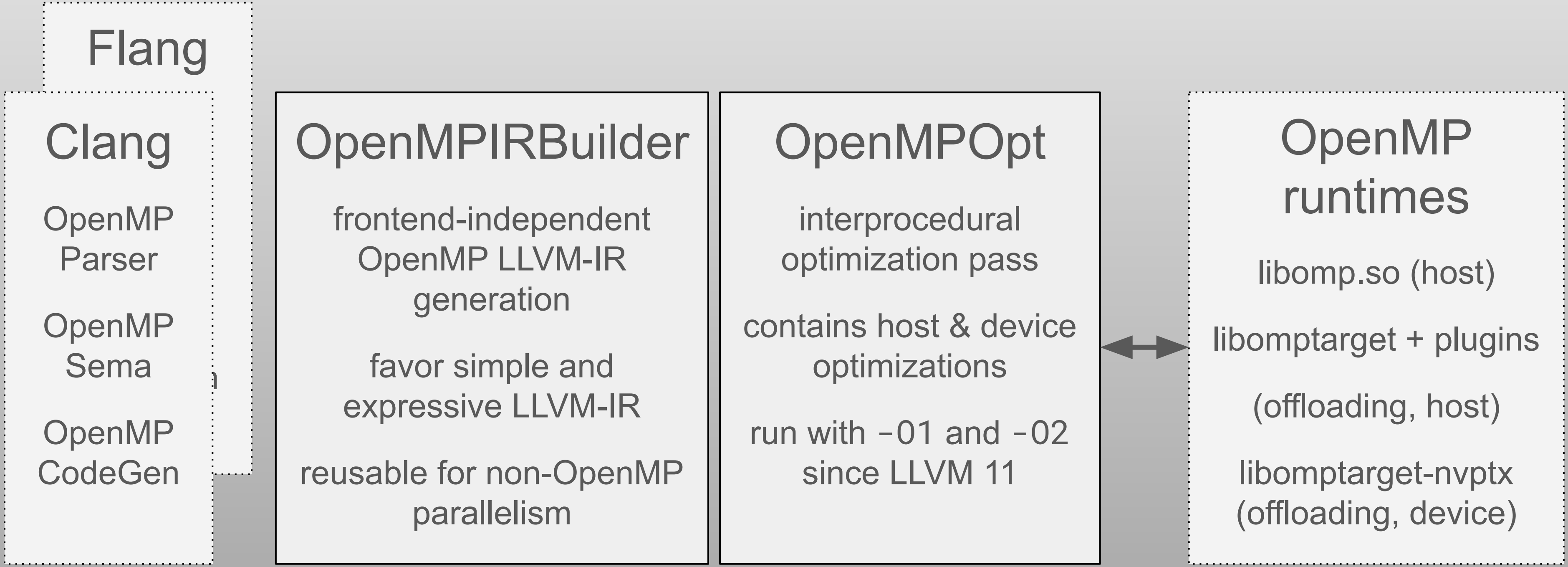
Optimizing OpenMP GPU Execution in LLVM

Melanie Cornelius (IIT), Jose Monsalve Diaz (ANL), Johannes Doerfert (ANL),
Giorgis Georgakoudis (LLNL), Joseph Huber (ORNL),
Atmn Patel (U of Waterloo), Shilei Tian (Stony Brook University)

Background

OpenMP in LLVM

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



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 −01 and −02 since LLVM 11

OpenMP runtimes

libomp.so (host)

