A Compiler's View of OpenMP Offloading

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

Background

LLVM in a Nutshell

Thanks 2 Ryan Houdek

- open (source/community/...)

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

- performant and correct ;)

[😉 eventually]

LLVM/OpenMP - A Community Effort

Weekly Meeting: https://bit.ly/2Zqt49v

“Academia”

- Joseph Huber (ORNL)
- Shilei Tian (SBU)
- Giorgis Georgakoudis (LLNL)
- Michael Kruse (ANL)
- Joachim Protze (RWTH A.)
- Joel Denny (ORNL)
- Valentin Clement (ORNL, now NVIDIA)
- Many, many, more

Industry

- Alexey Bataev (Intel)
- Jon Chesterfield (AMD)
- George Rokos (Intel)
- Pushpinder Singh (AMD)
- Kiran Chandramohan (ARM)
- Chi Chun Chen (HPE/Cray)
- Andrey Churbanov (Intel)
- Carlo Bertolli (AMD)
- Many, many, more

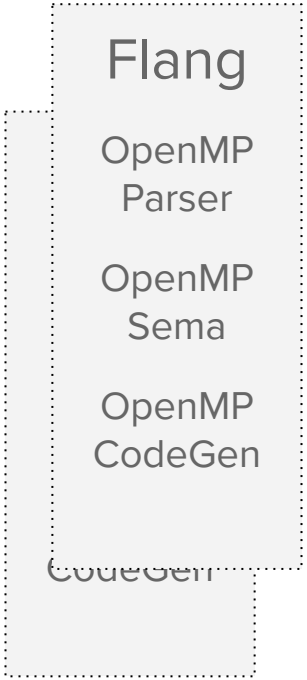
Power Users

- Ye Luo (ANL)
- Christopher Daley (NERSC)
- John Tramm (ANL)
- Rahul Gayatri (NERSC)
- Itaru Kitayama (RIKEN)
- Wael Elwasif (ORNL)
- More that I have forgotten

OpenMP in LLVM

Weekly Meeting: https://bit.ly/2Zqt49v

Johannes Doerfert
jdoerfert@anl.gov
Argonne National Lab



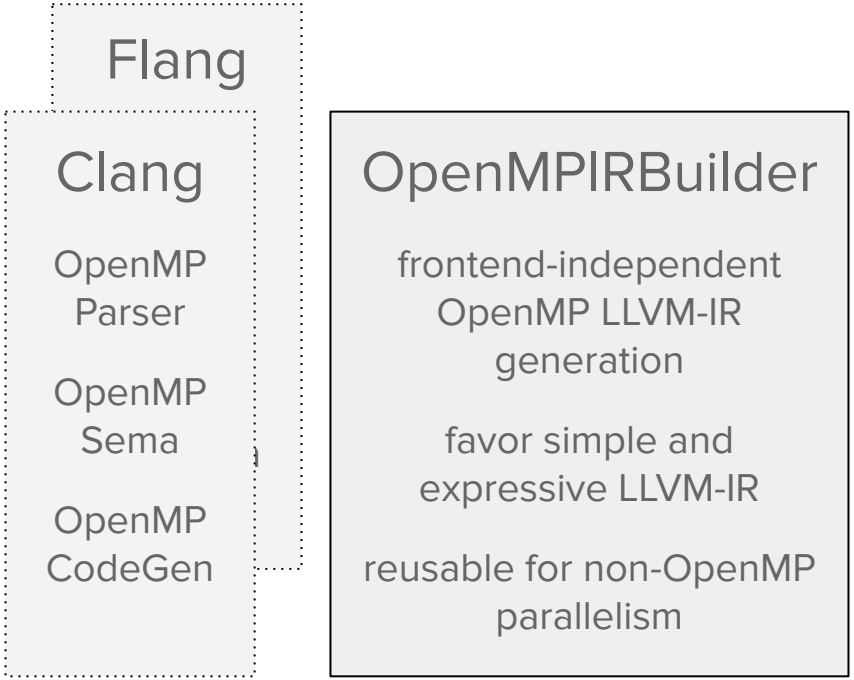
Slide originally presented at LLVM Dev'21

https://youtu.be/4EPI7De3cKg

OpenMP in LLVM

Weekly Meeting: https://bit.ly/2Zqt49v

Johannes Doerfert
jdoerfert@anl.gov
Argonne National Lab



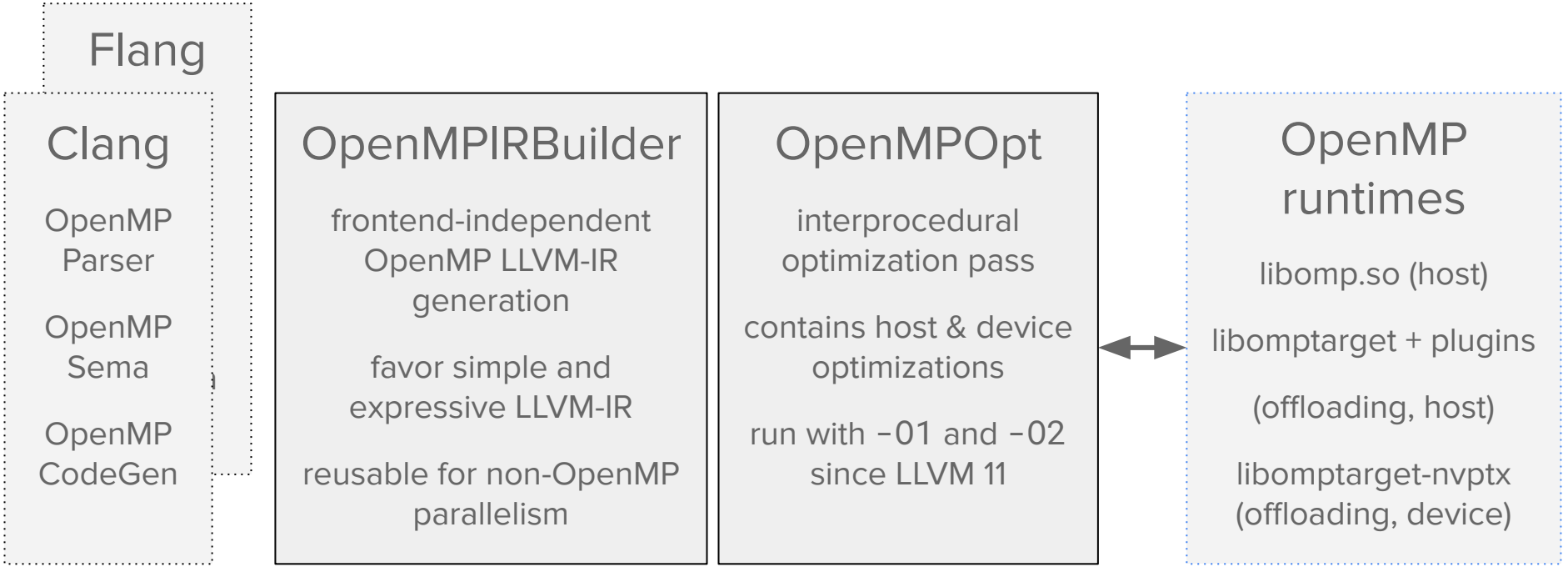
Slide originally presented at LLVM Dev'21

https://youtu.be/4EPI7De3cKg

OpenMP in LLVM

Weekly Meeting: https://bit.ly/2Zqt49v

Johannes Doerfert
jdoerfert@anl.gov
Argonne National Lab



Slide originally presented at LLVM Dev'21

https://youtu.be/4EPI7De3cKg

OpenMP in LLVM

Weekly Meeting: https://bit.ly/2Zqt49v

Johannes Doerfert
jdoerfert@anl.gov
Argonne National Lab

New optimizations for GPUs

Flang

Clang

OpenMP Parser

OpenMP Sema

OpenMP CodeGen

OpenMPIRBuilder

frontend-independent 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 -O1 and -O2 since LLVM 11

OpenMP runtimes

libomp.so (host)

libomptarget + plugins

(offloading, host)

libomptarget-nvptx (offloading, device)

Slide originally presented at LLVM Dev'21

https://youtu.be/4EPI7De3cKg

OpenMP in LLVM

Weekly Meeting: https://bit.ly/2Zqt49v

Johannes Doerfert
jdoerfert@anl.gov
Argonne National Lab

New optimizations for GPUs

Re-design for optimization

Flang

Clang

OpenMP Parser

OpenMP Sema

OpenMP CodeGen

OpenMPIRBuilder

frontend-independent 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 -O1 and -O2 since LLVM 11

OpenMP runtimes

libomp.so (host)

libomptarget + plugins

(offloading, host)

libomptarget-nvptx (offloading, device)

