- We transform the IR into a DFA
- We perform traditional DFA minimization
- We generate C++ for the IR

```
dfa @Main {
  %eps = eps
  %terminal = terminal #syntax.char_class<"A-Z_a-z">
  %terminal_0 = terminal #syntax.char_class<"0-9A-Z_a-z">
  lex_state @StateMain0 {
    lex_transition %terminal -> @StateMain1
  }
  lex_state @StateMain1 {
    lex_transition %terminal_0 -> @StateMain1
    lex_transition %eps -> @StateMain2
  }
  lex_state final @StateMain2 id = @Identifier
}
```

Code after minimizing the DFA

Lexical transformations

- We begin with inlining, canonicalizing, and applying CSE
- We transform the IR into a DFA
- We perform traditional DFA minimization
- We generate C++ for the IR

```
Lexer::TokenID
Lexer::lexMain(SourceState &state, SourceLocation &beginLoc,
               llvm::StringRef &spelling) const {
StateMain0: {
  if (((/*A*/ (65 <= *state) && (*state <= 90) /*Z*/) ||
      *state == 95 /*_*/ ||
      (/*a*/ (97 <= *state) && (*state <= 122) /*z*/)) {
    state.advance();
    goto StateMain1;
  }
  return Invalid;
}
StateMain1: {
  if (((/*0*/ (48 <= *state) && (*state <= 57) /*9*/) ||
      (/*A*/ (65 <= *state) && (*state <= 90) /*Z*/) ||
      *state == 95 /*_*/ ||
      (/*a*/ (97 <= *state) && (*state <= 122) /*z*/)) {
    state.advance();
    goto StateMain1;
  }
  spelling = getSpelling(beginLoc, state.getLoc());
  return Identifier;
}
  return Invalid;
}
```

Syntax analysis with syntax

Generating parsers: TableGen backend

- We created a TableGen backend to construct syntax analyzers
- The MLIR code then gets analyzed and optimized and then translated to C++
- The TableGen backend generates a Packrat-PEG parser
- It has native support for dynamic combinators

```
def Dim3: ParserMacro<"Dim3", ["expr"], [{
  expr ("," expr ("," expr)? )?
}]>;
/*example:
region<<[n], [32, 16]>> {
  // statement
}
*/
def GPU_RegionStx:
    Production<"Region", "::mlir::Operation*"> {
  let rule = [{
    /*parse launch bounds*/
    "region" "<<"
      "[" @Dim3(#dyn("Expr"):$bv { bsz.push_back(
            bv.get()); }) "]"
      ","
      "[" @Dim3(#dyn("Expr"):$gv { gsz.push_back(
            gv.get()); }) "]"
    ">>"
    /*parse region statement*/
    #dyn("Stmt")
  }];
}
```

TableGen code for parsing a GPU region

Generating parsers: TableGen backend

- We created a TableGen backend to construct syntax analyzers
- The MLIR code then gets analyzed and optimized and then translated to C++
- The TableGen backend generates a Packrat-PEG parser
- It has native support for dynamic combinators

```
def Dim3: ParserMacro<"Dim3", ["expr"], [{
  expr ("," expr ("," expr)? )?
}]>;
/*example:
region<<[n], [32, 16]>> {
  // statement
}
*/
def GPU_RegionStx:
    Production<"Region", "::mlir::Operation*"> {
  let rule = [{
    /*parse launch bounds*/
    "region" "<<"
      "[" @Dim3(#dyn("Expr"):$bv { bsz.push_back(bv.get()); }) "]"
      ","
      "[" @Dim3(#dyn("Expr"):$gv { gsz.push_back(gv.get()); }) "]"
    ">>"
    /*parse region statement*/
    #dyn("Stmt")
  }];
}
```

TableGen code for parsing a GPU region

Generating parsers: TableGen backend

- We created a TableGen backend to construct syntax analyzers
- The MLIR code then gets analyzed and optimized and then translated to C++
- The TableGen backend generates a Packrat-PEG parser
- It has native support for dynamic combinators

```
def Dim3: ParserMacro<"Dim3", ["expr"], [{
  expr ("," expr ("," expr)? )?
}]>;
/*example:
region<<[n], [32, 16]>> {
  // statement
}
*/
def GPU_RegionStx:
    Production<"Region", "::mlir::Operation*"> {
  let rule = [{
    /*parse launch bounds*/
    "region" "<<"
      "[" @Dim3(#dyn("Expr"):$bv { bsz.push_back(
           bv.get()); }) "]"
      ","
      "[" @Dim3(#dyn("Expr"):$gv { gsz.push_back(
           gv.get()); }) "]"
    ">>"
    /*parse region statement*/
    #dyn("Stmt")
  }];
}
```

