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

/\   /\
\------------- me in Zurich ---------------/

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

Background

LLVM in a Nutshell

- open (source/community/...)

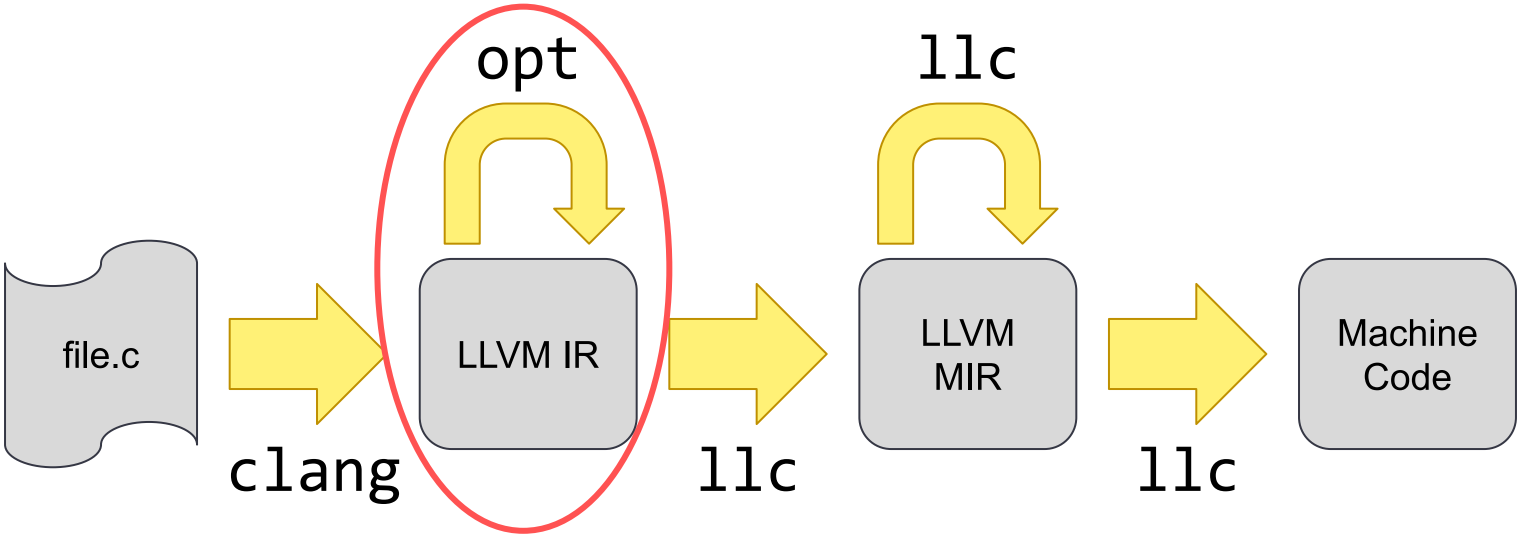
- extensible, “fixable”
- portable (GPUs, CPUs, ...)
- C++/OpenMP/SYCL/HIP/CUDA/... feature complete😉
- early access to *the coolest* features

- performant and correct ;)

THANKS 2 RYAN HOUDEK

[😉 eventually]

LLVM/Clang 101



Slide originally by *Eric Christopher* and *Johannes Doerfert* https://youtu.be/J5xExRGaIIY

OpenMP in LLVM

http://openmp.llvm.org/docs

Johannes Doerfert
jdoerfert@anl.gov
Argonne National Lab

OpenMP in LLVM

http://openmp.llvm.org/docs

Johannes Doerfert
jdoerfert@anl.gov
Argonne National Lab

Clang

OpenMP Parser

OpenMP Sema

OpenMP CodeGen

OpenMP in LLVM

http://openmp.llvm.org/docs

Johannes Doerfert
jdoerfert@anl.gov
Argonne National Lab

Clang

OpenMP Parser

OpenMP Sema

OpenMP CodeGen

OpenMP runtimes

libomp.so (classic, host)

OpenMP in LLVM

http://openmp.llvm.org/docs

Johannes Doerfert
jdoerfert@anl.gov
Argonne National Lab

Clang

- OpenMP Parser
- OpenMP Sema
- OpenMP CodeGen

OpenMP runtimes

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

OpenMP in LLVM

http://openmp.llvm.org/docs

Johannes Doerfert
jdoerfert@anl.gov
Argonne National Lab

Flang

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

OpenMP in LLVM

http://openmp.llvm.org/docs

Johannes Doerfert
jdoerfert@anl.gov
Argonne National Lab

Flang

Clang

OpenMP Parser

OpenMP Sema

OpenMP 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

Johannes Doerfert
jdoerfert@anl.gov
Argonne National Lab

Flang

Clang

OpenMP Parser

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

Context — Compiler Optimization

Original Program

After Optimizations

```
int y = 7;

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

Context — Compiler Optimization

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);
```