Slide originally presented at LLVM Dev'21

https://youtu.be/4EPI7De3cKg

The Beginning - OpenMP Implementation Consequences

Context — Compiler Optimization

Original Program

```
int y = 7;

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

After Optimizations

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

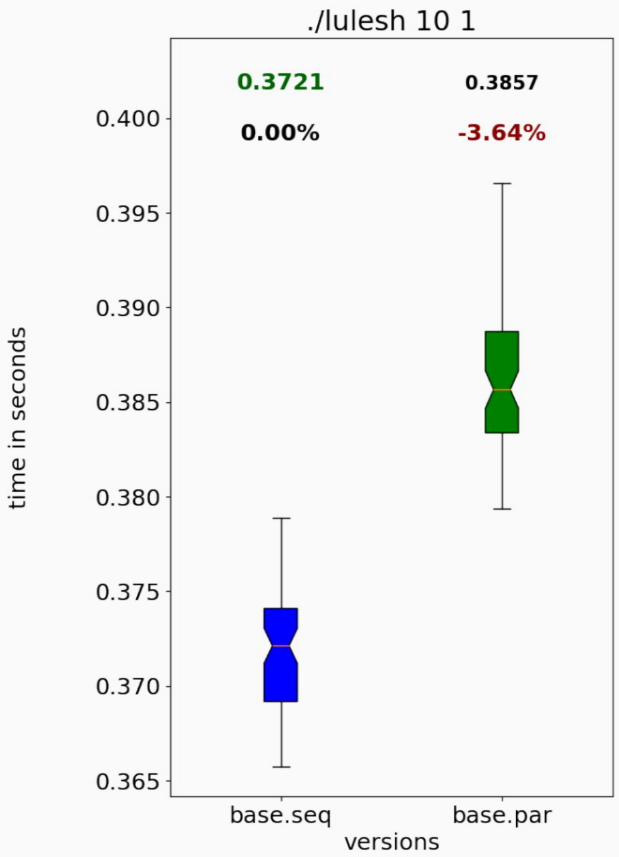
```
int y = 7;
#pragma omp parallel for
for (i = 0; i < N; i++) {
  f(y, i);
}
g(y);
```

After Optimizations

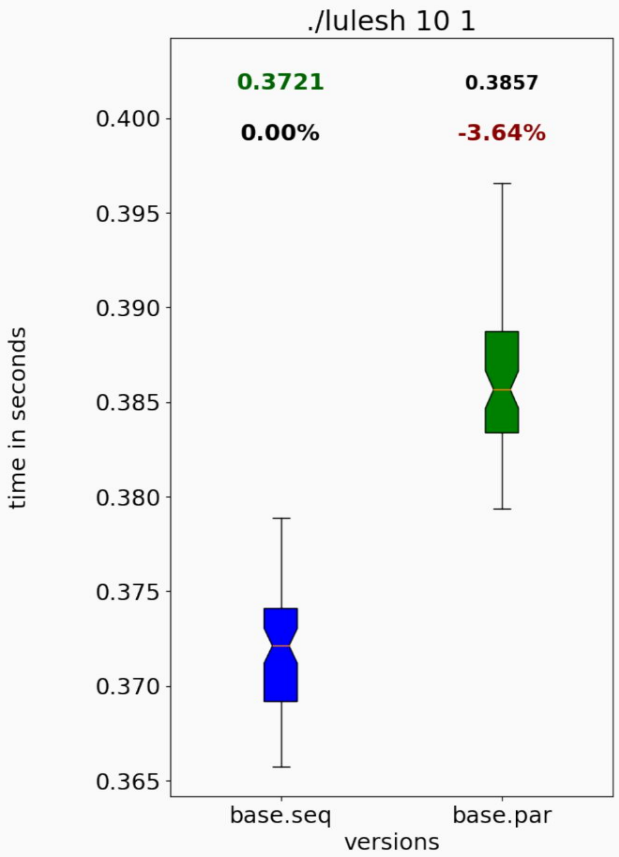
```
int y = 7;
#pragma omp parallel for
for (i = 0; i < N; i++) {
  f(y, i);
}
g(y);
```

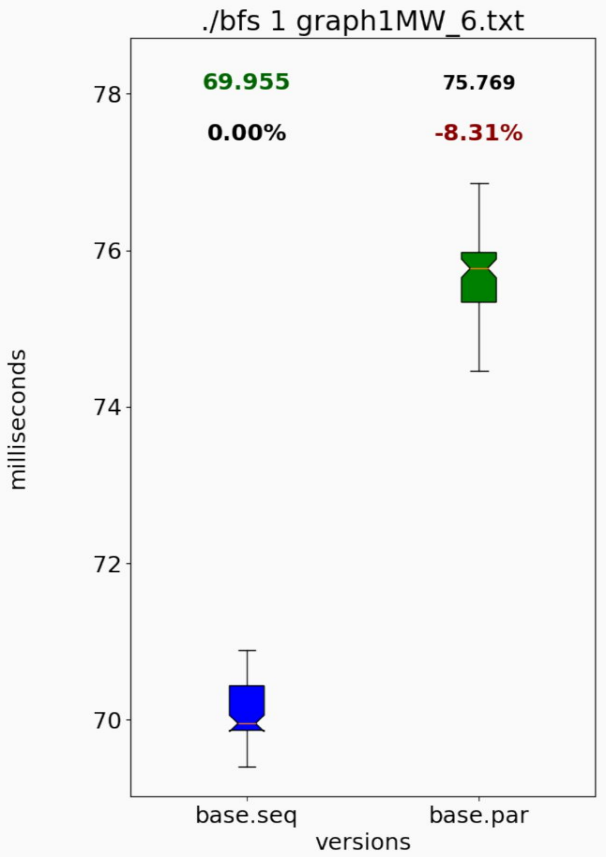
Why is this important?

Sequential Performance of Parallel Programs

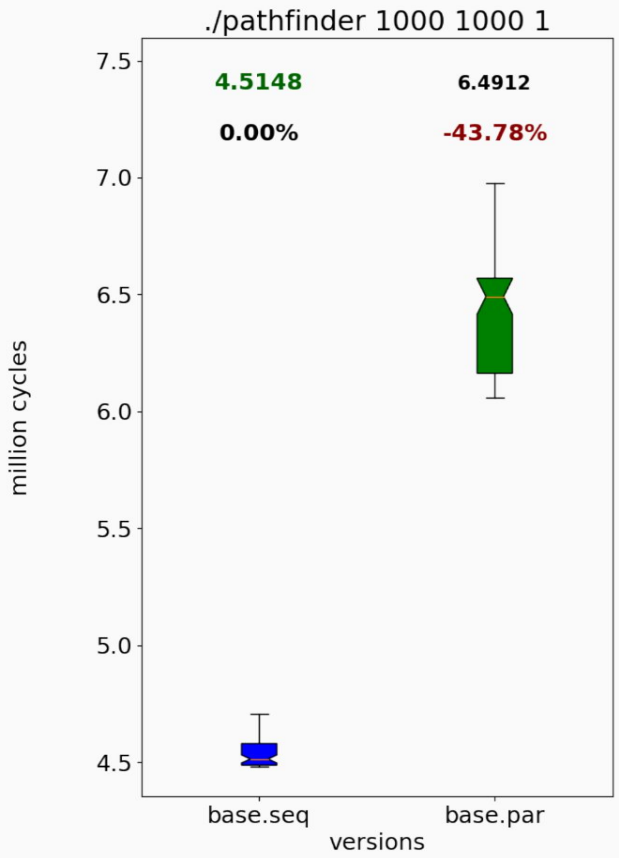


Sequential Performance of Parallel Programs

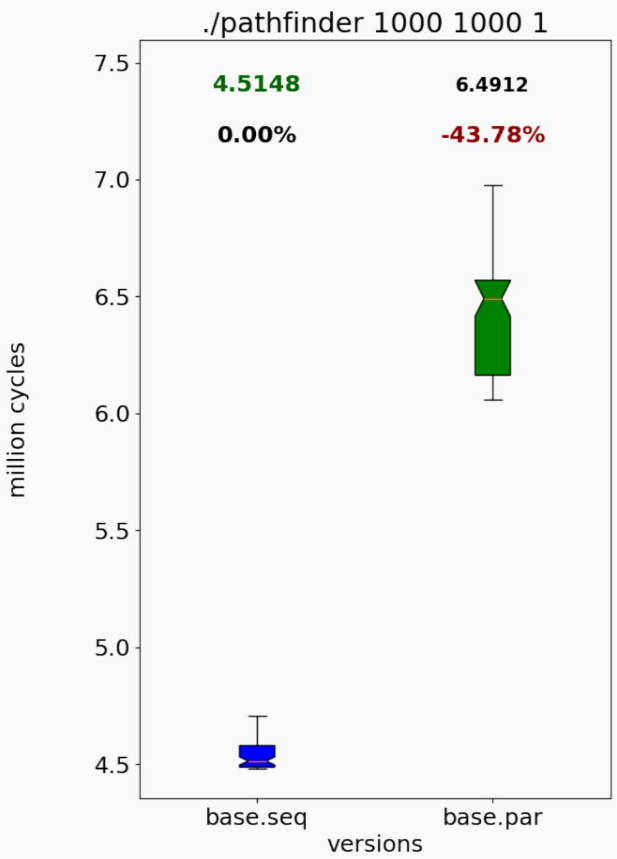


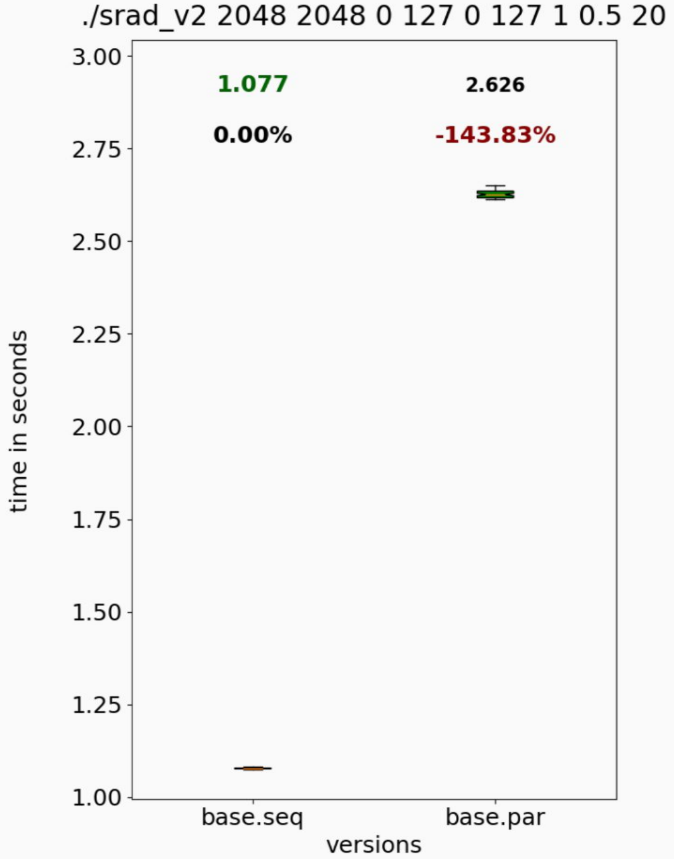


Sequential Performance of Parallel Programs



Sequential Performance of Parallel Programs





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 almost done 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

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

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

#pragma omp parallel
do_work(A, B);
```

Optimization Remarks

Example: OpenMP runtime call deduplication

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

```
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