TableGen code for parsing a GPU region

Generating parsers: TableGen backend

- We created a TableGen backend to construct syntax analyzers
- The MLIR code then gets analyzed and optimized and then translated to C++
- The TableGen backend generates a Packrat-PEG parser
- It has native support for dynamic combinators

```
def Dim3: ParserMacro<"Dim3", ["expr"], [{
  expr ("," expr ("," expr)? )?
}]>;
/*example:
region<<[n], [32, 16]>> {
  // statement
}
*/
def GPU_RegionStx:
    Production<"Region", "::mlir::Operation*"> {
  let rule = [{
    /*parse launch bounds*/
    "region" "<<"
      "[" @Dim3(#dyn("Expr"):$bv { bsz.push_back(
            bv.get()); }) "]"
      ","
      "[" @Dim3(#dyn("Expr"):$gv { gsz.push_back(
            gv.get()); }) "]"
    ">>"
    /*parse region statement*/
    #dyn("Stmt")
  }];
}
```

TableGen code for parsing a GPU region

Syntax transformations

- We take our TableGen spec and transform it into `syntax`
- We then perform inlining, canonicalize, and apply CSE
- We analyze the grammar and lower its complexity
- We generate C++ for the IR

```
def Top : Production<"Top", "::mlir::Value"> {
  let rule = [{
    ("a" "b") |
    ("a" "c") |
    ("c" "d")
  }];
}
def Syntax : Parser<"Syntax", XBLangLexer> {
  let macros = [];
  let productions = [
    Top,
  ];
  let startSymbol = "Top";
  let cppNamespace = "::";
  let defaultToken = "Identifier";
}
```

Syntax specification

Syntax transformations

- We take our TableGen spec and transform it into `syntax`
- We then perform inlining, canonicalize, and apply CSE
- We analyze the grammar and lower its complexity
- We generate C++ for the IR

```
syntax.parser @Syntax start = @Top {
 rule @Top  {
  %Identifier = terminal #syntax.lex_terminal<@Identifier, unk, "a">
  %md = md_node %Identifier @_0
  %Identifier_0 = terminal #syntax.lex_terminal<@Identifier, unk, "b">
  %md_1 = md_node %Identifier_0 @_1
  %and = and %md, %md_1
  %Identifier_2 = terminal #syntax.lex_terminal<@Identifier, unk, "a">
  %md_3 = md_node %Identifier_2 @_0
  %Identifier_4 = terminal #syntax.lex_terminal<@Identifier, unk, "c">
  %md_5 = md_node %Identifier_4 @_1
  %and_6 = and %md_3, %md_5
  %or = or %and, %and_6
  %Identifier_7 = terminal #syntax.lex_terminal<@Identifier, unk, "c">
  %md_8 = md_node %Identifier_7 @_0
  %Identifier_9 = terminal #syntax.lex_terminal<@Identifier, unk, "d">
  %md_10 = md_node %Identifier_9 @_1
  %and_11 = and %md_8, %md_10
  %or_12 = or %or, %and_11
  return %or_12
 }
}
```

Generated syntax code

Syntax transformations

- We take our TableGen spec and transform it into `syntax`
- We then perform inlining, canonicalize, and apply CSE
- We analyze the grammar and lower its complexity
- We generate C++ for the IR

```
syntax.parser @Syntax start = @Top {
 rule @Top  {
  %Identifier = terminal #syntax.lex_terminal<@Identifier, unk, "a">
  %md = md_node %Identifier @_0
  %Identifier_0 = terminal #syntax.lex_terminal<@Identifier, unk, "b">
  %md_1 = md_node %Identifier_0 @_1
  %and = and %md, %md_1
  %Identifier_2 = terminal #syntax.lex_terminal<@Identifier, unk, "a">
  %md_3 = md_node %Identifier_2 @_0
  %Identifier_4 = terminal #syntax.lex_terminal<@Identifier, unk, "c">
  %md_5 = md_node %Identifier_4 @_1
  %and_6 = and %md_3, %md_5
  %or = or %and, %and_6
  %Identifier_7 = terminal #syntax.lex_terminal<@Identifier, unk, "c">
  %md_8 = md_node %Identifier_7 @_0
  %Identifier_9 = terminal #syntax.lex_terminal<@Identifier, unk, "d">
  %md_10 = md_node %Identifier_9 @_1
  %and_11 = and %md_8, %md_10
  %or_12 = or %or, %and_11
  return %or_12
 }
}
```

