Compiler Optimization For (OpenMP) Accelerator Offloading

EuroLLVM — April 8, 2019 — Brussels, Belgium

Johannes Doerfert and Hal Finkel

Leadership Computing Facility
Argonne National Laboratory
`https://www.alcf.anl.gov/`

Acknowledgment

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration) responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering, and early testbed platforms, in support of the nation's exascale computing imperative.

Original Program

```
int y = 7;

for (i = 0; i < N; i++) {
  f(y, i);
}
g(y);
```

After Optimizations

```
for (i = 0; i < N; i++) {
  f(7, i);
}
g(7);
```

Original Program

After Optimizations

```
int y = 7;
#pragma omp parallel for
for (i = 0; i < N; i++) {
  f(y, i);
}
g(y);
```

Original Program

```
int y = 7;
#pragma omp parallel for
for (i = 0; i < N; i++) {
  f(y, i);
}
g(y);
```

After Optimizations

```
int y = 7;
#pragma omp parallel for
for (i = 0; i < N; i++) {
  f(y, i);
}
g(y);
```

Current Compiler Optimization For Parallelism

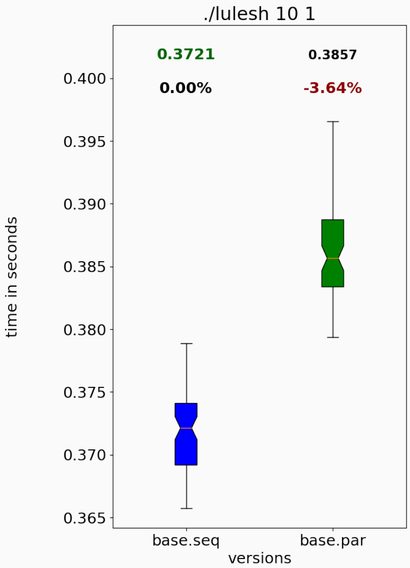
None*†

*At least for LLVM/Clang up to 8.0

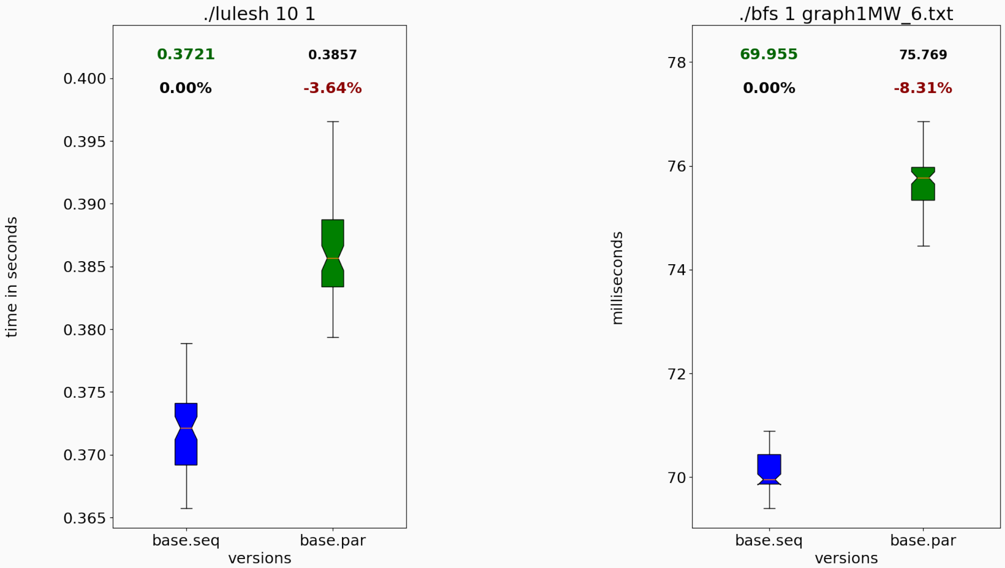
†And not considering smart runtime libraries!

Why is this important?

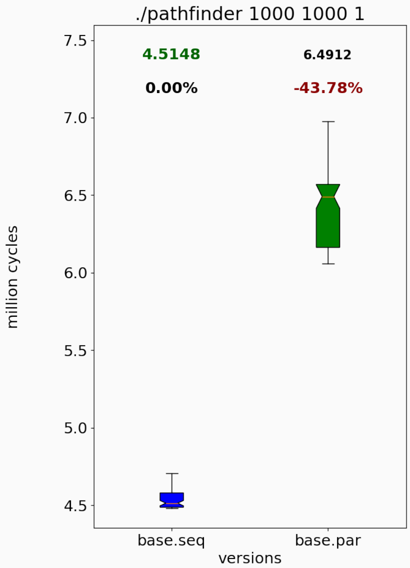
Performance Implications



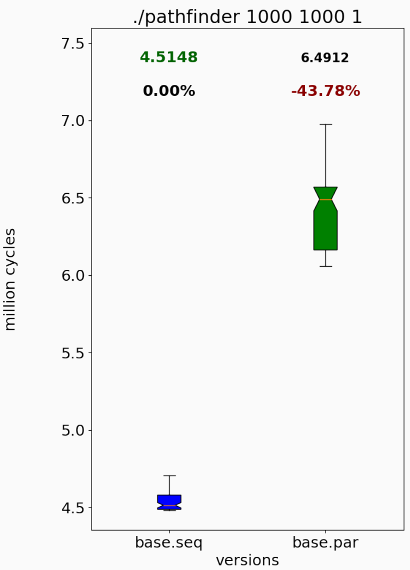
Performance Implications

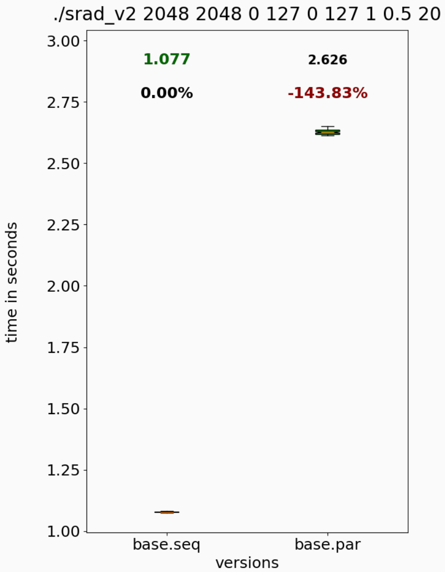


Performance Implications



Performance Implications





Optimization Categories

Optimizations for *sequential* aspects

Optimizations for *parallel* aspects

---

a

b

Optimization Categories

Optimizations for *sequential* aspects

- May reuse *existing* transformations (patches up for review!)

Optimizations for *parallel* aspects

- New *explicit parallelism-aware transformations*

---

[a]

[b]

Optimizations for *sequential* aspects

- May reuse *existing* transformations (patches up for review!)

⇒ Introduce *suitable abstractions* to bridge the indirection (DONE!)

Optimizations for *parallel* aspects

- New *explicit parallelism-aware transformations*

[a]

[b]

Optimizations for *sequential* aspects

- May reuse *existing* transformations (patches up for review!)

⇒ Introduce *suitable abstractions* to bridge the indirection (DONE!)

Optimizations for *parallel* aspects

- New *explicit parallelism-aware transformations*

⇒ Introduce a unifying abstraction layer

a

b

Optimization Categories

Optimizations for *sequential* aspects

- May reuse *existing* transformations (patches up for review!)

⇒ Introduce *suitable abstractions* to bridge the indirection (DONE!)

Optimizations for *parallel* aspects

- New *explicit parallelism-aware transformations* (see IWOMP'18[a])

⇒ Introduce a unifying abstraction layer (see EuroLLVM'18[b])

[a] *Compiler Optimizations For OpenMP*, J. Doerfert, H. Finkel, IWOMP 2018

[b] *A Parallel IR in Real Life: Optimizing OpenMP*, H. Finkel, J. Doerfert, X. Tian, G. Stelle, Euro LLVM Meeting 2018

Optimizations for *sequential* aspects

- May reuse *existing* transformations (patches up for review!)

⇒ Introduce *suitable abstractions* to bridge the indirection (DONE!)

Optimizations for *parallel* aspects

- New *explicit parallelism-aware transformations* (see IWOMP'18[a])

⇒ Introduce a unifying abstraction layer (see EuroLLVM'18[b])

---

[a] *Compiler Optimizations For OpenMP*, J. Doerfert, H. Finkel, IWOMP 2018

[b] *A Parallel IR in Real Life: Optimizing OpenMP*, H. Finkel, J. Doerfert, X. Tian, G. Stelle, Euro LLVM Meeting 2018

The Compiler Black Box

```
{
  #pragma omp target teams parallel
  work1();




}
```

```
>> clang -O3 -fopenmp-targets=...
```

```
{
  #pragma omp target teams parallel
  work1();



}
```

```
>> clang -O3 -fopenmp-targets=...
```

```
{
  #pragma omp target teams parallel
  work1();




}
```

“relatively” good performance :)

```
>> clang -O3 -fopenmp-targets=...
```

```
{
  #pragma omp target teams parallel
  work1();

  #pragma omp target teams
  #pragma omp parallel
  work2();
}
```

The Compiler Black Box

```
>> clang -O3 -fopenmp-targets=...
```

```
{
  #pragma omp target teams parallel
  work1();

  #pragma omp target teams
  #pragma omp parallel
  work2();
}
```

"relatively" good performance :)

```
>> clang -O3 -fopenmp-targets=...

  #pragma omp target teams
  {
    #pragma omp parallel
    work1();


    #pragma omp parallel
    work2();
  }
```

```
>> clang -O3 -fopenmp-targets=...

  #pragma omp target teams
  {
    #pragma omp parallel
    work1();


    #pragma omp parallel
    work2();
  }
```

probably poor performance :(

```
{
  #pragma omp target teams
  foo();
  #pragma omp target teams parallel
  work();                              // <- Hotspot
  #pragma omp target teams
  bar();
}
```

```
{
  #pragma omp target teams
  foo();
  #pragma omp target teams parallel
  work();                               // <- Hotspot
  #pragma omp target teams
  bar();
}
```

N teams,

```
{
  #pragma omp target teams
  foo();
  #pragma omp target teams parallel
  work();                                // <- Hotspot
  #pragma omp target teams
  bar();
}
```

N teams, with M threads each,

```
{
  #pragma omp target teams
  foo();
  #pragma omp target teams parallel
  work();                              // <- Hotspot
  #pragma omp target teams
  bar();
}
```

N teams, with M threads each, all executing `work` concurrently.

```c
#pragma omp target teams
{

  foo();
  #pragma omp parallel
  work();                                   // <- Hotspot

  bar();
}
```

```
#pragma omp target teams
{

  foo();
  #pragma omp parallel
  work();                                    // <- Hotspot

  bar();
}
```

1 master and N-1 worker teams,

```
#pragma omp target teams
{

  foo();
  #pragma omp parallel
  work();                              // <- Hotspot

  bar();
}
```

1 master and N-1 worker teams, worker teams M threads:

```
#pragma omp target teams
{

  foo();
  #pragma omp parallel
  work();                                    // <- Hotspot

  bar();
}
```

1 master and N-1 worker teams, worker teams M threads:
Masters execute `foo` concurrently, workers idle.

```
#pragma omp target teams
{

  foo();
  #pragma omp parallel
  work();                                    // <- Hotspot

  bar();
}
```

1 master and N-1 worker teams, worker teams M threads:
Masters execute `foo` concurrently, workers idle.
Masters delegate `work` for concurrent execution.

```
#pragma omp target teams
{

  foo();
  #pragma omp parallel
  work();                                       // <- Hotspot

  bar();
}
```

1 master and N-1 worker teams, worker teams M threads:
Masters execute `foo` concurrently, workers idle.
Masters delegate `work` for concurrent execution.
Masters execute `bar` concurrently, workers idle.

```
#pragma omp target teams
{
```

```
}
```

Problems:

1 m

Masters execute `foo` concurrently, workers idle.