Motivation — Compiler Optimization For Parallelism

Original Program

```
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

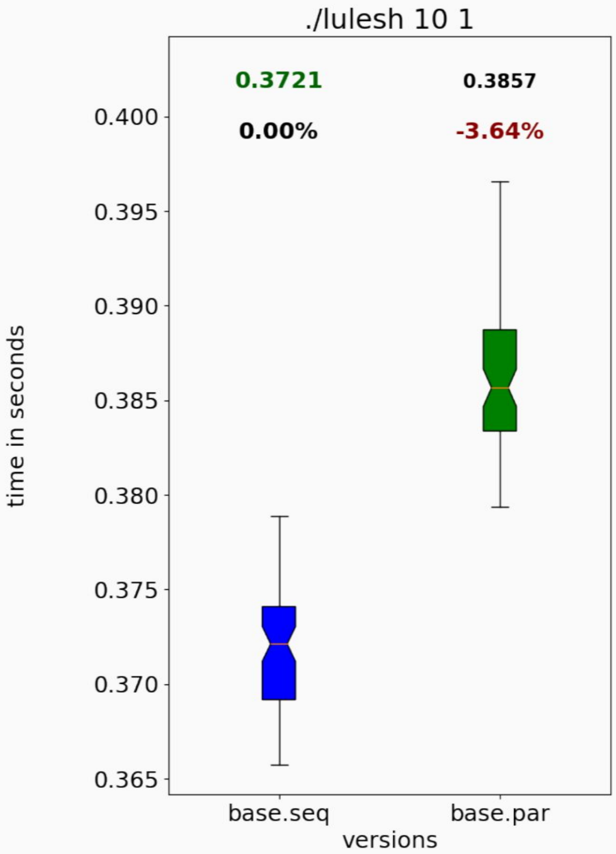
```
int y = 7;
#pragma omp parallel for
for (i = 0; i < N; i++) {
  f(y, i);
}
g(y);
```

After Optimizations

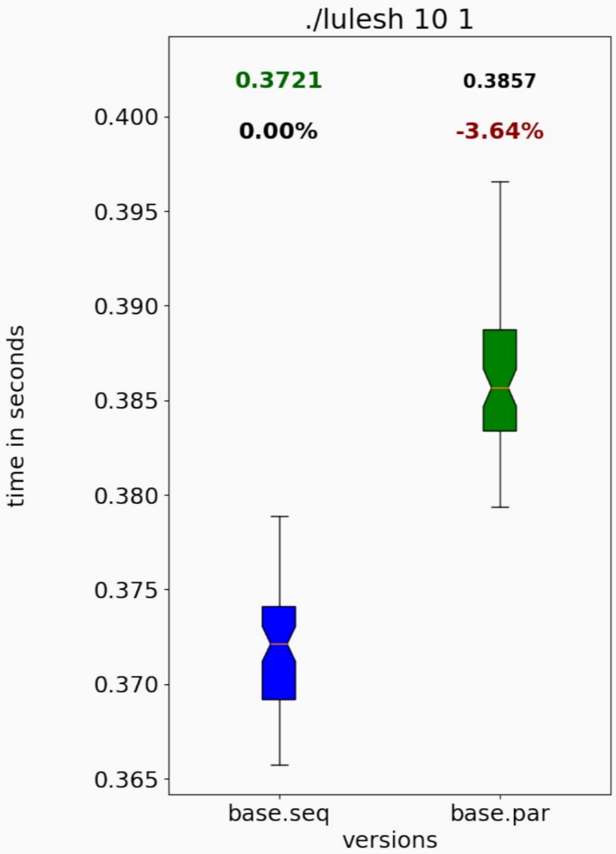
```
int y = 7;
#pragma omp parallel for
for (i = 0; i < N; i++) {
  f(y, i);
}
g(y);
```

