The LLVM/OpenMP Ecosystem

Optimizations, Features, and Outlook

Johannes Doerfert <jdoerfert@anl.gov>

About Me

2018 — PhD in CS from Saarland University, Saarbrücken, Germany

Since 2018 — Researcher at Argonne National Laboratory (ANL), Chicago, USA

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

Me @ ETH

Code owner for OpenMP offloading in LLVM (officially) since 2021

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)
- **Atmn Patel (U Waterloo)**
- **Konstantinos Parasyris (LLNL)**
- **Marc Jasper (LLNL)**
- 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)
- Valentin Clement (NVIDIA)
- Many, many, more

Power Users

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

Papers & Presentations
(selection)

Papers:

SCREENSHOT ME!

- Efficient Execution of OpenMP on GPUs (CGO'22)
- Co-Designing an OpenMP GPU Runtime Optimizations for Near-Zero Overhead Execution (IPDPS'22)
- Remote OpenMP Offloading (ISC'22, **best paper**)
- A Virtual GPU as Developer-Friendly OpenMP Offload Target (LLPP'21)
- Advancing OpenMP Offload Debugging Capabilities in LLVM (LLPP'21)
- Experience Report: Writing A Portable GPU Runtime with OpenMP 5.1 (IWOMP'21)
- Compiler Optimizations For Parallel Programs (LCPC'18)

Presentations:

- LTO and JIT Support in LLVM OpenMP Target Offloading
  https://www.youtube.com/watch?v=T43nhJNSGHg
- A Compiler's View of OpenMP
  https://www.openmp.org/events/webinar-a-compilers-view-of-the-openmp-api/

OpenMP in LLVM

https://openmp.llvm.org/docs

Slide originally presented at LLVM-Dev Meeting 2020

https://youtu.be/M0DrhQbjrro

OpenMP in LLVM

https://openmp.llvm.org/docs

Slide originally presented at LLVM-Dev Meeting 2020

https://youtu.be/M0DrhQbjrro

OpenMP in LLVM

https://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

https://openmp.llvm.org/docs

Clang

OpenMP Parser

OpenMP Sema

OpenMP CodeGen

OpenMP runtimes

libomp.so (classic, host)

libomptarget + plugins (offloading, host)

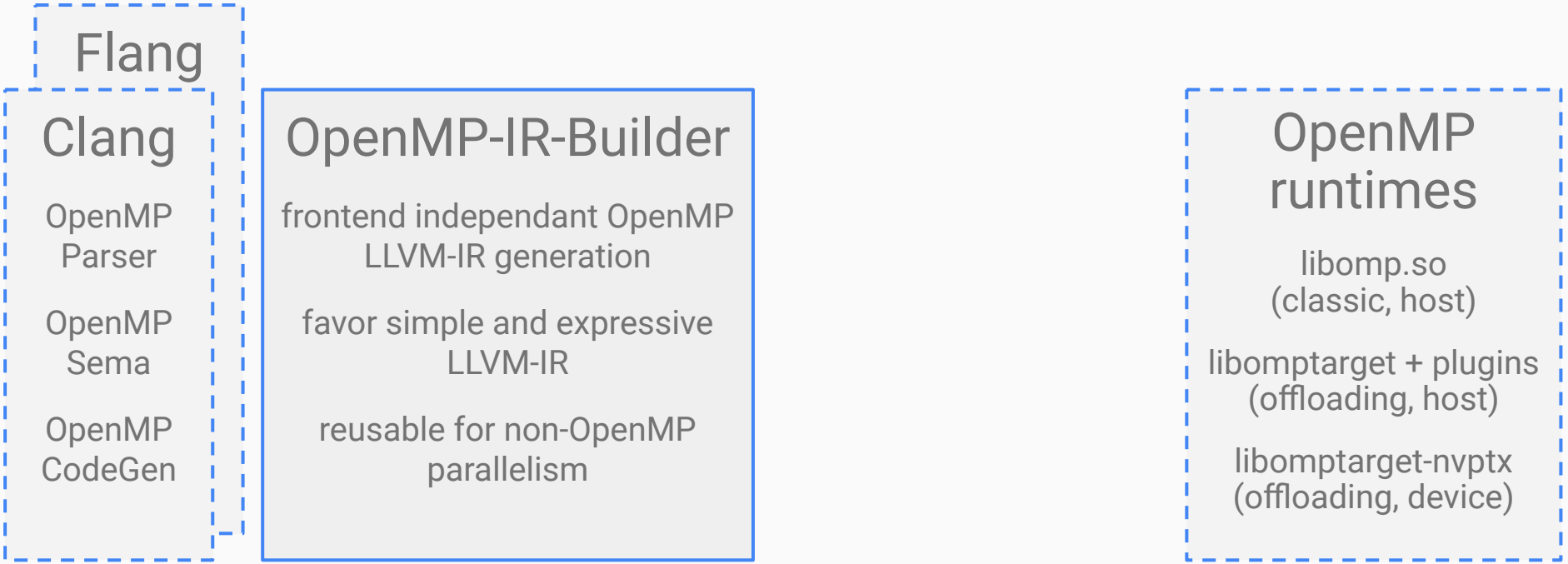
libomptarget-nvptx (offloading, device)

Slide originally presented at LLVM-Dev Meeting 2020

https://youtu.be/M0DrhQbjrro

OpenMP in LLVM

https://openmp.llvm.org/docs



Slide originally presented at LLVM-Dev Meeting 2020

https://youtu.be/M0DrhQbjrro

OpenMP in LLVM

https://openmp.llvm.org/docs

Flang

Clang

OpenMP Parser

OpenMP Sema

OpenMP CodeGen

OpenMP-IR-Builder

frontend independant OpenMP LLVM-IR generation

favor simple and expressive LLVM-IR

reusable for non-OpenMP parallelism

OpenMP-Opt

interprocedural optimization pass

contains host & device optimizations

run with -O1 and higher 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

(Some) Initial Problems

Or, how we started.

- Debugging Challenges
- Indirection Overheads
- GPU Performance

Debugging Challenges

Improved OpenMP Offload Error Diagnostic

```
$ clang++ -fopenmp -fopenmp-targets=nvptx64 -O3 -gline-tables-only sum.cpp -o sum
$ ./sum
CUDA error: an illegal memory access was encountered
Libomptarget error: Copying data from device failed.
Libomptarget error: Call to targetDataEnd failed, abort target.
Libomptarget error: Failed to process data after launching the kernel.
Libomptarget error: Consult https://openmp.llvm.org/design/Runtimes.html for debugging options.
sum.cpp:5:1: Libomptarget error 1: failure of target construct while offloading is mandatory
```

See: *Advancing OpenMP Offload Debugging Capabilities in LLVM* (LLPP'21)

Improved OpenMP GPU Runtime Information

```
$ clang++ -fopenmp -fopenmp-targets=nvptx64 -O3 -gline-tables-only sum.cpp -o sum
$ env LIBOMPTARGET_INFO=$((0x1 | 0x10 | 0x20)) ./sum
Entering OpenMP kernel at sum.cpp:5:1 with 3 arguments:
      firstprivate(N)[8] (implicit)
      tofrom(sum)[8] (implicit)
      to(A[:N])[8192]
Copying data from host to device, Size=8, Name=sum
Copying data from host to device, Size=8192, Name=A[:N]
Launching kernel __omp_offloading_fd02_60a38a2f__Z3sumPdm_l5 with 1 blocks and 128 threads in SPMD mode
```

See: *Advancing OpenMP Offload Debugging Capabilities in LLVM* (LLPP'21)

Improved OpenMP GPU Runtime Checks

```
$ clang++ -fopenmp -fopenmp-targets=nvptx64 -fopenmp-target-debug=0x5 sum.cpp -o sum
$ env LIBOMPTARGET_DEVICE_RTL_DEBUG=0x5 ./sum
Shared memory stack full, fallback to dynamic allocation of global memory will negatively impact performance.
nullptr returned by malloc!
CUDA error: an illegal memory access was encountered
```

See: *Advancing OpenMP Offload Debugging Capabilities in LLVM* (LLPP'21)

Indirection Overheads

Compiler Optimizations

Original Program

```
int y = 7;

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

After Optimizations

Compiler Optimizations

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);
```

Compiler Optimizations For Parallel Programs

Original Program

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

After Optimizations

Compiler Optimizations For Parallel Programs?