```
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 ...
```

```
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 surroundi unction 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 targ ested inside a target construct without intermediate code. This can lead to excessive register usage for unrelated target e translation unit due to spurious call edges assumed by ptxas.
remark: Specialize parallel region that is only reached from a single target regi us call edges and excessive register usage in other target regions. (parallel region ID: __omp_outlined__1_wrapper, ker loading_35_a1e179_foo_l7)
remark: Target region containing the parallel region that is specializ ID: __omp_outlined__1_wrapper, kernel ID: __omp_offloading_35_a1e179_foo_l7)
remark: Found a parallel region that is called in a target re of a combined target construct nor nested inside a target construct without intermediate code. This can lead to excessiv or unrelated target regions in the same translation unit due to spurious call edges assumed by ptxas.
remark: Specialize parallel region that is o a single target region to avoid spurious call edges and excessive register usage in other target regions. (parallel region ID ed__3_wrapper, kernel ID: __omp_offloading_35_a1e179_foo_l7)
remark: Target region containing the pa on 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
```

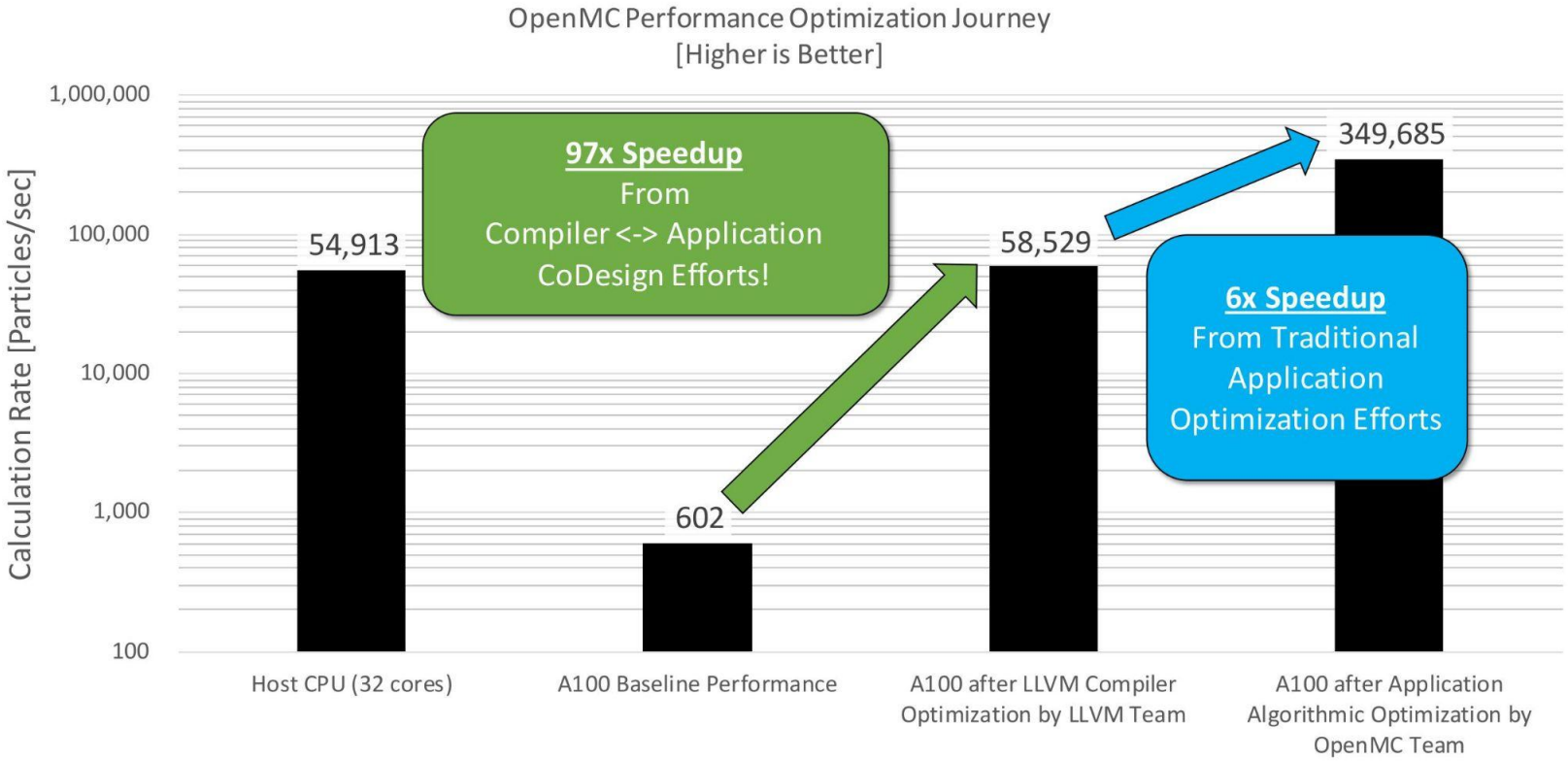
Explained Online!

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

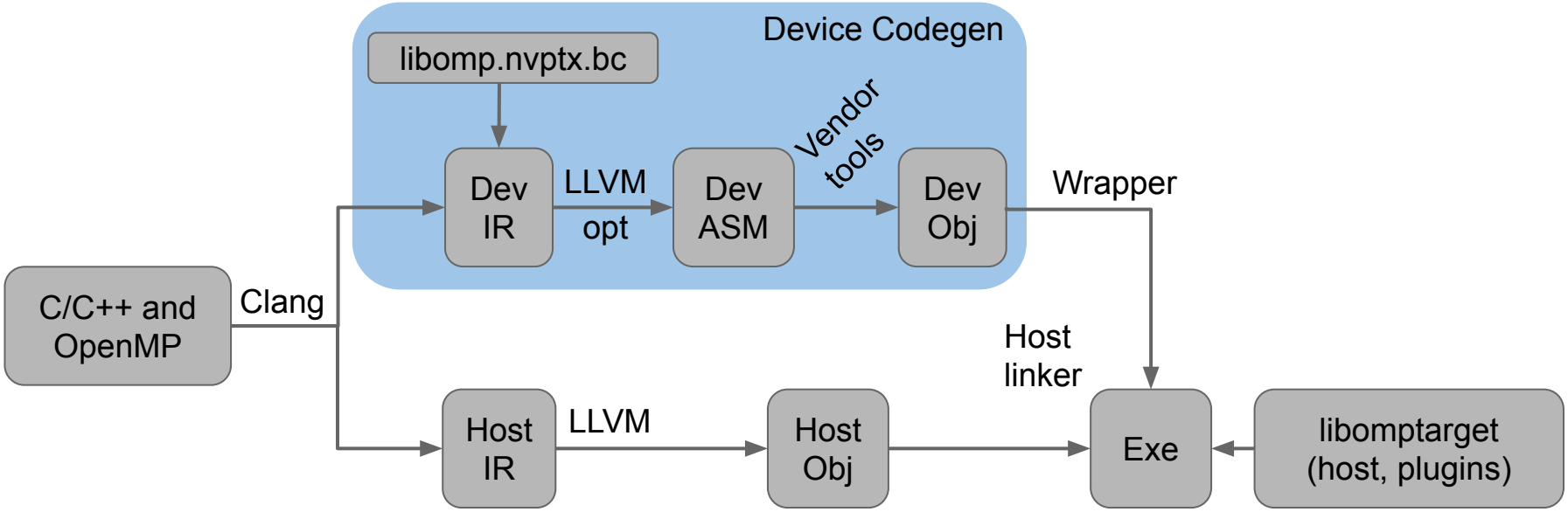
```
$ clang -O2 generic.c -fopenmp -fopenmp-targets=nvptx64-nvidia-cuda -o generic
$ env LIBOMPTARGET_INFO=16 ./generic

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
```

The Goal: Application Compiler Co-Design

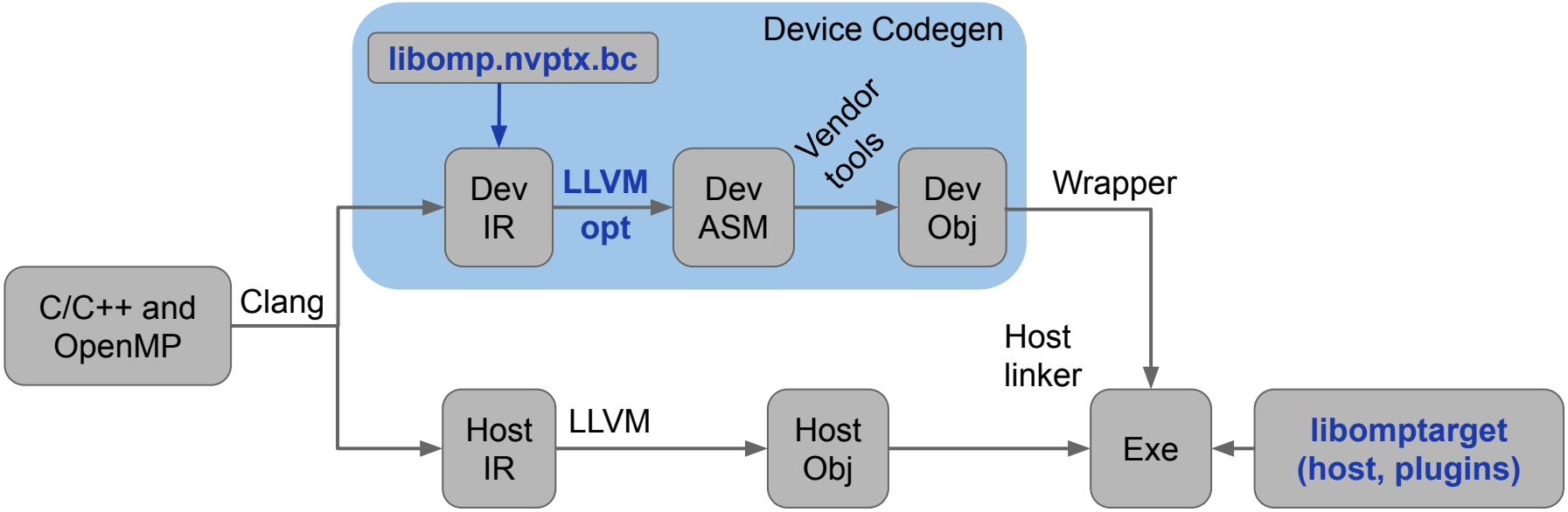


OpenMP Offloading (in LLVM)

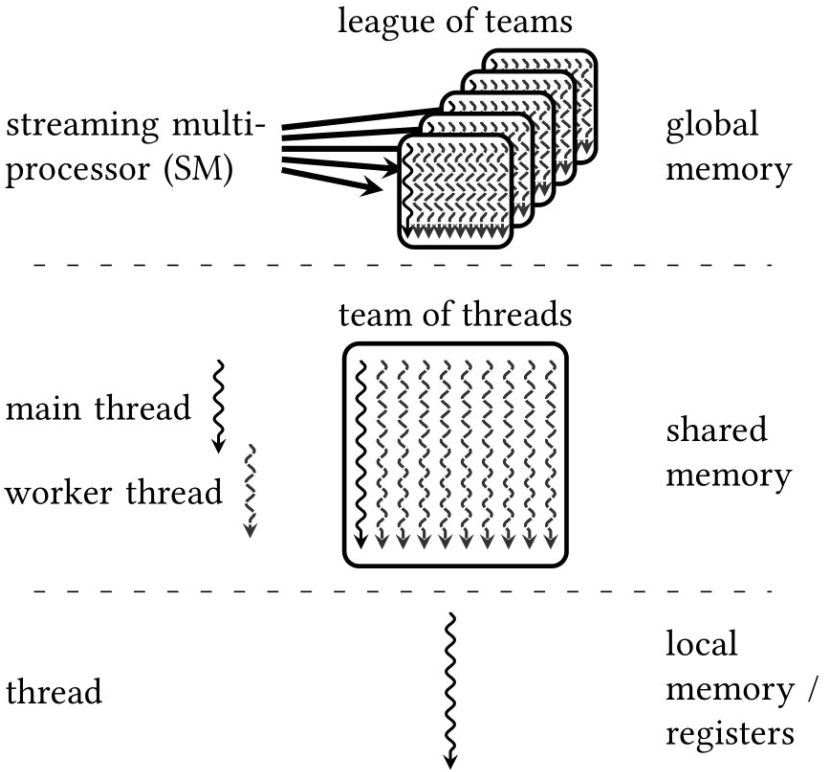
OpenMP Offload Compilation Toolchain



OpenMP Offload Compilation Toolchain



OpenMP to GPU Mapping



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