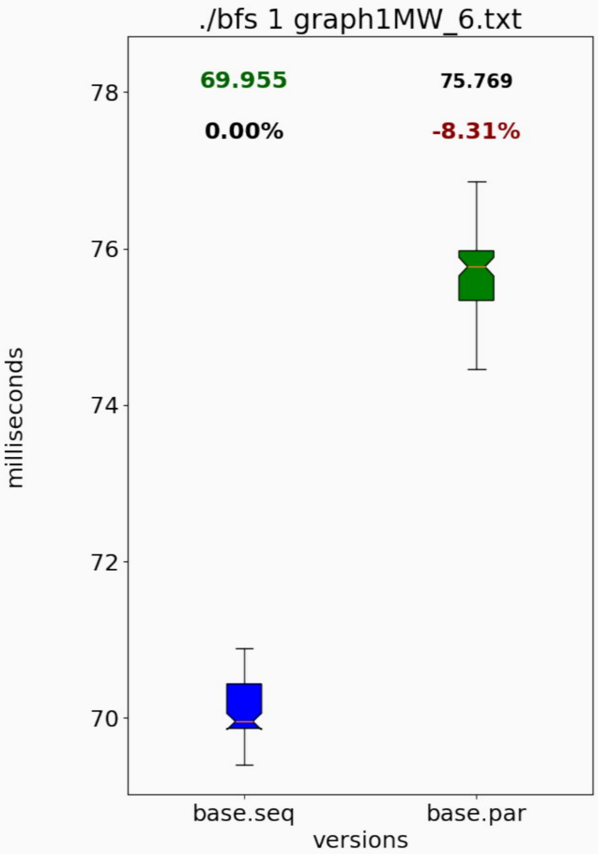
Why is this important?

Sequential Performance of Parallel Programs

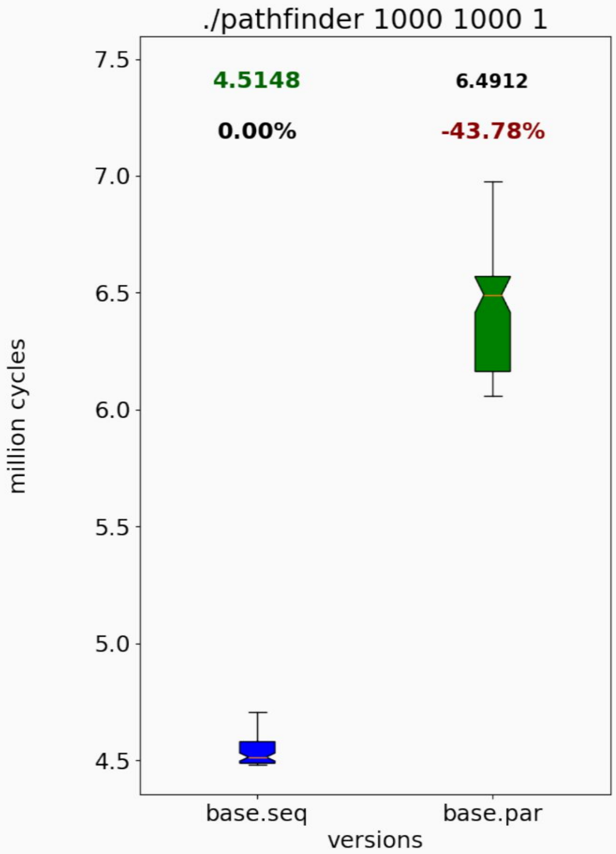


Sequential Performance of Parallel Programs

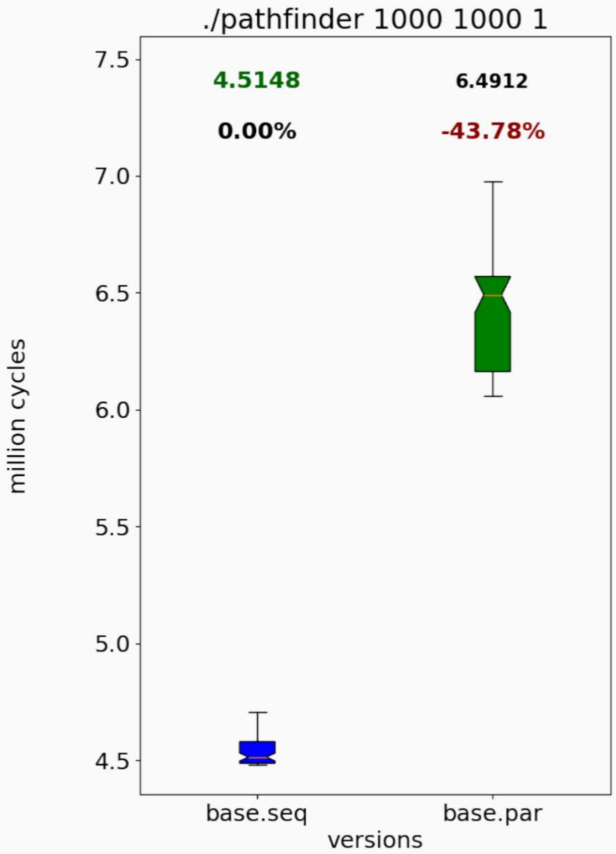


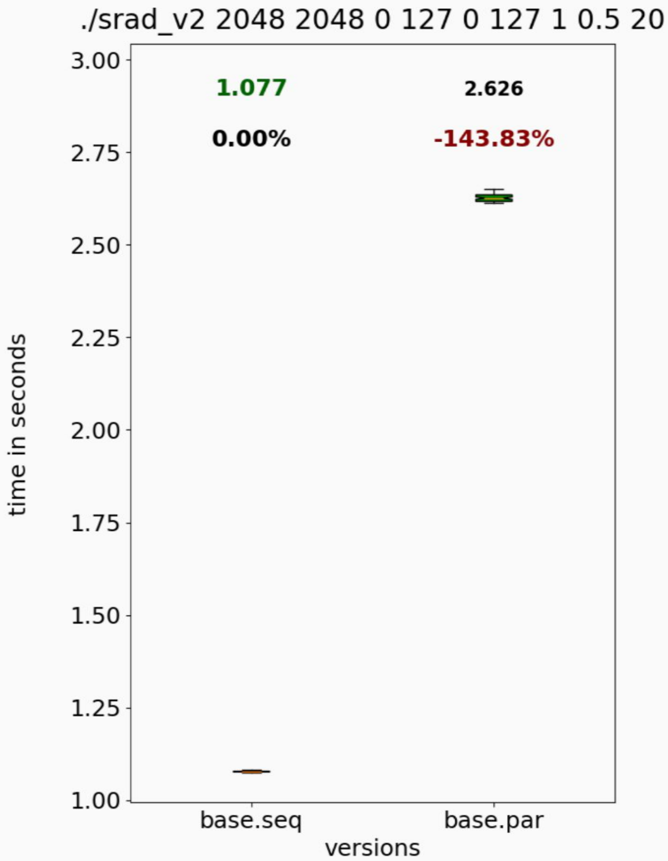


Sequential Performance of Parallel Programs



Sequential Performance of Parallel Programs





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)]
}
```

