CIL : Common MLIR abstraction for C / C++/ Fortran

Vinay/Ranjith/Prashantha/Srihari

Compiler Tree Technologies

CIL: Introduction and Status

- Middle level IR written as MLIR dialect.
- Comes in between AST and LLVM IR.
- Current focus is C / C++ / Fortran.
- Target / ABI independent.
- C/C++ input is parsed using clang AST plugin and converted to CIL.
- Fortran is parsed by FC compiler and converted to CIL.
- Working LTO Framework.
- Working Loop Nest Optimizer and Data Layout optimizations.
- SPEC CPU 2017 : Passes 2 C, 2 Fortran and 1 C++ benchmarks.

Why CIL?

- Flang is planning to use FIR, a MLIR dialect, for middle end optimizations.
- C/C++ remain without middle end optimizer.
- There is a necessity to design a common IR to share analysis and transformations across C/C++/Fortran.

Why CIL?

- LLVM is too low level: C++ STL constructs needs higher level IR to optimize effectively.
- Loop Transforms and Multi-dimensional arrays needs to analyzed accurately.
- Lack of optimizations for Parallel / Heterogeneous Programming constructs like OpenMP, OpenACC, etc.
- Need ABI independent IR

Limitations of LLVM IR (1) :C++ library optimizations

Found in hot part of a SPEC CPU 2017 benchmark omnetpp:

```
#include <map>

int func(std::map<int,int> &some_map) {

  for (auto &pair : some_map) {

  }

  return 0;

}
```

LLVM doesn’t remove the dead loop

Limitations of LLVM IR (1) :C++ library optimizations

```
#include <string>
int main() {
  std::string str = "";
  if (str == "") {
    return 0;
  }
  return 1;
```

Limitations of LLVM IR (2) : Multi-dimensional arrays

- Linearized access using GEP
- Dilinearization is non-trivial effort
- Overcome by various methods like intrinsics, metadata, etc.
- Existing solutions doesn’t fit well with LLVM IR.
- Leads to poor Dependence Analysis and hence Loop optimizations.

Limitations of LLVM IR (3) : Heterogeneous computing

- Various Heterogeneous programming paradigms supported in Clang
  - OpenMP
  - OpenACC
  - OpenCL
- Not natively supported in LLVM IR
- Mostly handled using runtime calls and intrinsics.
- Cross module optimizations
- Increasing use of C / C++ in DL / RL Frameworks / libraries.

CIL - Definition

Types

- Very close correspondence with C types.
- **Target independent**
- **Scalars**: cil.int, cil.char cil.float, cil.long,...
- Support for access / type qualifiers
- **Pointers**: cil.**pointer**<cil.int>,...
- **Arrays:**
  - !cil.**array**<10 x !cil.int>
  - !cil.**array**<10 x !cil.**array**<10 x !cil.int>>
- **Derived Types:**
  - !cil.**struct**.example<!cil.int, !cil.float>>

Operations:

- Close correspondence with LLVM instructions with some exceptions:
  - Alloca and dealloca op
  - LoadOp / StoreOp
  - CILConstant
  - CILGlobalOp
  - **CILCastToMemRef**
  - **CILIfOp** (terminator operation)
- **Struct element access**:
  - `cil.struct_element %<struct_pointer>, %<field_index>`
- **Array element access**:
  - `cil.array_index %3, %4 : (!cil.pointer<!cil.array<10 x !cil.int>>, !cil.int) -> !cil.pointer<!cil.int>`
- **Pointer index access** (unbounded access via pointer type)
  - `cil.pointer_index %2, %1 : (!cil.pointer<!cil.int>, !cil.int) -> !cil.pointer<!cil.int>`

CIL Example 1:

```c
struct example {
  int a;
  int b;
};
struct example e2;
int a[10];
int sum() { return a[3] + e2.a; }
```

```
module {
  %0 = cil.global @e2 : !cil.struct.example<!cil.int, !cil.int>
  %1 = cil.global @a : !cil.array<10 x !cil.int>
  func @sum() -> !cil.int {
    %2 = cil.global_address_of @a
    %4 = cil.constant( 3 : i32 ): !cil.int
    %5 = cil.array_index %2, %4
    %6 = cil.load %5 : !cil.pointer<!cil.int> -> !cil.int
    %7 = cil.global_address_of @e2
    %8 = cil.global_address_of @e2
    %9 = cil.constant( 0 : i32 ): !cil.int
    %10 = cil.struct_element %8, %9
    %11 = cil.load %10 : !cil.pointer<!cil.int> -> !cil.int
    %12 = cil.addi %6, %11 : !cil.int
    return %12 : !cil.int
  }
}
```

