A Compiler’s View of OpenMP

Johannes Doerfert, Argonne National Laboratory
A Compiler’s View of OpenMP

Johannes Doerfert  (Argonne National Laboratory)
About Me

PhD in CS from Saarland University, Saarbrücken, Germany

Researcher at Argonne National Laboratory (ANL), Chicago, USA

Active in the LLVM community since 2014, in the OpenMP community since 2018

Code owner for OpenMP offloading in LLVM (officially) since recently
Background
LLVM in a Nutshell

- open (source/community/...)
- extensible, “fixable”
- portable (GPUs, CPUs, ...)
- C++/OpenMP/SYCL/HIP/CUDA/... feature complete 😏
- early access to *the coolest* features

- performant and correct ;)
LLVM/Clang 101

file.c

LLVM IR

opt

LLVM MIR

llc

llc

Machine Code

Slide originally by Eric Christopher and Johannes Doerfert

https://youtu.be/J5xExRGaIIY
OpenMP in LLVM

http://openmp.llvm.org/docs

Clang

OpenMP Parser
OpenMP Sema
OpenMP CodeGen

Slide originally presented at LLVM-Dev Meeting 2020

https://youtu.be/M0DrhQbjrro
OpenMP in LLVM

http://openmp.llvm.org/docs

Clang

OpenMP Parser
OpenMP Sema
OpenMP CodeGen

OpenMP runtimes

libomp.so (classic, host)

Slide originally presented at LLVM-Dev Meeting 2020

https://youtu.be/M0DrhQbjrro
OpenMP in LLVM

http://openmp.llvm.org/docs

Clang
- OpenMP Parser
- OpenMP Sema
- OpenMP CodeGen

OpenMP runtimes
- libomp.so (classic, host)
- libomptarget + plugins (offloading, host)
- libomptarget-nvptx (offloading, device)

Slide originally presented at LLVM-Dev Meeting 2020

https://youtu.be/M0DrhQbjrro

Johannes Doerfert
jdoerfert@anl.gov
Argonne National Lab
OpenMP in LLVM

http://openmp.llvm.org/docs

Flang

OpenMP
Parser
OpenMP
Sema
OpenMP
CodeGen

Clang

OpenMP
Parser
OpenMP
Sema
OpenMP
CodeGen

OpenMP runtimes

libomp.so (classic, host)

libomptarget + plugins (offloading, host)

libomptarget-nvptx (offloading, device)

Johannes Doerfert
jdoerfert@anl.gov
Argonne National Lab

Slide originally presented at LLVM-Dev Meeting 2020

https://youtu.be/M0DrhQbjrro
OpenMP in LLVM

http://openmp.llvm.org/docs

Flang

OpenMP

Parser

OpenMP

Sema

CodeGen

OpenMPIRBuilder

frontend independant
OpenMP LLVM-IR generation

favor simple and expressive
LLVM-IR

reusable for non-OpenMP parallelism

OpenMP runtimes

libomp.so (classic, host)

libomptarget + plugins (offloading, host)

libomptarget-nvptx (offloading, device)

Slide originally presented at LLVM-Dev Meeting 2020

https://youtu.be/M0DrhQbjrro
OpenMP in LLVM

http://openmp.llvm.org/docs

Flang

OpenMPParser

OpenMP Sema

OpenMP CodeGen

OpenMPIRBuilder

frontend independant
OpenMP LLVM-IR generation
favor simple and expressive
LLVM-IR
reusable for non-OpenMP parallelism

OpenMPOpt

interprocedural optimization pass
contains host & device optimizations
run with -O2 and -O3 since LLVM 11

OpenMP runtimes

libomp.so (classic, host)
libomptarget + plugins (offloading, host)
libomptarget-nvptx (offloading, device)

Slide originally presented at LLVM-Dev Meeting 2020

https://youtu.be/M0DrhQbjrro
OpenMP Implementation & Optimization
Original Program

```
int y = 7;

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

```java
int y = 7;

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

After Optimizations

```java
for (i = 0; i < N; i++) {
    f(7, i);
}
g(7);
```
Motivation — Compiler Optimization for Parallelism