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)]
}
```

Early Outlining: Sequential Optimization Problems

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

Declaration	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*

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)]
}
```

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
parfor (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 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)]
}
```

Early Outlining + Transitive Calls

OpenMP Input:

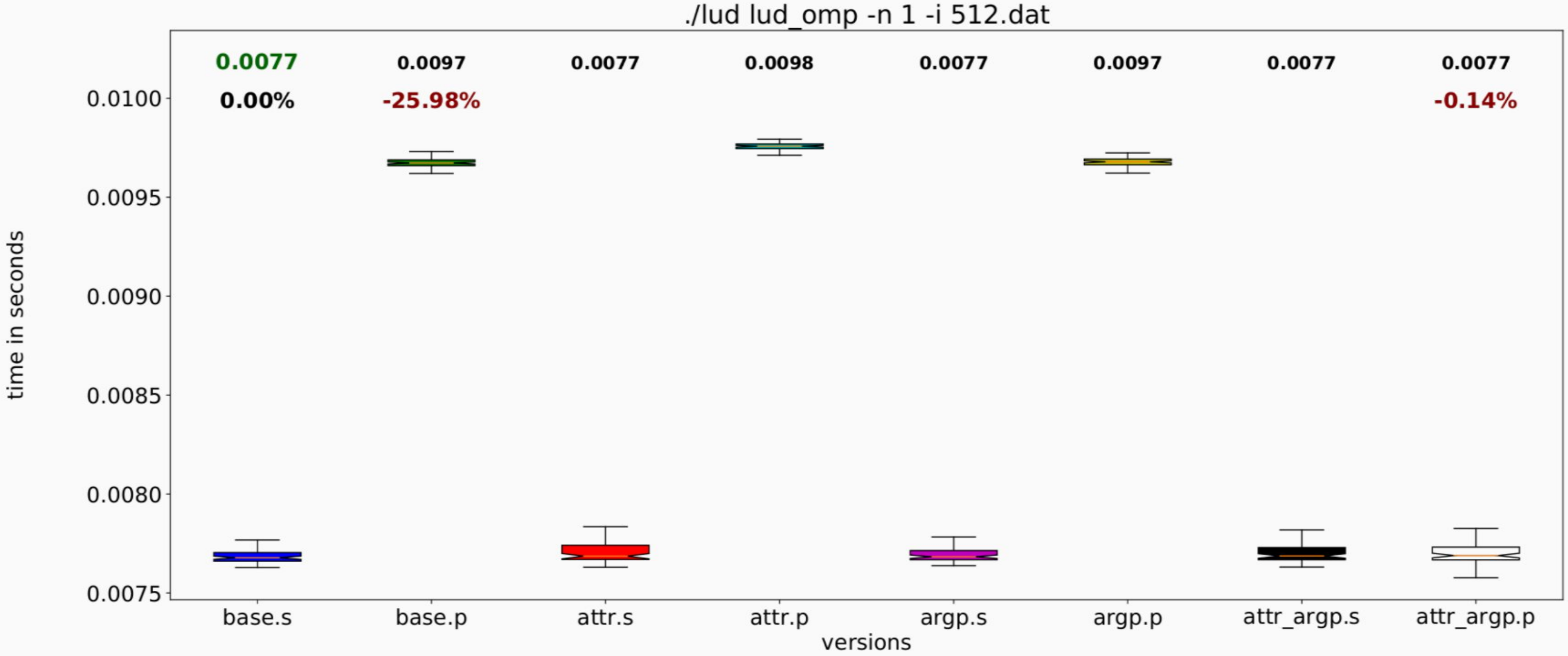
```
#pragma omp parallel for
for (int i = 0; i < N; i++)
  Out[i] = In[i] + In[i+N];
```

```
// 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)]
}
```

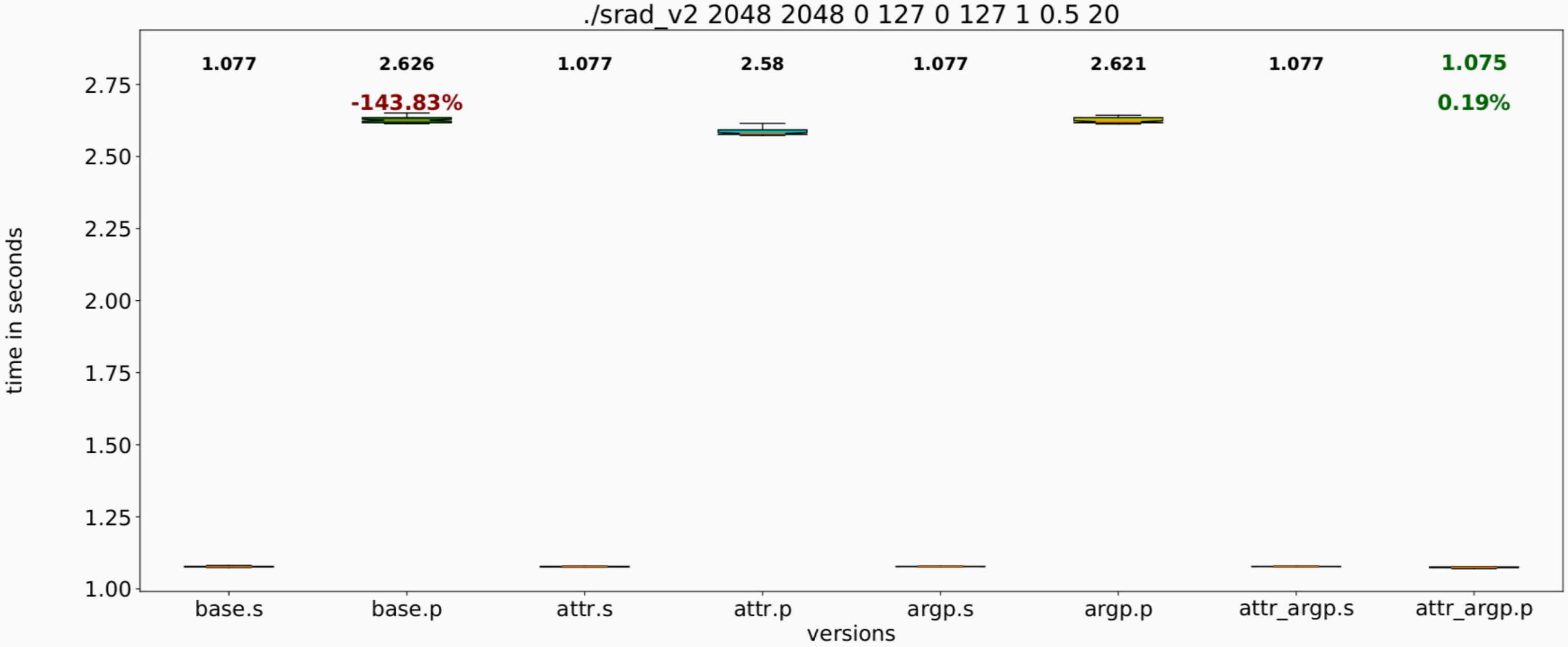
OpenMP Optimizations

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)	

OpenMP Optimizations — Performance Results



OpenMP Optimizations — Performance Results



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 -O2 and -O3 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);
```

OpenMP runtime calls with same return values can be merged to a single call

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

```
deduplicate.c:12:29: remark: OpenMP runtime call omp_get_thread_limit moved to deduplicate.c:11:29: [-Rpass=openmp-opt]
   double *B = malloc(size*omp_get_thread_limit());
deduplicate.c:11:29: remark: OpenMP runtime call omp_get_thread_limit deduplicated [-Rpass=openmp-opt]
   double *A = malloc(size*omp_get_thread_limit());
```