Loops

```c
int a[100][100];
int b[100][100],c[100][100],i,j;
...
for(i=0; i<100; i++) {
   for(j=0; j<100; j++) {
    c[i][j] = a[i][j] + b[i][j];
   }
 }
```

```
%2 = cil.alloca !cil.array<100 x !cil.array<100 x !cil.int>>
%3 = cil.alloca !cil.array<100 x !cil.array<100 x !cil.int>>
%4 = cil.alloca !cil.array<100 x !cil.array<100 x !cil.int>>
...
cil.for_loop %arg0 = %10, %11, %12 {
  %17 = cil.constant( 0 : i32 ): !cil.int
  %18 = cil.constant( 100 : i32 ): !cil.int
  %19 = cil.constant( 1 : i32 ): i32
  cil.for_loop %arg1 = %17, %18, %19 {
    %20 = cil.array_index %4, %arg0 :
    %21 = cil.array_index %20, %arg1 : !cil.pointer<!cil.int>
    %22 = cil.array_index %2, %arg0
    %23 = cil.array_index %22, %arg1 : !cil.pointer<!cil.int>
    %24 = cil.load %23 : !cil.int
    %25 = cil.array_index %3, %arg0
    %26 = cil.array_index %25, %arg1 : !cil.pointer<!cil.int>
    %27 = cil.load %26 : !cil.int
    %28 = cil.addi %24, %27 : !cil.int
    cil.store %28, %21 : !cil.pointer<!cil.int>
    %29 = cil.load %21 : !cil.int
  }
}
```

Unstructured Control Flow

- Unstructured control flow: break , continue, goto, ..
- Nested regions would cause problems
- Needs an analysis to convert them to Loop operations like loop.for, affine.for, etc.

C++ representation in CIL

Classes in CIL

```
class A {
  public:
    int a;
    int b;
    void func() { b = a + 3; }
};
```

```
cil.class A {
    cil.field_decl a :   !cil.int
    cil.field_decl b :   !cil.int
    func @_ZN1A4funcEv() attributes {original_name="func"} {
      %2 = cil.cxx_this : !cil.pointer<!cil.class<A>>
      %3 = cil.field_access %2, @b::@A
      %4 = cil.cxx_this : !cil.pointer<!cil.class<A>>
      %5 = cil.field_access %4, @a::@A
      %6 = cil.load %5 : !cil.int
      %7 = cil.constant( 3 : i32 ): !cil.int
      %8 = cil.addi %6, %7 : !cil.int
      cil.store %8, %3 : !cil.pointer<!cil.int>
      %9 = cil.load %3 : !cil.int
      return
    }
}
```

Lowering CIL Classes to Low Level CIL

- Converting high level CIL to low level CIL

```
cil.class A {
    cil.field_decl a :   !cil.int
    cil.field_decl b :   !cil.int
    func @_ZN1A4funcEv() {
    %2 = cil.cxx_this
    %3 = cil.field_access %2, @b::@A
    %4 = cil.cxx_this
    %5 = cil.field_access %4, @a::@A
    %6 = cil.load %5 : !cil.int
    %7 = cil.constant( 3 : i32 ): !cil.int
    %8 = cil.addi %6, %7 : !cil.int
    cil.store %8, %3 : !cil.pointer<!cil.int>
    %9 = cil.load %3 : !cil.int
    return
    }
 }
```

```
func @_ZN1A4funcEv(%arg0:
!cil.pointer<!cil.struct.A<!cil.int, !cil.int>>) attributes
{original_name = "func"} {
     %2 = cil.constant( 1 : i32 ): !cil.int
     %3 = struct_element %arg0, %2 : cil.pointer<!cil.int>
     %4 = cil.constant( 0 : i32 ): !cil.int
     %5 = struct_element %arg0, %4 : !cil.pointer<!cil.int>
     %6 = cil.load %5 : !cil.int
     %7 = cil.constant( 3 : i32 ): !cil.int
     %8 = cil.addi %6, %7 : !cil.int
     cil.store %8, %3 : !cil.pointer<!cil.int>
     %9 = cil.load %3 : !cil.int
     return
  }
```