Original Program

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

After Optimizations
**Motivation — Compiler Optimization For Parallelism**

**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);
```
Why is this important?
SEQUENTIAL PERFORMANCE OF PARALLEL PROGRAMS

/lulesh 10 1

<table>
<thead>
<tr>
<th></th>
<th>0.3721</th>
<th>0.3857</th>
</tr>
</thead>
<tbody>
<tr>
<td>%</td>
<td>0.00%</td>
<td>-3.64%</td>
</tr>
</tbody>
</table>

- Base seq: 0.3721, 0.3700
- Base par: 0.3857, 0.3870

Time in seconds

Versions: base.seq, base.par
SEQUENTIAL PERFORMANCE OF PARALLEL PROGRAMS

/.lulesh 10 l

0.3721 0.3857
0.00% -3.64%

./bfs 1 graph1MW_6.txt

69.955 75.769
0.00% -8.31%
SEQUENTIAL PERFORMANCE OF PARALLEL PROGRAMS

4.5148  6.4912

0.00%  -43.78%

./pathfinder 1000 1000 1

million cycles

versions

base.seq  base.par
SEQUENTIAL PERFORMANCE OF PARALLEL PROGRAMS

- **./pathfinder 1000 1000 1**
  - Million cycles:
    - Base.seq: 4.5148
    - Base.par: 6.4912
  - Baseline time: 0.00%
  - Parallel time: -43.78%

- **./srad_v2 2048 2048 0 127 0 127 1 0.5 20**
  - Time in seconds:
    - Base.seq: 1.077
    - Base.par: 2.626
  - Baseline time: 0.00%
  - Parallel time: -143.83%
OpenMP Input:

```c
#pragma omp parallel for
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.
omp_rt_parallel_for(0, N, &body_fn, &N, &In, &Out);


```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:

```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
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)]
}
```
Use `default(firstprivate)`, or `default(none) + firstprivate(...)`) for (almost) all values!

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

OpenMP Input:
```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)!
```c
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:  

```c
#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  
```c
parfor (int i = 0; i < N; i++)
    body_fn(i, &N, &In, &Out);
```

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

    (*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);

// Parallel region outlined in the front-end (clang)!

```c
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:

```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);

// 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)]
}
# OpenMP Optimizations

<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>
OpenMP Optimizations — Performance Results

```bash
./lud lud_omp -n 1 -i 512.dat
```

<table>
<thead>
<tr>
<th>Version</th>
<th>Time in Seconds</th>
<th>% Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>base.s</td>
<td>0.0077</td>
<td>0.00%</td>
</tr>
<tr>
<td>base.p</td>
<td>0.0097</td>
<td>-25.98%</td>
</tr>
<tr>
<td>attr.s</td>
<td>0.0077</td>
<td></td>
</tr>
<tr>
<td>attr.p</td>
<td>0.0098</td>
<td></td>
</tr>
<tr>
<td>argp.s</td>
<td>0.0077</td>
<td></td>
</tr>
<tr>
<td>argp.p</td>
<td>0.0077</td>
<td></td>
</tr>
<tr>
<td>attr_argp.s</td>
<td>0.0077</td>
<td>-0.14%</td>
</tr>
<tr>
<td>attr_argp.p</td>
<td>0.0077</td>
<td></td>
</tr>
</tbody>
</table>

The diagram shows the performance results for different versions of the program, with the time in seconds on the y-axis and the versions on the x-axis. The % change is also indicated, showing improvement or degradation in performance.
## OpenMP Optimizations — Performance Results

![Graph showing performance results](attachment:image.png)

<table>
<thead>
<tr>
<th>Versions</th>
<th>Time in Seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>base.s</td>
<td>1.077</td>
</tr>
<tr>
<td>base.p</td>
<td>2.626</td>
</tr>
<tr>
<td>attr.s</td>
<td>1.077</td>
</tr>
<tr>
<td>attr.p</td>
<td>2.58</td>
</tr>
<tr>
<td>argp.s</td>
<td>1.077</td>
</tr>
<tr>
<td>argp.p</td>
<td>2.621</td>
</tr>
<tr>
<td>attr_argp.s</td>
<td>1.077</td>
</tr>
<tr>
<td>attr_argp.p</td>
<td>1.075</td>
</tr>
</tbody>
</table>

Time improvement: **-143.83%**

Overall improvement: **0.19%**
LLVM’s OpenMP-Aware Optimizations
LLVM’s OpenMP-Aware Optimizations

Towards OpenMP-aware compiler optimizations

- LLVM “knows” about OpenMP API and (internal) runtime calls, incl. their potential effects (e.g., they won’t throw exceptions).
- LLVM performs “high-level” optimizations, e.g., parallel region merging, and various GPU-specific optimizations late.
- Some LLVM/Clang “optimizations” remain, but we are in the process of removing them: simple frontend, smart middle-end.