Optimization Remarks

Example: OpenMP Target Scheduling

```
clang12 -Rpass=openmp-opt ...
```

```c
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 rrounding 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 c nstruct nor nested inside a target construct without intermediate code. This can lead to excessive register usage for un ons in the same translation unit due to spurious call edges assumed by ptxas.
remark: Specialize parallel region that is only reached from a sing avoid spurious call edges and excessive register usage in other target regions. (parallel region ID: __omp_outlined__1 D: __omp_offloading_35_a1e179_foo_l7)
remark: Target region containing the parallel region tha arallel region ID: __omp_outlined__1_wrapper, kernel ID: __omp_offloading_35_a1e179_foo_l7)
remark: Found a parallel region that is called but not part of a combined target construct nor nested inside a target construct without intermediate code. This can lea gister usage for unrelated target regions in the same translation unit due to spurious call edges assumed by ptxas.
remark: Specialize parallel regi eached from a single target region to avoid spurious call edges and excessive register usage in other target regions. (parallel reg _omp_outlined__3_wrapper, kernel ID: __omp_offloading_35_a1e179_foo_l7)
remark: Target region containing t 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
```

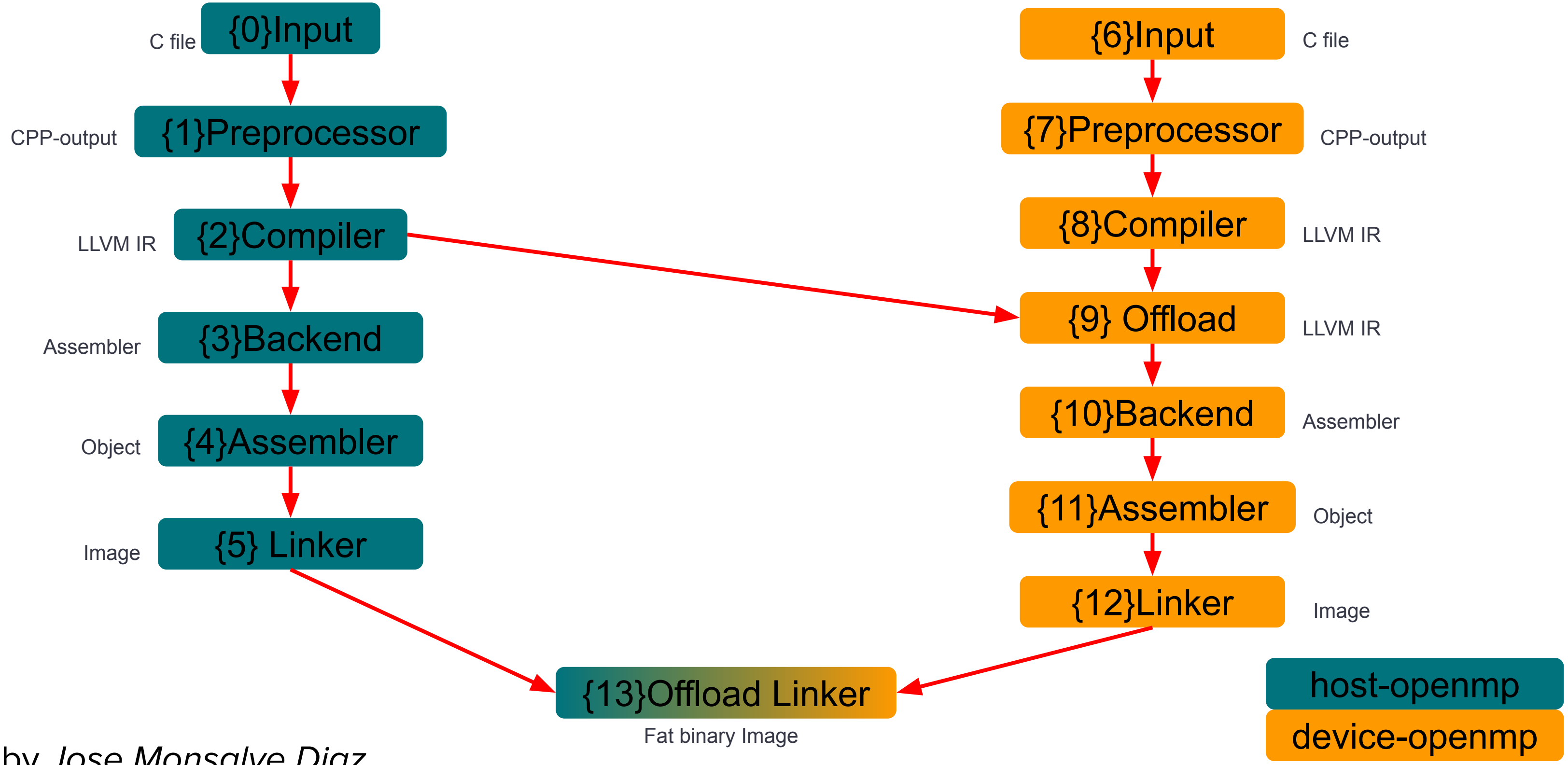
```
CUDA device 0 info: Device supports up to 65536 CUDA blocks and 1024 threads with a warp size of 32
CUDA device 0 info: Launching kernel __omp_offloading_fd02_c2a59832_main_l106 with 48 blocks and 128 threads in Generic mode
```

OpenMP Offloading (in LLVM)

Compiling

Clang Actions

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



Slide originally by *Jose Monsalve Diaz*

OpenMP Offloading

The Tricky Bits

```
#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

math.h