Lowering CIL Classes to LLVM

- Converting low level CIL to LLVM

```
func @_ZN1A4funcEv(%arg0:
!cil.pointer<!cil.struct.A<!cil.int, !cil.int>>)
attributes {original_name = "func"} {
      %2 = cil.constant( 1 : i32 ): !cil.int
      %3 = struct_element %arg0, %2
      %4 = cil.constant( 0 : i32 ): !cil.int
      %5 = struct_element %arg0, %4
      %6 = cil.load %5 : !cil.int
      %7 = cil.constant( 3 : i32 ): !cil.int
      %8 = cil.addi %6, %7 : !cil.int
      cil.store %8, %3 : !cil.pointer<!cil.int>
      %9 = cil.load %3 : !cil.int
      return
  }
```

```
%0 = type { i32, i32 }

; Function Attrs: nofree norecurse nounwind
define void @_ZN1A4funcEv(%0* nocapture %0)
local_unnamed_addr #0 !dbg !3 {
  %2 = getelementptr %0, %0* %0, i64 0, i32 1, !dbg !7
  %3 = getelementptr %0, %0* %0, i64 0, i32 0, !dbg !9
  %4 = load i32, i32* %3, align 4, !dbg !9
  %5 = add i32 %4, 3, !dbg !10
  store i32 %5, i32* %2, align 4, !dbg !10
  ret void, !dbg !11
}
```

Templates in CIL

```
template <class T> class A {
public:
  T a;
};

A<int> o;
A<float> o2;
```

```
cil.class A.0 {
      cil.field_decl a :   !cil.float
}
cil.class A {
      cil.field_decl a :   !cil.int
}

%2 = cil.alloca !cil.class<A> : !cil.pointer<!cil.class<A>>
%3 = cil.alloca !cil.class<A.0> : !cil.pointer<!cil.class<A.0>>
```

Operator Overloading in CIL

```
class cout {
 public:
   int dummy;
 };

 void
 operator<<(cout &obj, int val)
 {
     printf("%d \n", val);
 }
```

```
cil.class cout {
      cil.field_decl dummy :   !cil.int
   }

func @_ZlsR4couti(%arg0:
!cil.ref_pointer<!cil.class<cout>>, %arg1: !cil.int) {
      %1 = cil.alloca !cil.ref_pointer<!cil.class<cout>>
      cil.store %arg0, %1
      %2 = cil.alloca !cil.int : !cil.pointer<!cil.int>
      cil.store %arg1, %2 : !cil.pointer<!cil.int>
      %3 = cil.global_address_of @__str_tmp0
      %4 = cil.pointer_bitcast %3
      %5 = cil.load %2 : !cil.int
      %6 = cil.call @printf(%4, %5) : !cil.int,
      return
   }
```

Inheritance in CIL

```
class C {
private:
  int c;

public:
  void set_c(int val) { c = val; }
  int get_c() { return c; }
};

class B : public C {
public:
  int e;
 };
```

```
cil.class C {
     cil.field_decl c :   !cil.int
     func @_ZN1C5set_cEi(%arg0: !cil.int) {
     %1 = cil.alloca !cil.int : !cil.pointer<!cil.int>
     cil.store %arg0, %1 : !cil.pointer<!cil.int>
     %2 = cil.cxx_this : !cil.pointer<!cil.class<C>>
     %3 = cil.field_access %2, @c::@C
     %4 = cil.load %1 : !cil.int
     cil.store %4, %3 : !cil.pointer<!cil.int>
     %5 = cil.load %3 : !cil.int
     return
     }
     func @_ZN1C5get_cEv() -> !cil.int {
     %1 = cil.cxx_this : !cil.pointer<!cil.class<C>>
     %2 = cil.field_access %1, @c::@C
     %3 = cil.load %2 : !cil.int
     return %3 : !cil.int
     ^bb1:      // no predecessors
     cil.unreachable
     }
}

cil.class B inherits [!cil.class<C>] {
     cil.field_decl e :   !cil.int
}
```