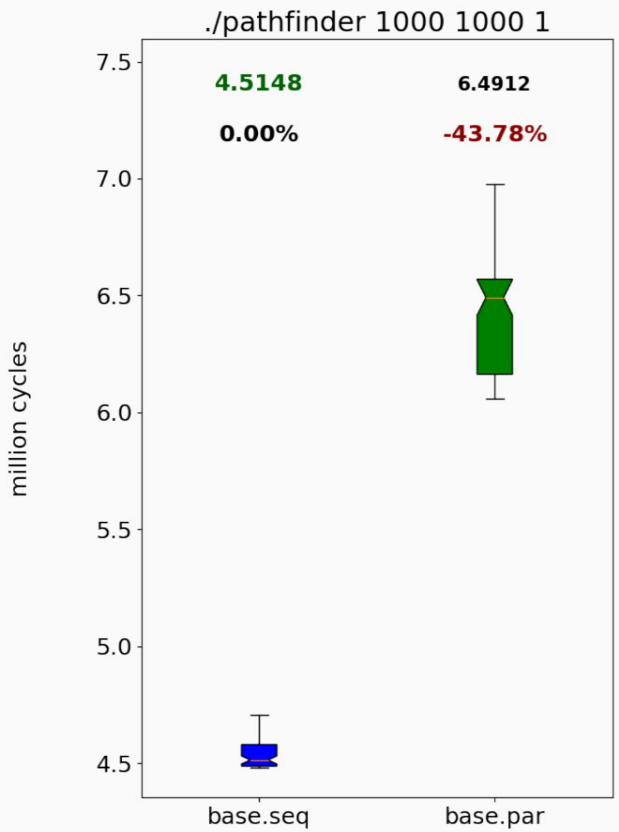
Original Program

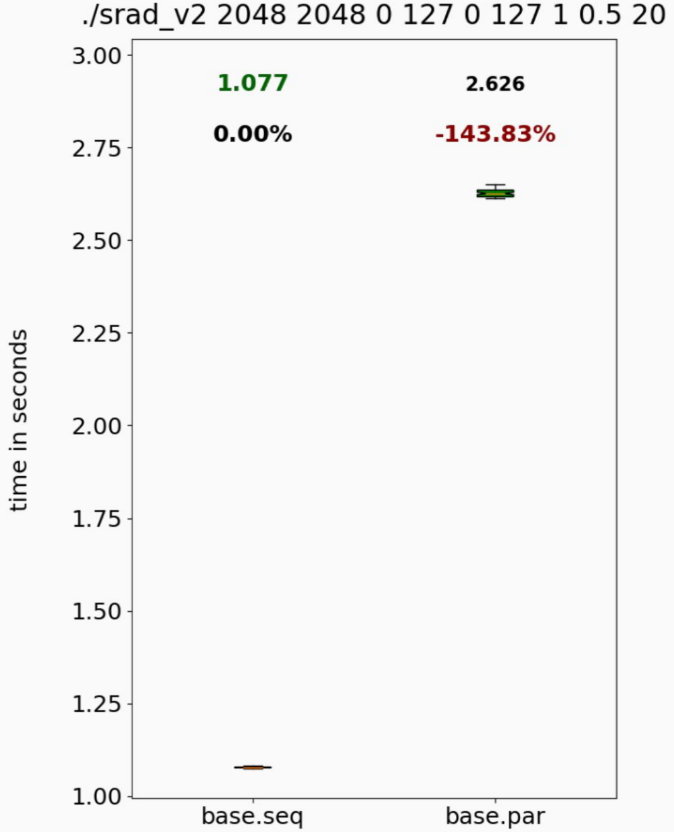
```
int y = 7;
#pragma omp parallel for
for (i = 0; i < N; i++) {
  f(y, i);
}
g(y);
```

After Optimizations

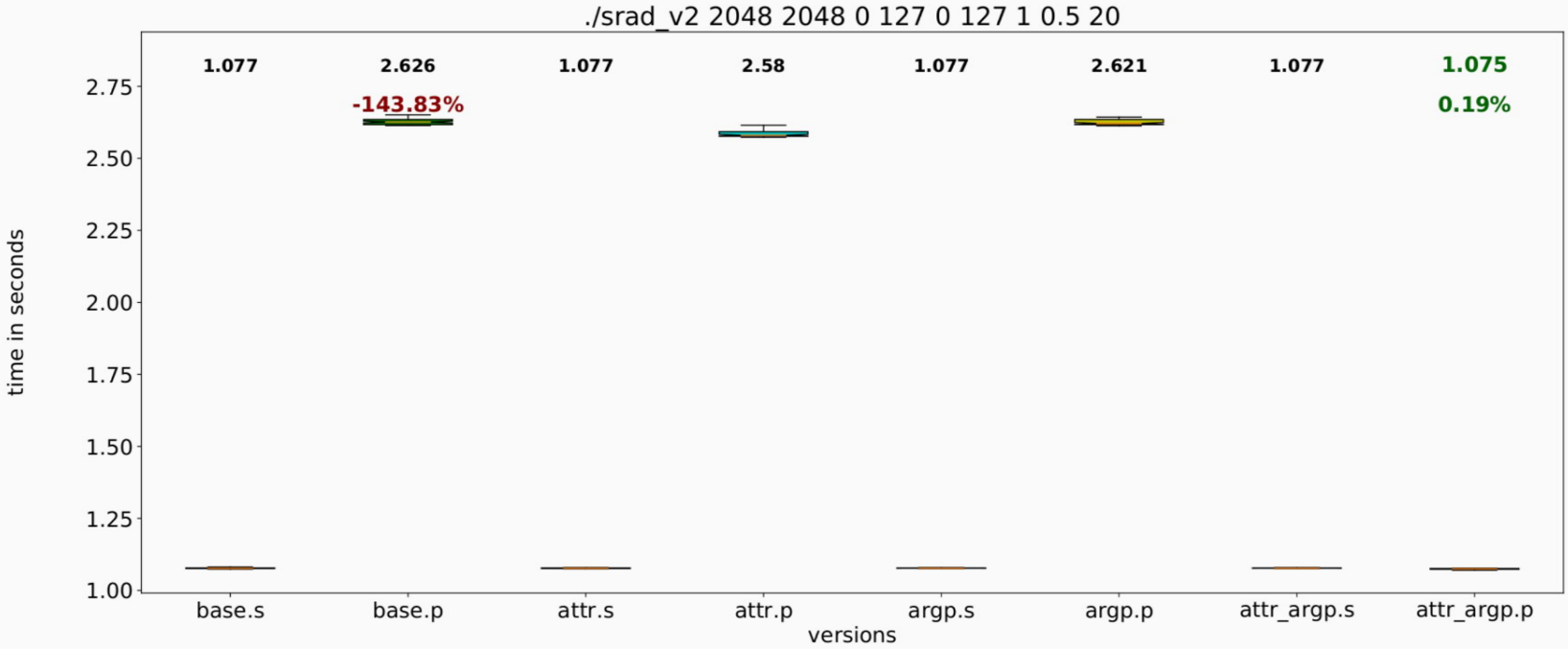
```
int y = 7;
#pragma omp parallel for
for (i = 0; i < N; i++) {
  f(y, i);
}
g(y);
```

Compiler Optimizations For Parallel Programs?





Compiler Optimizations For Parallel Programs?



See: *Compiler Optimizations For OpenMP* (IWOMP'18)

CPU vs GPU Execution Model

CUDA vs OpenMP Offload

```
__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);


}
```

CUDA vs OpenMP Offload

```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 num_threads(...)
    for (int j = 0; j < BLOCK_SIZE; ++j)
      L = load_data(Buffer, j);
    // Synchronization is implicit

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


  }
}
```

CUDA vs OpenMP Offload — Globalization of Locals

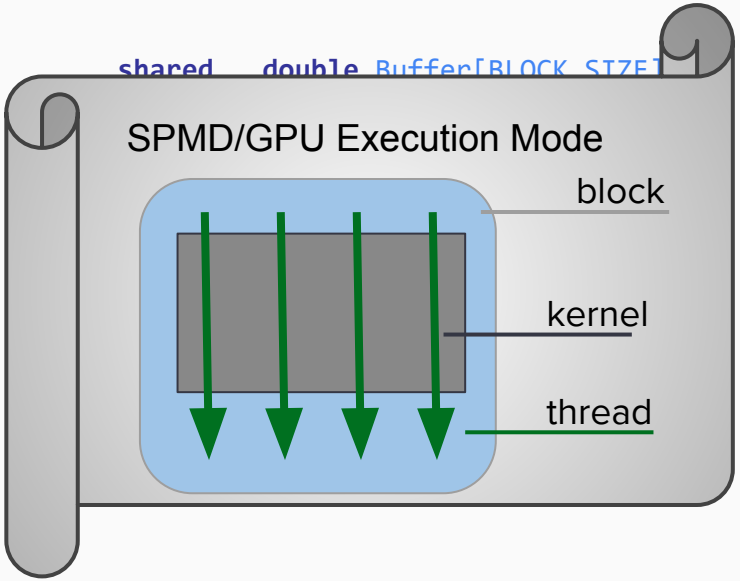
```
__global__ void cuda() {


  __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();
                                        one thread

                                    threads(...) shared(L)
                                    E; ++j)

    // Synchronization is implicit

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

    __omp_free(...)
  }
}
```

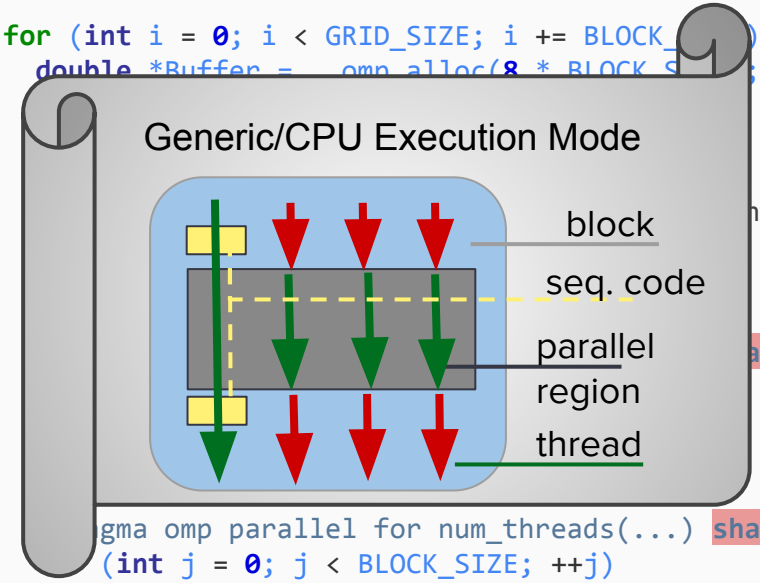
Local/Stack variables cannot be shared across GPU threads.