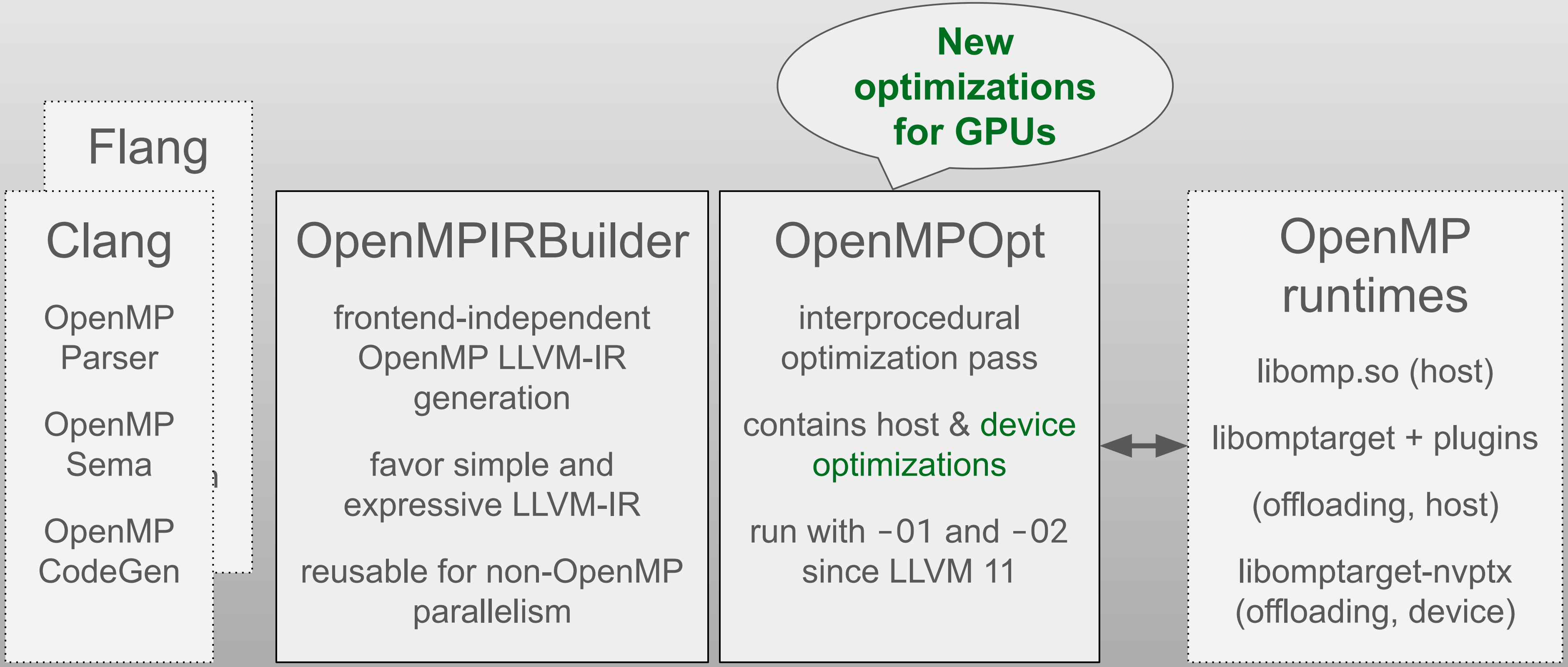
libomptarget + plugins

(offloading, host)

libomptarget-nvptx (offloading, device)

OpenMP in LLVM

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



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)