Masters delegate `work` for concurrent execution.

Masters execute `bar` concurrently, workers idle.

```
#pragma omp target teams
{
```

Problems:

- a separate master team costs resources

```
}
```

1 m

Masters execute `foo` concurrently, workers idle.

Masters delegate `work` for concurrent execution.

Masters execute `bar` concurrently, workers idle.

```
#pragma omp target teams
{
```

Problems:

- a separate master team costs resources
- synchronization has overhead

```
}
```

1 m

Masters execute `foo` concurrently, workers idle.

Masters delegate `work` for concurrent execution.

Masters execute `bar` concurrently, workers idle.

```
#pragma omp target teams
{
```

```
}
```

Problems:

- a separate master team costs resources
- synchronization has overhead
- currently impossible to optimization

1 m

Masters execute `foo` concurrently, workers idle.

Masters delegate `work` for concurrent execution.

Masters execute `bar` concurrently, workers idle.

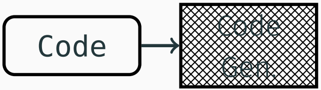
OpenMP Offload — Overview

`OpenMP` `Clang` `LLVM-IR` `LLVM` `Assembly`

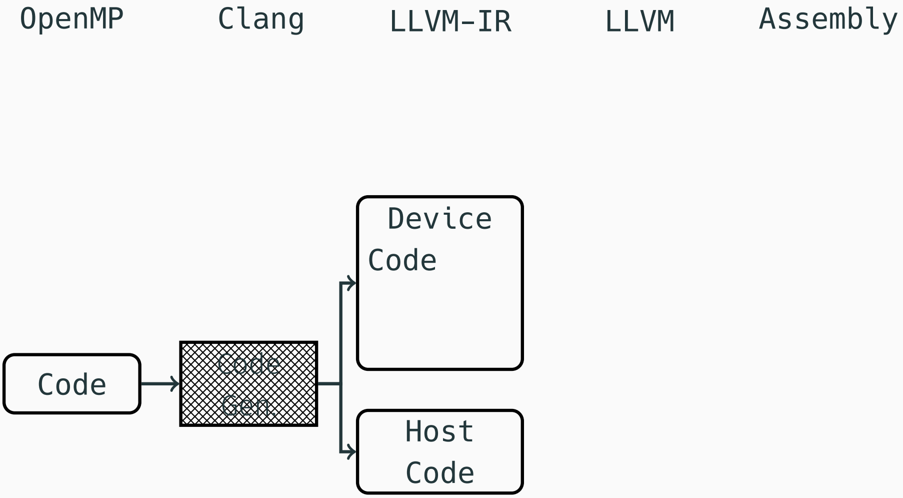
OpenMP Clang LLVM-IR LLVM Assembly

Code

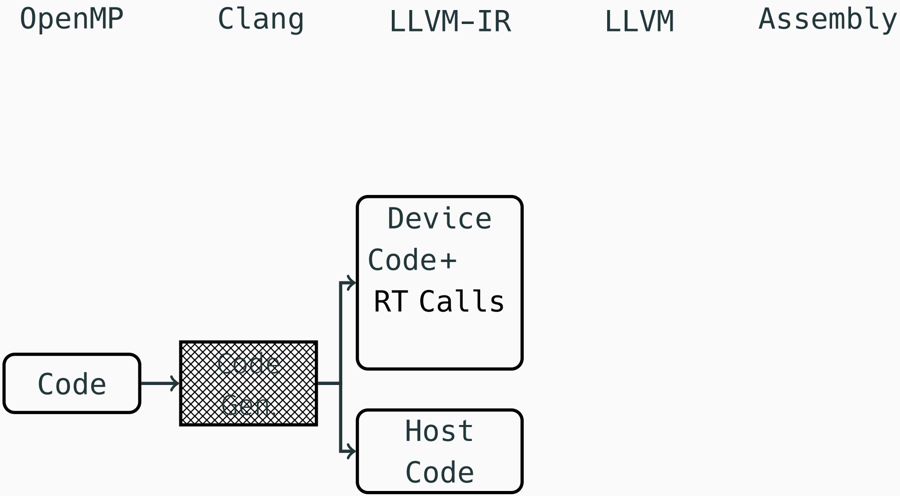
OpenMP Clang LLVM-IR LLVM Assembly



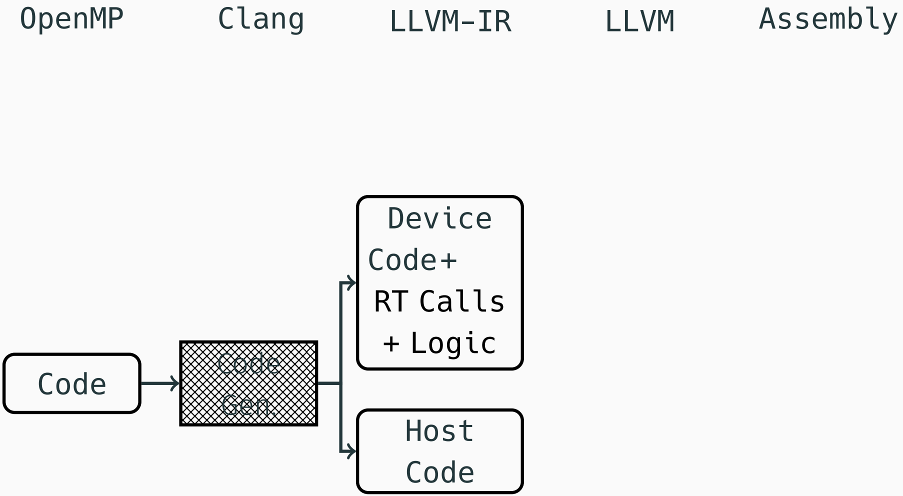
OpenMP Offload — Overview

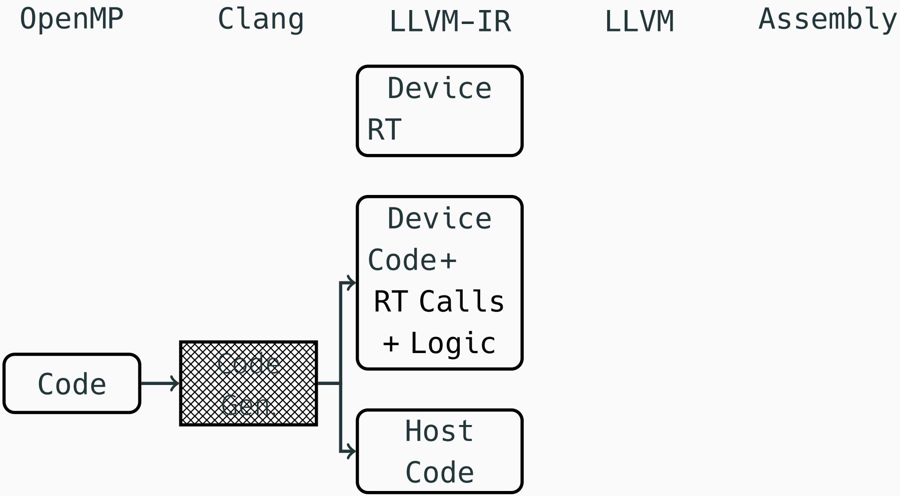


OpenMP Offload — Overview

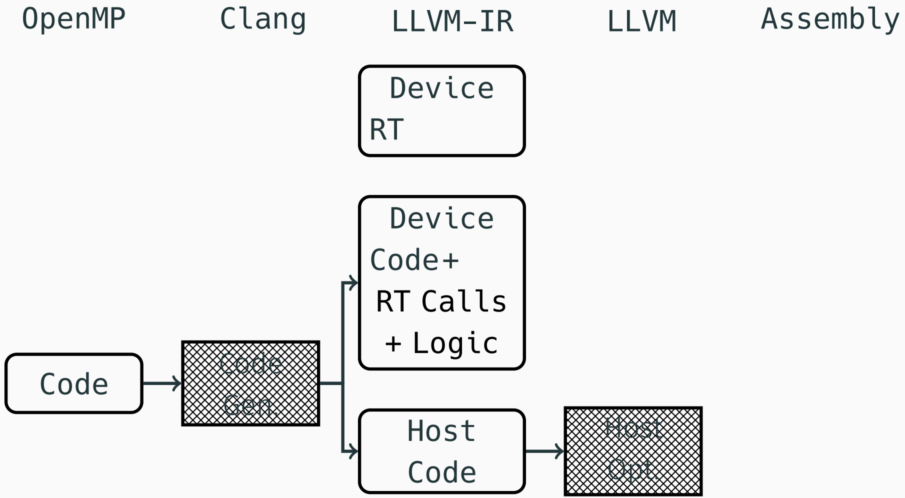


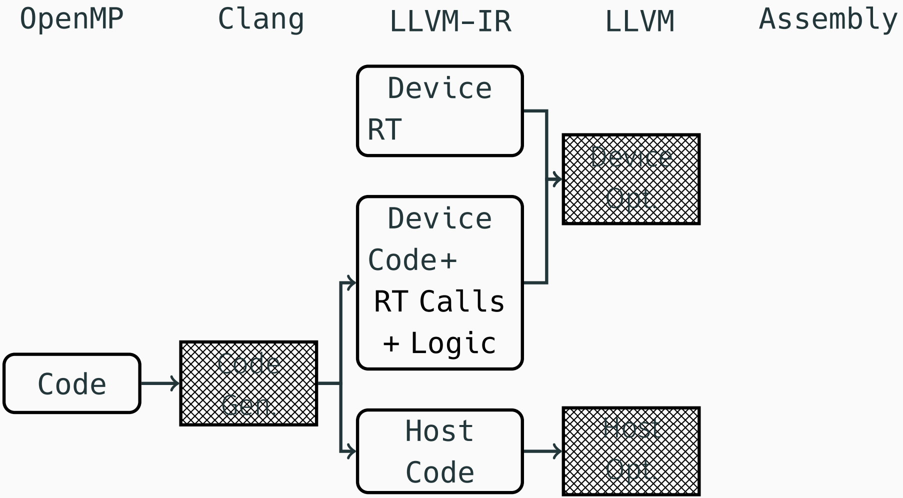
OpenMP Offload — Overview



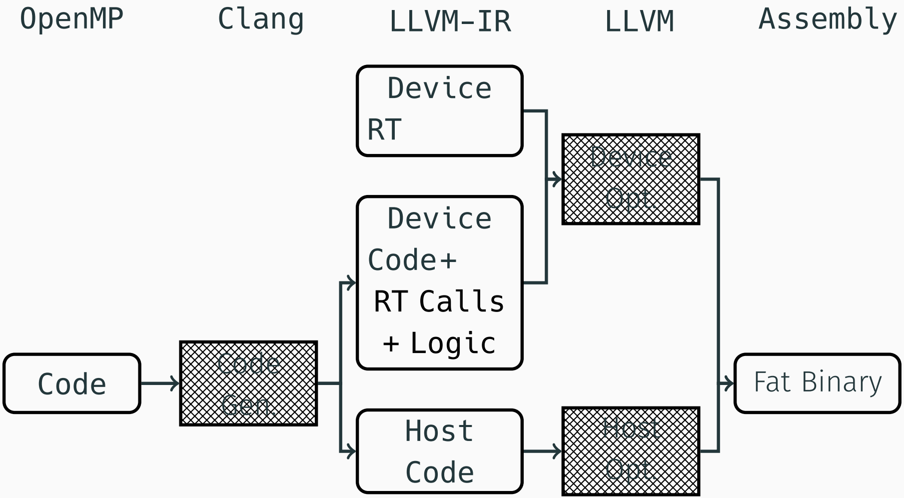

OpenMP
Clang
LLVM-IR
LLVM
Assembly
Device RT
Device Code+ RT Calls + Logic
Code
Code Gen.
Host Code