```
/* Test for negative number.  Used in the signbit() macro.  */
__MATH_INLINE int
__NTH (__signbitf (float __x))
{
# ifdef __SSE2_MATH__
  int __m;
  __asm (""pmovmskb %1, %0"" : ""=r"" (__m) : ""x"" (__x));
  return (__m & 0x8) != 0;
# else
  __extension__ union { float __f; int __i; } __u = { __f: __x };
  return __u.__i < 0;
# endif
}
```

OpenMP Offloading

The Tricky Bits

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

```
#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

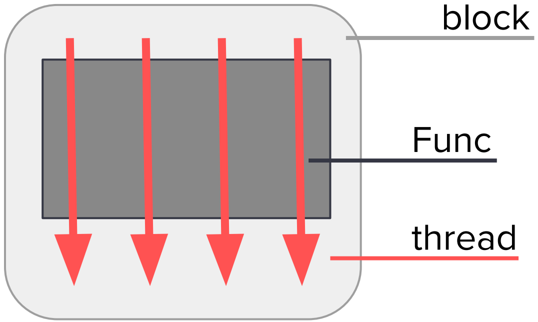
Linking

not today 😢

OpenMP Offloading vs Kernel Languages

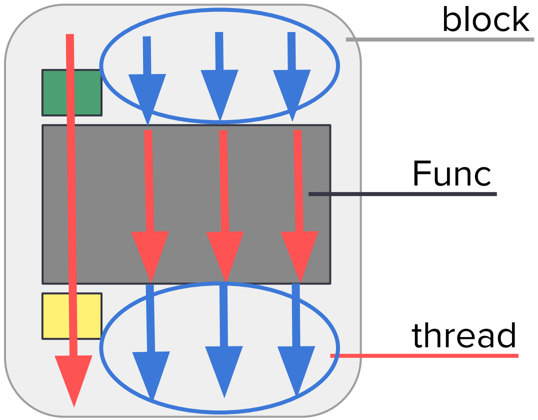
LLVM/OpenMP

```
Func<<</* blocks */ 1, /* threads */ 4>>>(args);
```



SPMD-mode

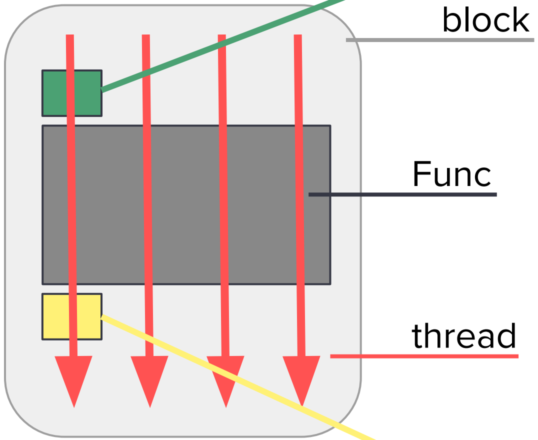
```
#pragma omp target teams num_teams(1)
{
  A();
  #pragma omp parallel num_threads(4) default(firstprivate)
  {
    Func(args);
  }
  B();
}
```



Generic-mode

OpenMP Offloading vs Kernel Languages

```
#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();
  }
}
```



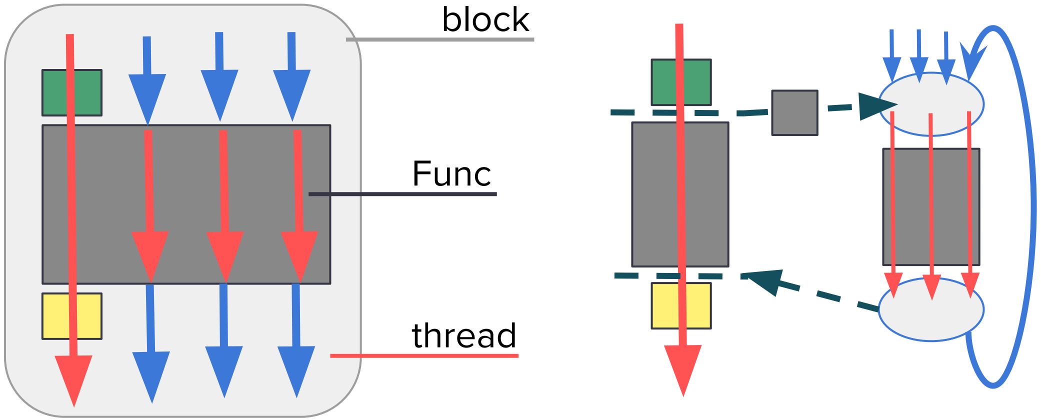

```
void A() {
  #pragma omp parallel
  Kernel2();
}
```

```
void B() {
  #pragma omp barrier
}
```

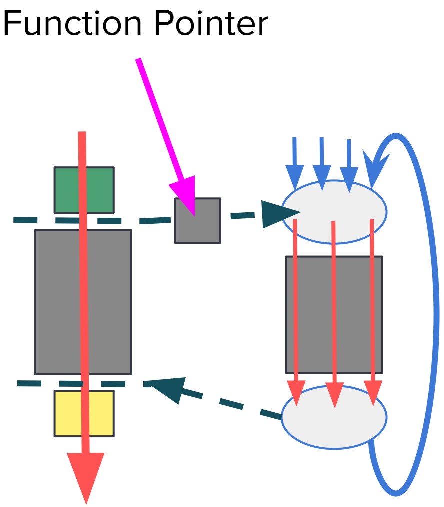
SPMD-zation, coming soon!