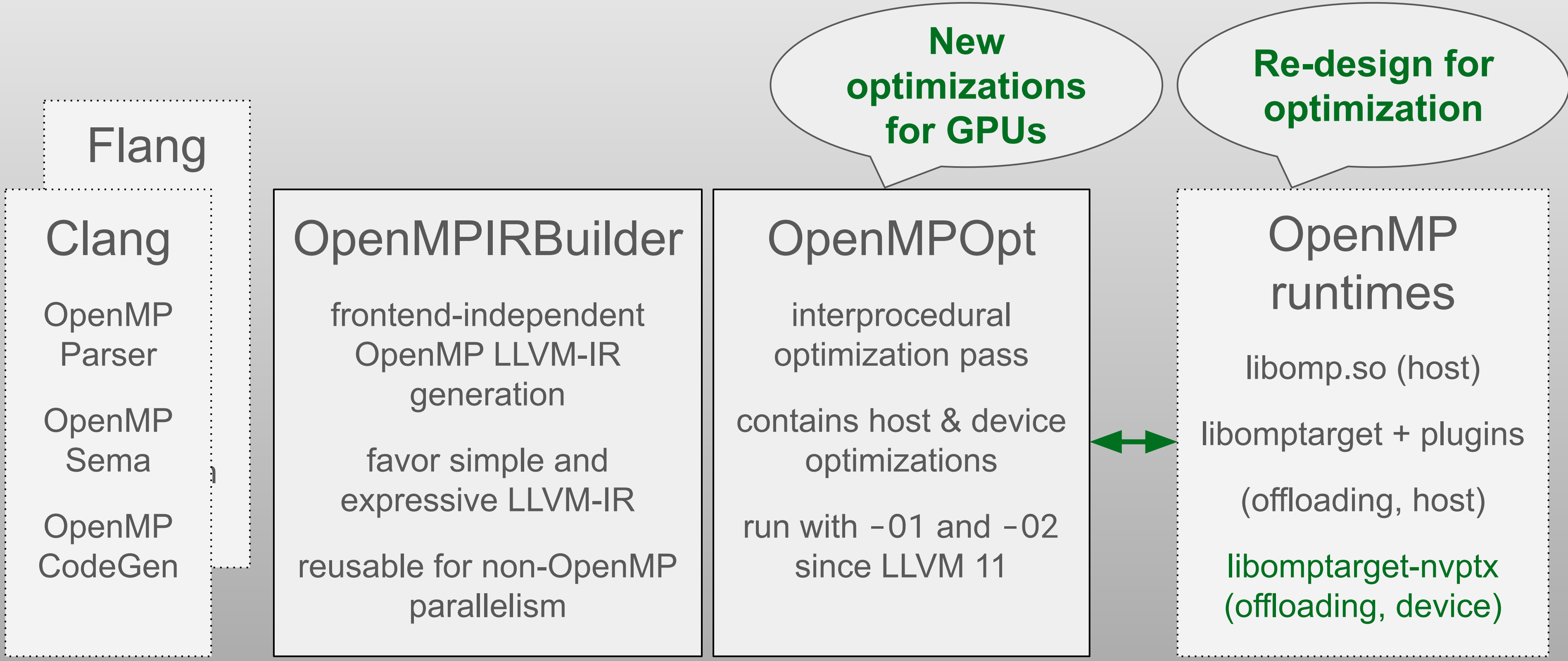
libomptarget + plugins

(offloading, host)

libomptarget-nvptx (offloading, device)

OpenMP in LLVM

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



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

New optimizations for GPUs

OpenMP runtimes

libomp.so (host)