OpenMP Offload — Overview

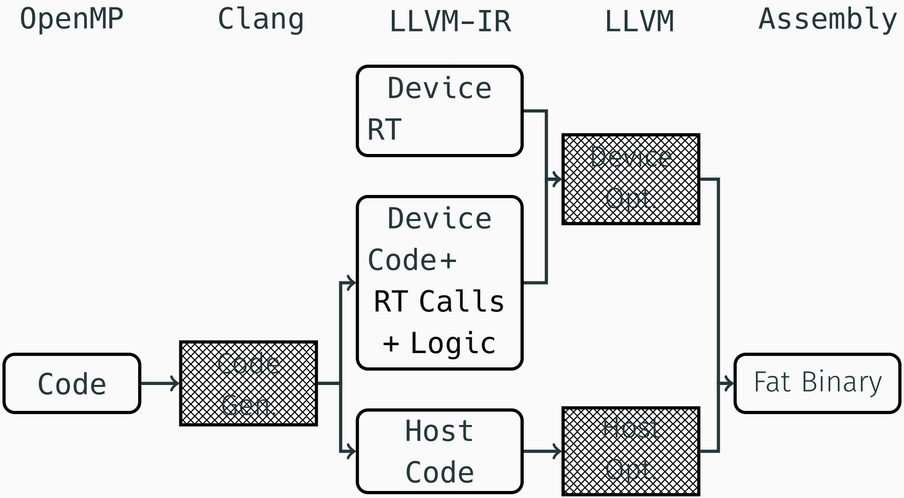



OpenMP
Clang
LLVM-IR
LLVM
Assembly
Device RT
Device Code+ RT Calls + Logic
Host Code
Code
Code Gen.
Device Opt.
Host Opt.

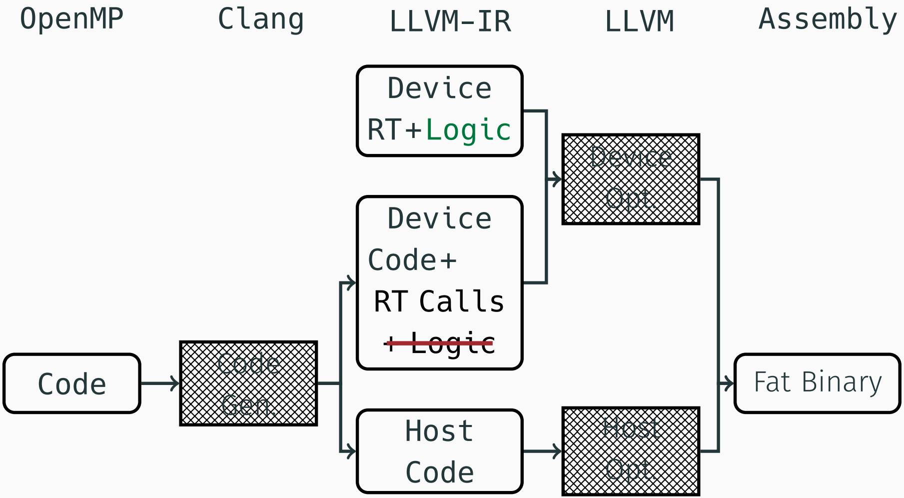
OpenMP Offload — Overview



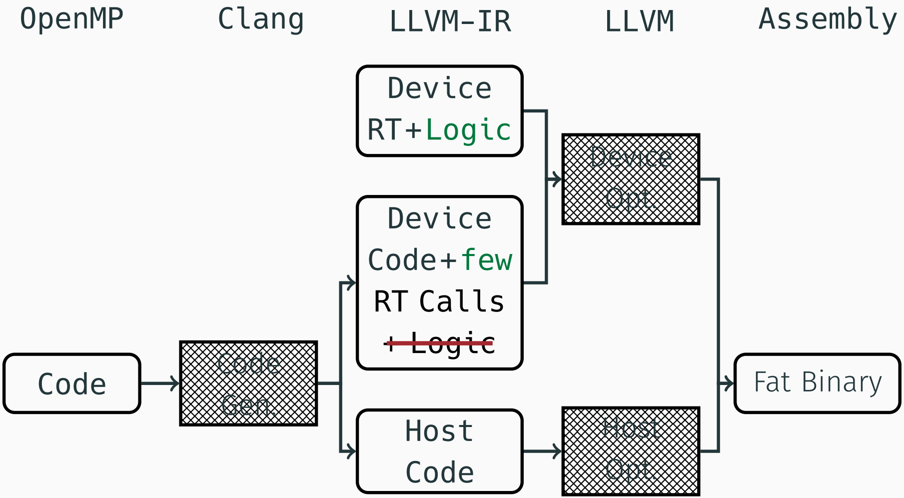
1. Offload-Specific Optimizations on Device Code



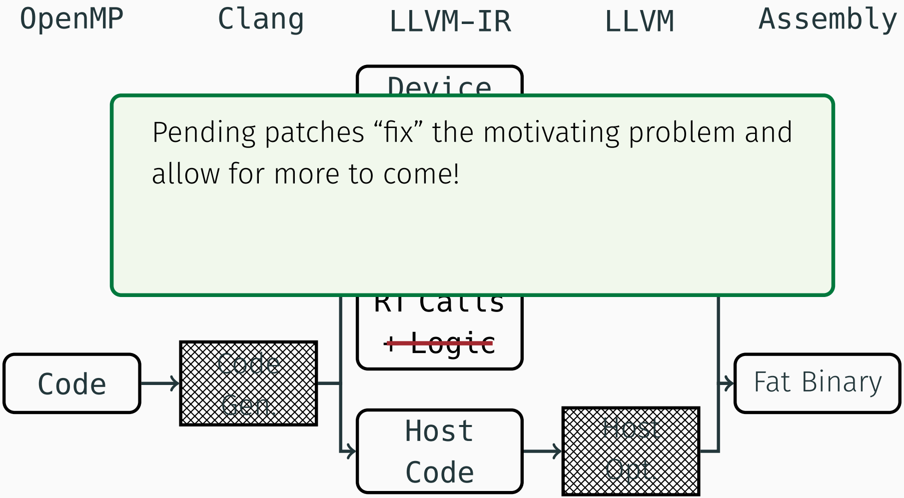
1. Offload-Specific Optimizations on Device Code



1. Offload-Specific Optimizations on Device Code

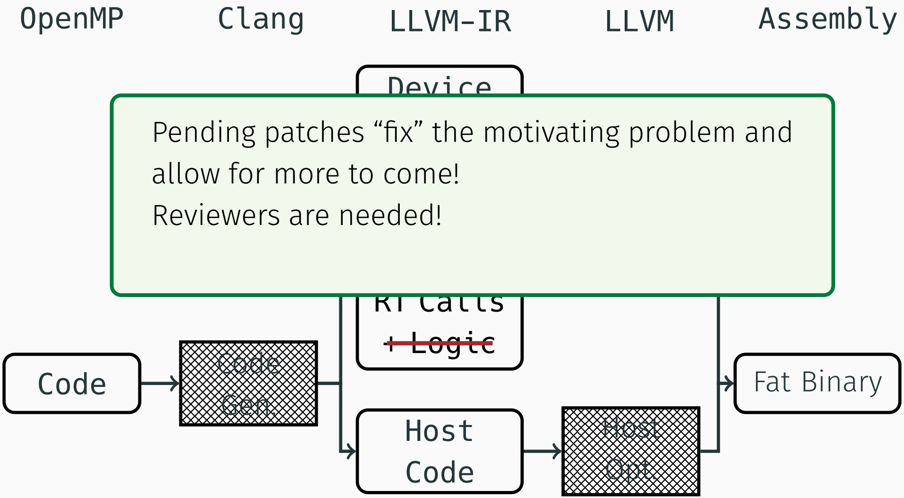


1. Offload-Specific Optimizations on Device Code



Pending patches “fix” the motivating problem and allow for more to come!

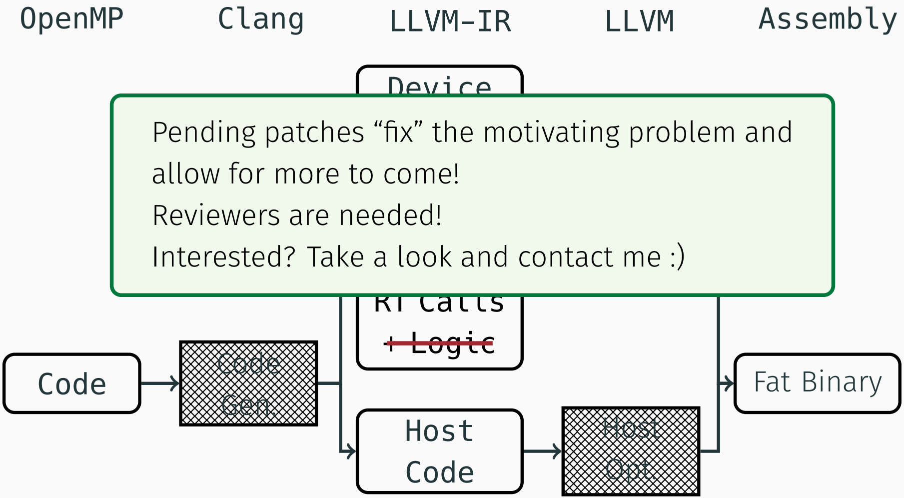
1. Offload-Specific Optimizations on Device Code



Pending patches "fix" the motivating problem and allow for more to come!

Reviewers are needed!

1. Offload-Specific Optimizations on Device Code

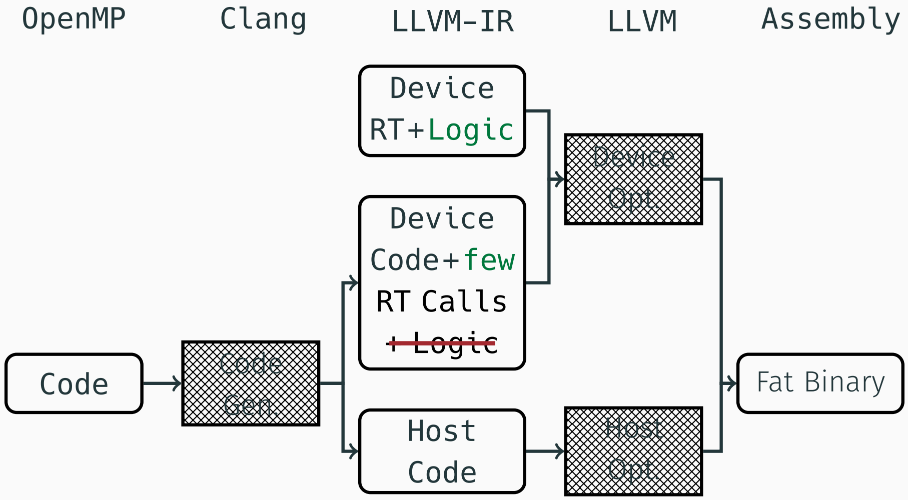


Pending patches "fix" the motivating problem and allow for more to come!