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();
}
```



OpenMP Offloading vs Kernel Languages (simplified)



Q: How do you identify a parallel region?

A: Via the function (pointer) we outlined it into.

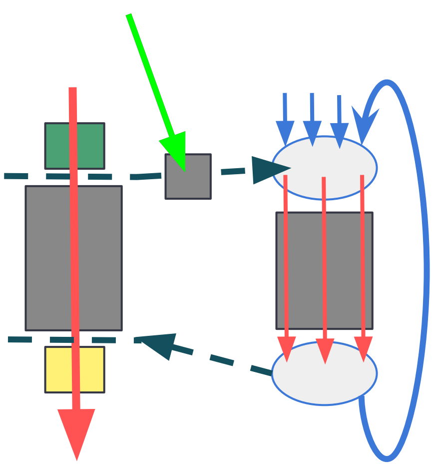
Q: Won't that cause indirect calls and spurious call edges?

A: Yes. That's why we try to use non-function pointer IDs.

OpenMP Offloading vs Kernel Languages (simplified)

```
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();
     (fn == &parFnId) ? parFn() : fn();
    __omp_inform_parallel_done();
    }
  } else {
    __omp_inform_workers(&parFnId, ...)
    parFn();
    __omp_wait_for_workers();
  }
}
```

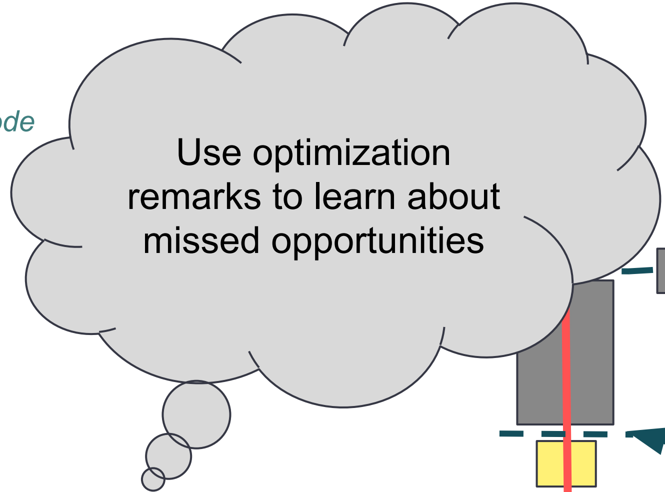


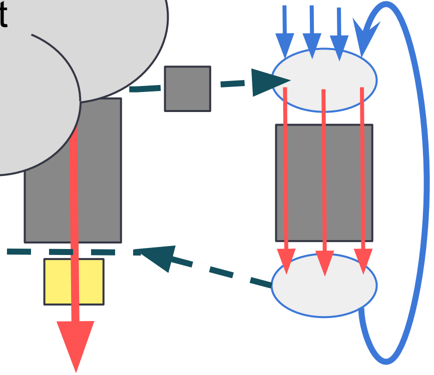
Performed since LLVM 12

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
```





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:

```
static int X;

static int PleaseDont[alignof(X)];

int* whileWeAreHere(void) { return &X; }

#pragma omp allocate(X) allocator(...) align(...)
```

The fixation on syntactic nesting:

```
#pragma omp target
{
  #pragma omp atomic update
  ++X;
}
```

```
#pragma omp target teams
{
  #pragma omp atomic update  // error
  ++X;
}
```

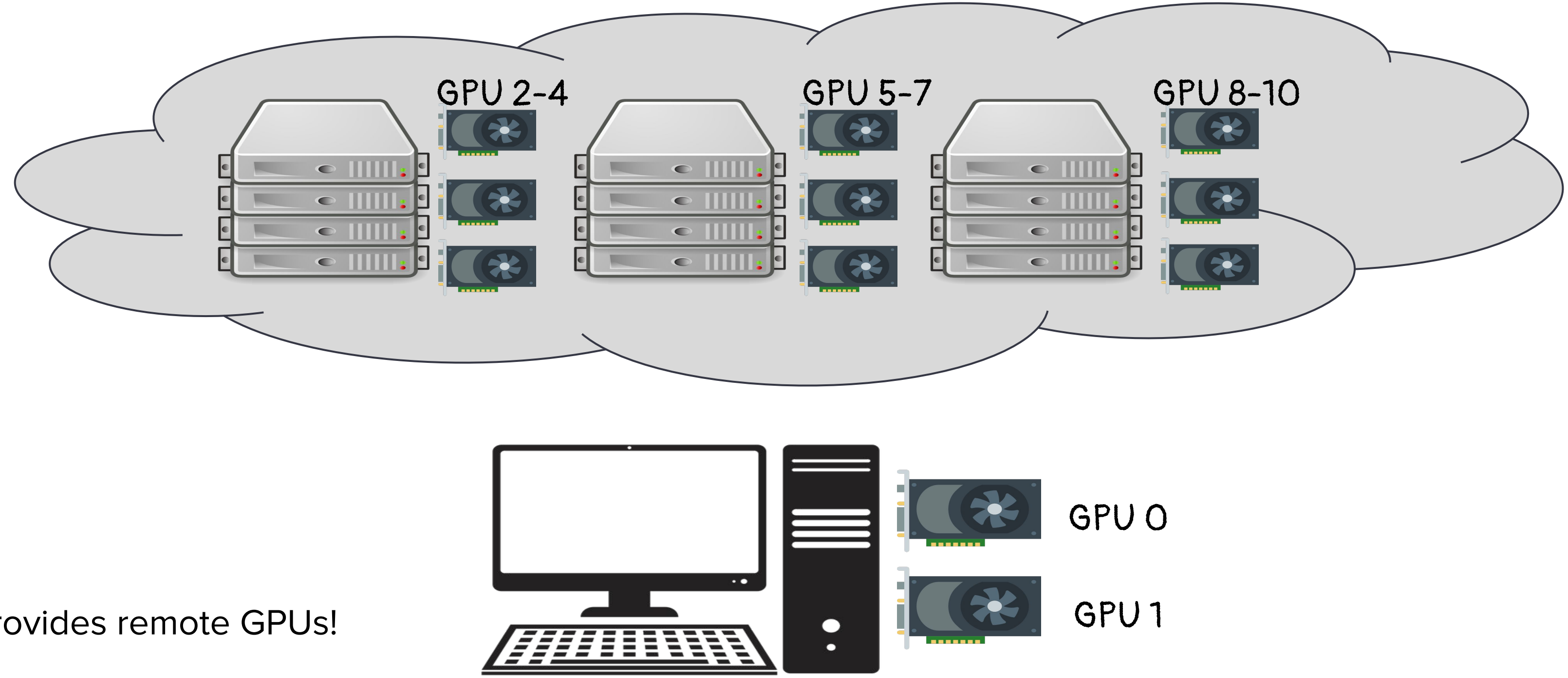
```
#pragma omp target teams
{
  // pragma omp atomic in foo is fine
  foo();
}
```

