

LLVM in HPC

Enabling Performance Portability, Interoperability, and Novel Features

Johannes Doerfert <jdoerfert@llnl.gov>

LLVM/OpenMP - A Community Effort

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

“Academia”

- Shilei Tian (SBU)
- Giorgis Georgakoudis (LLNL)
- Michael Kruse (ANL)
- Joachim Protze (RWTH A.)
- Joel Denny (ORNL)
- Atmn Patel (Northwestern)
- Konstantinos Parasyris (LLNL)
- Marc Jasper (LLNL)
- Many, many, more

Industry

- Joseph Huber (AMD)
- 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

About Me

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

Since last week — Researcher at Lawrence Livermore National Laboratory (LLNL) — at ANL before

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

LLVM in a Nutshell

- open (source/community/...)
- extensible, “fixable”
- portable (GPUs, CPUs, ...)
- C++/OpenMP/SYCL/HIP/CUDA/... feature complete 😊
- early access to *the coolest* features
- performant and correct ;)
- the basis for most vendor compilers



THANKS 2 RYAN HOUDEK

[😊 eventually]

C-family frontend (compile C/C++/Cuda/...)



memory, address, UB, thread, ... sanitizers



"modern" Fortran frontend, aka F18 (WIP)



C++ standard library (replaces libstdc++)



"fast" and feature rich linker (replaces ld)



LLVM-centric debugger (replaces gdb)



"core-LLVM", middle-end & backends

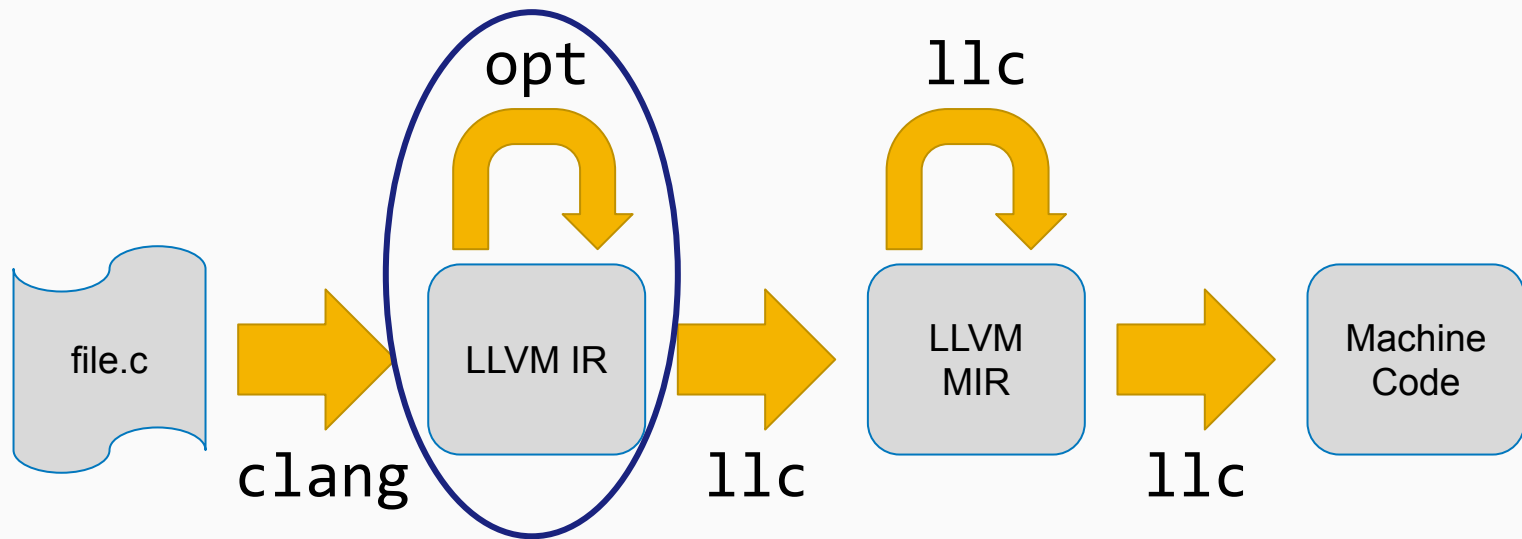


OpenMP runtimes (host, device, and plugins)