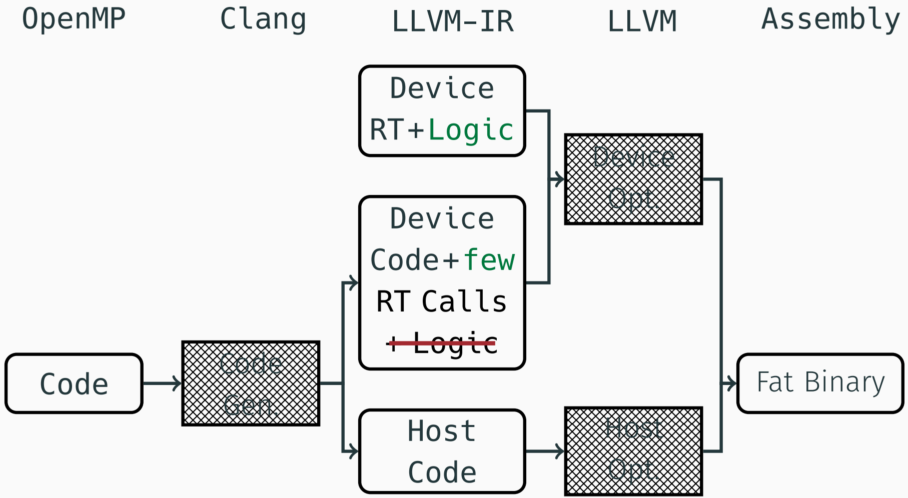
Reviewers are needed!

Interested? Take a look and contact me :)

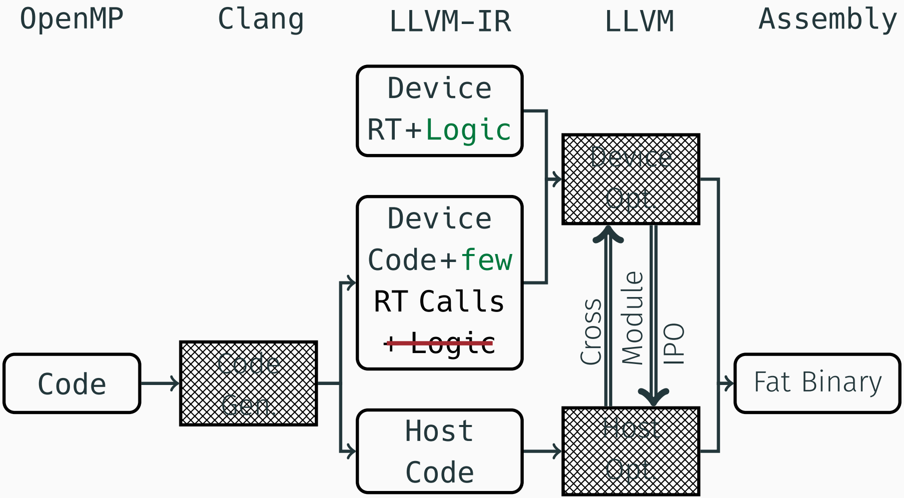
OpenMP Offload — Overview & Directions



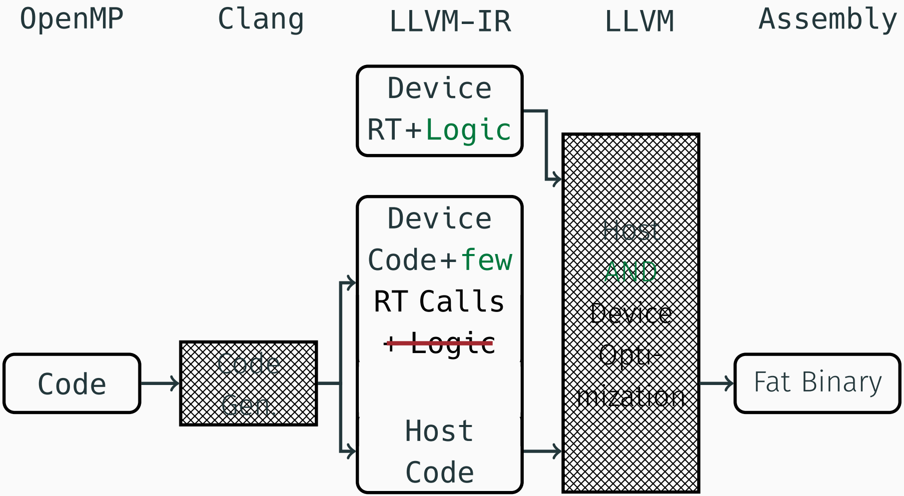
2. Optimize Device and Host Code Together



2. Optimize Device and Host Code Together



2. Optimize Device and Host Code Together



Pending Patches — Target Region Interface

Pending Patches — Target Region Interface

- A straight-forward `#pragma omp target` front-end:
  - simplified implementation
  - improved reusability (F18, ...)

Pending Patches — Target Region Interface

- A straight-forward `#pragma omp target` front-end:
  - ◇ simplified implementation — CGOpenMPRuntimeNVPTX.cpp ~5.0k loc
  - ◇ improved reusability (F18, ...) — CGOpenMPRuntimeTRegion.cpp ~0.5k loc

Pending Patches — Target Region Interface

- A straight-forward `#pragma omp target` front-end:
  - simplified implementation — CGOpenMPRuntimeNVPTX.cpp ~5.0k loc
  - improved reusability (F18, ...) — CGOpenMPRuntimeTRegion.cpp ~0.5k loc
- Interface exposes information and implementation choices:
  - “smartness” is moved in the compiler middle-end
  - simplifies analyses and transformations in LLVM

Pending Patches — Target Region Interface

- A straight-forward `#pragma omp target` front-end:
  - simplified implementation — CGOpenMPRuntimeNVPTX.cpp ~5.0k loc
  - improved reusability (F18, ...) — CGOpenMPRuntimeTRegion.cpp ~0.5k loc
- Interface exposes information and implementation choices:
  - “smartness” is moved in the compiler middle-end
  - simplifies analyses and transformations in LLVM
- Device RT interface & implementation are separated:
  - simplifies generated LLVM-IR
  - most LLVM & Clang parts become target agnostic

1. Offload-Specific Optimizations — "SPMD-zation"

- use inter-procedural reasoning to place minimal guards/synchronization

1. Offload-Specific Optimizations — "SPMD-zation"

- use inter-procedural reasoning to place minimal guards/synchronization
- if legal, switch all boolean `UseSPMDMode` flags to `true`

- use inter-procedural reasoning to place minimal guards/synchronization
- if legal, switch all boolean `UseSPMDMode` flags to `true`
- currently, no (unknown) global side-effects allowed outside parallel regions.

- use optimized state-machines when unavoidable

1. Offload-Specific Optimizations — Custom State Machines

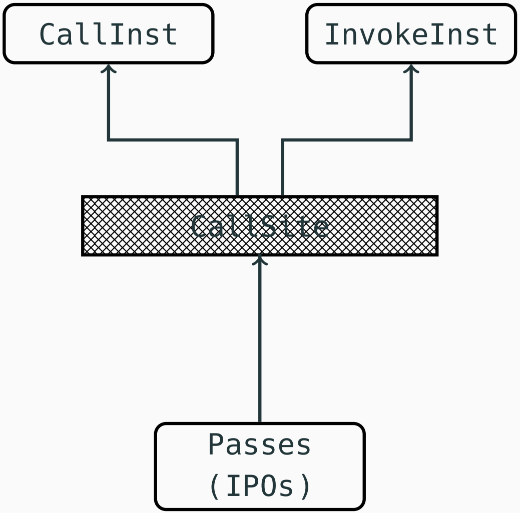
- use optimized state-machines when unavoidable
- reachability & post-dominance restrict the set of potential next parallel regions to work on

1. Offload-Specific Optimizations — Custom State Machines

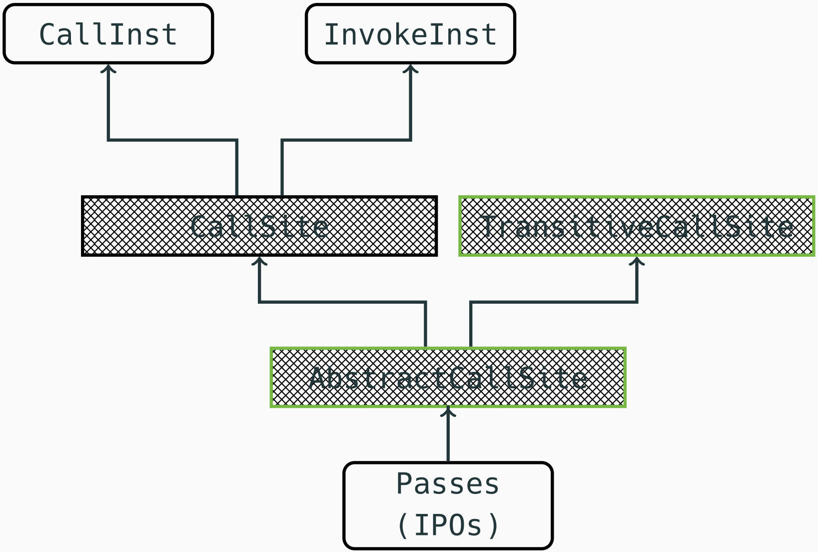
- use optimized state-machines when unavoidable
- reachability & post-dominance restrict the set of potential next parallel regions to work on
- reuse already communicated/shared values if possible

- use optimized state-machines when unavoidable
- reachability & post-dominance restrict the set of potential next parallel regions to work on
- reuse already communicated/shared values if possible
- currently, a simple state machine is generated with explicit conditionals for all known parallel regions in the module

2. Optimize Device and Host Together — Abstract Call Sites



2. Optimize Device and Host Together — Abstract Call Sites



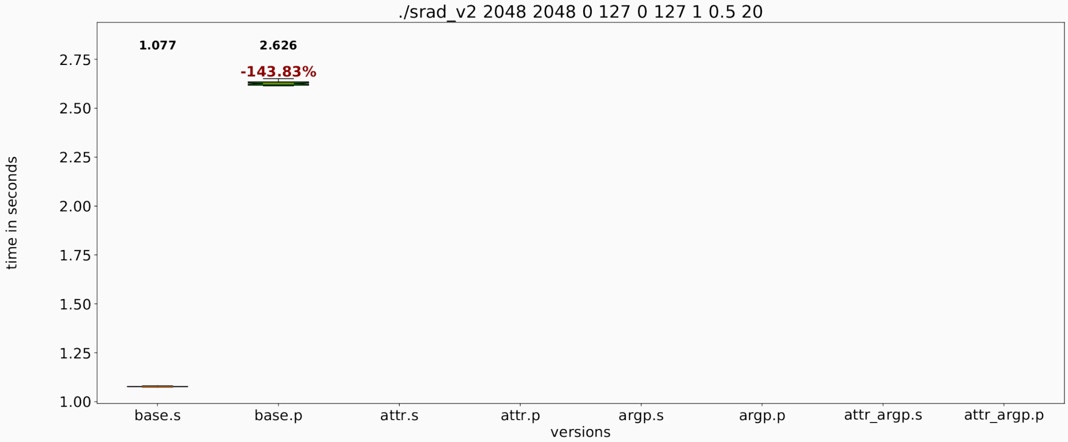
Call

Functional changes required for
Inter-procedural Constant Propagation:

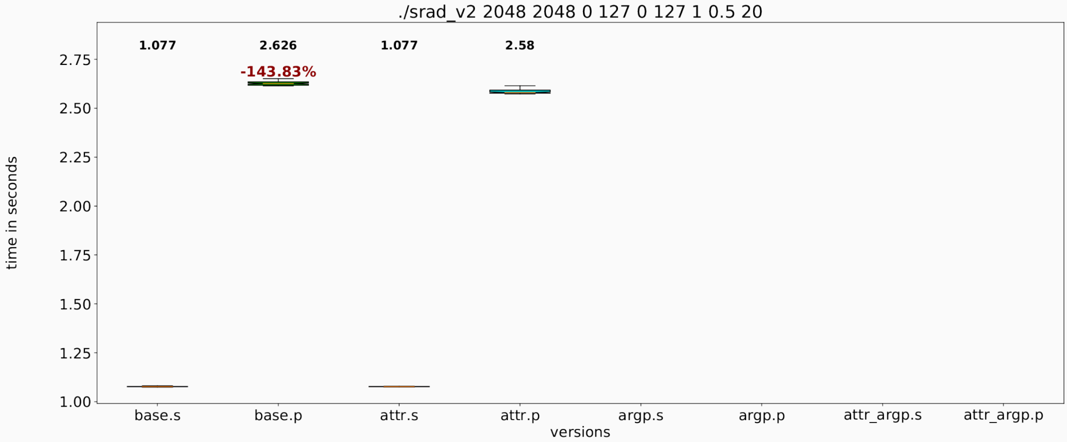
```
for (int i = 0; i < NumArgs; i++) {
  Value *ArgOp = ACS.getArgOperand(i);
  if (!ArgOp) {
    // handle non-constant
    continue;
  }
  ...
}
```