OpenMPOpt
Interprocedural optimization pass
Contains host & device optimizations
Run with -02 and -03 since LLVM 11.
Optimization Remarks
Example: OpenMP runtime call deduplication

double *A = malloc(size * omp_get_thread_limit());
double *B = malloc(size * omp_get_thread_limit());

#pragma omp parallel
do_work(A, B);

OpenMP runtime calls with same return values can be merged to a single call
Optimization Remarks
Example: OpenMP runtime call deduplication

```c
double *A = malloc(size * omp_get_thread_limit());
double *B = malloc(size * omp_get_thread_limit());
#pragma omp parallel
do_work(A, B);
```

$ clang -g -O2 deduplicate.c -fopenmp -Rpass=openmp-opt$

OpenMP runtime calls with same return values can be merged to a single call
void bar(void) {  
#pragma omp parallel  
{}  
}
void foo(void) {  
#pragma omp target teams  
{  
#pragma omp parallel  
{}  
bar();  
#pragma omp parallel  
{}  
}

remark: Found a parallel region that is called in a target region but not part of a combined target construct nor nested inside a target construct without intermediate code. This can lead to excessive register usage for unrelated target regions in the same translation unit due to spurious call edges assumed by ptxas.
remark: Parallel region is not known to be called from a unique single target region, maybe the surrounding function has external linkage?; will not attempt to rewrite the state machine use.
remark: Found a parallel region that is called in a target region but not part of a combined target construct nor nested inside a target construct without intermediate code. This can lead to excessive register usage for unrelated target regions in the same translation unit due to spurious call edges assumed by ptxas.
remark: Specialize parallel region that is only reached from a single target region to avoid spurious call edges and excessive register usage in other target regions. (parallel region ID: __omp_outlined__1_wrapper, kernel ID: __omp_offloading_35_a1e179_foo_l7)
remark: Target region containing the parallel region that is specialized. (parallel region ID: __omp_outlined__1_wrapper, kernel ID: __omp_offloading_35_a1e179_foo_l7)
remark: Found a parallel region that is called in a target region but not part of a combined target construct nor nested inside a target construct without intermediate code. This can lead to excessive register usage for unrelated target regions in the same translation unit due to spurious call edges assumed by ptxas.
remark: Specialize parallel region that is only reached from a single target region to avoid spurious call edges and excessive register usage in other target regions. (parallel region ID: __omp_outlined__3_wrapper, kernel ID: __omp_offloading_35_a1e179_foo_l7)
remark: Target region containing the parallel region that is specialized. (parallel region ID: __omp_outlined__3_wrapper, kernel ID: __omp_offloading_35_a1e179_foo_l7)
remark: OpenMP GPU kernel __omp_offloading_35_a1e179_foo_l7

Explained Later!
OpenMP Compile-Time and Runtime Information

- Use OpenMP optimization remarks
- Optimization remark explanations, examples, FAQs, ...
  all gradually added to http://openmp.llvm.org/docs
- Use LIBOMPTARGET_INFO for runtime library interactions

$ clang -O2 generic.c -fopenmp -fopenmp-targets=nvptx64-nvidia-cuda -o generic
$ env LIBOMPTARGET_INFO=1 ./generic

CUDA device 0 info: Device supports up to 65536 CUDA blocks and 1024 threads with a warp size of 32
CUDA device 0 info: Launching kernel __omp_offloading_fd02_c2a59832_main_l106 with 48 blocks and 128 threads in Generic mode
OpenMP Offloading (in LLVM)
Compiling Clang Actions

clang -fopenmp -fopenmp-targets=nvptx64 file.c

Slide originally by Jose Monsalve Diaz
#include <math.h>

#pragma omp begin declare target
void science(float f) {
    if (signbitf(f)) {
        // some science
    } else {
        // some other science
    }
}
#pragma omp end declare target

science can be called from the host and device
OpenMP Offloading
The Tricky Bits

GPUs do not provide a math.h, and more importantly, no libm.

```c
#include <math.h>

#pragma omp begin declare target
void science(float f) {
    if (signbitf(f)) {
        // some science
    } else {
        // some other science
    }
}
#pragma omp end declare target

// LLVM/Clang's "math.h" wrapper for NVPTX (CUDA)
int __signbitf(float __a) { return __nv_signbitf(__a); }

#pragma omp begin declare variant match(device={kind(gpu)})
bool signbit(float __x) { return ::__signbitf(__x); }
#pragma omp end declare variant

science can be called from the host and device
```
OpenMP Offloading