```
#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/C[😢]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

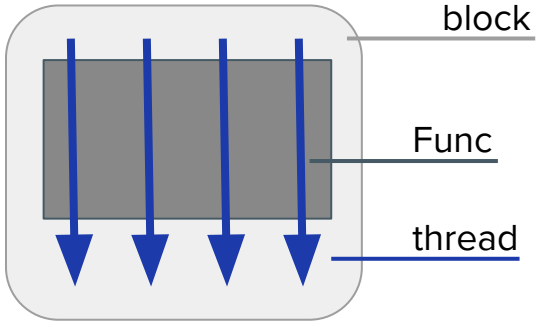
not today 😢

But the new device linker work including **device-side LTO** is almost done!

OpenMP Offloading vs Kernel Languages

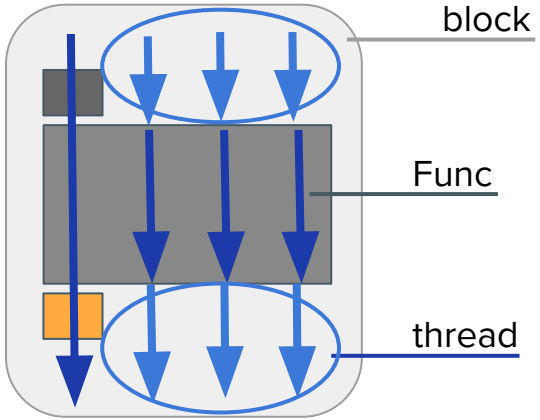
LLVM/OpenMP

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



SPMD-mode

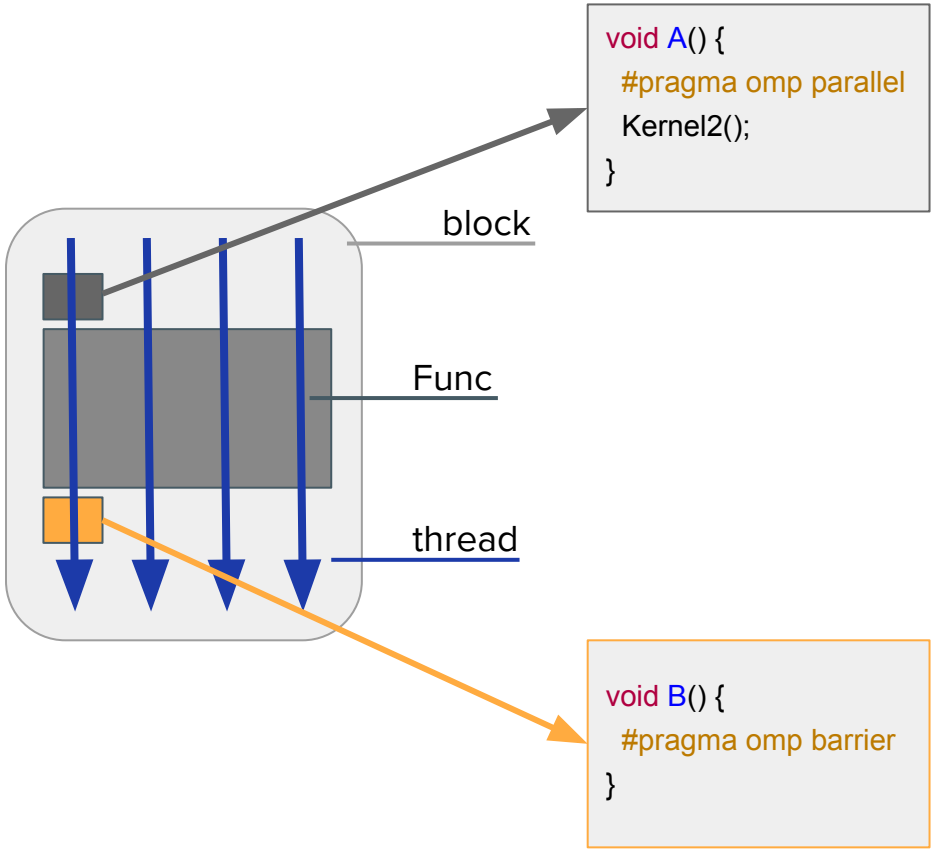
```
#pragma omp target teams num_teams(1)
{
  A();
  #pragma omp parallel num_threads(4) default(firstprivate)
  {
    Func(args);
  }
  B();
}
```



Generic-mode

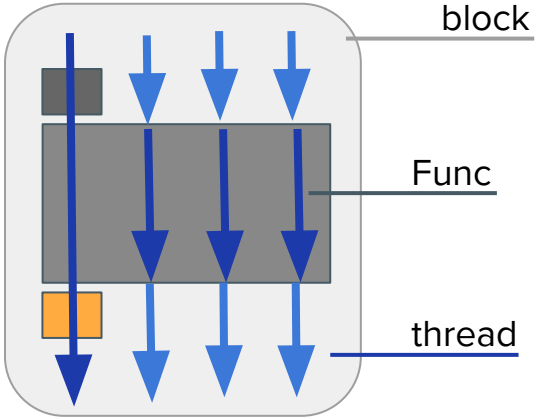
OpenMP Offloading vs Kernel Languages

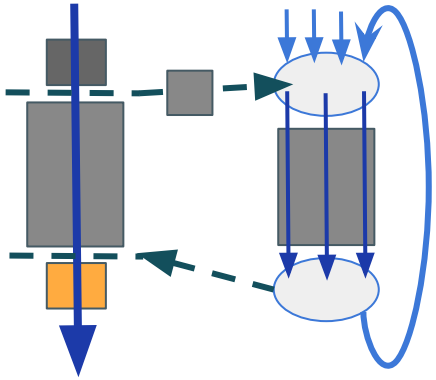
```
#pragma omp target teams num_teams(1)
{
  #pragma omp parallel num_threads(4) default(firstprivate)
  {
    if (omp_get_thread_num() == 0)
      A();
    #pragma omp barrier
    Func(args);
    #pragma omp barrier
    if (omp_get_thread_num() == 0)
      B();
  }
}
```



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();
}
```





(Some) Motivational Problems

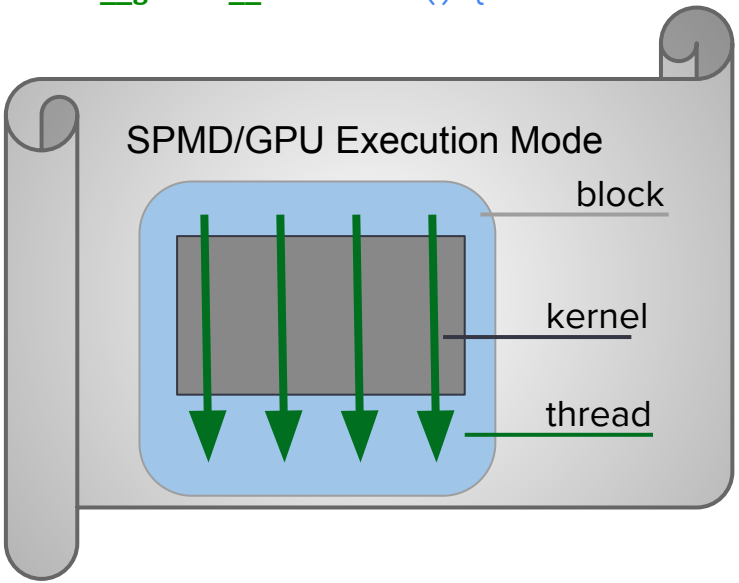
OpenMP Offload vs CUDA

```cuda
__global__ void cuda() {

  __shared__ double Buffer[BLOCK_SIZE];

  int L;

  if (threadIdx.x == 0)
    single_thread_init();
  __syncthreads();


  L = load_data(Buffer, threadIdx.x);
  __syncthreads();


  if (L != 0)
    parallel_work(Buffer, threadIdx.x);


}
```

```openmp
void openmp() {
#pragma omp target teams distribute
  for (int i = 0; i < GRID_SIZE; i += BLOCK_SIZE) {
    double Buffer[BLOCK_SIZE];

    int L;

    // No conditional, conceptually one thread only
    single_thread_init();
    // No synchronization, again, one thread

    #pragma omp parallel for
    for (int j = 0; j < BLOCK_SIZE; ++j)
      L = load_data(Buffer, j);
    // Synchronization is implicit

    #pragma omp parallel for
    for (int j = 0; j < BLOCK_SIZE; ++j)
      if (L != 0)
        parallel_work(Buffer, j);


  }
}
```

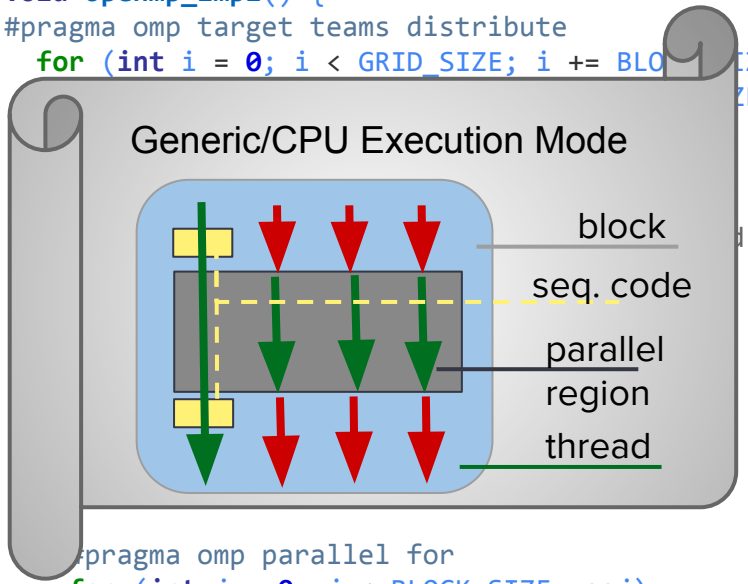
OpenMP Offload vs CUDA - Execution Mode