Passes
(IPOs)

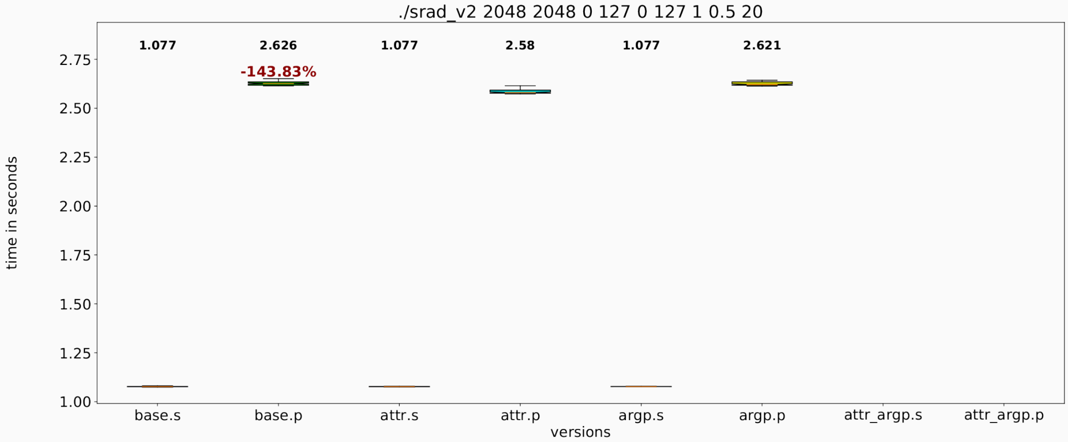
Abstract Call Sites — Performance Results



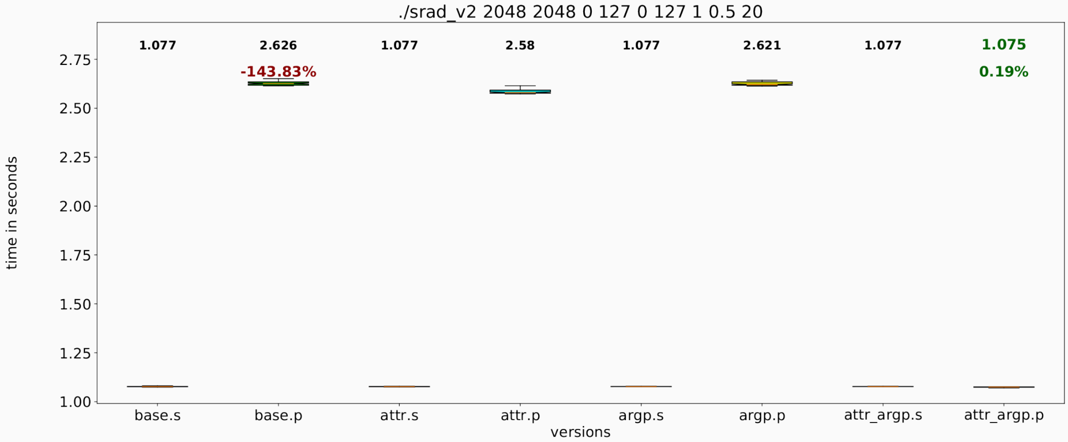
Abstract Call Sites — Performance Results



Abstract Call Sites — Performance Results



Abstract Call Sites — Performance Results



Conclusion

Conclusion

The Compiler Black Box

```
>> clang -O3 -fopenmp-targets=...

  #pragma omp target teams
  {
    #pragma omp parallel
    work1();


    #pragma omp parallel
    work2();
  }
```

probably poor performance :(

Conclusion

The Compiler Black Box

```
>> clang -O3 -fopenmp-targets=...

  #pragma omp target teams
  {
    #pragma omp parallel
    work1();


    #pragma omp parallel
    work2();
  }
```

probably poor performance :(

Optimization Categories

Optimizations for *sequential* aspects

- May reuse *existing* transformations (patches up for review!)

⇒ Introduce *suitable abstractions* to bridge the indirection (DONE!)

Optimizations for *parallel* aspects

- New *explicit parallelism-aware transformations* (see IWOMP'18[a])

⇒ Introduce a unifying abstraction layer (see EuroLLVM'18[b])

[a] *Compiler Optimizations For OpenMP*, J. Doerfert, H. Finkel, IWOMP 2018

[b] *A Parallel IR in Real Life: Optimizing OpenMP*, H. Finkel, J. Doerfert, X. Tian, G. Stelle, Euro LLVM Meeting 2018

Conclusion

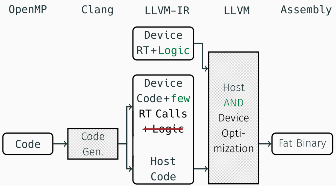
The Compiler Black Box

```
>> clang -O3 -fopenmp-targets=...

  #pragma omp target teams
  {
    #pragma omp parallel
    work1();


    #pragma omp parallel
    work2();
  }
```

probably poor performance :(

OpenMP Offload — Overview & Directions

2. Optimize Device and Host Code Together



Optimization Categories

Optimizations for *sequential* aspects

- May reuse *existing* transformations (patches up for review!)

⇒ Introduce *suitable abstractions* to bridge the indirection (DONE!)

Optimizations for *parallel* aspects

- New *explicit parallelism-aware transformations* (see IWOMP'18[a])

⇒ Introduce a unifying abstraction layer (see EuroLLVM'18[b])

[a] *Compiler Optimizations For OpenMP*, J. Doerfert, H. Finkel, IWOMP 2018

[b] *A Parallel IR in Real Life: Optimizing OpenMP*, H. Finkel, J. Doerfert, X. Tian, G. Stelle, Euro LLVM Meeting 2018

Conclusion

The Compiler Black Box

```
>> clang -O3 -fopenmp-targets=...

  #pragma omp target teams
  {
    #pragma omp parallel
    work1();


    #pragma omp parallel
    work2();
  }
```

probably poor performance :(

Optimization Categories

Optimizations for *sequential* aspects

- May reuse *existing* transformations (patches up for review!)

⇒ Introduce *suitable abstractions* to bridge the indirection (DONE!)

Optimizations for *parallel* aspects

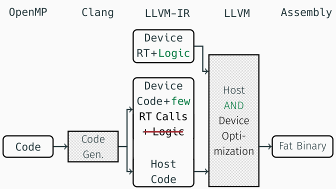
- New *explicit parallelism-aware transformations* (see IWOMP'18[a])

⇒ Introduce a unifying abstraction layer (see EuroLLVM'18[b])

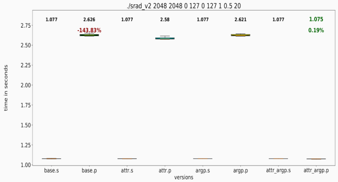
[a] *Compiler Optimizations For OpenMP*, J. Doerfert, H. Finkel, IWOMP 2018
[b] *A Parallel IR in Real Life: Optimizing OpenMP*, H. Finkel, J. Doerfert, X. Tian, G. Stelle, Euro LLVM Meeting 2018

OpenMP Offload — Overview & Directions

2. Optimize Device and Host Code Together



Abstract Call Sites — Performance Results



OpenMP-Aware Optimizations (see IWOMP'18)

I: Attribute Propagation — Bidirectional Information Transfer:
`read/write-only`, **`restrict`/`noalias`**, …

OpenMP-Aware Optimizations (see IWOMP'18)

I: Attribute Propagation — Bidirectional Information Transfer:
`read/write-only`, **`restrict`**/**`noalias`**, ...

II: Variable Privatization — Limit Variable Lifetimes:
`shared(var)` ⟶ `firstprivate(var)` ⟶ `private(var)`

OpenMP-Aware Optimizations (see IWOMP'18)

I: Attribute Propagation — Bidirectional Information Transfer:
`read/write-only`, **`restrict`/`noalias`**, ...

II: Variable Privatization — Limit Variable Lifetimes:
`shared(var)` $\longrightarrow$ `firstprivate(var)` $\longrightarrow$ `private(var)`

III: Parallel Region Expansion — Maximize Parallel Contexts:
$\Longrightarrow$ reduce start/stop overheads and expose barriers

OpenMP-Aware Optimizations (see IWOMP'18)

I: Attribute Propagation — Bidirectional Information Transfer:
`read/write-only`, **`restrict`/`noalias`**, …

II: Variable Privatization — Limit Variable Lifetimes:
`shared(var)` $\longrightarrow$ `firstprivate(var)` $\longrightarrow$ `private(var)`

III: Parallel Region Expansion — Maximize Parallel Contexts:
$\Longrightarrow$ reduce start/stop overheads and expose barriers

IV: Barrier Elimination — Eliminate Redundant Barriers

OpenMP-Aware Optimizations (see IWOMP'18)

I: Attribute Propagation — Bidirectional Information Transfer:
`read/write-only`, **`restrict`/`noalias`**, …

II: Variable Privatization — Limit Variable Lifetimes:
`shared(var)` $\longrightarrow$ `firstprivate(var)` $\longrightarrow$ `private(var)`

III: Parallel Region Expansion — Maximize Parallel Contexts:
$\Longrightarrow$ reduce start/stop overheads and expose barriers

IV: Barrier Elimination — Eliminate Redundant Barriers

V: Communication Optimization — Move Computations Around: seq. compute & result comm. **vs.** operand comm. & par. compute

OpenMP-Aware Optimizations (see IWOMP'18)

I: Attribute Propagation — Bidirectional Information Transfer:
`read/write-only`, **`restrict`/`noalias`**, ...

II: Variable Privatization — Limit Variable Lifetimes:
`shared(var)` $\longrightarrow$ `firstprivate(var)` $\longrightarrow$ `private(var)`

III: Parallel Region Expansion — Maximize Parallel Contexts:
$\Longrightarrow$ reduce start/stop overheads and expose barriers

IV: Barrier Elimination — Eliminate Redundant Barriers

V: Communication Optimization — Move Computations Around: seq. compute & result comm. **vs.** operand comm. & par. compute

OpenMP-Aware Optimizations (see IWOMP'18)

I: Attribute Propagation — In LLVM: *Attribute Deduction (IPO!)*
`read/write-only`, **`restrict`**/**`noalias`**, ...