CUDA vs OpenMP Offload — Execution Mode Mismatch

```
__global__ void cuda() {
  __shared__ double Buffer[BLOCK_SIZE];

  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);

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

    __omp_free(...)
  }
}
```



CUDA vs OpenMP Offload — Globalization of Locals

```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 num_threads(...) shared(L)
    for (int j = 0; j < BLOCK_SIZE; ++j)
      *L = load_data(Buffer, j);
    // Synchronization is implicit

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

    __omp_free(...)
  }
}
```

OpenMP-Opt — Deglobalization

See: CGO'22

```
__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-Opt — “SPMDzation”

See: CGO’22

```
__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-Opt — Loop Oversubscription (User Assumption)

See: CGO'22

```
__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-Opt — Aligned Barrier Removal

See: CGO'22

```
__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-Opt — Runtime Call Specialization

See: CGO'22

```
__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
    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);

  }
}
```

num_teams(GRID_SIZE)
thread_limit(BLOCK_SIZE)

OpenMP-Opt — CUDA-like Kernel Code

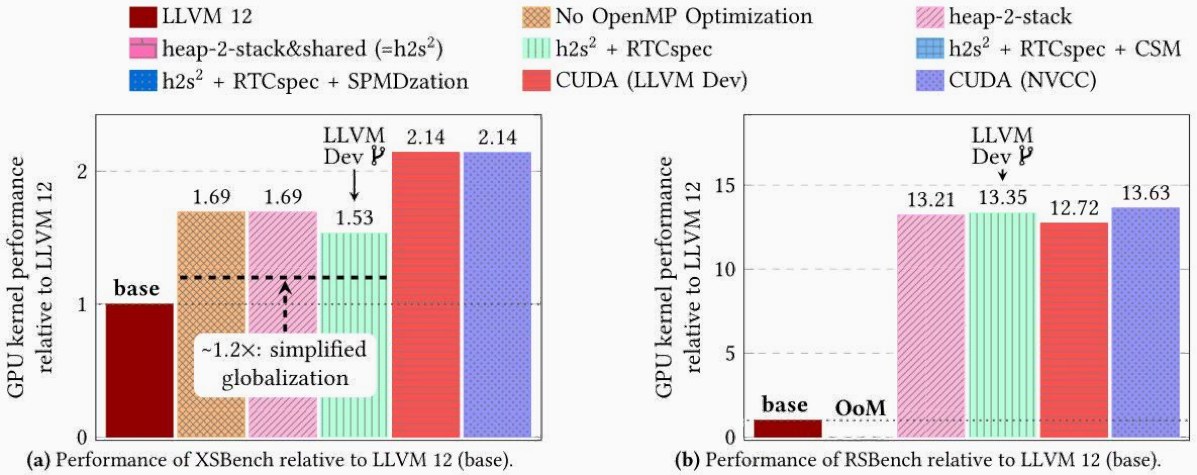
See: CGO'22

```
__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

  {
    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



    L = load_data(Buffer, j);
    #pragma omp barrier // aligned



    if (L != 0)
      parallel_work(Buffer, j);

  }
}
```

Optimized OpenMP Offload Performance

See: CGO'22

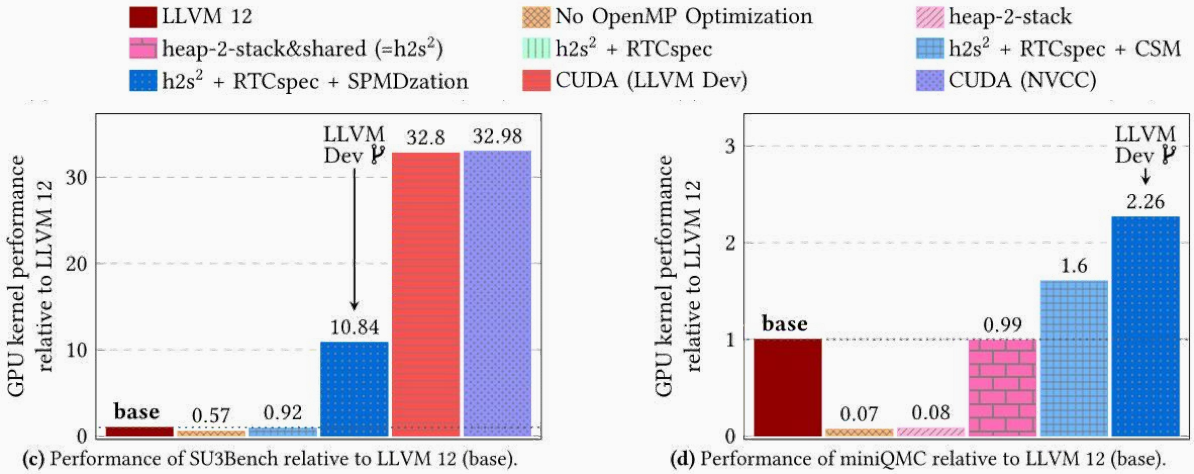


(a) Performance of XSBench relative to LLVM 12 (base).

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

Optimized OpenMP Offload Performance

See: CGO'22



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

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

Co-Designing Opt. & Portable GPU Runtime

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, ...);
}
```

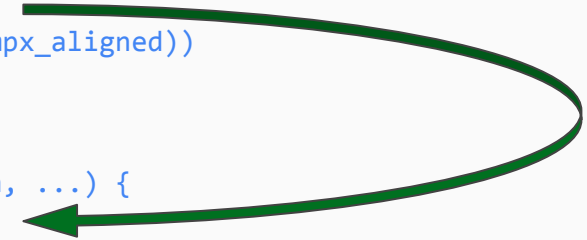
```
__device__ static void outlined_fn(...) {
  // Do not (transitively) call __omp_parallel.
  use(State.TeamSize);
}
```

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

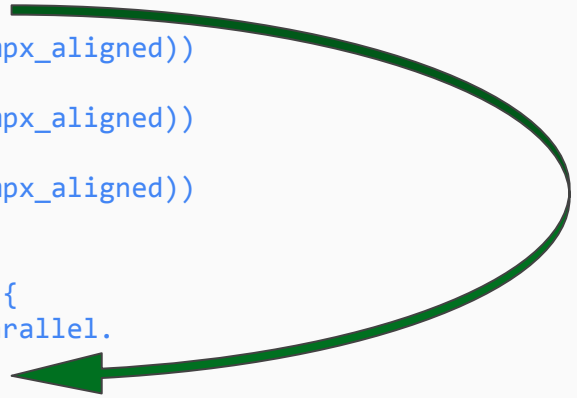
```
__shared__ StateTy State;

