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/
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.
Compiler Optimization

Original Program

```c
int y = 7;

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

g(y);
```

After Optimizations

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

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

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

After Optimizations

```c
int y = 7;
#pragma omp parallel for
for (i = 0; i < N; i++) {
    f(y, i);
}
g(y);
```
CURRENT COMPILER OPTIMIZATION FOR PARALLELISM
None\textsuperscript{*†}

\textsuperscript{*}At least for LLVM/Clang up to 8.0
\textsuperscript{†}And not considering smart runtime libraries!
Why is this important?
Performance Implications

![Box plot comparing ./lulesh 10 1 results]

- **0.3721**
- **0.3857**
- **0.00%**
- **-3.64%**

*Versions: base.seq, base.par*
Performance Implications

./pathfinder 1000 1000 1

4.5148  
0.00%

6.4912  
-43.78%

million cycles

base.seq  base.par

versions
Performance Implications

./pathfinder 1000 1000 1

4.5148
0.00%
6.4912
-43.78%

./srad_v2 2048 2048 0 127 0 127 1 0.5 20

1.077
0.00%
2.626
-143.83%
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

\[ \text{a} \]
\[ \text{b} \]
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
Optimization Categories

Optimizations for *sequential* aspects

- May reuse *existing* transformations  
  ⇒ Introduce *suitable abstractions* to bridge the indirection  
  (patches up for review!)

Optimizations for *parallel* aspects

- New *explicit parallelism-aware transformations*  
  ⇒ Introduce a unifying abstraction layer  
  (DONE!)

---

*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
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
{  
    #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 :}
The Compiler Black Box

>> clang -03 -fopenmp-targets=...
>> clang -O3 -fopenmp-targets=...

{
    #pragma omp target teams parallel
    work1();

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

“relatively” good performance :(
The Compiler Black Box

>> clang -03 -fopenmp-targets=...

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

    #pragma omp parallel
    work2();
}
The Compiler Black Box

>> clang -03 -fopenmp-targets=...

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

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

probably poor performance :(
```c
{
    #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,
The Compiler Black Box — Behind the Curtain (of Clang)

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

    bar();
}

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

Problems:
- A separate master team costs resources.
- Synchronization has overhead.
- Currently impossible to optimization.
The Compiler Black Box — Behind the Curtain (of Clang)

```c
#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.

Problems:
- a separate master team costs resources
- synchronization has overhead
- currently impossible to optimization
#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.

Problems:
- a separate master team costs resources
- synchronization has overhead
#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.

Problems:
• a separate master team costs resources
• synchronization has overhead
• currently impossible to optimization
## OpenMP Offload — Overview

<table>
<thead>
<tr>
<th>OpenMP</th>
<th>Clang</th>
<th>LLVM-IR</th>
<th>LLVM</th>
<th>Assembly</th>
</tr>
</thead>
</table>

- Few RT Calls
- Device Opt.
- Host Code
- Fat Binary
- Cross Module IPO
### OpenMP Offload — Overview

<table>
<thead>
<tr>
<th>OpenMP</th>
<th>Clang</th>
<th>LLVM-IR</th>
<th>LLVM</th>
<th>Assembly</th>
</tr>
</thead>
</table>

- **Code**
# OpenMP Offload — Overview

<table>
<thead>
<tr>
<th>OpenMP</th>
<th>Clang</th>
<th>LLVM-IR</th>
<th>LLVM</th>
<th>Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>+</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>RT Calls + Logic</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Host</td>
<td>Code</td>
<td>Code Gen.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fat Binary
OpenMP Offload — Overview

1. Offload-Specific Optimizations on Device Code

- Code
- Code Gen.
- Device Code
- Host Code
- Host Code

OpenMP
Clang
LLVM-IR
LLVM
Assembly
OpenMP Offload — Overview

1. Offload-Specific Optimizations on Device Code

- RT Calls
- Device Code + RT Calls
- Host Code

OpenMP → Code Generation → Device Code + RT Calls → Host Code
OpenMP Offload — Overview

1. Offload-Specific Optimizations on Device Code

- Code
- Code Gen.
- Device Code
- Device Code + RT Calls + Logic
- Host Code
- Fat Binary

OpenMP
Clang
LLVM-IR
LLVM
Assembly

---

OpenMP Offload — Overview

1. Offload-Specific Optimizations on Device Code

- Code
- Code Gen.
- Device Code
- Device Code + RT Calls + Logic
- Host Code
- Fat Binary