STL Optimizations using CIL (1)

```
void func() {
  std::vector<int> vec;
  vec.push_back(1);
  vec.push_back(2);
  vec.push_back(3);

  vec.insert(vec.end(),{4,5,6});
}
```

```
void func() {
  std::vector<int> vec;
  vec.insert(vec.end(),{1,2,3});
  vec.insert(vec.end(),{4,5,6});
}
```

```
%15 = cil.alloca !cil.class<std::__1::vector>,
@_ZNSt3__16vectorIiNS_9allocatorIiEEEC1Ev::@vector() :
!cil.pointer<!cil.class<std::__1::vector>>
%16 = cil.constant( 1 : i32 ): !cil.int
cil.member_call %15, @_ZNSt3__16vectorIiNS_9allocatorIiEEE9push_backEOi::@vector(%16)
%17 = cil.constant( 2 : i32 ): !cil.int
cil.member_call %15, @_ZNSt3__16vectorIiNS_9allocatorIiEEE9push_backEOi::@vector(%17)
%18 = cil.constant( 3 : i32 ): !cil.int
cil.member_call %15, @_ZNSt3__16vectorIiNS_9allocatorIiEEE9push_backEOi::@vector(%18)
%19 = cil.alloca !cil.class<std::__1::__wrap_iter> :
!cil.pointer<!cil.class<std::__1::__wrap_iter>>
%20 = cil.load %19 : !cil.class<std::__1::__wrap_iter>
%21 = cil.global  { [4 : i32, 5 : i32, 6 : i32] } : !cil.class<std::initializer_list>
{sym_name = "", value = [4 : i32, 5 : i32, 6 : i32]}
```

```
%15 = cil.alloca !cil.class<std::__1::vector>, @_ZNSt3__16vectorIiNS_9allocatorIiEEEC1Ev::@vector() :
!cil.pointer<!cil.class<std::__1::vector>>
%16 = cil.global  { [1 : i32, 2 : i32, 3 : i32] } : !cil.class<std::initializer_list> {sym_name = "", value = [1 :
i32, 2 : i32, 3 : i32]}
%17 = cil.alloca !cil.class<std::__1::__wrap_iter> : !cil.pointer<!cil.class<std::__1::__wrap_iter>>
%18 = cil.load %17 : !cil.class<std::__1::__wrap_iter>
%19 = cil.member_call %15,
@_ZNSt3__16vectorIiNS_9allocatorIiEEE6insertENS_11__wrap_iterIPKiEESt16initializer_listIiE::@vector(%18, %16) :
!cil.class<std::__1::__wrap_iter.1>,
%20 = cil.alloca !cil.class<std::__1::__wrap_iter> : !cil.pointer<!cil.class<std::__1::__wrap_iter>>
%21 = cil.load %20 : !cil.class<std::__1::__wrap_iter>
%22 = cil.global  { [4 : i32, 5 : i32, 6 : i32] } : !cil.class<std::initializer_list> {sym_name = "", value = [4
: i32, 5 : i32, 6 : i32]}
```

STL Optimizations using CIL (2)

- Adding std::vector::reserve() before loop

```
void func() {
  std::vector<int> vecA;

  for(int i=0; i<10; i++) {
    vecA.push_back(i);
  }
}
```

```
void func() {
  std::vector<int> vecA;


  vecA.reserve(10);
  for(int i=0; i<10; i++) {
    vecA.push_back(i);
  }
}
```

```
^bb2: // 2 preds: ^bb1, ^bb4
  %11 = cil.load %9 : !cil.int
  %12 = cil.constant( 10 : i32 ): !cil.int
  %13 = cil.cmpi slt %11, %12 : !cil.bool
  cil.for %13, ^bb3, ^bb5
^bb3: // pred: ^bb2
  cil.member_call %7,
@_ZNSt3__16vectorIiNS_9allocatorIiEEE9push_ba
ckERKi::@vector(%9)
  br ^bb4
^bb4: // pred: ^bb3
  %14 = cil.load %9 : !cil.int
  %15 = cil.constant( 1 : i32 ): !cil.int
  %16 = cil.addi %14, %15 : !cil.int
  cil.store %16, %9 : !cil.pointer<!cil.int>
  br ^bb2
```

```
 %9 = cil.alloca !cil.int :
!cil.pointer<!cil.int>
 %10 = cil.constant( 0 : i32 ): !cil.int
 cil.store %10, %9 : !cil.pointer<!cil.int>
 br ^bb1
^bb1: // 2 preds: ^bb0, ^bb2
  %11 = cil.constant( 10 : i32 ): !cil.int
```