II: Variable Privatization — In LLVM: *Argument Promotion (IPO!)*
`shared(var)` $\longrightarrow$ `firstprivate(var)` $\longrightarrow$ `private(var)`

III: Parallel Region Expansion — Maximize Parallel Contexts:
$\Longrightarrow$ reduce start/stop overheads and expose barriers

IV: Barrier Elimination — Eliminate Redundant Barriers

V: Communication Optimization — Move Computations Around: seq. compute & result comm. **vs.** operand comm. & par. compute

Early Outlining

OpenMP Input:
```
#pragma omp parallel for
for (int i = 0; i < N; i++)
  Out[i] = In[i] + In[i+N];
```

Early Outlining

OpenMP Input:
```
#pragma omp parallel for
for (int i = 0; i < N; i++)
  Out[i] = In[i] + In[i+N];
```

```
// Parallel region replaced by a runtime call.
omp_rt_parallel_for(0, N, &body_fn,   &N, &In, &Out);
```

Early Outlining

OpenMP Input:
```
#pragma omp parallel for
for (int i = 0; i < N; i++)
  Out[i] = In[i] + In[i+N];
```

```
// Parallel region replaced by a runtime call.
omp_rt_parallel_for(0, N, &body_fn,   &N, &In, &Out);


// Parallel region outlined in the front-end (clang)!
static void body_fn(int tid, int *N,
                    float** In, float** Out) {
  int lb = omp_get_lb(tid), ub = omp_get_ub(tid);
  for (int i = lb; i < ub; i++)
    (*Out)[i] = (*In)[i] + (*In)[i + (*N)]
}
```

OpenMP Input:
```
#pragma omp parallel for
for (int i = 0; i < N; i++)
  Out[i] = In[i] + In[i+N];
```

```
// Parallel region replaced by a runtime call.
omp_rt_parallel_for(0, N, &body_fn,   &N, &In, &Out);


// Parallel region outlined in the front-end (clang)!
static void body_fn(int tid, int* N,
                    float** In, float** Out) {
  int lb = omp_get_lb(tid), ub = omp_get_ub(tid);
  for (int i = lb; i < ub; i++)
    (*Out)[i] = (*In)[i] + (*In)[i + (*N)]
}
```

An Abstract Parallel IR

OpenMP Input:

```
#pragma omp parallel for
for (int i = 0; i < N; i++)
  Out[i] = In[i] + In[i+N];
```

```
// Parallel region replaced by an annotated loop
for /* parallel */ (int i = 0; i < N; i++)
  body_fn(i, &N, &In, &Out);

// Parallel region outlined in the front-end (clang)!
static void body_fn(int  i , int* N,
                    float** In, float** Out) {


    (*Out)[i] = (*In)[i] + (*In)[i + (*N)]
}
```

Early Outlined + Transitive Calls

OpenMP Input:

```
#pragma omp parallel for
for (int i = 0; i < N; i++)
  Out[i] = In[i] + In[i+N];
```

```
// Parallel region replaced by a runtime call.
omp_rt_parallel_for(0, N, &body_fn,   &N, &In, &Out);
// Model transitive call:  body_fn(?, &N, &In, &Out);

// Parallel region outlined in the front-end (clang)!
static void body_fn(int tid, int* N,
                    float** In, float** Out) {
  int lb = omp_get_lb(tid), ub = omp_get_ub(tid);
  for (int i = lb; i < ub; i++)
    (*Out)[i] = (*In)[i] + (*In)[i + (*N)]
}
```

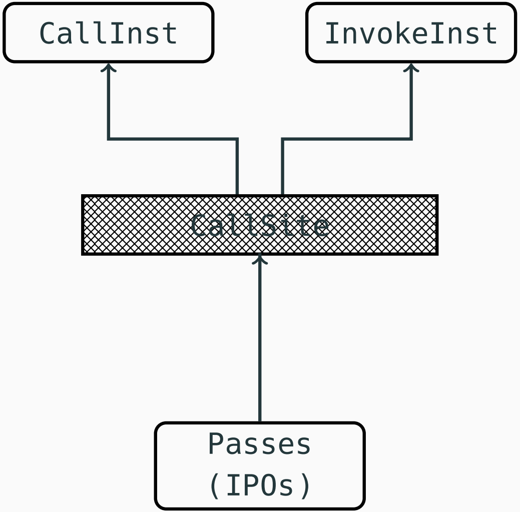
Early Outlined + Transitive Calls

OpenMP Input:
```
#pragma omp parallel for
for (int i = 0; i < N; i++)
  Out[i] = In[i] + In[i+N];
```

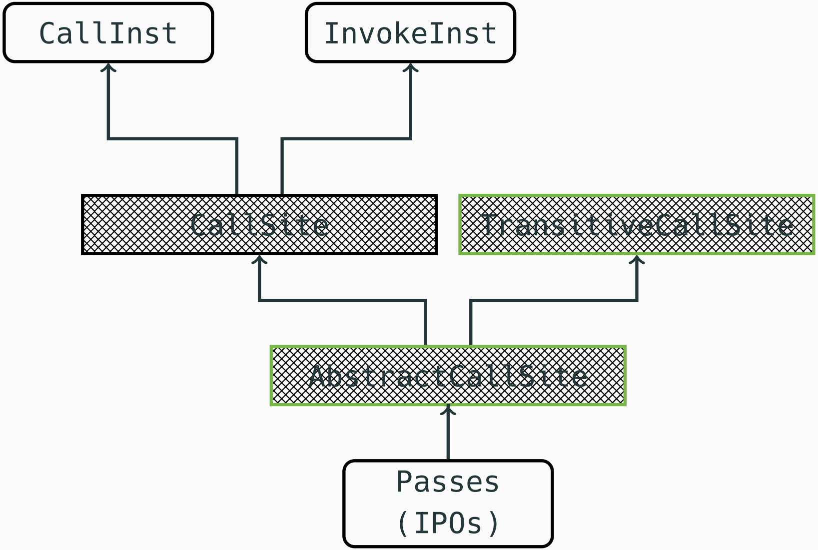
```
// Parallel region replaced by a runtime call.
omp_rt_parallel_for(0, N, &body_fn,   &N, &In, &Out);
// Model transitive call:  body_fn(?, &N, &In, &Out);

// Parallel region outlined in the front-end (clang)!
static void body_fn(int tid, int* N,
                    float** In, float** Out) {
  int lb = o
  for (int i
    (*Out)[i
}
```

- \+ *valid* and *executable* IR
- – *integration cost* per IPO
- \+ *no unintended interactions*

IPO in LLVM



Transitive Call Sites in LLVM



Transitive Call Sites in LLVM

Call

Functional changes required for
Inter-procedural Constant Propagation:

```
for (int i = 0; i < NumArgs; i++) {
  Value *ArgOp = ACS.getArgOperand(i);
  if (!ArgOp) {
    // handle non-constant
    continue;
  }
  ...
}
```

(IPOs)

Evaluated Version

Version	Description	Opt.
base	plain "-O3", thus no parallel optimizations	
attr	attribute propagation through attr. deduction (IPO)	I
argp	variable privatization through arg. promotion (IPO)	II
n/a	constant propagation (IPO)	

Some Context

Some Context

Examples

Examples are given in a C-like language with OpenMP annotations.

Transformations

Our transformations work on the LLVM intermediate representation (LLVM-IR), thus take and produce LLVM-IR.

OpenMP Runtime Library

We experience OpenMP annotations as OpenMP runtime library calls and the situation is most often more complicated than presented here.

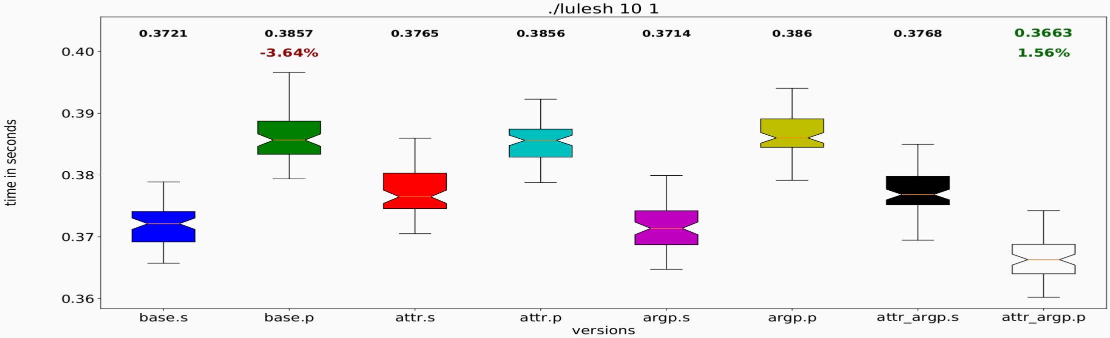
Evaluation Environment

- Run with 1 Thread[2]
- Median and variance of 51 runs is shown
- Rodiana 3.1 benchmarks and LULESH v1.0 (OpenMP)
- Only time in parallel constructs was measured

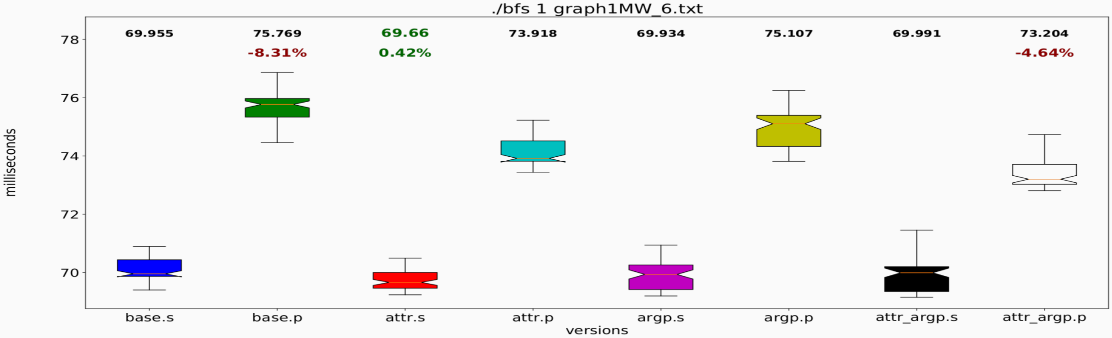
[2] Intel(R) Core(TM) i7-4800MQ CPU @ 2.70GHz

Performance Results

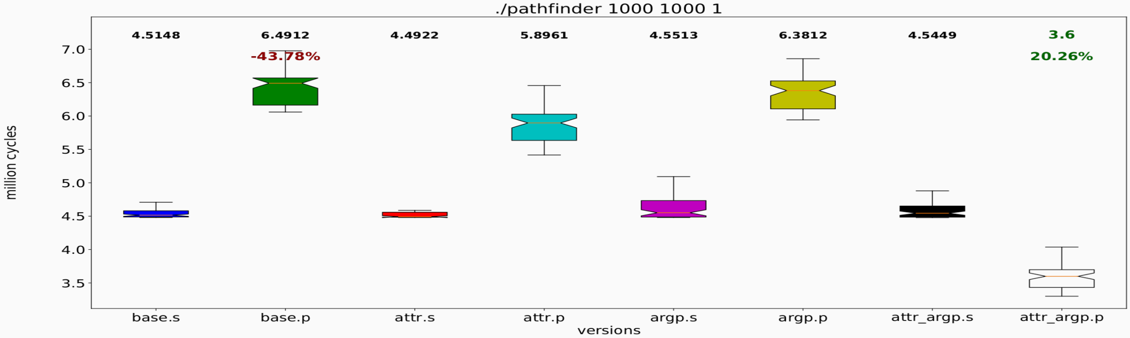
Performance Results



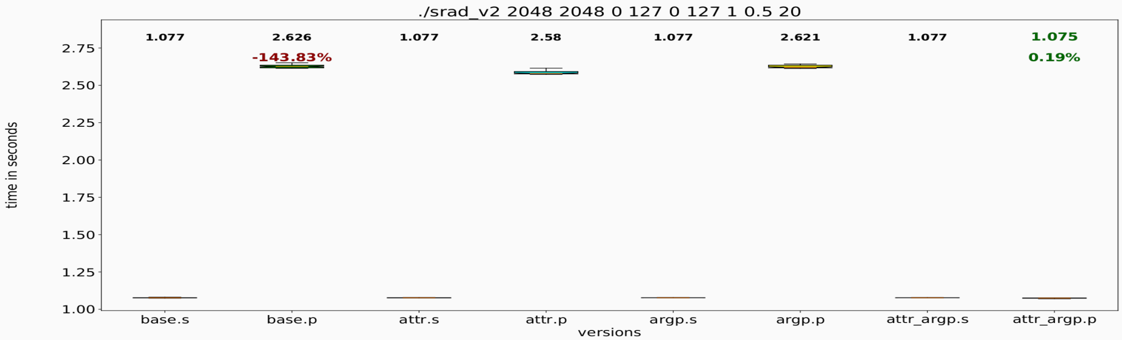
Performance Results



Performance Results



Performance Results



Action Item I

⋆

†

Action Item I

1) Run *your* OpenMP code sequentially[*],
   with and without OpenMP.