The Tricky Bits

Linking

not today 😢
OpenMP Offloading vs Kernel Languages

```
#pragma omp target teams num_teams(1)
{
  A();
}
#pragma omp parallel num_threads(4) default(firstprivate)
{
  Func(args);
}
B();
```

LLVM/OpenMP

SPMD-mode

Generic-mode
OpenMP Offloading vs Kernel Languages

```c
#pragma omp target teams num_teams(1)
{
    #pragma omp parallel num_threads(4) default(firstprivate)
    {
        if (omp_get_thread_num() == 0)
            A();
        #pragma omp barrier
        Func(args);
        #pragma omp barrier
        if (omp_get_thread_num() == 0)
            B();
    }
}
```

SPMD-zation, coming soon!
OpenMP Offloading vs Kernel Languages (simplified)

```
#pragma omp target teams num_teams(1)
{
    A();
    #pragma omp parallel num_threads(4) default(firstprivate)
    {
        Func(args);
    }
    B();
}
```
Q: How do you identify a parallel region?
A: Via the function (pointer) we outlined it into.

Q: Won’t that cause indirect calls and spurious call edges?
A: Yes. That’s why we try to use non-function pointer IDs.
OpenMP Offloading vs Kernel Languages (simplified)

```c
static void parFn() {
  // parallel function code
}

void kernel() {
  if (is_worker()) {
    while (1) {
      fn = __omp_wait_for_parallel();
      fn();
      __omp_inform_parallel_done();
    }
  } else {
    __omp_inform_workers(&parFn, ...)
    parFn();
    __omp_wait_for_workers();
  }
}

static char parFnId;
static void parFn() {
  // parallel function code
}

void kernel() {
  if (is_worker()) {
    while (1) {
      fn = __omp_wait_for_parallel();
      if (fn == &parFnId) parFn(); else fn();
      __omp_inform_parallel_done();
    }
  } else {
    __omp_inform_workers(&parFnId, ...)
    parFn();
    __omp_wait_for_workers();
  }
}
```
OpenMP Offloading vs Kernel Languages (simplified)

static void parFn() {
    // parallel function code
}

void kernel() {
    if (is_worker()) {
        // ...
    } else {
        visible();
    }
}

void visible() {
    __omp_inform_workers(&parFn, ...)
    parFn();
    __omp_wait_for_workers();
}

static char parFnId;
static void parFn() {
    // parallel function code
}

void kernel() {
    if (is_worker()) {
        // ...
    } else {
        visible();
    }
}

#pragma omp begin assumes ompx_no_external_callers
void visible() {
    __omp_inform_workers(&parFnId, ...)
    parFn();
    __omp_wait_for_workers();
}
#pragma omp end assumes

Use optimization remarks to learn about missed opportunities

LLVM 13 will know more tricks :)
What OpenMP got Wrong

(non exhaustive list)
What OpenMP got Wrong

All instances where a directive retroactively changes something:

```c
static int X;
static int PleaseDont[alignof(X)];
int* whileWeAreHere(void) { return &X; }
#pragma omp allocate(X) allocator(...) align(...)
```

The fixation on syntactic nesting:

```c
#pragma omp target
{
#pragma omp atomic update
++X;
}
#pragma omp target
{
#pragma omp atomic update // error
++X;
}
#pragma omp target teams
{
#pragma omp atomic update // error
foo();
}
#pragma omp target teams
{
// pragma omp atomic in foo is fine
foo();
}
```
What OpenMP got (kinda) Right

(non exhaustive list)
What OpenMP got (kinda) Right

The target device abstraction:

LLVM 12 provides remote GPUs!
What OpenMP got (kinda) Right

The target device abstraction:

LLVM 13 will provide a VGPU :)
What OpenMP got (kinda) Right

The target device abstraction:

Application + OpenMP World:
1. Application
2. Fat Binary
3. libomptarget

Device (Abstraction) World:
4. libomptarget.rtl.cuda
5. Device Runtime Library

libomptarget.rtl.vgpu
6. Device Runtime Library
What’s Next?
What’s Next?