```
__global__ void cuda() {
```




```
  if (L != 0)
    parallel_work(Buffer, threadIdx.x);


}
```

```
void openmp_impl() {
#pragma omp target teams distribute
  for (int i = 0; i < GRID_SIZE; i += BLO      IZE) {
                                              ZE);


                                          d only
```




```
    #pragma omp parallel for
    for (int j = 0; j < BLOCK_SIZE; ++j)
      if (*L != 0)
        parallel_work(Buffer, j);

    // __omp_free(...)
  }
}
```

OpenMP Offload vs CUDA - Globalization of Locals

```
__global__ void cuda() {


  __shared__ double Buffer[BLOCK_SIZE];

  int L;

  if (threadIdx.x == 0)
    single_thread_init();
  __syncthreads();



  L = load_data(Buffer, threadIdx.x);
  __syncthreads();



  if (L != 0)
    parallel_work(Buffer, threadIdx.x);



}
```

```
void openmp_impl() {
#pragma omp target teams distribute
  for (int i = 0; i < GRID_SIZE; i += BLOCK_SIZE) {
    double *Buffer = __omp_alloc(8 * BLOCK_SIZE);

    int *L = __omp_alloc(sizeof(int));

    // No conditional, conceptually one thread only
    single_thread_init();
    // No synchronization, again, one thread

    #pragma omp parallel for shared(L)
    for (int j = 0; j < BLOCK_SIZE; ++j)
      *L = load_data(Buffer, j);
    // Synchronization is implicit

    #pragma omp parallel for shared(L)
    for (int j = 0; j < BLOCK_SIZE; ++j)
      if (*L != 0)
        parallel_work(Buffer, j);

    // __omp_free(...)
  }
}
```

OpenMP Offload vs CUDA - Explicit Loop Backedges

```cuda
__global__ void cuda() {

  __shared__ double Buffer[BLOCK_SIZE];

  int L;

  if (threadIdx.x == 0)
    single_thread_init();
  __syncthreads();



  L = load_data(Buffer, threadIdx.x);
  __syncthreads();



  if (L != 0)
    parallel_work(Buffer, threadIdx.x);



}
```

```openmp
void openmp_impl() {
#pragma omp target teams distribute
  for (int i = 0; i < GRID_SIZE; i += BLOCK_SIZE) {
    double *Buffer = __omp_alloc(8 * BLOCK_SIZE);

    int *L = __omp_alloc(sizeof(int));

    // No conditional, conceptually one thread only
    single_thread_init();
    // No synchronization, again, one thread

    #pragma omp parallel for
    for (int j = 0; j < BLOCK_SIZE; ++j)
      *L = load_data(Buffer, j);
    // Synchronization is implicit

    #pragma omp parallel for
    for (int j = 0; j < BLOCK_SIZE; ++j)
      if (*L != 0)
        parallel_work(Buffer, j);

    // __omp_free(...)
  }
}
```

OpenMP GPU Optimizations

OpenMP Offloading - Deglobalization

```cuda
__global__ void cuda() {


  __shared__ double Buffer[BLOCK_SIZE];

  int L;

  if (threadIdx.x == 0)
    single_thread_init();
  __syncthreads();



  L = load_data(Buffer, threadIdx.x);
  __syncthreads();



  if (L != 0)
    parallel_work(Buffer, threadIdx.x);


}
```

```openmp
void openmp_impl() {
#pragma omp target teams distribute

  for (int i = 0; i < GRID_SIZE; i += BLOCK_SIZE) {
    double Buffer[BLOCK_SIZE];
    #pragma omp allocate(Buffer) allocator(cgroup)
    int L;
    #pragma omp allocate(L) allocator(thread)
    // No conditional, conceptually one thread only
    single_thread_init();
    // No synchronization, again, one thread

    #pragma omp parallel for
    for (int j = 0; j < BLOCK_SIZE; ++j)
      L = load_data(Buffer, j);
    // Synchronization is implicit

    #pragma omp parallel for
    for (int j = 0; j < BLOCK_SIZE; ++j)
      if (L != 0)
        parallel_work(Buffer, j);

  }
}
```

OpenMP Offloading - SPMDzation

```
__global__ void cuda() {


  __shared__ double Buffer[BLOCK_SIZE];

  int L;

  if (threadIdx.x == 0)
    single_thread_init();
  __syncthreads();



  L = load_data(Buffer, threadIdx.x);
  __syncthreads();



  if (L != 0)
    parallel_work(Buffer, threadIdx.x);


}
```

```
void openmp_impl() {
#pragma omp target teams distribute
#pragma omp parallel
  for (int i = 0; i < GRID_SIZE; i += BLOCK_SIZE) {
    double Buffer[BLOCK_SIZE];
    #pragma omp allocate(Buffer) allocator(cgroup)
    int L;
    #pragma omp allocate(L) allocator(thread)
    if (__omp_get_thread_id() == 0)
      single_thread_init();
    #pragma omp barrier // aligned

    #pragma omp for nowait
    for (int j = 0; j < BLOCK_SIZE; ++j)
      L = load_data(Buffer, j);
    #pragma omp barrier // aligned

    #pragma omp for nowait
    for (int j = 0; j < BLOCK_SIZE; ++j)
      if (L != 0)
        parallel_work(Buffer, j);
    #pragma omp barrier // aligned
  }
}
```

OpenMP Offloading - Loop Oversubscription

```
__global__ void cuda() {


  __shared__ double Buffer[BLOCK_SIZE];

  int L;

  if (threadIdx.x == 0)
    single_thread_init();
  __syncthreads();



  L = load_data(Buffer, threadIdx.x);
  __syncthreads();



  if (L != 0)
    parallel_work(Buffer, threadIdx.x);


}
```

```
void openmp_impl() {
#pragma omp target teams parallel
int i = omp_get_team_num();
if (i < GRID_SIZE) {
    double Buffer[BLOCK_SIZE];
    #pragma omp allocate(Buffer) allocator(cgroup)
    int L;
    #pragma omp allocate(L) allocator(thread)
    if (__omp_get_thread_id() == 0)
      single_thread_init();
    #pragma omp barrier // aligned

    int j = omp_get_thread_num();
    if (j < BLOCK_SIZE)
      L = load_data(Buffer, j);
    #pragma omp barrier // aligned

    int j = omp_get_thread_num();
    if (j < BLOCK_SIZE)
      if (L != 0)
        parallel_work(Buffer, j);
    #pragma omp barrier // aligned
  }
}
```

OpenMP Offloading - (Aligned) Barrier Removal

```
__global__ void cuda() {



  __shared__ double Buffer[BLOCK_SIZE];

  int L;

  if (threadIdx.x == 0)
    single_thread_init();
  __syncthreads();



  L = load_data(Buffer, threadIdx.x);
  __syncthreads();



  if (L != 0)
    parallel_work(Buffer, threadIdx.x);


}
```

```
void openmp_impl() {
#pragma omp target teams parallel
int i = omp_get_team_num();
if (i < GRID_SIZE) {
    double Buffer[BLOCK_SIZE];
    #pragma omp allocate(Buffer) allocator(cgroup)
    int L;
    #pragma omp allocate(L) allocator(thread)
    if (__omp_get_thread_id() == 0)
      single_thread_init();
    #pragma omp barrier // aligned

    int j = omp_get_thread_num();
    if (j < BLOCK_SIZE)
      L = load_data(Buffer, j);
    #pragma omp barrier // aligned

    int j = omp_get_thread_num();
    if (j < BLOCK_SIZE)
      if (L != 0)
        parallel_work(Buffer, j);
    #pragma omp barrier // aligned
  }
}
```

Optimization Implementation

The OpenMP-Opt Pass

- OpenMP-specific optimizations (= instant no-op for non-openmp codes)
- Run early (as module pass) and late (as CG-SCC pass)
- Embedded Domain Knowledge (recognizes `omp_*` and `__kmpc_*` calls)
- Uses the Attributor IPO framework and provides custom Abstract Attributes:
  - AAExecutionDomain - Determine if a block is executed by the main thread only, or by all threads in an “aligned” fashion.
  - AAFoldRuntimeCall - Replace runtime calls with their constant return value (if known).
  - AAHeapToStack - Replace globalized memory with an alloca.
  - AAHeapToShared - Replace globalized memory with shared memory (if heap-2-stack failed).
  - AAKernelInfo - Track reaching kernels, optimize kernel execution mode, ...
- *Inter-procedural by design*

Optimization - Runtime Co-Design

Remarks & Assumptions - Interactive Optimization

OpenMP-Opt emits remarks:

- -Rpass=openmp-opt
- -Rpass-missed=openmp-opt
- -Rpass-analysis=openmp-opt

to report success and failure,

and utilizes assumptions:

- `#pragma omp assumes ...`
- `__attribute__((assume("...")))`
- command line flags

to enhance static analysis.

`omp_no_openmp`

`omp_no_parallelism`

`omp_no_openmp_routines`

} OpenMP 5.1 spec assumptions

`ompx_spmd_amenable`

`ompx_aligned_barrier`

`ompx_no_sync`

} LLVM assumption extensions

`-fopenmp-assume-teams-oversubscription`

`-fopenmp-assume-threads-oversubscription`

Explicit (Shared) Global State and Powerful IPO

```
__shared__ StateTy State;

__global__ void kernel() {
  State.TeamSize = 1;
  __omp_aligned_barrier(); // assume((ompx_aligned))
  __omp_parallel(outlined_fn, ...);
}
```

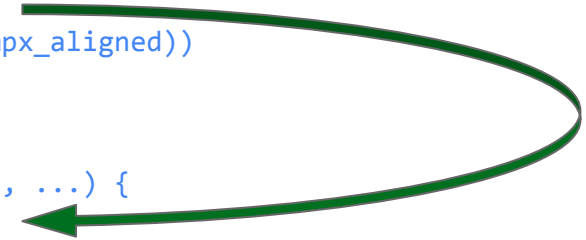
```
__device__ static void outlined_fn(...) {
  // Do not (transitively) call __omp_parallel.
  use(State.TeamSize);
}
```

Explicit (Shared) Global State and Powerful IPO