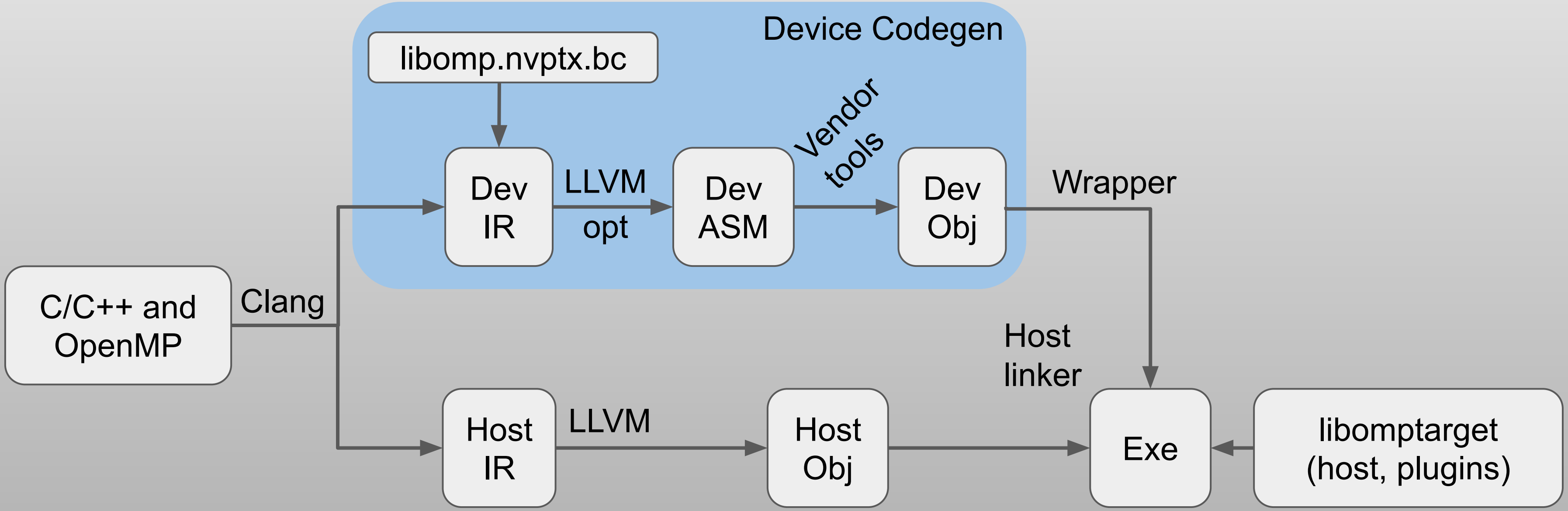
libomptarget + plugins

(offloading, host)

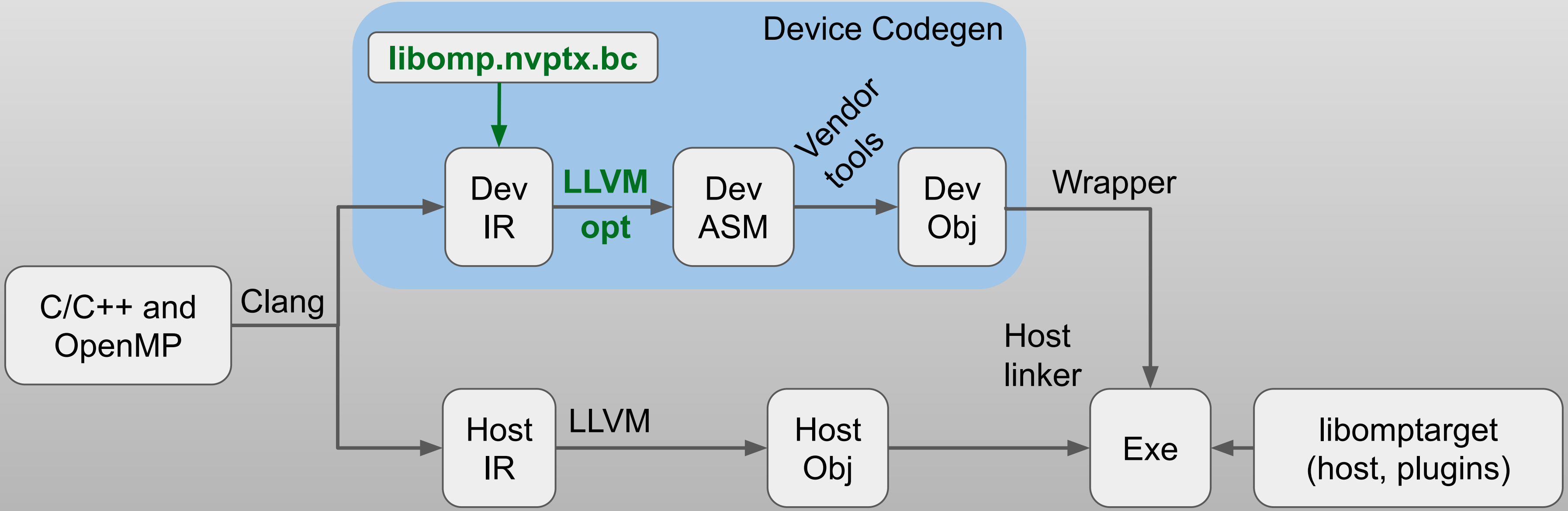
libomptarget-nvptx (offloading, device)