__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

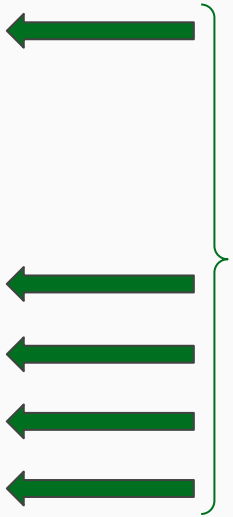
```
__shared__ StateTy State;

__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)

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`

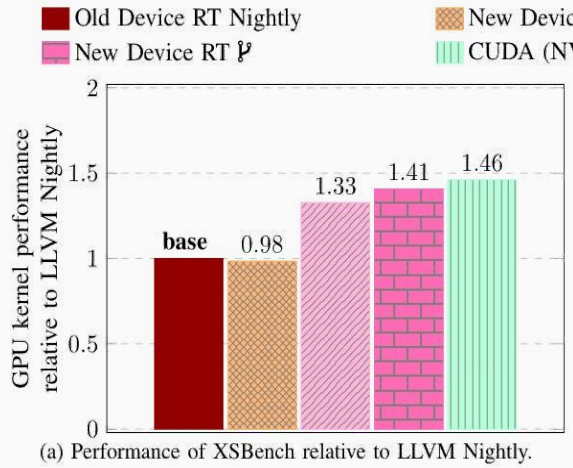
} LLVM assumption extensions

`-fopenmp-assume-teams-oversubscription`

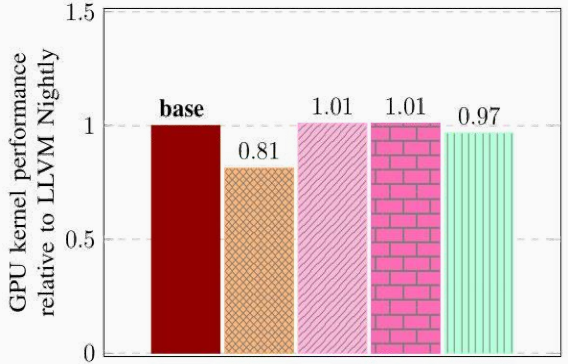
`-fopenmp-assume-threads-oversubscription`

Optimized OpenMP Offload Performance

See: IPDPS'22



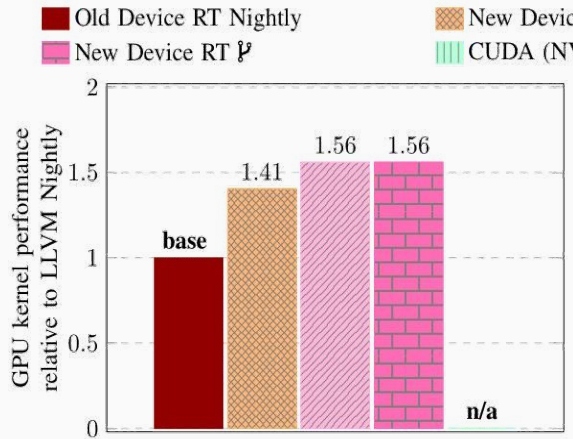
(a) Performance of XSBench relative to LLVM Nightly.



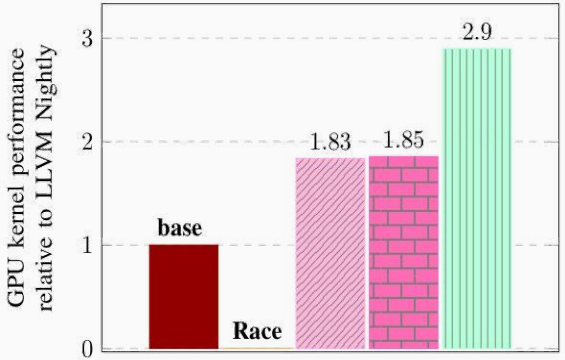
(b) Performance of RSBench relative to LLVM Nightly.

Optimized OpenMP Offload Performance

See: IPDPS'22



(c) Performance of TestSNAP relative to LLVM Nightly.



(d) Performance of MiniFMM relative to LLVM Nightly.

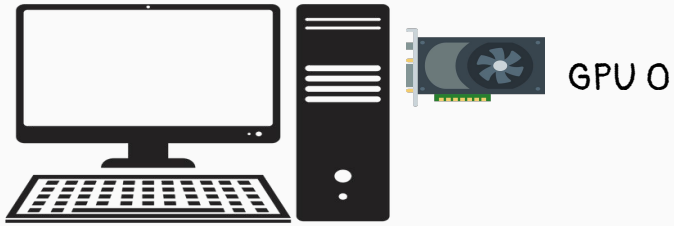
Beyond “Standard” OpenMP

Or, playing the long game.

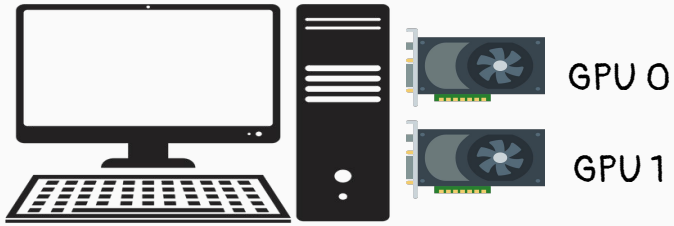
- Remote OpenMP Offloading
- OpenMP as Intermediate Layer

Remote OpenMP Offloading

Remote OpenMP Offloading (Plugin)

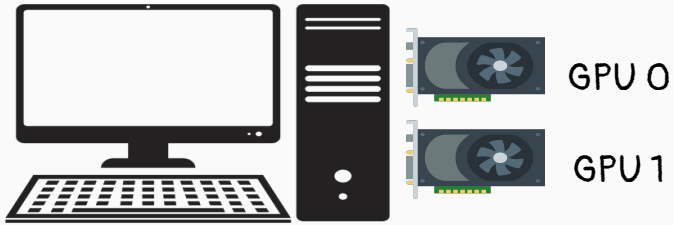


Remote OpenMP Offloading (Plugin)

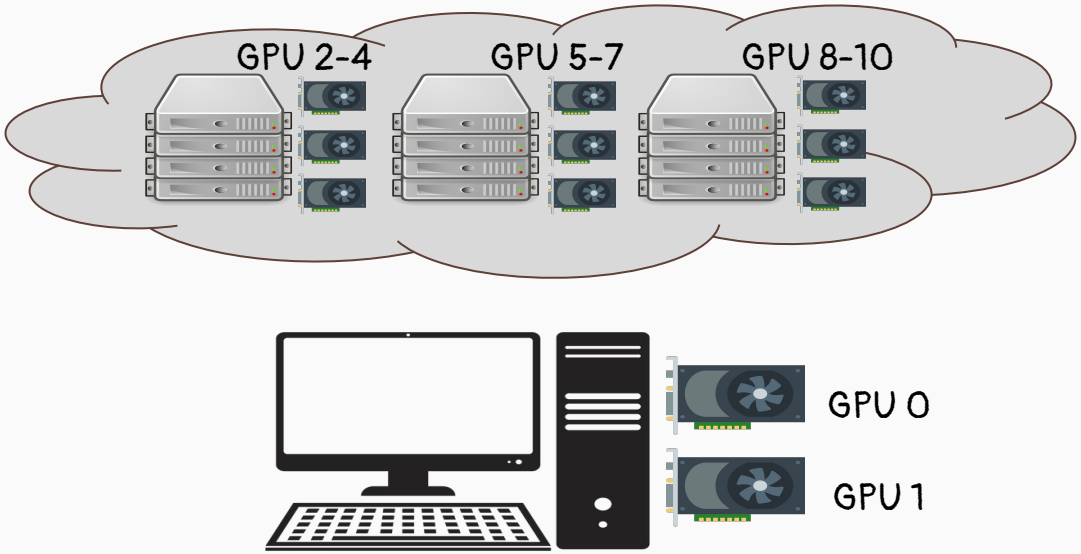


Remote OpenMP Offloading (Plugin)

```
#pragma omp parallel for num_threads(num_devices)
for (auto K = 0; K < num_devices; K++) {
    #pragma omp target ... device(K)
    for (auto i = 0; i < lookups_per_device; i++) {

        ...