```
__shared__ StateTy State;

__global__ void kernel() {
  State.TeamSize = 1;
  __omp_aligned_barrier(); // assume((ompx_aligned))
  __omp_parallel(outlined_fn, ...);
}

__device__ static void __omp_parallel(fn, ...) {
  if (State.TeamSize > 1)
    return __omp_parallel_sequentialized(fn, ...);
  __omp_aligned_barrier(); // assume((ompx_aligned))
  State.TeamSize = blockDim.x;
  __omp_aligned_barrier(); // assume((ompx_aligned))
  fn();
  __omp_aligned_barrier(); // assume((ompx_aligned))
  State.TeamSize = 1;
  __omp_aligned_barrier(); // assume((ompx_aligned))
}

__device__ static void outlined_fn(...) {
  // Do not (transitively) call __omp_parallel.
  use(State.TeamSize);
}
```

Explicit (Shared) Global State and Powerful IPO

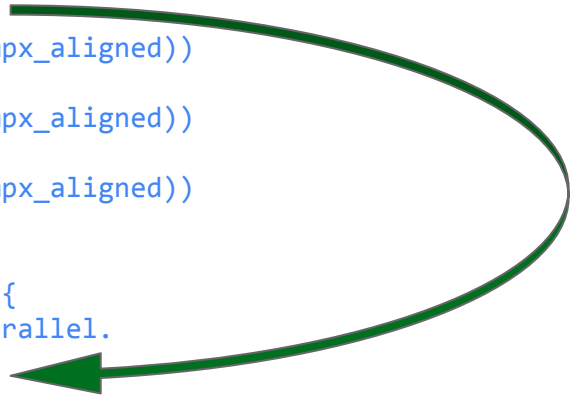
```
__shared__ StateTy State;

__global__ void kernel() {
  State.TeamSize = 1;
  __omp_aligned_barrier(); // assume((ompx_aligned))
  __omp_parallel(outlined_fn, ...);
}

__device__ static void __omp_parallel(fn, ...) {
  if (1 > 1)
    return __omp_parallel_sequentialized(fn, ...);
  __omp_aligned_barrier(); // assume((ompx_aligned))
  State.TeamSize = blockDim.x;
  __omp_aligned_barrier(); // assume((ompx_aligned))
  fn();
  __omp_aligned_barrier(); // assume((ompx_aligned))
  State.TeamSize = 1;
  __omp_aligned_barrier(); // assume((ompx_aligned))
}

__device__ static void outlined_fn(...) {
  // Do not (transitively) call __omp_parallel.
  use(State.TeamSize);
}
```



IP-Reachability +
shared memory lifetime

Explicit (Shared) Global State and Powerful IPO

```
__shared__ StateTy State;

__global__ void kernel() {
  State.TeamSize = 1;
  __omp_aligned_barrier(); // assume((ompx_aligned))
  __omp_parallel(outlined_fn, ...);
}

__device__ static void __omp_parallel(fn, ...) {


  __omp_aligned_barrier(); // assume((ompx_aligned))
  State.TeamSize = blockDim.x;
  __omp_aligned_barrier(); // assume((ompx_aligned))
  fn();
  __omp_aligned_barrier(); // assume((ompx_aligned))
  State.TeamSize = 1;
  __omp_aligned_barrier(); // assume((ompx_aligned))
}

__device__ static void outlined_fn(...) {
  // Do not (transitively) call __omp_parallel.
  use(blockDim.x);
}
```

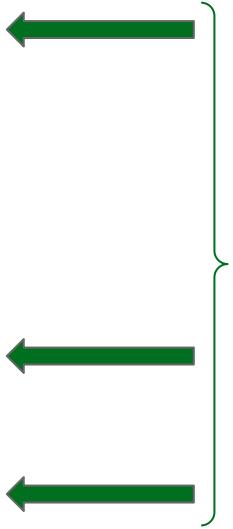


IP-Reachability + shared memory lifetime + IP-Dominance + intrinsic annotations

Explicit (Shared) Global State and Powerful IPO

```
__shared__ StateTy State;

__global__ void kernel() {
  State.TeamSize = 1;
  __omp_aligned_barrier(); // assume((ompx_aligned))
  __omp_parallel(outlined_fn, ...);
}

__device__ static void __omp_parallel(fn, ...) {


  __omp_aligned_barrier(); // assume((ompx_aligned))
  State.TeamSize = blockDim.x;
  __omp_aligned_barrier(); // assume((ompx_aligned))
  fn();
  __omp_aligned_barrier(); // assume((ompx_aligned))
  State.TeamSize = 1;
  __omp_aligned_barrier(); // assume((ompx_aligned))
}

__device__ static void outlined_fn(...) {
  // Do not (transitively) call __omp_parallel.
  use(blockDim.x);
}
```

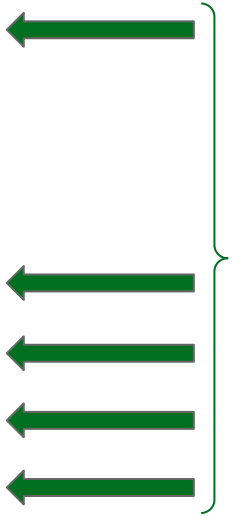
shared memory lifetime + IP-write-only



shared memory lifetime + IP-DSE

Explicit (Shared) Global State and Powerful IPO

```
__global__ void kernel() {
  __omp_aligned_barrier(); // assume((ompx_aligned))
  __omp_parallel(outlined_fn, ...);
}

__device__ static void __omp_parallel(fn, ...) {

  __omp_aligned_barrier(); // assume((ompx_aligned))
  __omp_aligned_barrier(); // assume((ompx_aligned))
  fn();
  __omp_aligned_barrier(); // assume((ompx_aligned))
  __omp_aligned_barrier(); // assume((ompx_aligned))
}

__device__ static void outlined_fn(...) {
  // Do not (transitively) call __omp_parallel.
  use(blockDim.x);
}
```

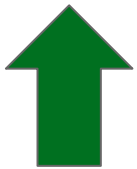


IP-aligned barrier elimination

Explicit (Shared) Global State and Powerful IPO

```
__global__ void kernel() {

  __omp_parallel(outlined_fn, ...);
}

__device__ static void __omp_parallel(fn, ...) {


  fn();