OpenMP
Clang
LLVM-IR
LLVM
Assembly

---

OpenMP Offload — Overview

1. Offload-Specific Optimizations on Device Code

- Code
- Code Gen.
- Device Code
- Device Code + RT Calls + Logic
- Host Code
- Fat Binary

OpenMP
Clang
LLVM-IR
LLVM
Assembly

---
OpenMP Offload — Overview

1. Offload-Specific Optimizations on Device Code

- RT Calls
- Logic

Device RT

Device Code + RT Calls + Logic

Host Code

OpenMP  Clang  LLVM-IR  LLVM  Assembly

OpenMP Offload — Overview

1. Offload-Specific Optimizations on Device Code

- Code
- Code Gen.
- Device RT
- Device Code + RT Calls + Logic
- Host Code
- Host Opt.

OpenMP
Clang
LLVM-IR
LLVM
Assembly

Fat Binary

Clang
LLVM-IR
LLVM
Assembly

Cross Module IPO

8/14
OpenMP Offload — Overview

1. Offload-Specific Optimizations on Device Code

- Code
- RT Calls
- Logic

- Host Code
- Host Opt.

- Device Code
- RT Calls
- Logic

- Device RT
- Device Opt.

- Code Gen.
1. Offload-Specific Optimizations on Device Code

- Code
- Code Gen.
- Device RT
- Device Code + RT Calls + Logic
- Host Code
- Host Opt.
- Device Opt.
- Fat Binary

Diagram shows the flow of code generation and optimization processes from OpenMP, Clang, LLVM-IR, LLVM to Assembly, including device-specific optimizations and runtime calls.
1. Offload-Specific Optimizations on Device Code

- Code
- Code Gen.
- Device Code + RT Calls + Logic
- Host Code
- Host Opt.
- Device Opt.
- Device RT + Logic
- Code
- Device RT Calls
- Assembly
- Fat Binary

OpenMP Offload — Overview & Directions
1. Offload-Specific Optimizations on Device Code

- Code
- Code Gen.
- Code
- Device Code + few RT Calls + Logic
- Device RT + Logic
- Device Opt.
- Host Code
- Host Opt.
- Fat Binary

OpenMP Offload — Overview & Directions
1. Offload-Specific Optimizations on Device Code

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

1. Offload-Specific Optimizations on Device Code

- Code
- Code Gen.
- Host Code
- Device Code (few RT Calls + Logic)
- Device RT + Logic

- Host Opt.
- Device Opt.

- Fat Binary

OpenMP → Clang → LLVM-IR → LLVM → Assembly
2. Optimize Device and Host Code Together

- Code generation:
  - OpenMP Code
  - Clang Code
  - LLVM-IR Device Code + few RT Calls
  - Host Code

- Optimization:
  - Device RT + Logic
  - Host Code

- Assembly:
  - Device Opt.
  - Host Opt.

- Fat Binary
2. Optimize Device and Host Code Together

- Code
- Code Gen.
- Host Code
- Host Opt.
- Device RT + Logic
- Device Code + few RT Calls
- Cross Module IPO
- Fat Binary

OpenMP
Clang
LLVM-IR
LLVM
Assembly
2. Optimize Device and Host Code Together
Pending Patches — Target Region Interface

• A straightforward #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
Pending Patches — Target Region Interface

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

[Image 9x13 to 18x20]
• A straight-forward `#pragma omp target` front-end:

  ◦ simplified implementation  
  ◦ improved reusability (F18, …)

  \[\text{CGOpenMPRuntimeNVPTX.cpp} \sim 5.0\text{ k loc}\]

  \[\text{CGOpenMPRuntimeTRegion.cpp} \sim 0.5\text{ k loc}\]
• A straight-forward `#pragma omp target` front-end:
  ◊ simplified implementation
  ◊ improved reusability (F18, ...)

• 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
  ◊ improved reusability (F18, ...)