    }
}
```

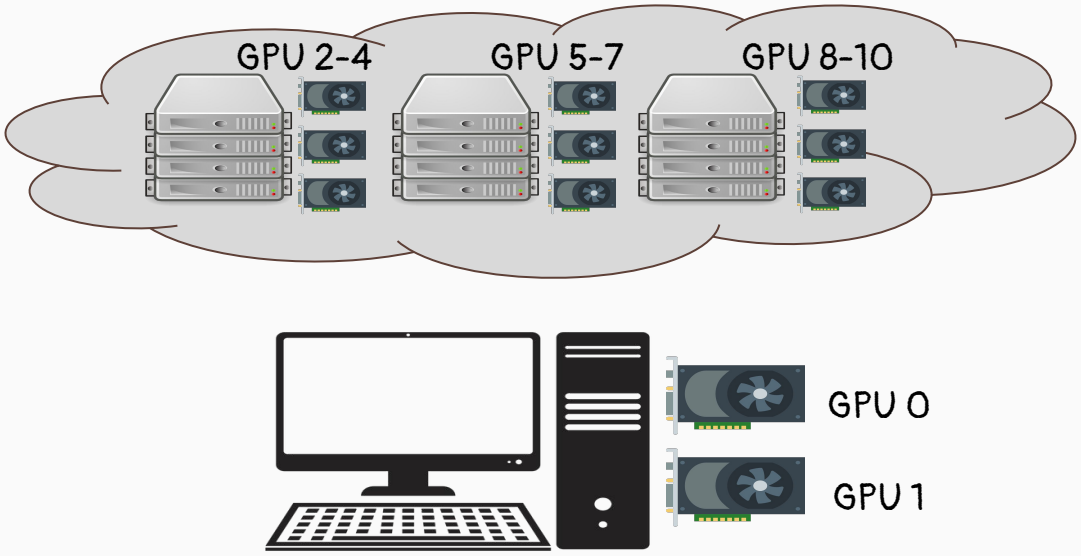


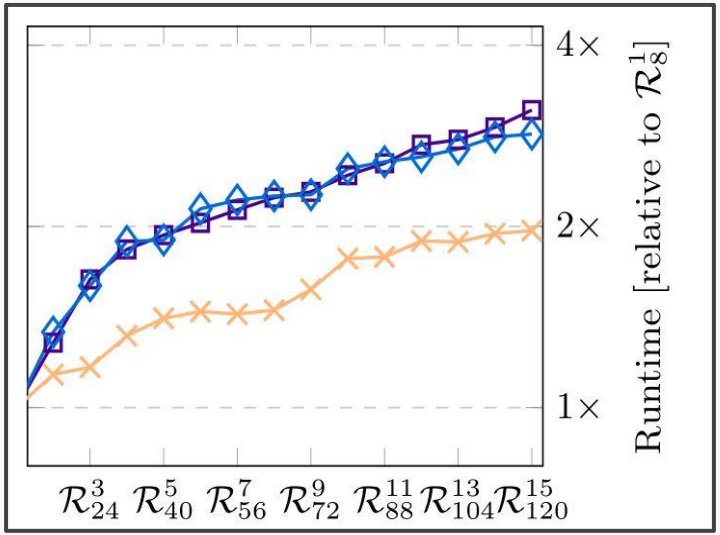
Remote OpenMP Offloading (Plugin)



See: *Remote OpenMP Offloading* (ISC'22, **best paper**)

Remote OpenMP Offloading (Plugin)





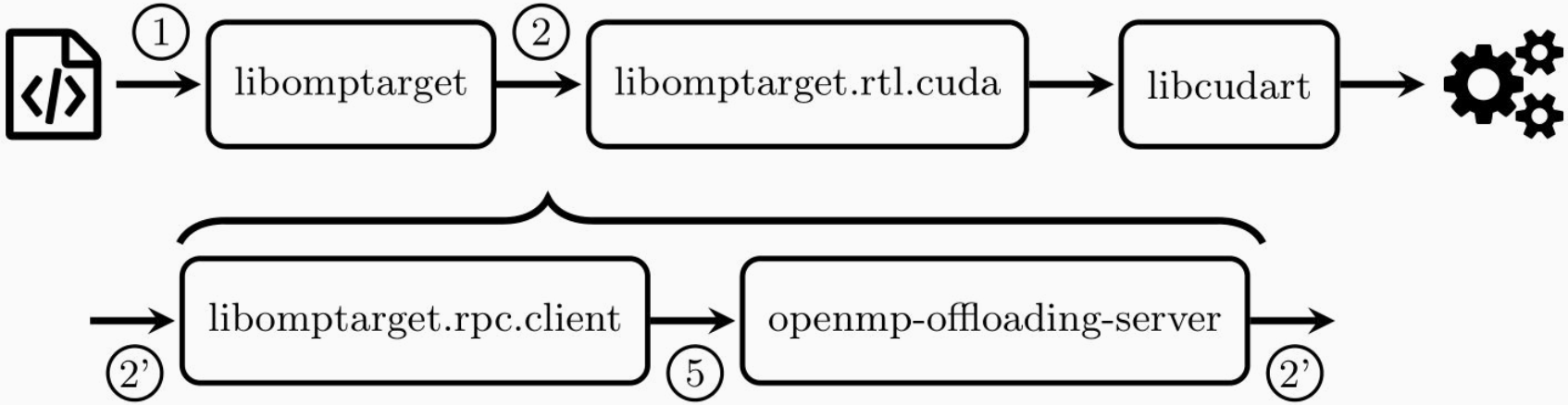
XSBench - Strong scaling - up to 15x8 A100 GPUs compared to 1x8

See: *Remote OpenMP Offloading* (ISC'22, **best paper**)

Remote OpenMP Offloading (Plugin)

See: *Remote OpenMP Offloading* (ISC'22, **best paper**)

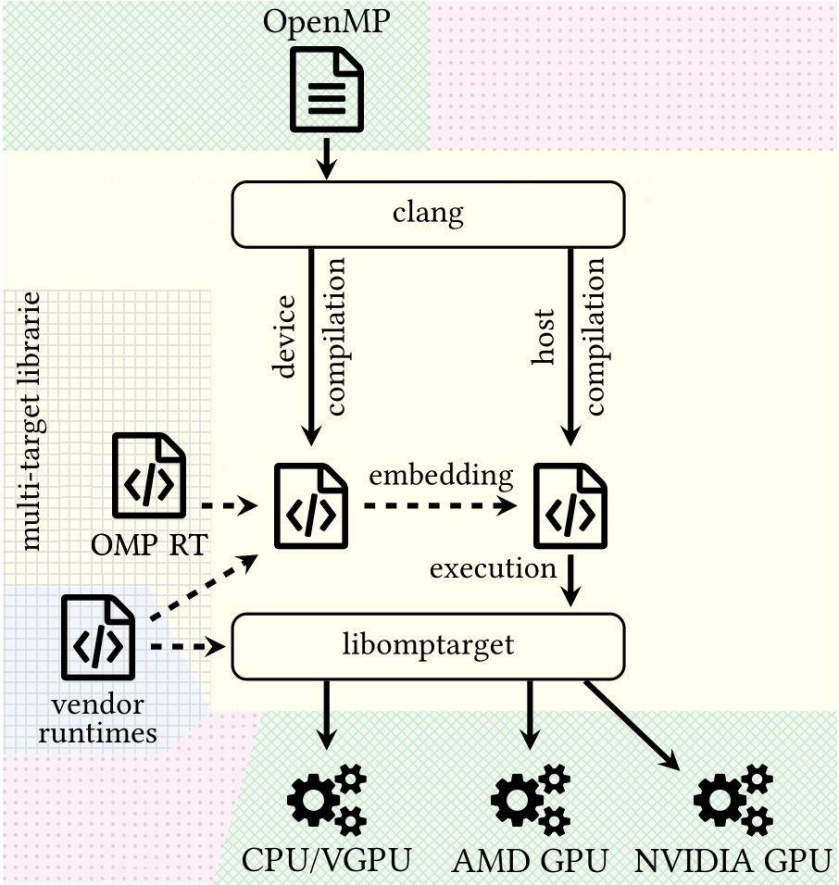
Remote OpenMP Offloading (Plugin)



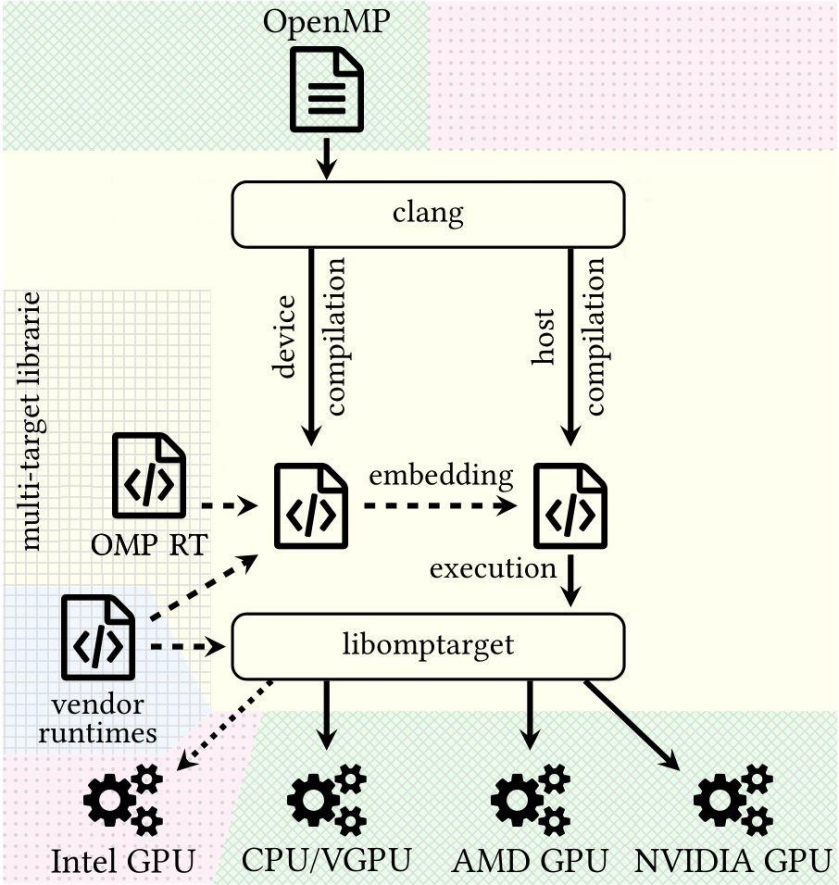
See: *Remote OpenMP Offloading* (ISC'22, **best paper**)

OpenMP as Intermediate Layer

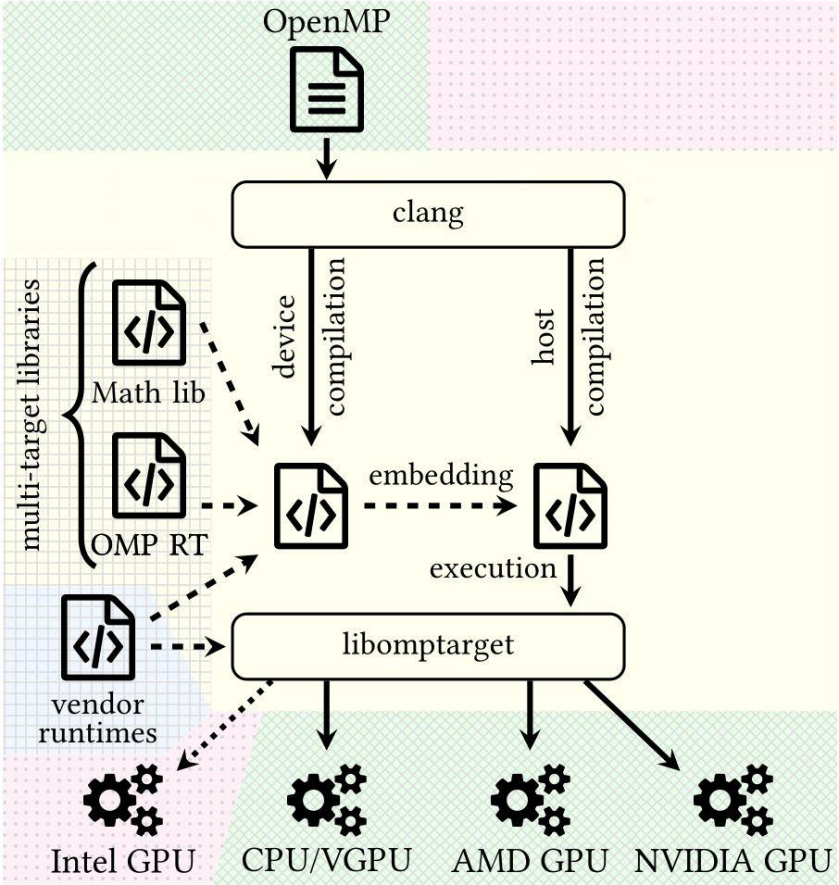
LLVM/OpenMP Target Offloading



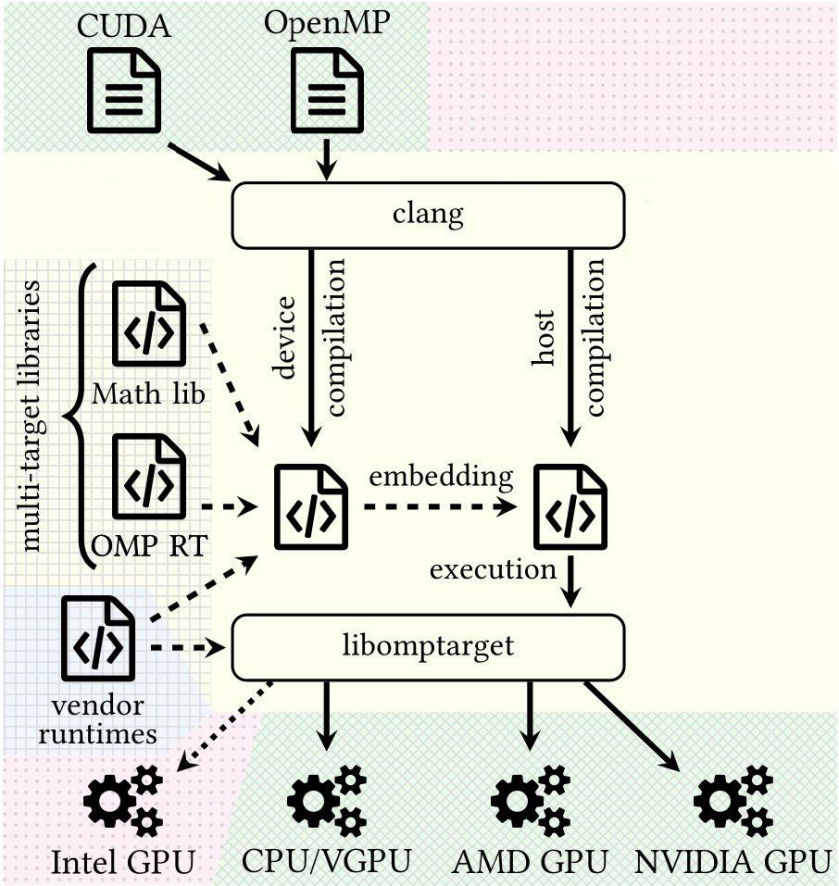