Re-design for optimization

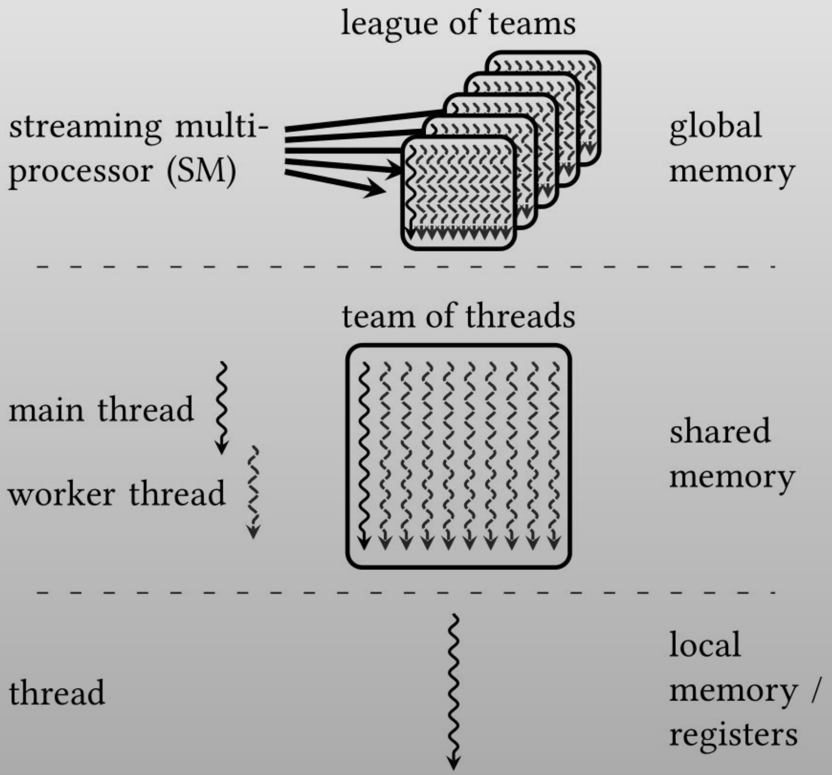
OpenMP Offload Compilation Toolchain



OpenMP Offload Compilation Toolchain



OpenMP to GPU Mapping



(Some) Motivational Problems

OpenMP Offload vs 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);



}
```

```
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);


 }
}
```

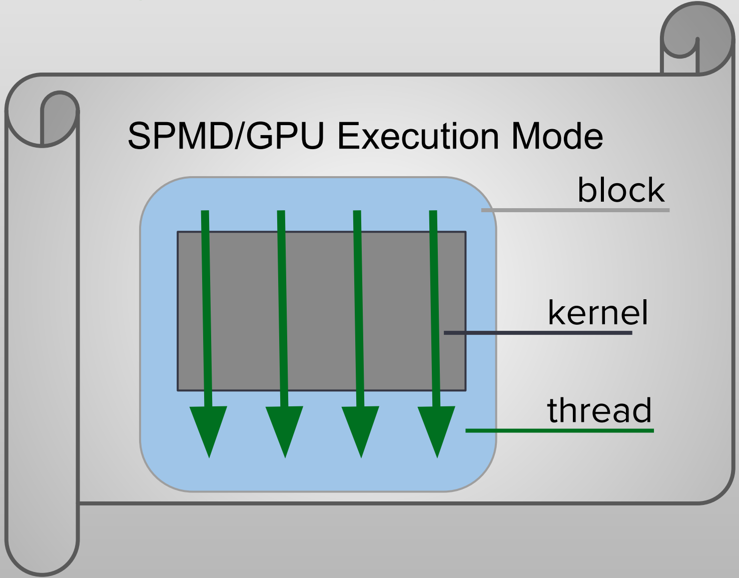
OpenMP Offload vs CUDA - Globalization of Locals