• 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
• 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`
1. **Offload-Specific Optimizations — “SPMD-zation”**

- 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.
1. **Offload-Specific Optimizations — Custom State Machines**

- 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
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
• 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

- CallInst
- InvokeInst
- CallSite
- Passes (IPOs)
2. Optimize Device and Host Together — Abstract Call Sites

- CallInst
- InvokeInst
- CallSite
- TransitiveCallSite
- AbstractCallSite
- Passes (IPOs)
Functional changes required for Inter-procedural Constant Propagation:

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

```
./srad_v2 2048 2048 0 127 0 127 1 0.5 20
```

-143.83%
Abstract Call Sites — Performance Results

![Graph showing performance results with various versions.](image)
Abstract Call Sites — Performance Results

![Graph showing performance results for different versions of a program with improvements or decrements indicated.]
Conclusion
The Compiler Black Box

```c
>> clang -03 -fopenmp-targets=...

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

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

probably poor performance:
Conclusion

The Compiler Black Box

```c
>> 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\(^1\))
- Introduce a unifying abstraction layer (see EuroLLVM'18\(^2\))

\(^1\)Compiler Optimizations For OpenMP, J. Doerffert, H. Finkel, IWOMP 2018
\(^2\)A Parallel IR in Real Life: Optimizing OpenMP. H. Finkel, J. Doerffert, X. Tian, G. Stelle, Euro LLVM Meeting 2018
Conclusion

The Compiler Black Box

```c
#include "omp.h"

#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

<table>
<thead>
<tr>
<th>OpenMP</th>
<th>Clang</th>
<th>LLVM-IR</th>
<th>LLVM</th>
<th>Assembly</th>
</tr>
</thead>
</table>

- **Device RT+Logic**
- **Device Code + few RT Calls + Logic**
- **Host AND Device Optimization**
- **Fat Binary**

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)
- Introduce a unifying abstraction layer (see EuroLLVM’18)

---

*Compiler Optimizations for OpenMP, J. Doerfler, H. Finkel, IWOMP 2018
1A Parallel IR in Real Life: Optimizing OpenMP, H. Finkel, J. Doerfler, X. Tian, G. Stelle, Euro LLVM Meeting 2018
Conclusion

**The Compiler Black Box**

```c
#include <openmp>
void work1()
{
    #pragma omp target
    #pragma omp parallel
    work1();
}

void work2()
{
    #pragma omp target
    #pragma omp parallel
    work2();
}
```

probably poor performance :(

---

**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, …

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

III: Parallel Region Expansion — Maximize Parallel Contexts: ⟹ 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

I: Attribute Propagation — Bidirectional Information Transfer:
read/write-only, restrict/noalias, ...
OpenMP-Aware Optimizations

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

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

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

III: Parallel Region Expansion — Maximize Parallel Contexts:
    → 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. & parallel compute
I: Attribute Propagation — Bidirectional Information Transfer:
read/write-only, restrict/noalias, ...

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

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

IV: Barrier Elimination — Eliminate Redundant Barriers
OpenMP-Aware Optimizations

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

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

III: Parallel Region Expansion — Maximize Parallel Contexts:
    → 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

(See IWOMP’18)
I: Attribute Propagation — Bidirectional Information Transfer: 
read/write-only, restrict/noalias, ...

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

III: Parallel Region Expansion — Maximize Parallel Contexts:
$\rightarrow$ 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

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

II: Variable Privatization — In LLVM: Argument Promotion (IPO!)
   shared(var) ⟷ firstprivate(var) ⟷ private(var)

III: Parallel Region Expansion — Maximize Parallel Contexts:
   ⟷ 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

(see IWOMP’18)
#pragma omp parallel for

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

OpenMP Input:  

```c
for (int i = 0; i < N; i++)
    Out[i] = In[i] + In[i+N];
```

// Parallel region replaced by a runtime call.

```c
omp_rt_parallel_for(0, N, &body_fn, &N, &In, &Out);
```
Early Outlining

```c
#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)]
}
```
#pragma omp parallel for

OpenMP Input:  
\begin{verbatim}
  for (int i = 0; i < N; i++)
    Out[i] = In[i] + In[i+N];
\end{verbatim}

// 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

#pragma omp parallel for

OpenMP Input:  
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)]
}


#pragma omp parallel for

OpenMP Input: 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