LLVM/OpenMP Target Offloading



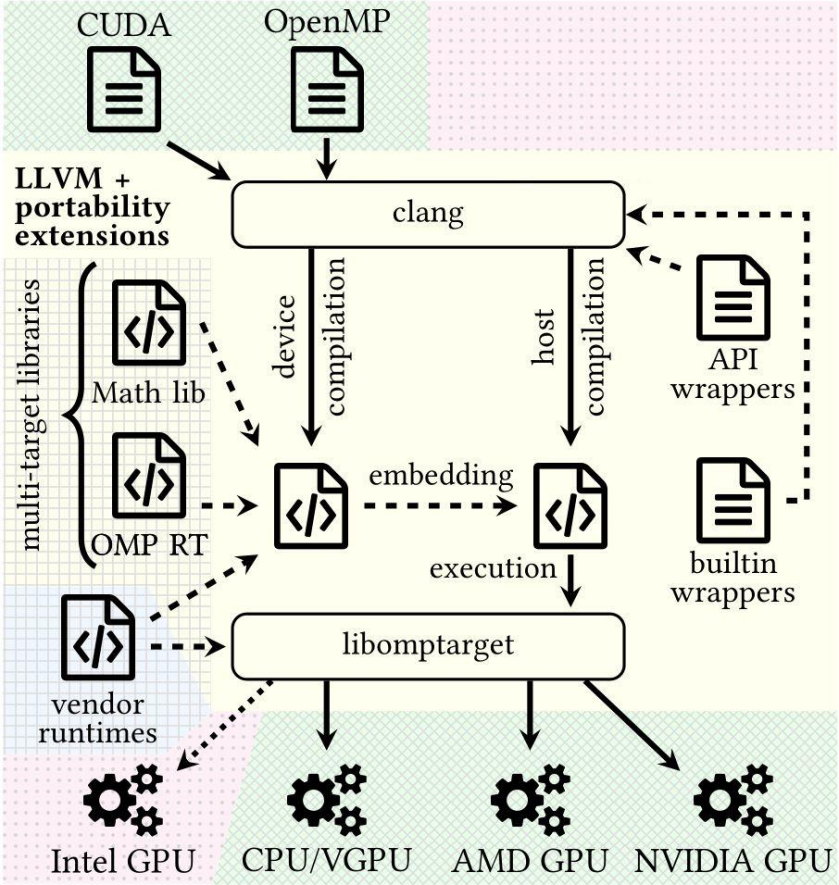
LLVM/OpenMP Target Offloading + Math Runtimes



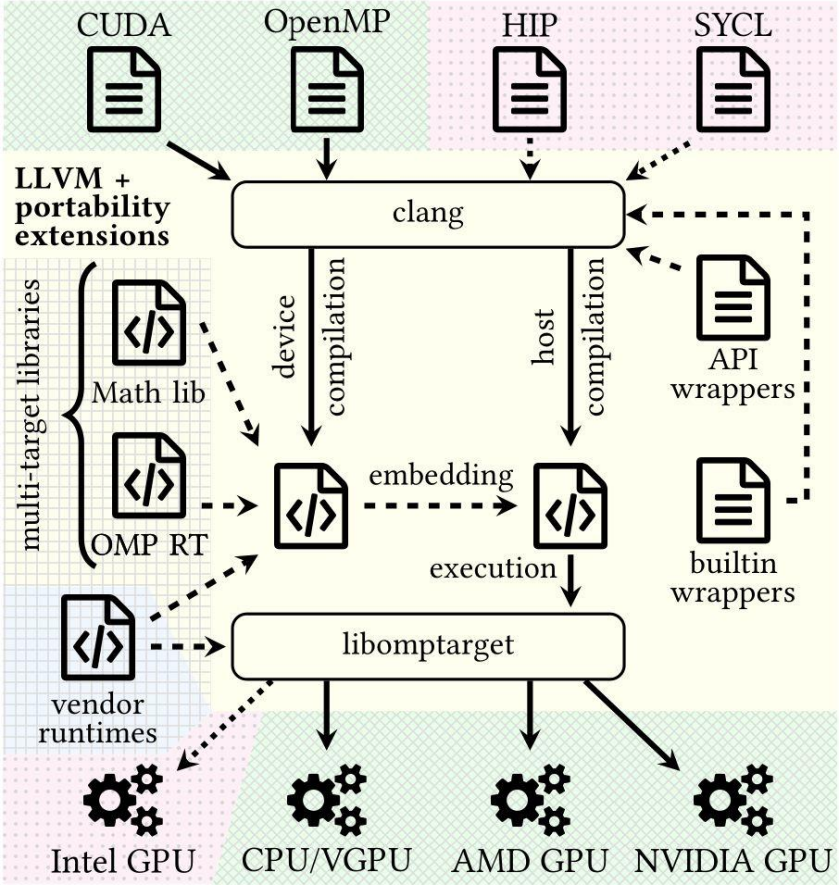
LLVM/OpenMP Target Offloading + CUDA Device Compilation



LLVM/OpenMP as Target Independent Runtime Layer (for CUDA)



LLVM/OpenMP as Target Independent Runtime Layer (WIP)



Brief Recap & Outlook

Brief Recap

Brief Recap

Enhanced GPU Debugging & Profiling

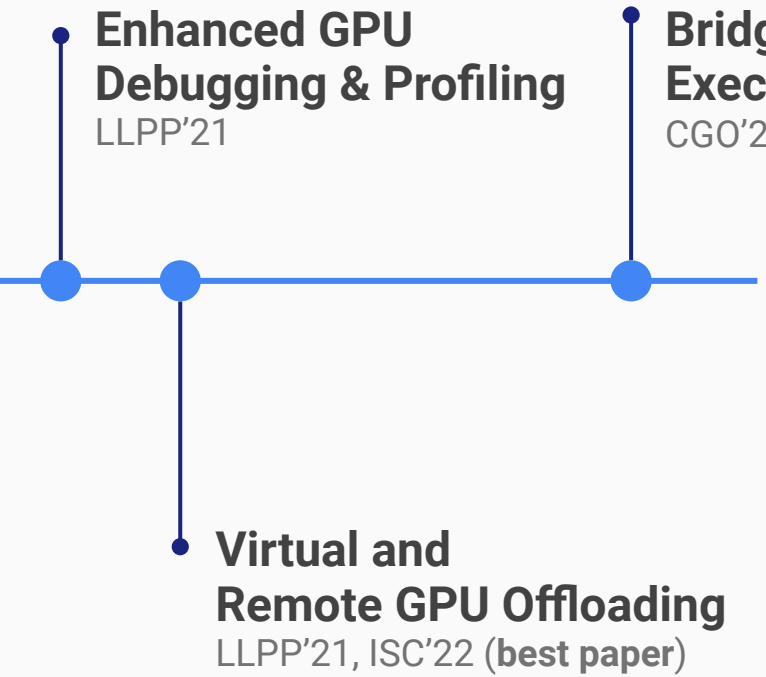
LLPP'21

Brief Recap

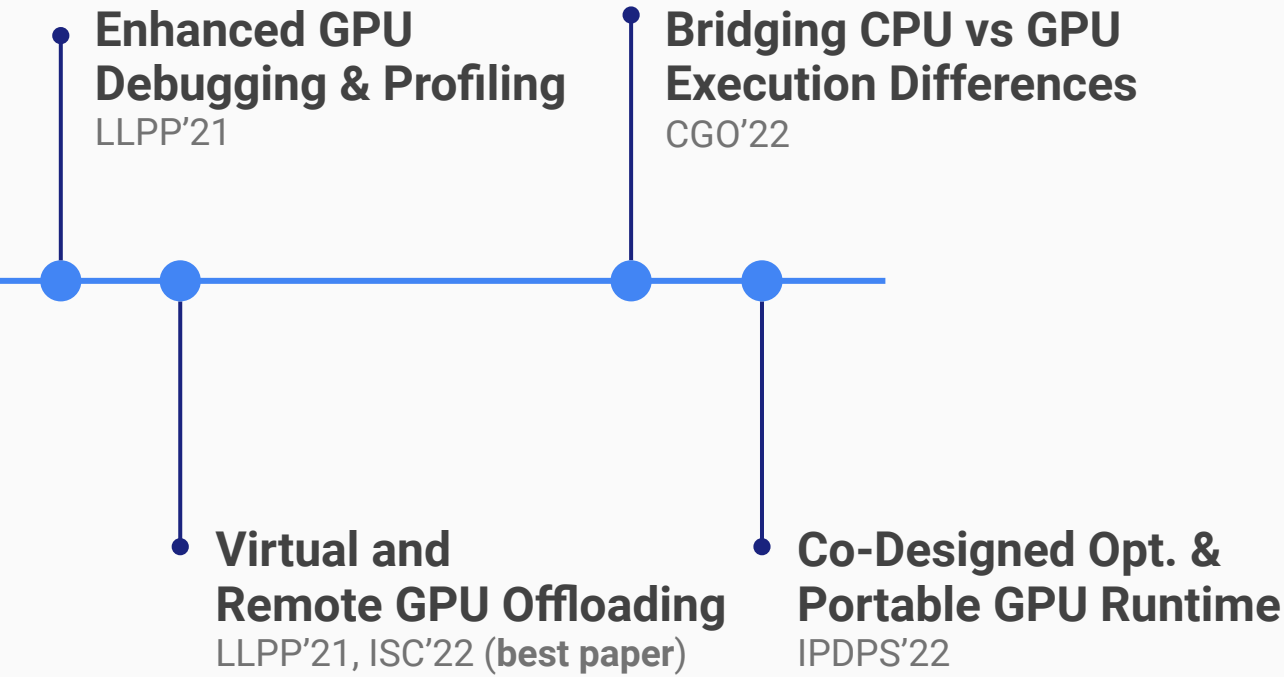
Enhanced GPU Debugging & Profiling
LLPP'21

Virtual and Remote GPU Offloading
LLPP'21, ISC'22 (**best paper**)

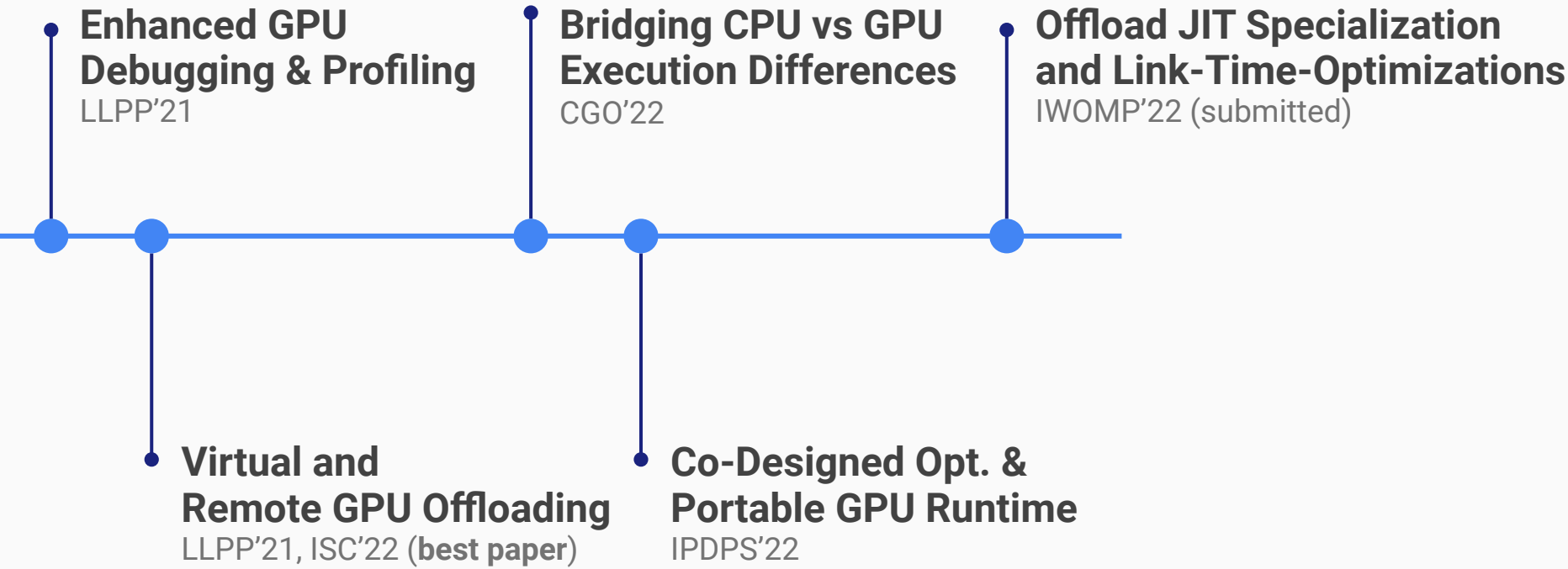
Brief Recap


Enhanced GPU Debugging & Profiling
LLPP'21
Bridging CPU vs GPU Execution Differences
CGO'22