}

__device__ static void outlined_fn(...) {
  // Do not (transitively) call __omp_parallel.
  use(blockDim.x);
}
```

```
__global__ void kernel() {
  use(blockDim.x);
}
```

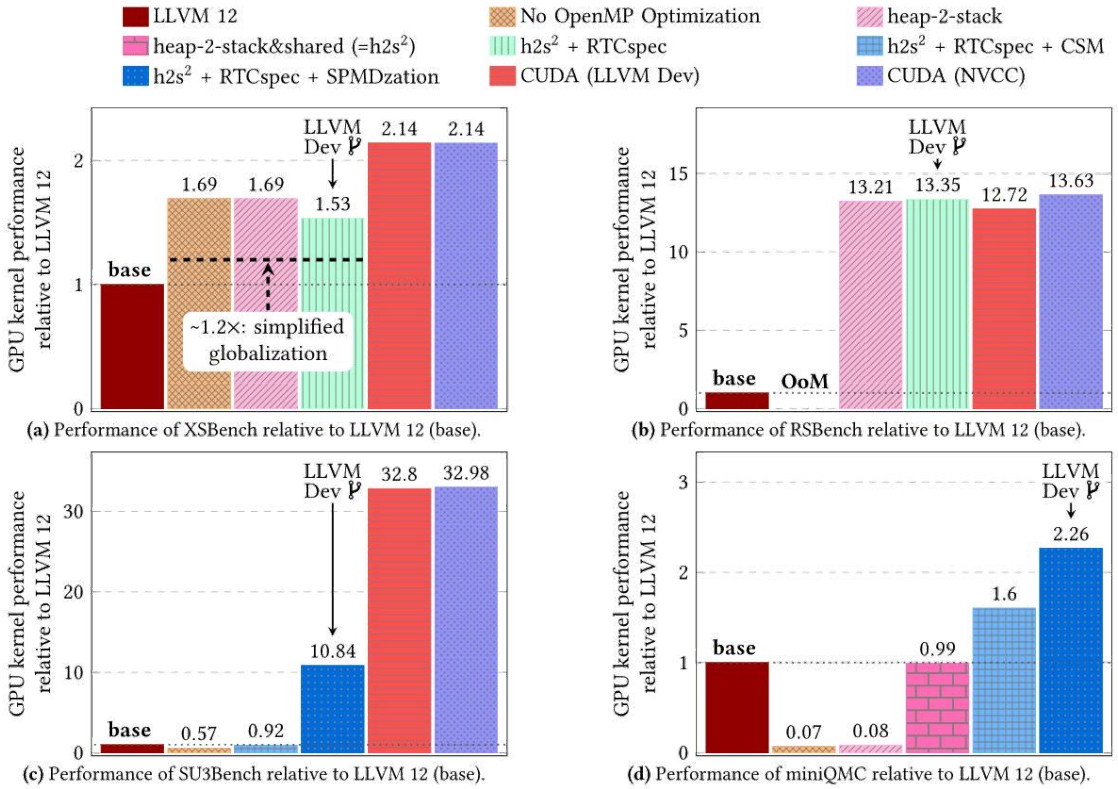


Simplifications, e.g., inlining, remove (now empty) abstraction layers.

⇒ CUDA-like code (IR and PTX)

Evaluation

OpenMP Offloading Performance (Part I)



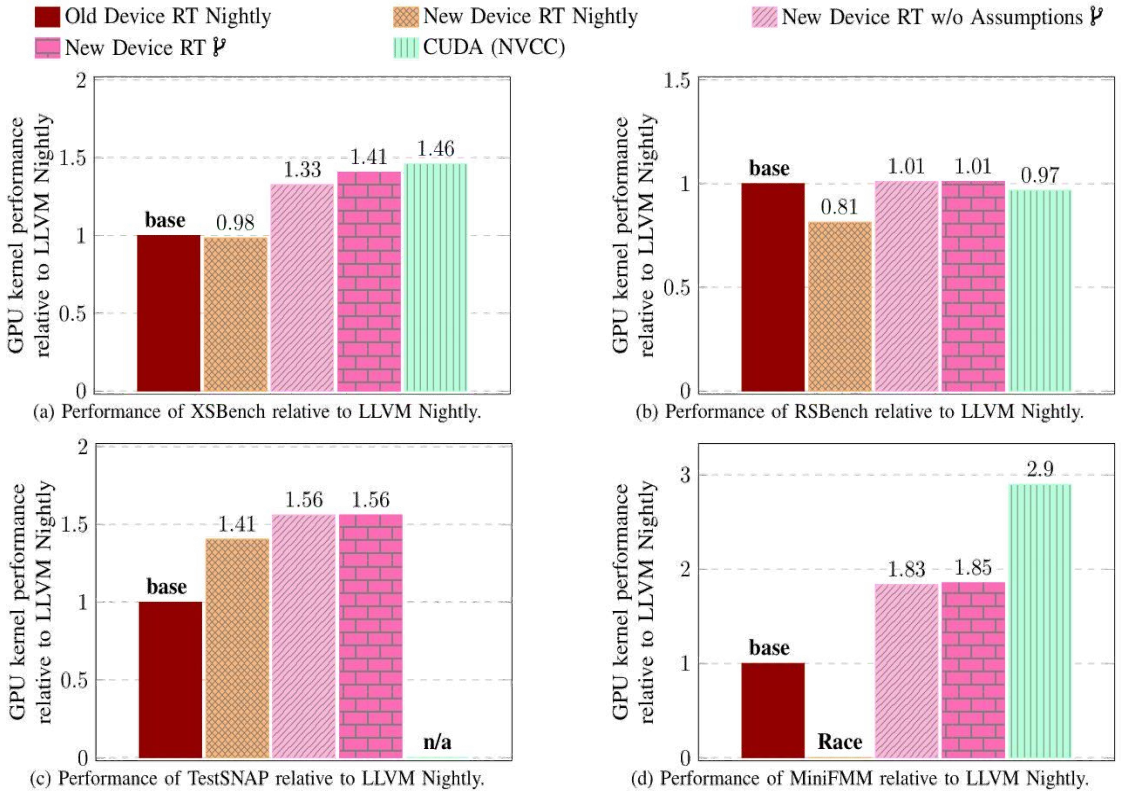
(a) Performance of XSBench relative to LLVM 12 (base).

(b) Performance of RSBench relative to LLVM 12 (base).

(c) Performance of SU3Bench relative to LLVM 12 (base).

(d) Performance of miniQMC relative to LLVM 12 (base).

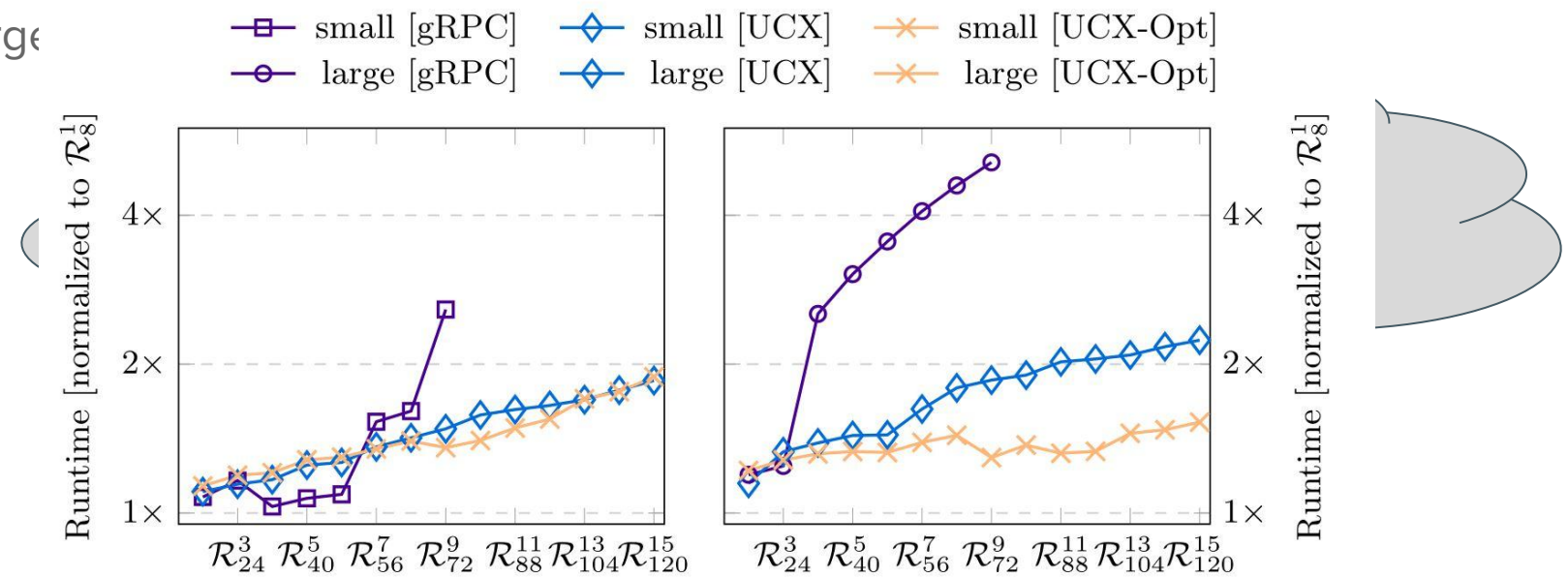
OpenMP Offloading Performance (Part II)



Cool Things to do with OpenMP

What OpenMP got (kinda) Right

The targe



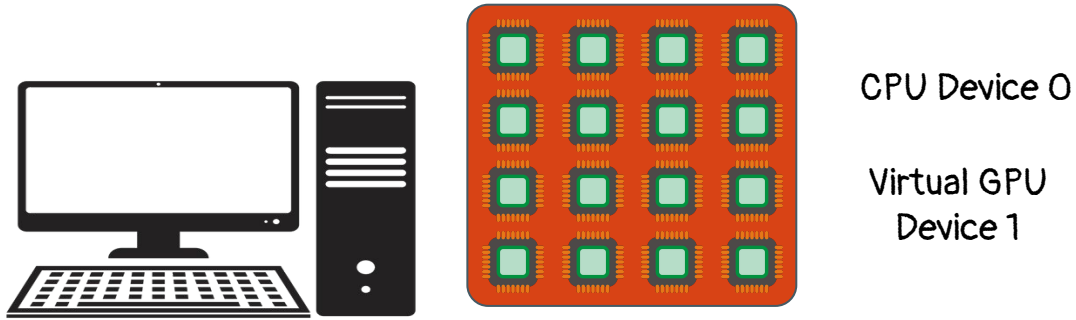
(a) Configurations running Small instance. (b) Configurations running Large instance.

Fig. 8: RSBench remote offloading performance for the ThetaGPU environment.

LLVM 12 prov

What OpenMP got (kinda) Right

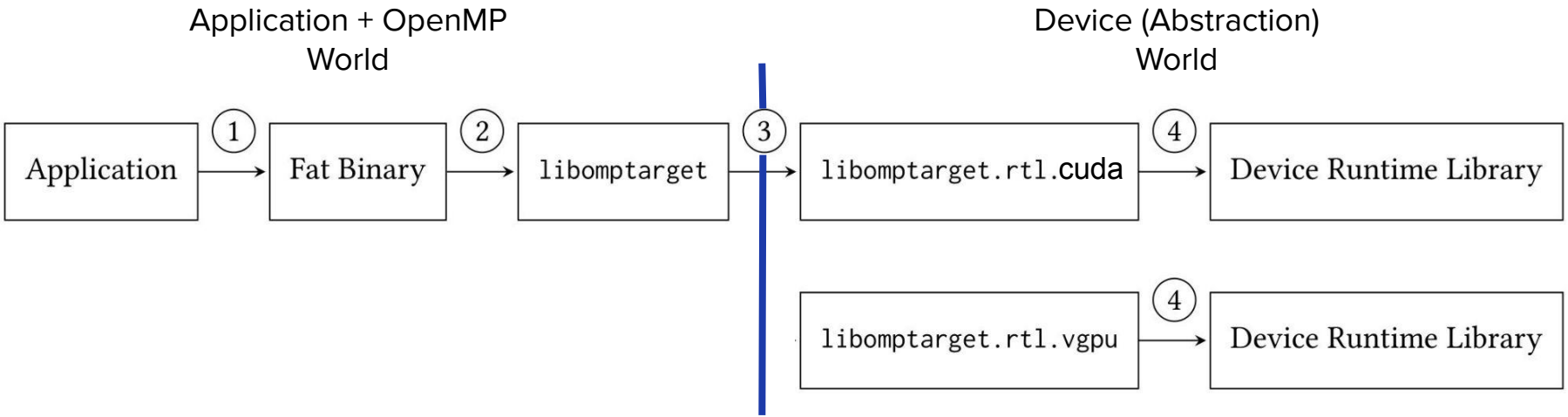
The target device abstraction:



LLVM 14 will provide a VGPU :)

What OpenMP got (kinda) Right

The target device abstraction:



Conclusion & Future Work

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 + new process for debugging, or
  - JIT for performance
- Host-Device optimizations
- Enhanced profiling support

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

Further Resources

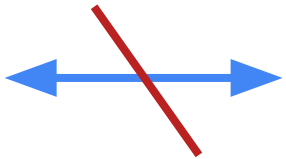
- Official LLVM OpenMP documentation page
  - https://openmp.llvm.org/
- (OpenMP) Parallelism-Aware Optimizations (LLVM Dev'20)
  - https://youtu.be/gtxWkeLCxmU
- OpenMP Webinar ('20)
  - https://www.openmp.org/events/webinar-a-compilers-view-of-the-openmp-api/
- Advancing OpenMP Offload Debugging Capabilities in LLVM (LLPP'21)
- Experience Report: Writing A Portable GPU Runtime with OpenMP 5.1 (IWOMP'21)
- Efficient Execution of OpenMP on GPUs (CGO'22)
- Co-Designing an OpenMP GPU Runtime and Optimizations for Near-Zero Overhead Execution (IPDPS'22)

Final Thoughts

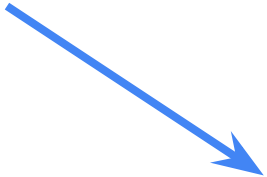
(aka. Rambling)

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) {
    // ...
  }
}
```



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

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

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

Inter-procedural analysis with OpenMP-awareness

- Internalize functions to improve inter-procedural analysis
  - All calls to the internalized version are known
- Analyse uses of known OpenMP runtime calls

```
define void @__omp_offloading_XXX() {
  call void @foo()
  ret void
}
```

Replace call →

```
define void @__omp_offloading_XXX() {
  call void @foo.internalized()
  ret void
}

define void @foo() { ... }
```

```
define void @foo() {
entry:
  %0 = call i8* @__omp_alloc(i64 4)
  call void @bar(i8* %0)
  call void @__omp_free(i8* %0)
  ret void
}
```