```c
#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(void, &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];
}
```

+ **valid and executable IR**
- integration cost per IPO
+ no unintended interactions
IPO in LLVM

- CallInst
- InvokeInst

CallSite

Passes
(IPOs)
Transitive Call Sites in LLVM

Diagram:
- CallInst
- InvokeInst
- CallSite
- TransitiveCallSite
- AbstractCallSite
- Passes (IPOs)
Functional changes required for Inter-procedural Constant Propagation:

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

<table>
<thead>
<tr>
<th>Version</th>
<th>Description</th>
<th>Opt.</th>
</tr>
</thead>
<tbody>
<tr>
<td>base</td>
<td>plain &quot;-O3&quot;, thus no parallel optimizations</td>
<td></td>
</tr>
<tr>
<td>attr</td>
<td>attribute propagation through attr. deduction (IPO)</td>
<td>I</td>
</tr>
<tr>
<td>argp</td>
<td>variable privatization through arg. promotion (IPO)</td>
<td>II</td>
</tr>
<tr>
<td>n/a</td>
<td>constant propagation (IPO)</td>
<td></td>
</tr>
</tbody>
</table>
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.
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

/bfs 1 graph1MW_6.txt

milliseconds

69.955 75.769 69.66 73.918 69.934 75.107 69.991 73.204

-8.31% 0.42% -4.64%

base.s  base.p  attr.s  attr.p  argp.s  argp.p  attr_argp.s  attr_argp.p
Performance Results

/pathfinder 1000 1000 1

<table>
<thead>
<tr>
<th>versions</th>
<th>cycles</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>base.s</td>
<td>4.5148</td>
<td></td>
</tr>
<tr>
<td>base.p</td>
<td>6.4912</td>
<td>-43.78%</td>
</tr>
<tr>
<td>attr.s</td>
<td>4.4922</td>
<td></td>
</tr>
<tr>
<td>attr.p</td>
<td>5.8961</td>
<td></td>
</tr>
<tr>
<td>argp.s</td>
<td>4.5513</td>
<td></td>
</tr>
<tr>
<td>argp.p</td>
<td>6.3812</td>
<td></td>
</tr>
<tr>
<td>attr_argp.s</td>
<td>4.5449</td>
<td>20.26%</td>
</tr>
<tr>
<td>attr_argp.p</td>
<td>3.6</td>
<td></td>
</tr>
</tbody>
</table>
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
1) Run your OpenMP code sequentially\(^*\), with and without OpenMP.

\[^*\text{export OMP\_NUM\_THREADS=1}\]

\[^{†}\text{jdoerfert@anl.gov}\]
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

1) Always \texttt{use default(none)} and \texttt{firstprivate(...)}

2) Revisit Action Item I for scalars/pointers if you do not have explicit synchronization.
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(...)` *For scalars/pointers if you do not have explicit synchronization.*

2) Revisit ACTION ITEM I
NO need to “share” the variable A!

```c
#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] ... gamma[3][7]
    }
Optimization I: Attribute Propagation

OpenMP Input:

```c
#pragma omp parallel for firstprivate(...) for (int i = 0; i < N; i++)
    Out[i] = In[i] + In[i+N];
```
#pragma omp parallel for firstprivate(...) 
OpenMP Input:  
```c
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)!
```c
void body_fn(int i, int N,
             float* In,
             float* Out) {
    Out[i] = In[i] + In[i + N];
}
```
#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];
}


#pragma omp parallel for firstprivate(...) 

OpenMP Input: 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];
}
```c
int foo() {
    int a = 0;

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

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

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

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);
    }
}
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 = 2 * (*p + 1);
        omp_critical_end(tid);
    }
}

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);
    }
}

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 II: Variable Privatization

OpenMP Input:

```c
#pragma omp parallel for shared(...) for (int i = 0; i < N; i++)
    Out[i] = In[i] + In[i+N];
```
Optimization II: Variable Privatization

#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)];  
}
#pragma omp parallel for shared(...)