Virtual and Remote GPU Offloading
LLPP'21, ISC'22 (best paper)

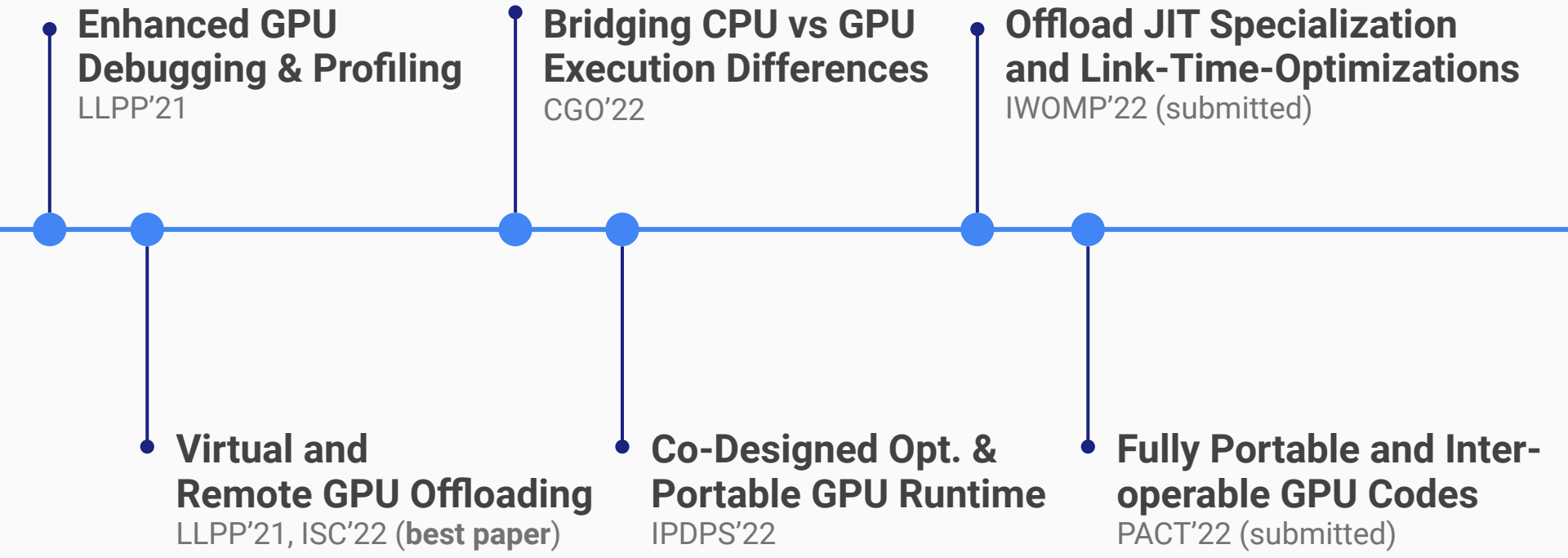
Brief Recap


Enhanced GPU Debugging & Profiling
LLPP'21
Bridging CPU vs GPU Execution Differences
CGO'22
Virtual and Remote GPU Offloading
LLPP'21, ISC'22 (best paper)
Co-Designed Opt. & Portable GPU Runtime
IPDPS'22

Brief Recap


Enhanced GPU Debugging & Profiling
LLPP'21
Bridging CPU vs GPU Execution Differences
CGO'22
Offload JIT Specialization and Link-Time-Optimizations
IWOMP'22 (submitted)
Virtual and Remote GPU Offloading
LLPP'21, ISC'22 (best paper)
Co-Designed Opt. & Portable GPU Runtime
IPDPS'22

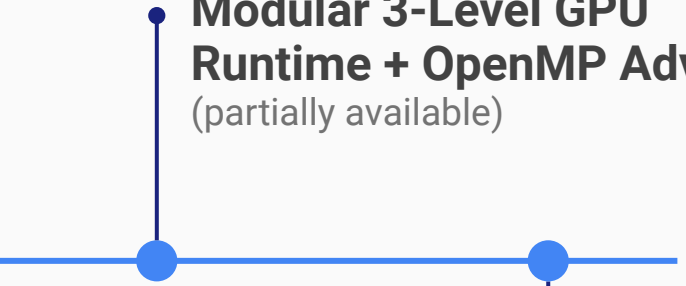
Brief Recap


Enhanced GPU Debugging & Profiling
LLPP'21
Virtual and Remote GPU Offloading
LLPP'21, ISC'22 (best paper)
Bridging CPU vs GPU Execution Differences