[*] `export OMP_NUM_THREADS=1`

†

Action Item I

1) Run *your* OpenMP code sequentially[*],
with and without OpenMP.

2) Email me[†] the results!

---

[*] `export OMP_NUM_THREADS=1`

[†] `jdoerfert@anl.gov`

Action Item II

⋆

Action Item II

1) Always[*] use `default(none)` and `firstprivate(...)`

[*] `For scalars/pointers if you do not have explicit synchronization.`

Action Item II

1) Always* use `default(none)` and `firstprivate(...)`

2) Revisit Action Item I

*For scalars/pointers if you do not have explicit synchronization.

ACTION ITEM II

NO need to “share” the *variable* A!

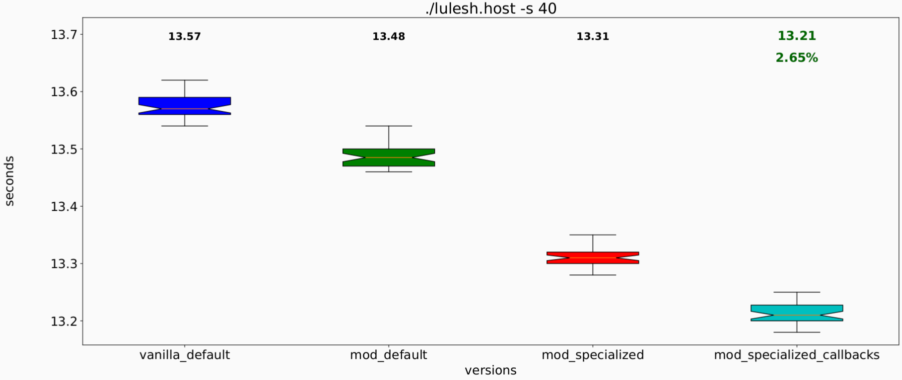
```
#pragma omp parallel for shared(A)
for (int i = 0; i < N; i++)
  A[i] = i;
```

* For scalars/pointers if you do not have explicit synchronization.

Constant Propagation Example

```
double gamma[4][8];
gamma[0][0] = 1;
// ... and so on till ...
gamma[3][7] = -1;

Kokkos::parallel_for(
  "CalcFBHourglassForceForElems A",
  numElem, KOKKOS_LAMBDA(const int &i2) {
  // Use gamma[0][0] ... gamme[3][7]
}
```

Constant Propagation Performance



Optimization I: Attribute Propagation

OpenMP Input:

```
#pragma omp parallel for firstprivate(...)
for (int i = 0; i < N; i++)
  Out[i] = In[i] + In[i+N];
```

Optimization I: Attribute Propagation

OpenMP Input:

```
#pragma omp parallel for firstprivate(...)
for (int i = 0; i < N; i++)
  Out[i] = In[i] + In[i+N];
```

```
// Parallel region replaced by a runtime call.
omp_rt_parallel_for(0, N, &body_fn, N, In, Out);


// Parallel region outlined in the front-end (clang)!
void body_fn(int i, int N,
             float* In,
             float* Out) {
  Out[i] = In[i] + In[i + N];
}
```

Optimization I: Attribute Propagation

OpenMP Input:

```
#pragma omp parallel for firstprivate(...)
for (int i = 0; i < N; i++)
  Out[i] = In[i] + In[i+N];
```

```
// Parallel region replaced by a runtime call.
omp_rt_parallel_for(0, N, &body_fn, N, In, Out);


// Parallel region outlined in the front-end (clang)!
void body_fn(int i, int N,
             float* /*  read-only & no-escape */ In,
             float* /* write-only & no-escape */ Out) {
  Out[i] = In[i] + In[i + N];
}
```

Optimization I: Attribute Propagation

OpenMP Input:
```
#pragma omp parallel for firstprivate(...)
for (int i = 0; i < N; i++)
  Out[i] = In[i] + In[i+N];
```

```
// Parallel region replaced by a runtime call.
omp_rt_parallel_for(0, N, &body_fn, N,
      /* ro & no-esc */ In, /* wo & no-esc */ Out);

// Parallel region outlined in the front-end (clang)!
void body_fn(int i, int N,
             float* /*  read-only & no-escape */ In,
             float* /* write-only & no-escape */ Out) {
  Out[i] = In[i] + In[i + N];
}
```

```
int foo() {
  int a = 0;






#pragma omp parallel
  {
#pragma omp critical
    { a += 1; }
    bar();
#pragma omp critical
    { a *= 2; }
  }
  return a;
}
```

Optimization I: Attribute Propagation (cont)

```c
int foo() {
  int a = 0;







#pragma omp parallel
  {
#pragma omp critical
    { a += 1; }
    bar();
#pragma omp critical
    { a *= 2; }
  }
  return a;
}
```

```c
int foo() {
  int a = 0;
  int *restrict p = &a;
  omp_rt_parallel_for(pwork, p);
  return a;
}
void pwork(int tid, int *p) {
  if (omp_critical(tid)) {
    *p = *p + 1;
    omp_critical_end(tid);
  }
  bar();
  if (omp_critical(tid)) {
    *p = *p * 2;
    omp_critical_end(tid);
  }
}
```

Optimization I: Attribute Propagation (cont)

```c
void pwork(int tid,
           int *restrict p) {
  if (omp_critical(tid)) {
    omp_critical_end(tid);
  }
  bar();
  if (omp_critical(tid)) {
    *p = 2 * (*p + 1);
    omp_critical_end(tid);
  }
}
```

```c
int foo() {
  int a = 0;
  int *restrict p = &a;
  omp_rt_parallel_for(pwork, p);
  return a;
}
void pwork(int tid, int *p) {
  if (omp_critical(tid)) {
    *p = *p + 1;
    omp_critical_end(tid);
  }
  bar();
  if (omp_critical(tid)) {
    *p = *p * 2;
    omp_critical_end(tid);
  }
}
```

Optimization I: Attribute Propagation (cont)

```
int foo() {
  int a = 0;
  int *restrict p = &a;
  omp_rt_parallel_for(pwork, p);
  return a;
}
void pwork(int tid, int *p) {
  if (omp_critical(tid)) {
    *p = *p + 1;
    omp_critical_end(tid);
  }
  bar();
  if (omp_critical(tid)) {
    *p = *p * 2;
    omp_critical_end(tid);
  }
}
```

```
void pwork(int tid,
           int *restrict p) {
  if (omp_critical(tid)) {
    *p = *p + 1;
    omp_critical_end(tid);
  }
  bar()[p]; // May "use" p.
  if (omp_critical(tid)) {
    *p = *p * 2;
    omp_critical_end(tid);
  }
}
```

Optimization II: Variable Privatization

OpenMP Input:

```
#pragma omp parallel for shared(...)
for (int i = 0; i < N; i++)
  Out[i] = In[i] + In[i+N];
```

Optimization II: Variable Privatization

OpenMP Input:

```
#pragma omp parallel for shared(...)
for (int i = 0; i < N; i++)
  Out[i] = In[i] + In[i+N];
```

```
// Parallel region replaced by a runtime call.
omp_rt_parallel_for(0, N, &body_fn, &N, &In, &Out);
```

```
// Parallel region outlined in the front-end (clang)!
void body_fn(int i, int* N,
             float** In,
             float** Out) {
  (*Out)[i] = (*In)[i] + (*In)[i + (*N)];
}
```

Optimization II: Variable Privatization

OpenMP Input:
```
#pragma omp parallel for shared(...)
for (int i = 0; i < N; i++)
  Out[i] = In[i] + In[i+N];
```

```
// Parallel region replaced by a runtime call.
omp_rt_parallel_for(0, N, &body_fn, &N, &In, &Out);


// Parallel region outlined in the front-end (clang)!
void body_fn(int i, int* /* ro & ne */ N,
             float** /* ro & ne */ In,
             float** /* ro & ne */ Out) {
  (*Out)[i] = (*In)[i] + (*In)[i + (*N)];
}
```

Optimization II: Variable Privatization

OpenMP Input:
```
#pragma omp parallel for firstprivate(...)
for (int i = 0; i < N; i++)
  Out[i] = In[i] + In[i+N];
```

```
// Parallel region replaced by a runtime call.
omp_rt_parallel_for(0, N, &body_fn, N, In, Out);


// Parallel region outlined in the front-end (clang)!
void body_fn(int i, int N,
             float* In,
             float* Out) {
  Out[i] = In[i] + In[i + N];
}
```

Optimization III: Parallel Region Expansion

Optimization III: Parallel Region Expansion

```
void copy(float* dst, float* src, int N) {
  #pragma omp parallel for
  for(int i = 0; i < N; i++) {
    dst[i] = src[i];
  } // implicit barrier!
}

void compute_step_factor(int nelr, float* vars,
                         float* areas, float* sf) {
  #pragma omp parallel for
  for (int blk = 0; blk < nelr / block_length; ++blk) {
    ...
  } // implicit barrier!
}
```

Optimization III: Parallel Region Expansion

```
for (int i = 0; i < iterations; i++) {
  copy(old_vars, vars, nelr * NVAR);



  compute_step_factor(nelr, vars, areas, sf);



  for (int j = 0; j < RK; j++) {
    compute_flux(nelr, ese, normals, vars, fluxes, ff_vars,
                 ff_m_x, ff_m_y, ff_m_z, ff_dnergy);

    time_step(j, nelr, old_vars, vars, sf, fluxes);
```

Optimization III: Parallel Region Expansion

```
for (int i = 0; i < iterations; i++) {
  #pragma omp parallel for          // copy
  for (...) {
    /* write old_vars, read vars */
  } // implicit barrier!
  compute_step_factor(nelr, vars, areas, sf);



  for (int j = 0; j < RK; j++) {
    compute_flux(nelr, ese, normals, vars, fluxes, ff_vars,
                 ff_m_x, ff_m_y, ff_m_z, ff_dnergy);

    time_step(j, nelr, old_vars, vars, sf, fluxes);
```

Optimization III: Parallel Region Expansion

```
for (int i = 0; i < iterations; i++) {
  #pragma omp parallel for              // copy
  for (...) {
    /* write old_vars, read vars */
  } // implicit barrier!
  #pragma omp parallel for              // compute_step_factor
  for (...) {
    /* write sf, read vars & area */
  } // implicit barrier!
  for (int j = 0; j < RK; j++) {
    #pragma omp parallel for            // compute_flux
    for (...) {
      /* write fluxes, read vars & ... */
    } // implicit barrier!
    ...
```

Optimization III: Parallel Region Expansion

```
#pragma omp parallel
for (int i = 0; i < iterations; i++) {
  #pragma omp for                       // copy
  for (...) {
    /* write old_vars, read vars */
  } // explicit barrier in LLVM-IR!
  #pragma omp for                       // compute_step_factor
  for (...) {
    /* write sf, read vars & area */
  } // explicit barrier in LLVM-IR!
  for (int j = 0; j < RK; j++) {
    #pragma omp for                     // compute_flux
    for (...) {
      /* write fluxes, read vars & ... */
    } // explicit barrier in LLVM-IR!
    ...
```