OpenMP Input:  
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)];
}
OpenMP Input:

```c
#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];
}
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!
}
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);
    }
}
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);
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!
    }
    ...

#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!
        ...
    }
#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!
...
#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!
    ...

#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!
    }
}
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
    }
}
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
    }
}

\( c_\infty = \infty \quad c_{\text{add}} = 5 \quad c_\omega = 15 \quad c_{\text{mul}} = 10 \)
\( c_{\text{id}} = 20 \quad c_N = c_{\text{cst}} \quad c_{\text{cst}} = 0 \)
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
    }
}

\[ c_\infty = \infty \quad c_{\text{add}} = 5 \quad c_\omega = 15 \quad c_{\text{mul}} = 10 \]
\[ c_{\text{id}} = 20 \quad c_N = c_{\text{cst}} \quad c_{\text{cst}} = 0 \]
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
    }
}

void g(int *X, int *restrict Y) {
    int L = *X; // immovable
    int K = *Y; // c_{ld} > c_{\omega}
    int M = 512 * K; // c_{mul} + c_{K} > c_{\omega}
    #pragma omp parallel
    firstprivate(X, M, L)
    {
        int A = 512 + L; // c_{add} < c_{\omega}
        #pragma omp for
        firstprivate(X, M, A, L)
        for (int i = 0; i < 512; i++) {
            X[i] = M + A * L * i; // immovable
        }
    }
}
Information Transfer

Value Transfer
NO Information Transfer:
outlined function ↔ runtime library call site

Value Transfer
NO Information Transfer:
outlined function ⇔ runtime library call site

<table>
<thead>
<tr>
<th>Declaration</th>
<th>Value Transfer</th>
<th>OpenMP Clause</th>
<th>Communication Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>T var;</td>
<td>default = shared</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T var;</td>
<td>shared(var)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T var;</td>
<td>lastprivate(var)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T var;</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## EARLY OUTLINING: SEQUENTIAL OPTIMIZATION PROBLEMS

**NO Information Transfer:**

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

<table>
<thead>
<tr>
<th>Declaration</th>
<th>Value Transfer</th>
<th>Communication Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>T var;</code></td>
<td><code>default = shared</code></td>
<td><code>&amp;var</code> of type <code>T</code>*</td>
</tr>
<tr>
<td><code>T var;</code></td>
<td><code>shared(var)</code></td>
<td><code>&amp;var</code> of type <code>T</code>*</td>
</tr>
<tr>
<td><code>T var;</code></td>
<td><code>lastprivate(var)</code></td>
<td><code>&amp;var</code> of type <code>T</code>*</td>
</tr>
</tbody>
</table>
### Early Outlining: Sequential Optimization Problems

**NO Information Transfer:**

outlined function \(\leftrightarrow\) runtime library *call site*

<table>
<thead>
<tr>
<th>Declaration</th>
<th>Value Transfer</th>
<th>OpenMP Clause</th>
<th>Communication Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>T var;</td>
<td></td>
<td>default = shared</td>
<td>&amp;var of type T*</td>
</tr>
<tr>
<td>T var;</td>
<td></td>
<td>shared(var)</td>
<td>&amp;var of type T*</td>
</tr>
<tr>
<td>T var;</td>
<td></td>
<td>lastprivate(var)</td>
<td>&amp;var of type T*</td>
</tr>
<tr>
<td>T var;</td>
<td></td>
<td>firstprivate(var)</td>
<td>var of type T</td>
</tr>
</tbody>
</table>
### NO Information Transfer:
outlined function ⟷ runtime library call site

<table>
<thead>
<tr>
<th>Declaration</th>
<th>Value Transfer</th>
<th>OpenMP Clause</th>
<th>Communication Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>T var;</td>
<td></td>
<td>default = shared</td>
<td>&amp;var of type T*</td>
</tr>
<tr>
<td>T var;</td>
<td></td>
<td>shared(var)</td>
<td>&amp;var of type T*</td>
</tr>
<tr>
<td>T var;</td>
<td></td>
<td>lastprivate(var)</td>
<td>&amp;var of type T*</td>
</tr>
<tr>
<td>T var;</td>
<td></td>
<td>firstprivate(var)</td>
<td>var of type T</td>
</tr>
<tr>
<td>T var;</td>
<td></td>
<td>private(var)</td>
<td>none</td>
</tr>
</tbody>
</table>
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

// 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);
• (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

1) Review your OpenMP target code.
2) Email me† if you use the “bad” pattern!

†jdoerfert@anl.gov
1) Review your OpenMP target code.

†
1) Review your OpenMP target code.

2) Email me† if you use the “bad” pattern!

---

†jdoerfert@anl.gov
• started the review process
• started the review process
• more test-cases needed to determine benefit
• started the review process
• more test-cases needed to determine benefit
• more developers needed to add missing features
• 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
• 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

Interested? Please let me know!
Future Work — Optimizations, Front-ends, Targets

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

• 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?, ...

Interested? Please let me know!