LLVM

- More OpenMP-aware optimizations:
  - hide memory transfer latencies
  - exploit OpenMP domain knowledge
  - ask for and utilize user assumptions
- GPU-specific optimizations
- More actionable optimization remarks
- OpenMP 5.1 features
- A new (portable and performant) GPU device runtime (*written in OpenMP 5.1!*)
- Helpful offloading “devices”:
  - VGPU + NewProcess for debugging, or
  - JIT for performance
- Host-Device optimizations

OpenMP

- OpenMP Interop and dynamic context selector implementations
- A community developed OMPX (header) library (think stdlib for OpenMP).
- Function variants shipped via libraries
- More powerful assumptions
- Less syntactic / more semantic reasoning*
- Deprecations*

* I hope
Final Thoughts

(aka. Rambling)
Parallel Worksharing Loops ≠ “Parallel Loops”

```c
void f(double *A, double *B) {
    #pragma omp parallel for
    for (int i = 0; i < N; ++i) {
        // ...
    }
}
```

```c
void f(double *A, double *B) {
    #pragma omp parallel for schedule(static, N)
    for (int i = 0; i < N; ++i) {
        // ...
    }
}
```

```c
void f(double *A, double *B) {
    #pragma omp parallel for order(concurrent)
    for (int i = 0; i < N; ++i) {
        // ...
    }
}
```

```c
omp_set_num_threads(1);
f(A, B);
```

```c
void f(double *A, double *B) {
    #pragma omp parallel for schedule(static, 1)
    for (int i = 0; i < N; ++i) {
        // ...
    }
}
```
What’s Next?

LLVM

● More OpenMP-aware optimizations:
  ○ hide memory transfer latencies
  ○ exploit OpenMP domain knowledge
  ○ ask for and utilize user assumptions

● GPU-specific optimizations

● More actionable optimization remarks

● OpenMP 5.1 features

● A new (portable and performant) GPU device runtime (*written in OpenMP 5.1!*)

● Helpful offloading “devices”:
  ○ VGPU + NewProcess for debugging, or
  ○ JIT for performance

● Host-Device optimizations

OpenMP

● OpenMP Interop and dynamic context selector implementations

● A community developed OMPX (header) library (think stdlib for OpenMP).

● Function variants shipped via libraries

● More powerful assumptions

● Less syntactic / more semantic reasoning*

● Deprecations*

* I hope

Thanks!

Interested?

Reach out!

Johannes Doerfert
jdoerfert@anl.gov
Argonne National Lab
Design Goal

Report every successful and failed optimization
Design Goal

Optimize offloading code

perform host + accelerator optimizations
OpenMP Offload Compilation

```c
void foo() {
    int N = 1024;

    #pragma omp target
    *mem = N;
}
```

OpenMP Offload Compilation (simplified)

---

**User Code 1.c**

```c
void foo() {
    int N = 1024;
    #pragma omp target
    *mem = N;
}
```

**Host C**

```c
extern void device_func7(int);

void foo() {
    int N = 1024;
    if (!offload(device_func7, N)) {
        // host fallback
        *mem = N;
    }
}
```

**Device C**

```c
void device_func7(int N) {
    *mem = N;
}
```

---

OpenMP Offload Compilation

### user_code_1.c

```c
void foo() {
    int N = 1024;
    #pragma omp target
    *mem = N;
}
```

### host.c

```c
extern void device_func7(int);

void foo() {
    int N = 1024;
    if (!offload(device_func7, 1024)) {
        // host fallback
        *mem = 1024;
    }
}
```

### device.c

```c
void device_func7(int N) {
    *mem = N;
}
```

OpenMP Offload Compilation

The constant is part of the "host code".

The constant is part of the "host code".

Heterogeneous LLVM-IR Module

user_code_1.c

```c
void foo() {
    int N = 1024;
    #pragma omp target
    *mem = N;
}
```

heterogeneous.c

```c
__attribute__((callback(Func, ...)))
int offload(void (*)(...) Func, ...);

target 0 void foo() {
    int N = 1024;
    if (!offload(device_func7, N)) {
        // host fallback
        *mem = N;
    }
}

target 1 void device_func7(int N) {
    *mem = N;
}
```


Heterogeneous LLVM-IR Module

```c
user_code_1.c

void foo() {
    int N = 1024;
    #pragma omp target
    *mem = N;
}

__attribute__((callback(Func, ...)))
int offload(void (*)(...) Func, ...);

target 0 void foo() {
    int N = 1024;
    if (!offload(device_func7(N))) {
        // host fallback
        *mem = 1024;
    }
}

target 1 void device_func7(int N) {
    *mem = 1024;
}
```