cil.member_call %7,
@_ZNSt3__16vectorIiNS_9allocatorIiEEE7reserveEm::@
vector(%11)

```
  %12 = cil.load %9 : !cil.int
  %13 = cil.constant( 10 : i32 ): !cil.int
  %14 = cil.cmpi slt %12, %13 : !cil.bool
  cil.for %14, ^bb2, ^bb3
^bb2: // pred: ^bb1
  cil.member_call %7,
@_ZNSt3__16vectorIiNS_9allocatorIiEEE9push_backE
RKi::@vector(%9)
  %15 = cil.load %9 : !cil.int
  %16 = cil.constant( 1 : i32 ): !cil.int
  %17 = cil.addi %15, %16 : !cil.int
  cil.store %17, %9 : !cil.pointer<!cil.int>
```

Current Progress in C++

- Major support features include:
  - **C part of C++**
  - **Templates**
  - **Inheritance**
  - **Overloading**
  - **Namespaces.**
- Considerable progress in compiling STL.
  - **Partial support for iostream, vector, string, set ,...**
- Compiles and runs 80+ cpp unit tests.
- Compiling and Running a SPEC CPU 2017 C++ benchmark

CIL and Fortran IR

CIL and Fortran

- FC - Fortran compiler, now emits CIL from high level fortran dialect.
- Multiple levels of IR.
- Optimisations at different abstractions.
- CIL is further lowered to LLVM dialect.
- All CIL optimisations can be reused
- FC emits CIL for 2 SPEC CPU 2017 benchmarks.
- Experimental LTO framework.

FC Fortran IR Example

```fortran
program hello
   print *, "hello"
 end program
```

```
module {
  fc.function @hello() -> !cil.int {
    %0 = fc.constant_string("hello^@") : !fc.array<0:5 *
!cil.char8>
    fc.print %0:!fc.array<0:5 * !cil.char8>
    %1 = cil.constant( 0 : i32 ): !cil.int
    fc.return %1
  }
}
```

Fortran IR: Nested functions

```fortran
subroutine sub1
  integer :: a
  print *, func()
  contains
  integer function func
    func = a
  end function func
end
```

```
fc.function @sub1() {
  fc.function @func() -> i32 {
    %2 = fc.get_element_ref @a::@sub1 : !fc.ref<i32>
    %3 = fc.allocate func : !fc.ref<i32>
    %4 = fc.load %2 {name = "a"} : i32
    fc.store %4, %3 {name = "func"} : !fc.ref<i32>
    %5 = fc.load %3 : i32
    fc.return %5
  }
  %0 = fc.allocate a, implicitly_captured !fc.ref<i32>
  %1 = fc.call @func::@sub1() : i32,
  fc.print %1 {arg_info = #fc.is_string< >}
  fc.return
}
```

Fortran CIL Example

```fortran
program hello
  integer :: a
  a = 10
  print *, a + 15
end program
```

```
module {
  %0 = cil.global @fc_internal_argv {sym_name =
"fc_internal_argv"} : !cil.pointer<!cil.pointer<!cil.char8>>
  %1 = cil.global @fc_internal_argc : !cil.int
  func @hello() -> !cil.int {
    %2 = cil.alloca !cil.int : !cil.pointer<!cil.int>
    %3 = cil.constant( 10 : i32 ): !cil.int
    cil.store %3, %2 : !cil.pointer<!cil.int>
    %4 = cil.load %2 : !cil.pointer<!cil.int> -> !cil.int
    %5 = cil.constant( 15 : i32 ): !cil.int
    %6 = cil.addi %4, %5 : !cil.int
    %7 = cil.constant( 3 : i32 ): !cil.int
    %8 = cil.constant( 2 : i32 ): !cil.int
    cil.call @__fc_runtime_print(%8, %7, %6)
    %9 = cil.constant( 0 : i32 ): !cil.int
    cil.return %9 : !cil.int
  }
 ...
```