Optimization IV: Barrier Elimination

```
#pragma omp parallel
for (int i = 0; i < iterations; i++) {
  #pragma omp for                        // copy
  for (...) {
    /* write old_vars, read vars */
  } // explicit barrier in LLVM-IR!
  #pragma omp for                        // compute_step_factor
  for (...) {
    /* write sf, read vars & area */
  } // explicit barrier in LLVM-IR!
  for (int j = 0; j < RK; j++) {
    #pragma omp for                      // compute_flux
    for (...) {
      /* write fluxes, read vars & ... */
    } // explicit barrier in LLVM-IR!
    ...
```

Optimization IV: Barrier Elimination

```
#pragma omp parallel
for (int i = 0; i < iterations; i++) {
  #pragma omp for                            // copy
  for (...) {
    /* write old_vars, read vars */
  } // explicit barrier in LLVM-IR!
  #pragma omp for                            // compute_step_factor
  for (...) {
    /* write sf, read vars & area */
  } // explicit barrier in LLVM-IR!
  for (int j = 0; j < RK; j++) {
    #pragma omp for                          // compute_flux
    for (...) {
      /* write fluxes, read vars & ... */
    } // explicit barrier in LLVM-IR!
    ...
```

Optimization IV: Barrier Elimination

```
#pragma omp parallel
for (int i = 0; i < iterations; i++) {
  #pragma omp for nowait                 // copy
  for (...) {
    /* write old_vars, read vars */
  }
  #pragma omp for nowait                 // compute_step_factor
  for (...) {
    /* write sf, read vars & area */
  }
  for (int j = 0; j < RK; j++) {
    #pragma omp for                      // compute_flux
    for (...) {
      /* write fluxes, read vars & ... */
    } // explicit barrier in LLVM-IR!
    ...
```

Optimization V: Communication Optimization

Optimization V: Communication Optimization

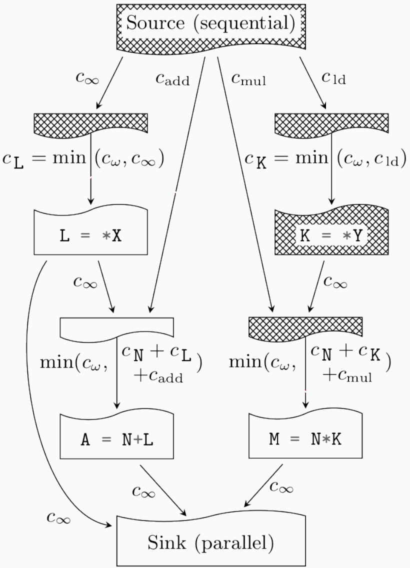
```
void f(int *X, int *restrict Y) {
  int L = *X;          // immovable
  int N = 512;         //   movable




  int A = N + L;       //   movable
  #pragma omp parallel for      \
      firstprivate(X, Y, N, L, A)
  for (int i = 0; i < N; i++) {
    int K = *Y;          //   movable
    int M = N * K;       //   movable
    X[i] = M+A*L*i;      // immovable
  }
}
```

Optimization V: Communication Optimization

```
void f(int *X, int *restrict Y) {
  int L = *X;          // immovable
  int N = 512;         //   movable



  int A = N + L;       //   movable
  #pragma omp parallel for        \
      firstprivate(X, Y, N, L, A)
  for (int i = 0; i < N; i++) {
    int K = *Y;         //   movable
    int M = N * K;      //   movable
    X[i] = M+A*L*i;     // immovable
  }
}
```

$c_\infty = \infty$ $c_{\text{add}} = 5$ $c_\omega = 15$ $c_{\text{mul}} = 10$

$c_{\text{ld}} = 20$ $c_{\text{N}} = c_{\text{cst}}$ $c_{\text{cst}} = 0$ cut

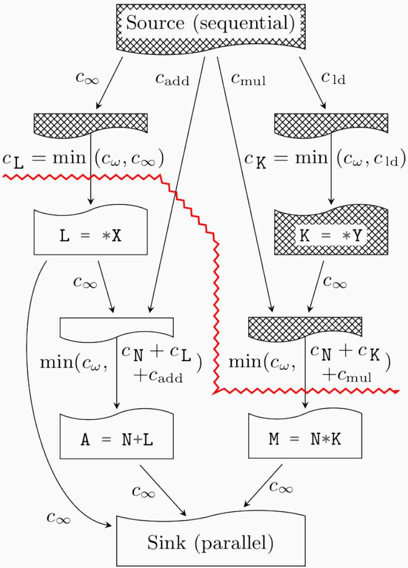


Optimization V: Communication Optimization

```
void f(int *X, int *restrict Y) {
  int L = *X;           // immovable
  int N = 512;          //   movable



  int A = N + L;        //   movable
  #pragma omp parallel for      \
      firstprivate(X, Y, N, L, A)
  for (int i = 0; i < N; i++) {
    int K = *Y;         //   movable
    int M = N * K;      //   movable
    X[i] = M+A*L*i;     // immovable
  }
}
```

$c_\infty = \infty$ $c_{\text{add}} = 5$ $c_\omega = 15$ $c_{\text{mul}} = 10$

$c_{\text{ld}} = 20$ $c_{\text{N}} = c_{\text{cst}}$ $c_{\text{cst}} = 0$ cut



Optimization V: Communication Optimization

```c
void f(int *X, int *restrict Y) {
  int L = *X;          // immovable
  int N = 512;         //   movable
  int A = N + L;       //   movable
  #pragma omp parallel for       \
      firstprivate(X, Y, N, L, A)
  for (int i = 0; i < N; i++) {
    int K = *Y;          //   movable
    int M = N * K;       //   movable
    X[i] = M+A*L*i;      // immovable
  }
}
```

```c
void g(int *X, int *restrict Y) {
  int L = *X;       // immovable
  int K = *Y;       // c_ld > c_ω
  int M = 512 * K;  // c_mul + c_K > c_ω
  #pragma omp parallel           \
      firstprivate(X, M, L)
  {
    int A = 512 + L; // c_add < c_ω
    #pragma omp for              \
        firstprivate(X, M, A, L)
    for (int i = 0; i < 512; i++) {
      X[i] = M+A*L*i; // immovable
    }
  }
}
```

Early Outlining: Sequential Optimization Problems

Information Transfer

Value Transfer

NO Information Transfer:

outlined function $\Longleftrightarrow$ *runtime library call site*

Value Transfer

Early Outlining: Sequential Optimization Problems

NO Information Transfer:

outlined function $\Longleftrightarrow$ *runtime library call site*

Declaration	Value Transfer OpenMP Clause	Communication Type
`T var;`	*default = shared*	
`T var;`	`shared(var)`	
`T var;`	`lastprivate(var)`	

Early Outlining: Sequential Optimization Problems

NO Information Transfer:

outlined function $\Longleftrightarrow$ *runtime library call site*

Declaration	Value Transfer OpenMP Clause	Communication Type
`T var;`	*default = shared*	`&var` of type `T*`
`T var;`	`shared(var)`	`&var` of type `T*`
`T var;`	`lastprivate(var)`	`&var` of type `T*`

Early Outlining: Sequential Optimization Problems

NO Information Transfer:

outlined function $\Longleftrightarrow$ *runtime library call site*

Declaration	Value Transfer OpenMP Clause	Communication Type
`T var;`	*default = shared*	`&var` of type `T*`
`T var;`	`shared(var)`	`&var` of type `T*`
`T var;`	`lastprivate(var)`	`&var` of type `T*`
`T var;`	`firstprivate(var)`	`var` of type `T`

Early Outlining: Sequential Optimization Problems

NO Information Transfer:

outlined function $\Longleftrightarrow$ *runtime library call site*

Declaration	Value Transfer OpenMP Clause	Communication Type
`T var;`	*default = shared*	`&var` of type `T*`
`T var;`	`shared(var)`	`&var` of type `T*`
`T var;`	`lastprivate(var)`	`&var` of type `T*`
`T var;`	`firstprivate(var)`	`var` of type `T`
`T var;`	`private(var)`	*none*

Target Region — The Interface

```
void kernel(...) {

init:
  char ThreadKind = __kmpc_target_region_kernel_init(...);
  if (ThreadKind == -1) {                //  actual worker thread
    if (!UsedLibraryStateMachine)
      user_code_state_machine();
    goto exit;
  } else if (ThreadKind == 0) {          // surplus worker thread
    goto exit;
  } else {                               //    team master thread
    goto user_code;
  }

user_code:
  // User defined kernel code, parallel regions are replaced by
  // by __kmpc_target_region_kernel_parallel(...) calls.

  // Fallthrough to de-initialization
deinit:
  __kmpc_target_region_kernel_deinit(...);

exit:
  /* exit the kernel */
}
```

Target Region — The Interface

```c
// Initialization
int8_t __kmpc_target_region_kernel_init(ident_t *Ident,
                                        bool UseSPMDMode,
                                        bool RequiresOMPRuntime,
                                        bool UseStateMachine,
                                        bool RequiresDataSharing);


// De-Initialization
void __kmpc_target_region_kernel_deinit(ident_t *Ident,
                                        bool UseSPMDMode,
                                        bool RequiredOMPRuntime);


// Parallel execution
typedef void (*ParallelWorkFnTy)(void * /* SharedValues */,
                                 void * /* PrivateValues */)

CALLBACK(ParallelWorkFnTy, SharedValues, PrivateValues)
void __kmpc_target_region_kernel_parallel(ident_t *Ident,
  bool UseSPMDMode, bool RequiredOMPRuntime,
  ParallelWorkFnTy ParallelWorkFn, void *SharedValues,
  uint16_t SharedValuesBytes, void *PrivateValues,
  uint16_t PrivateValuesBytes, bool SharedMemPointers);
```

Target Region — The Implementation

- (almost) the same as with the current NVPTX backend, except for shared/firstprivate variables
- implemented in Cuda as part of the library, not generated into the user code/module/TU by Clang
- the boolean flags are commonly constant, after inlining all target region abstractions is gone

Action Item III

†

Action Item III

1) Review your OpenMP target code.

†

1) Review your OpenMP target code.

2) Email me[†] if you use the "bad" pattern!

[†] jdoerfert@anl.gov

Current Work — Reviews, Evaluation, Features, Hardening

- started the review process

Current Work — Reviews, Evaluation, Features, Hardening

- started the review process
- more test-cases needed to determine benefit

Current Work — Reviews, Evaluation, Features, Hardening

- started the review process
- more test-cases needed to determine benefit
- more developers needed to add missing features

Current Work — Reviews, Evaluation, Features, Hardening

- started the review process
- more test-cases needed to determine benefit
- more developers needed to add missing features
- more users/developers needed to improve test coverage

Current Work — Reviews, Evaluation, Features, Hardening

Interested? Please let me know!

- started the review process
- more test-cases needed to determine benefit
- more developers needed to add missing features
- more users/developers needed to improve test coverage

Future Work — Optimizations, Front-ends, Targets

- improve and extend the LLVM's OpenMP optimizations: connection to abstract callsites, memory placement, ...

Future Work — Optimizations, Front-ends, Targets

- improve and extend the LLVM's OpenMP optimizations: connection to abstract callsites, memory placement, ...
- use target regions in other "front-ends": F18, Polly, Rust?, ...

Future Work — Optimizations, Front-ends, Targets

- improve and extend the LLVM's OpenMP optimizations: connection to abstract callsites, memory placement, ...
- use target regions in other "front-ends": F18, Polly, Rust?, ...
- implement the interface for other targets: GPUs, FPGAs?, ...

Future Work — Optimizations, Front-ends, Targets

Interested? Please let me know!

- improve and extend the LLVM's OpenMP optimizations: connection to abstract callsites, memory placement, ...
- use target regions in other "front-ends": F18, Polly, Rust?, ...
- implement the interface for other targets: GPUs, FPGAs?, ...