What OpenMP got (kinda) Right

(non exhaustive list)

What OpenMP got (kinda) Right

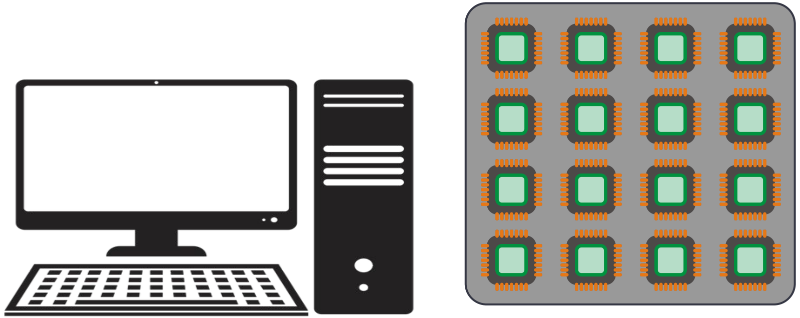
The target device abstraction:



LLVM 12 provides remote GPUs!

What OpenMP got (kinda) Right

The target device abstraction:



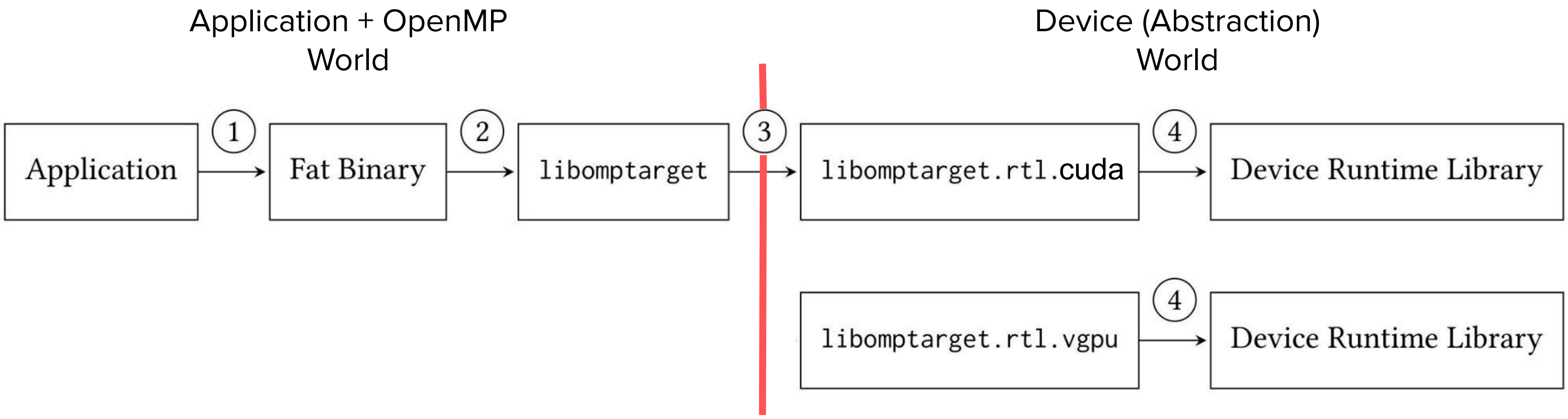
CPU Device 0

Virtual GPU
Device 1

LLVM 13 will provide a VGPU :)

What OpenMP got (kinda) Right

The target device abstraction:



What’s Next?

What's Next?

Johannes Doerfert
jdoerfert@anl.gov
Argonne National Lab

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”

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

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

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

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

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

What's Next?

Johannes Doerfert
jdoerfert@anl.gov
Argonne National Lab

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*

Thanks!
Interested?
Reach out!

* I hope

Joseph Huber
huberjn@ornl.gov
Oak Ridge National Lab

Design Goal

Report every successful and failed optimization

Shilei Tian
shilei.tian@stonybrook.edu
Stony Brook University

Design Goal

Optimize offloading code

perform host + accelerator optimizations

OpenMP Offload Compilation (simplified)

```
user_code_1.c
void foo() {
  int N = 1024;

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

* RFC: https://bit.ly/3bJzAx3

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;
}
```

* RFC: https://bit.ly/3bJzAx3

OpenMP Offload Compilation (simplified)

user_code_1.c

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

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

host.c

```
extern void device_func7(int);

void foo() {
  int N = 1024;

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

device.c

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

* RFC: https://bit.ly/3bJzAx3

OpenMP Offload Compilation (simplified)

user_code_1.c

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

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

host.c

```
extern void device_

void foo() {
  int N = 1024;

  if (!offload(device_func7, 1024)) {
    // host fallback
    *mem = 1024
  }
}
```

The constant is part of the "host code".

device.c

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

* RFC: https://bit.ly/3bJzAx3

Heterogeneous LLVM-IR Module

user_code_1.c

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

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

heterogeneous.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;
}
```

* RFC: https://bit.ly/3bJzAx3

* callback attribute: https://bit.ly/32I68ds

Heterogeneous LLVM-IR Module

user_code_1.c

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

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

heterogeneous.c

```
__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;
}
```

* RFC: https://bit.ly/3bJzAx3

* callback attribute: https://bit.ly/32l68ds