Clone function →

Analyze calls, replace uses →

```
define internal void @foo.internalized() {
entry:
  %0 = call i8* @__omp_alloc(i64 4)
  call void @bar.internalized(i8* %0)
  call void @__omp_free(i8* %0)
  ret void
}
```

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

What OpenMP got Wrong

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();
}
```

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:

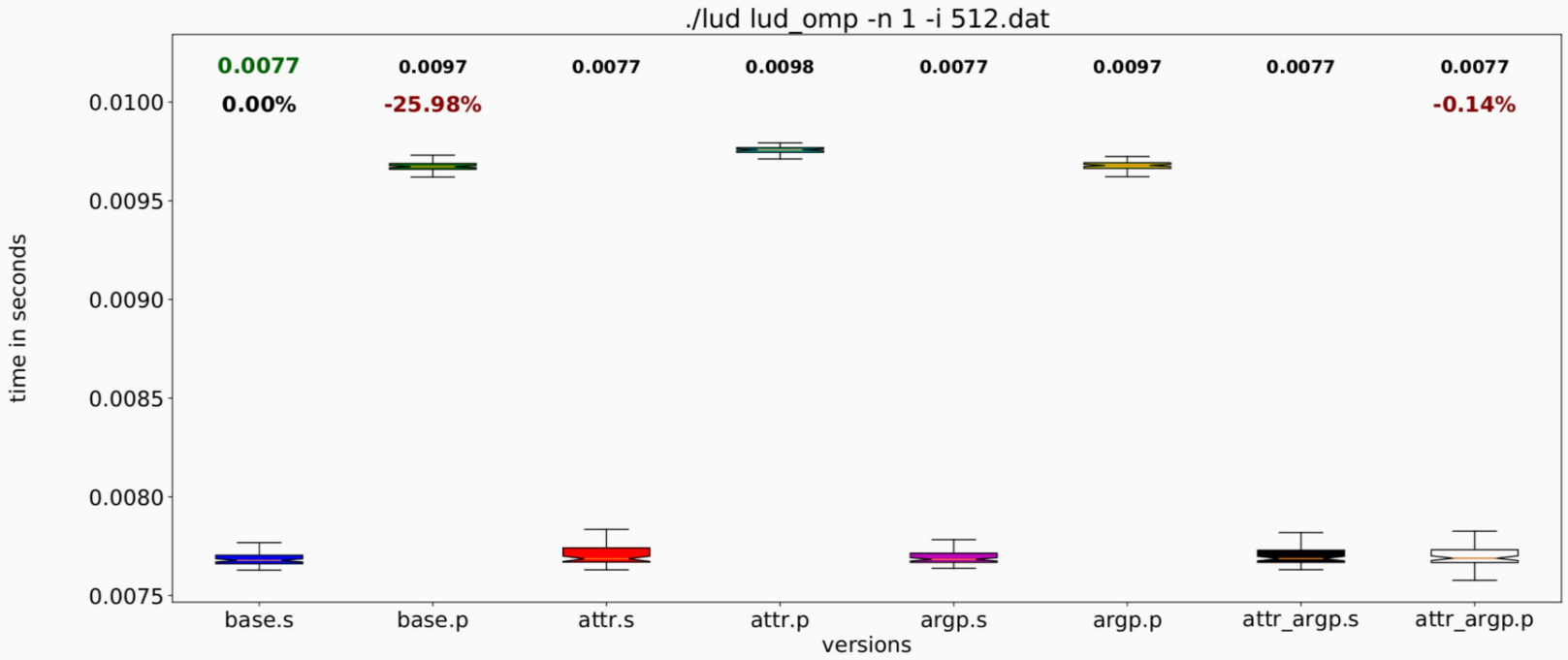
```
#pragma omp parallel for
for (int i = 0; i < N; i++)
  Out[i] = In[i] + In[i+N];
```

```
// 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)]
}
```

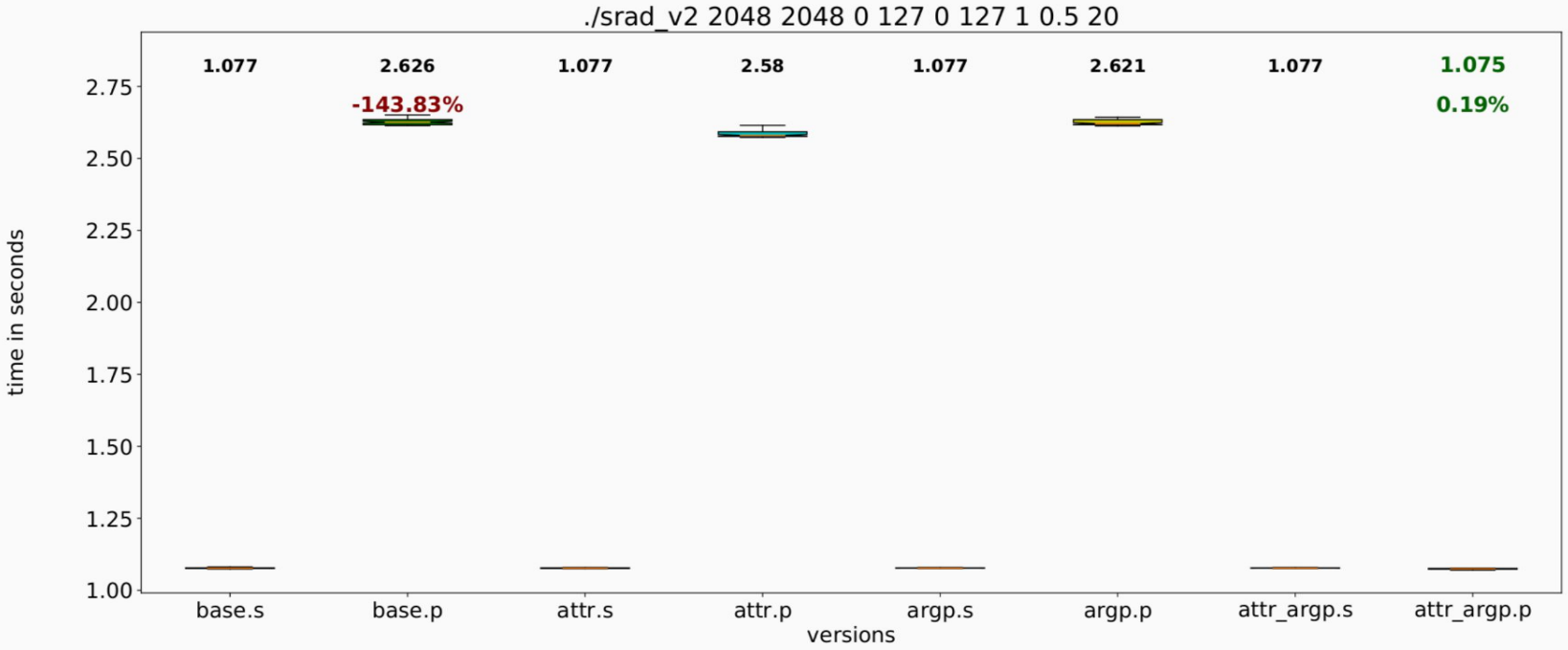
OpenMP Optimizations

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



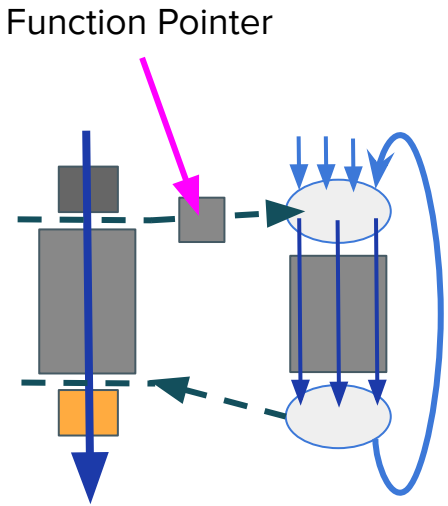
OpenMP Offloading vs Kernel Languages (simplified)

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?

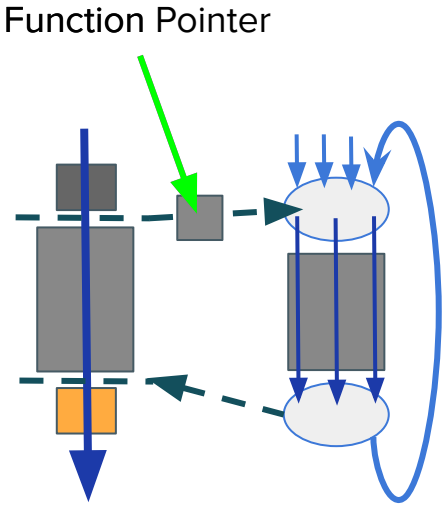
A: Yes. That's why we try to use non-function pointer IDs.



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();
  }
}
```

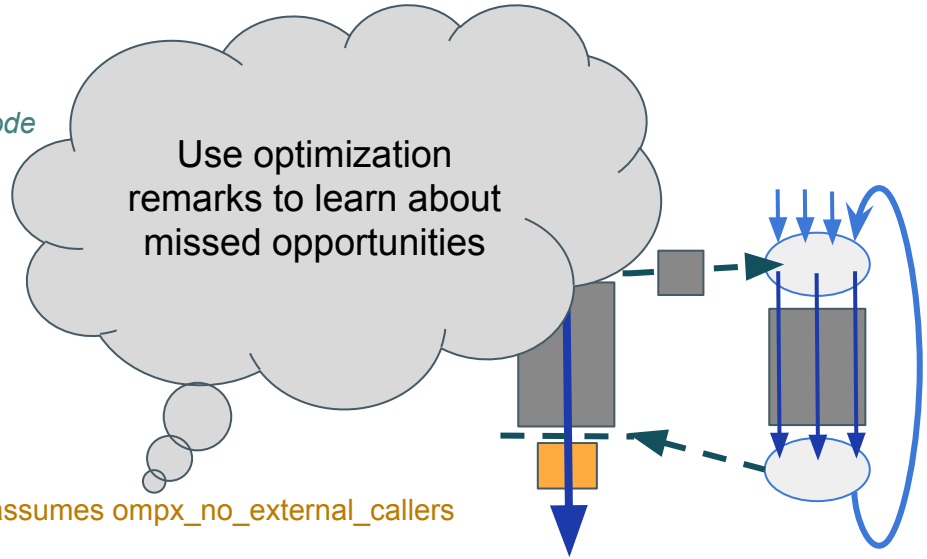


Performed since LLVM 12

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 knows more tricks :)