CIL - LTO

- Link multiple CIL modules into single CIL module.
- Resolves functions declarations with their definitions.
- Resolves global variable declarations.
- LTO helps to implements IPOs such as inliner, data layout optimisations in CIL

CIL LTO Example

```
module {
  %0 = cil.global @__str_tmp0 {constant,
    value = " %d \0A\00"} : ...
  func @main() -> !cil.int {
    ...
  }
  ...
  func @add(!cil.int, !cil.int) -> !cil.int
}
```

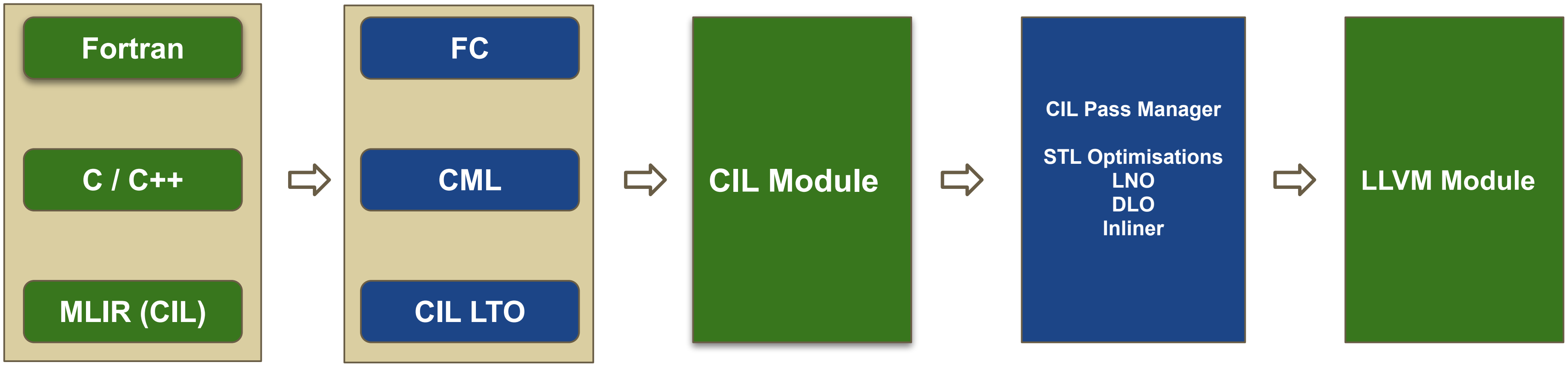
```
module {
  func @add(%arg0: !cil.int, %arg1: !cil.int)
-> !cil.int {
     ….
  }
}
```

```
module {
  %0 = cil.global @__str_tmp0 {constant, value = " %d
\0A\00"} : ...
  func @printf(!cil.pointer<!cil.char>) -> !cil.int
  func @main() -> !cil.int {
    ….
  }
  func @add(%arg0: !cil.int, %arg1: !cil.int) ->
!cil.int   {
    ….
  }
}
```

Data Layout Optimisations in CIL

- Instance interleaving and Dead field elimination optimisations are implemented in CIL
- Runs as LTO pass.
- Identification of struct access is simpler as compared to LLVM because there is separate operation for struct access.
- Approximately 35% improvement is seen in one of SPEC CPU 2017 benchmark.

C, C++, Fortran and CIL



CIL - Applications

- Optimising source codes for C, C++ and Fortran
- Optimising tensorflow graphs
- Optimising ONNX models
- Custom Hardware

Current Status

- Explored
  - C99
  - C++ : templates, operator overloading, Inheritance, ..
  - C++ STL
  - Fortran: low level ABI agnostic IR with LTO
  - Loop / Data layout optimization capability
- Yet to see:
  - Clang Integration
  - Clang static analysis toolchain
  - C++-11 and later standards
  - Other languages in Clang
  - C++ exceptions

Roadmap

CIL basic dialect to be open sourced soon

More focus on C++

Vectorizer integrated to Loop Nest Optimizer is work in progress.

LTO integration with Clang driver

Thank you