```
__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(...)
  }
}
```

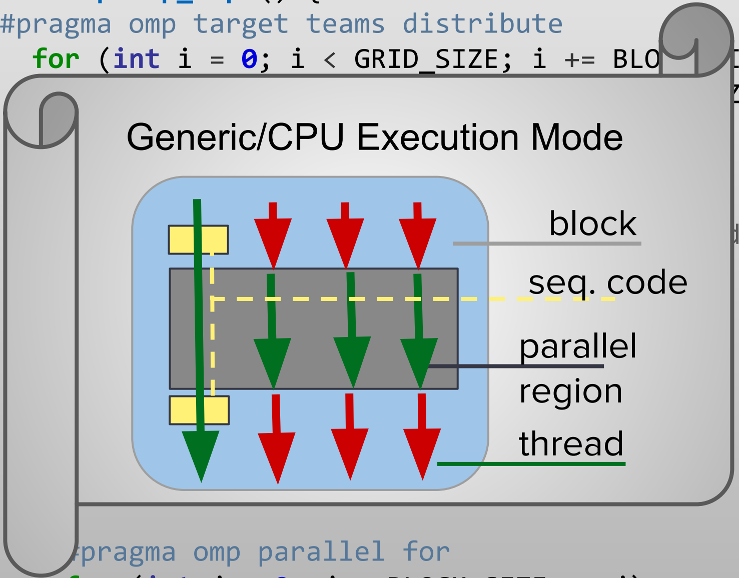
OpenMP Offload vs CUDA - Execution Mode

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



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



}
```

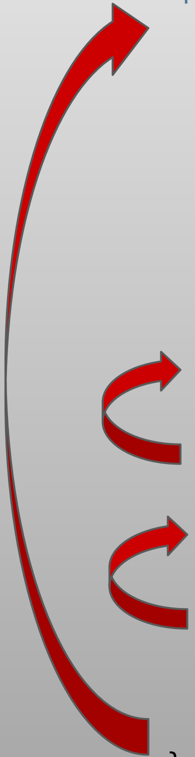
```
void openmp_impl() {
#pragma omp target teams distribute
  for (int i = 0; i < GRID_SIZE; i += BLO   IZE) {
                                             ZE);




                                             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 - Explicit Loop Backedges

```
__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
    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(...)
  }
}
```

Optimizations

OpenMP Offloading - Deglobalization

```
__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[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) {    // known true
    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) // known true
      L = load_data(Buffer, j);
    #pragma omp barrier // aligned