CGO'22
Co-Designed Opt. & Portable GPU Runtime
IPDPS'22
Offload JIT Specialization and Link-Time-Optimizations
IWOMP'22 (submitted)
Fully Portable and Inter-operable GPU Codes
PACT'22 (submitted)

Brief Outlook

Modular 3-Level GPU Runtime + OpenMP Advisor
(partially available)

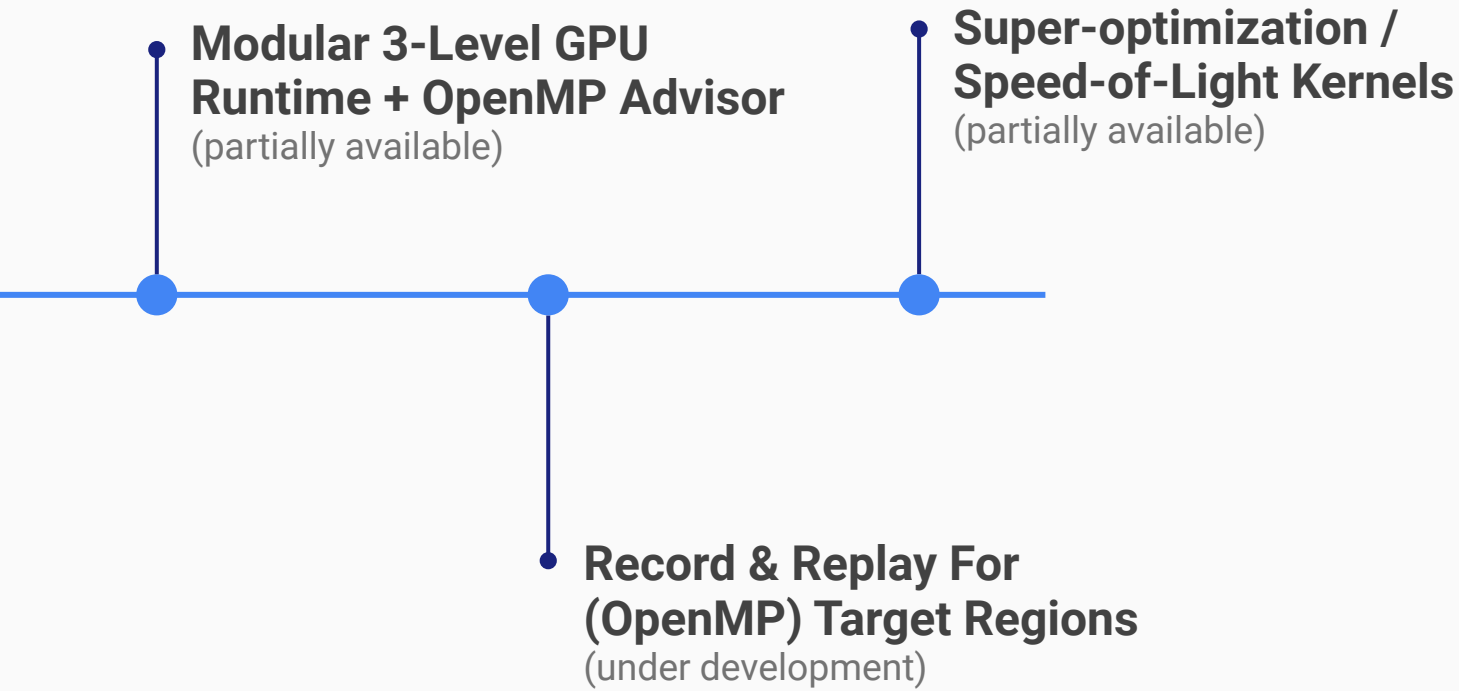
Brief Outlook



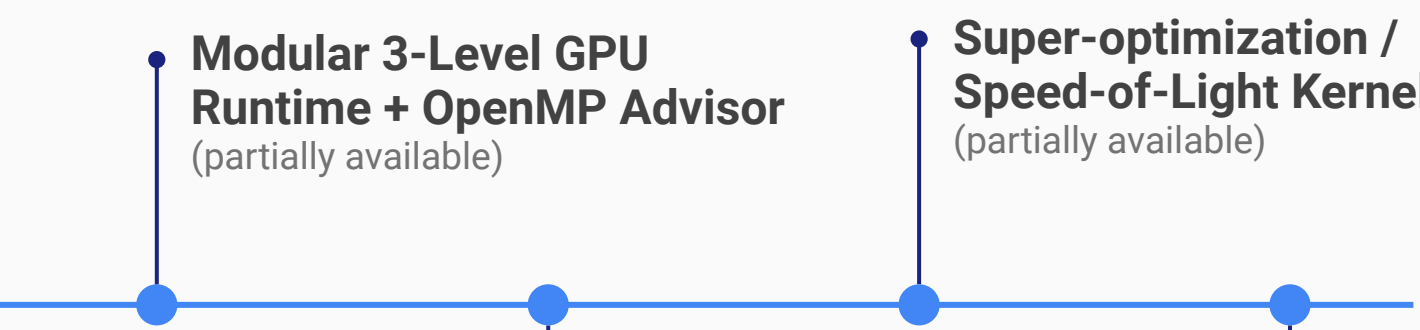
Modular 3-Level GPU Runtime + OpenMP Advisor
(partially available)

Record & Replay For (OpenMP) Target Regions
(under development)

Brief Outlook



Brief Outlook



Modular 3-Level GPU Runtime + OpenMP Advisor
(partially available)

Record & Replay For (OpenMP) Target Regions
(under development)

Super-optimization / Speed-of-Light Kernels
(partially available)

Smart Development and Runtime Environment
(Planned)

Brief Outlook