Generated syntax code

Syntax transformations

- We take our TableGen spec and transform it into `syntax`
- We then perform inlining, canonicalize, and apply CSE
- We analyze the grammar and lower its complexity
- We generate C++ for the IR

```
rule @Top  attributes {first_set = [#syntax.lex_terminal<@Identifier,
      unk, "c">, #syntax.lex_terminal<@Identifier, unk, "a">]} {
 %Identifier = terminal #syntax.lex_terminal<@Identifier, unk, "d">
 %Identifier_0 = terminal #syntax.lex_terminal<@Identifier, unk, "c">
 %Identifier_1 = terminal #syntax.lex_terminal<@Identifier, unk, "b">
 %Identifier_2 = terminal #syntax.lex_terminal<@Identifier, unk, "a">
 %md = md_node %Identifier_2 @_0
 %md_3 = md_node %Identifier_1 @_1
 %seq = seq %md, %md_3
 %md_4 = md_node %Identifier_2 @_0
 %md_5 = md_node %Identifier_0 @_1
 %seq_6 = seq %md_4, %md_5
 %md_7 = md_node %Identifier_0 @_0
 %md_8 = md_node %Identifier @_1
 %seq_9 = seq %md_7, %md_8
 %any = any %seq, %seq_6 first_sets = [[#syntax.lex_terminal<
      @Identifier, unk, "a">], [#syntax.lex_terminal<@Identifier, unk
      , "a">]] conflicts = [[#syntax.lex_terminal<@Identifier, unk, "
      a">], [#syntax.lex_terminal<@Identifier, unk, "a">]]
 %switch = switch %any, %seq_9 first_sets = [[#syntax.lex_terminal<
      @Identifier, unk, "a">], [#syntax.lex_terminal<@Identifier, unk
      , "c">]]
 return %switch
}
```

Future work: JIT compiling syntax

- We are currently working on JIT compiling `syntax`
  - Allowing the creation of LLVM-optimized regex recognizers on the fly
  - Allowing the creation of self-extensible languages
- The biggest barrier for JIT compilation is transforming the associated runtime library to be less C++ reliant

Future work: JIT compiling syntax

- We are currently working on JIT compiling `syntax`
  - Allowing the creation of LLVM-optimized regex recognizers on the fly
  - Allowing the creation of self-extensible languages
- The biggest barrier for JIT compilation is transforming the associated runtime library to be less C++ reliant

Future work: JIT compiling syntax

- We are currently working on JIT compiling `syntax`
  - Allowing the creation of LLVM-optimized regex recognizers on the fly
  - Allowing the creation of self-extensible languages
- The biggest barrier for JIT compilation is transforming the associated runtime library to be less C++ reliant

Future work: JIT compiling syntax

- We are currently working on JIT compiling `syntax`
  - Allowing the creation of LLVM-optimized regex recognizers on the fly
  - Allowing the creation of self-extensible languages
- The biggest barrier for JIT compilation is transforming the associated runtime library to be less C++ reliant

Summary

- `syntax` allows to represent and create lexical analyzers
- `syntax` can represent syntax analyzers
- It shows how traditional techniques such as CSE can be applied in non-traditional contexts
- This approach proves once more how MLIR can be used in many areas beyond traditional compilation

Summary

- `syntax` allows to represent and create lexical analyzers
- `syntax` can represent syntax analyzers
- It shows how traditional techniques such as CSE can be applied in non-traditional contexts
- This approach proves once more how MLIR can be used in many areas beyond traditional compilation

Summary

- `syntax` allows to represent and create lexical analyzers
- `syntax` can represent syntax analyzers
- It shows how traditional techniques such as CSE can be applied in non-traditional contexts
- This approach proves once more how MLIR can be used in many areas beyond traditional compilation

Summary

- `syntax` allows to represent and create lexical analyzers
- `syntax` can represent syntax analyzers
- It shows how traditional techniques such as CSE can be applied in non-traditional contexts
- This approach proves once more how MLIR can be used in many areas beyond traditional compilation

Acknowledgments

Acknowledgments

- This material is based upon work supported by the National Science Foundation (NSF) under grant no. 1814609.
- This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.
- This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility located at Lawrence Berkeley National Laboratory, operated under Contract No. DE-AC02-05CH11231.
- This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

Questions?