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

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

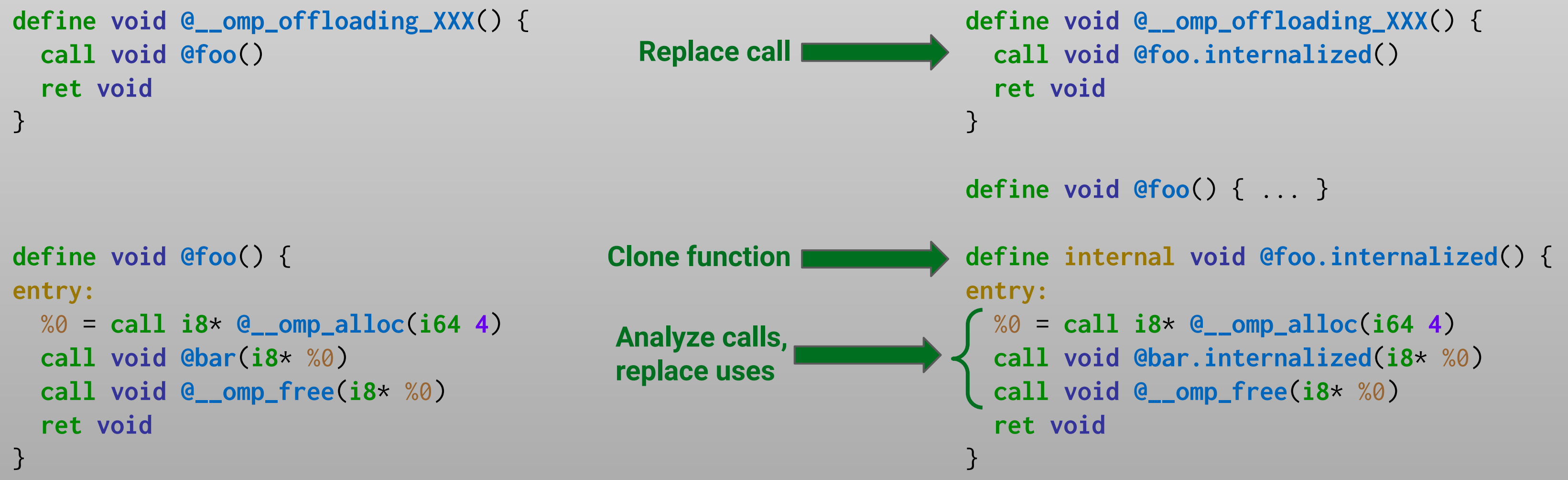
Optimization Implementation

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*

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
}
```

Runtime Co-Design

Remarks and Assumptions for 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, ...);
}
```

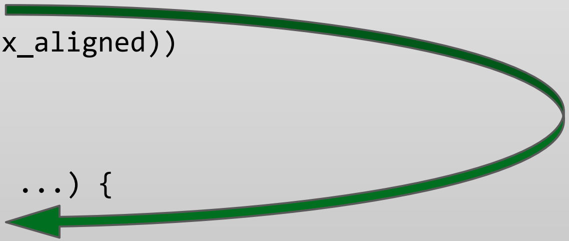
```
__device__ static void outlined_fn(...) {
  // Do not (transitively) call __omp_parallel.
  use(State.TeamSize);
}
```

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);
}
```

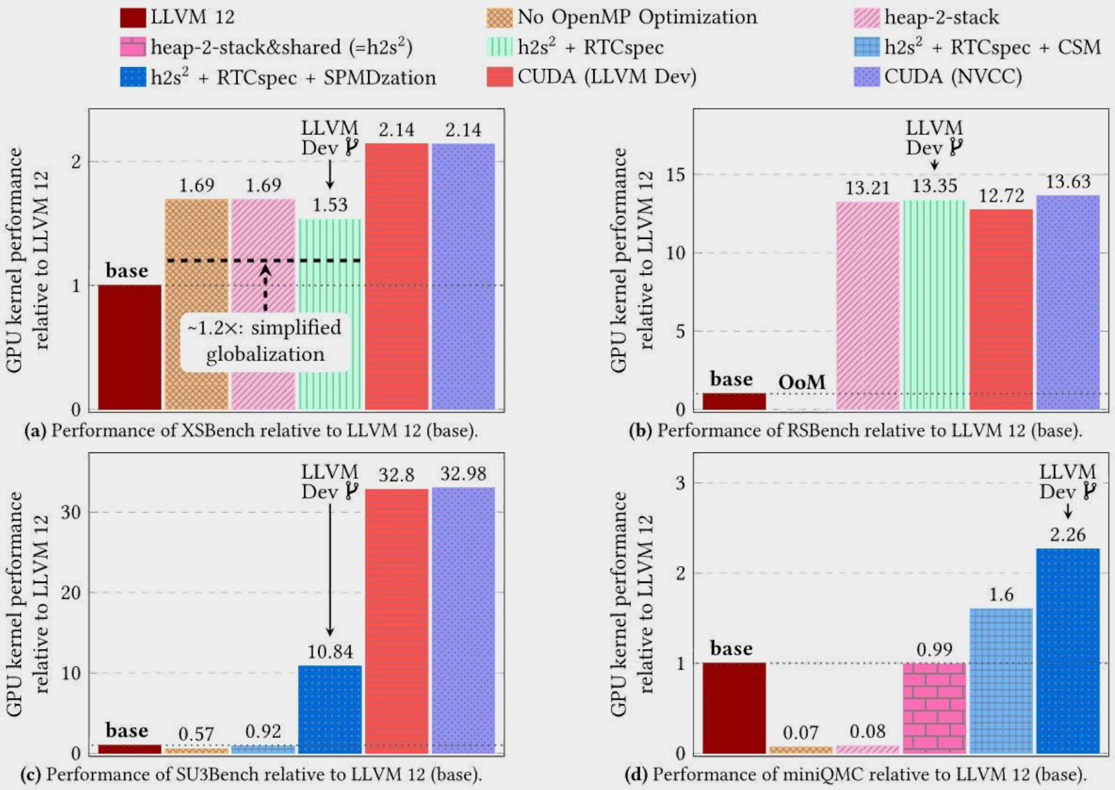
```
__global__ void kernel() {
  use(blockDim.x);
}
```

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

⇒ CUDA-like code (IR and PTX)

Evaluation

OpenMP Offloading Performance (Part I)



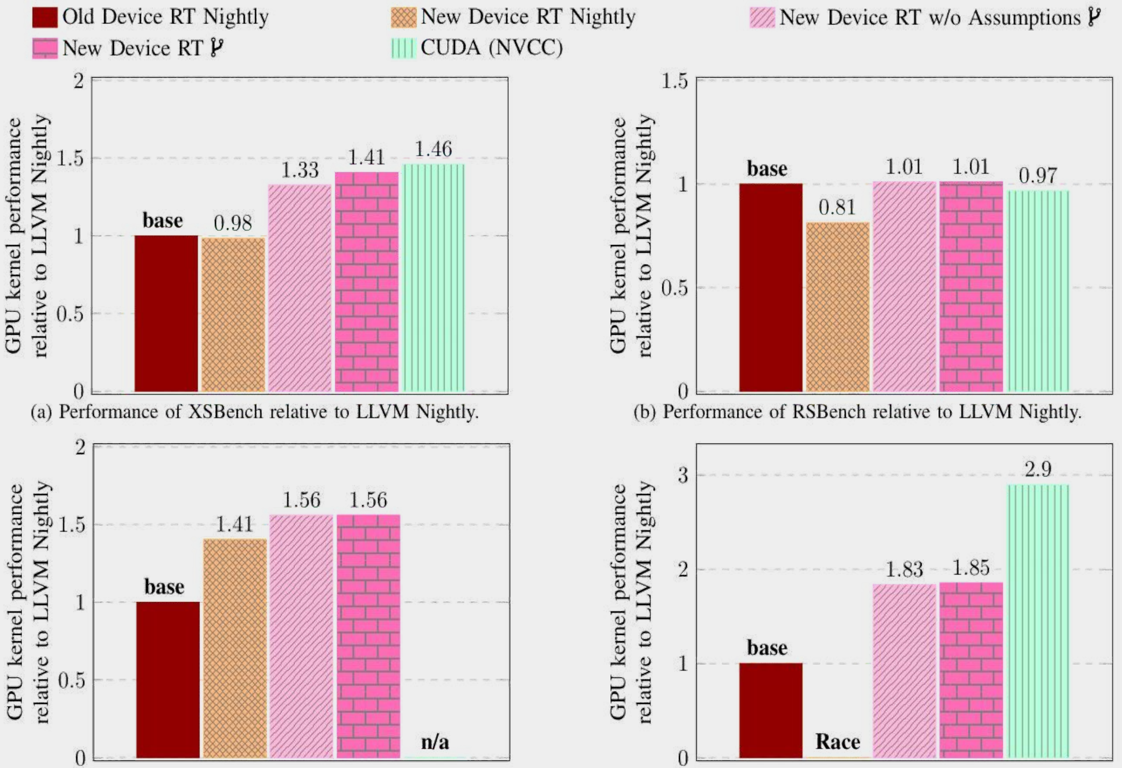
(a) Performance of XSBench relative to LLVM 12 (base).

(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)



(a) Performance of XSBench relative to LLVM Nightly.

(b) Performance of RSBench relative to LLVM Nightly.

(c) Performance of TestSNAP relative to LLVM Nightly.

(d) Performance of MiniFMM relative to LLVM Nightly.

Conclusion & Future Work

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)

Current and Future Work

- JIT and LTO for OpenMP offloading (see Lightning Talk)
- Profiling and runtime feature selection
- Heterogeneous host-device optimizations
- Generic GPU optimizations
- ...