	bolt	[BOLT][NFC] Return MCRegister::NoRegister from MCPlusBuilder::get...	4 days ago
	clang-tools-extra	[clang-tools-extra] Document clang tidy unit tests target	1 hour ago
	clang	[pseudo] Rename (Preprocess,PPStructure) -> DirectiveMap. NFC	4 minutes ago
	cmake	[NFC] Fix typo in CMake comment	5 days ago
	compiler-rt	[MSAN] extend ioctl interceptor to support BLKSSZGET	3 days ago
	cross-project-tests	[Dexter] Optimize breakpoint deletion in Visual Studio	6 days ago
	flang	[flang] Update tco after 0dc66b7	4 hours ago
	libc	[libc] Fix alignment logic in TLS image size calculation.	7 hours ago
	libclc	libclc: Add clspv64 target	2 months ago
	libcxx	[libc++][ranges] Implement ranges::max_element	34 minutes ago
	libcxxabi	[demangler] Make OutputBuffer non-copyable	3 days ago
	libunwind	[runtimes] Remove FOO_TARGET_TRIPLE, FOO_SYSROOT and FOO...	6 days ago
	lld	Reland D119909 [ELF] Parallelize initializeLocalSymbols	3 days ago
	lldb	[lldb/Test] Fix test_launch_scripted_process_stack_frames failure	3 days ago
	llvm-libgcc	[llvm-libgcc] initial commit	19 days ago
	llvm	[VE] Split v512.32 load store into interleaved v256.32 ops	7 minutes ago
	mlir	[mlir] Fix dumping invalid ops	13 minutes ago
	openmp	Revert "[OpenMP][NFCI] Use RAIL lock guards in libomptarget where p...	13 hours ago
	polly	Revert "[polly] Fix regression test after D110620."	3 days ago
	pstl	Bump the trunk major version to 15	last month
	runtimes	[llvm-libgcc] initial commit	19 days ago
	test	fix check-clang-tools tests that fail due to Windows CRLF line endings	24 days ago
	third-party	Ensure newlines at the end of files (NFC)	2 months ago
	utils	[mlir][shape] Migrate bufferization to BufferizableOpInterface	4 hours ago

LLVM/Clang 101



Building LLVM

Single command often suffices to configure:

```
cmake .../llvm-project/llvm -DLLVM_ENABLE_PROJECTS='clang' -DLLVM_ENABLE_RUNTIME='openmp'  
make -j
```

Useful options include:

- CMAKE_BUILD_TYPE={Release,Asserts,...}
- LLVM_ENABLE_ASSERTIONS={ON,OFF}
- LLVM_CCACHE_BUILD={ON,OFF}
- G Ninja

May need debug build to debug certain compiler-based issues,
release + assert is often used as trade off.

Various resources available online! Start here:

<http://llvm.org/docs/GettingStarted.html>

<https://openmp.llvm.org/SupportAndFAQ.html>

Using LLVM (cheat sheet)

- Use a fast linker (`lld`), `ccache`, and `ninja`
- Consider LTO, either thin or full
- Use tooling (`clang-format`, `clang-tidy`, `clang-modernize`, ...)
- Use `-O3/Ofast -march=native` as default
- Online documentation is not great but often not bad either
- Debug with sanitizers enabled
- A release + asserts build is best for every-day use

Ask the Community

Many ways to interact:

- Discourse (forum/mailling list)
- Discord (persistent chat)
- IRC (non-persistent chat)
- Online Sync-Ups:
 - AA, MLIR, ML, OpenMP, RISC-V, ...
- Office Hours ***NEW***
 - “AMA” with an “expert”
- Meetups (soon again!)

Getting Involved

LLVM welcomes contributions of all kinds

- Development Process
- Forums & Mailing Lists
- Online Sync-Ups
- Office hours
- IRC
- Meetups and social events
- Community wide proposals

Parallelism in LLVM

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?

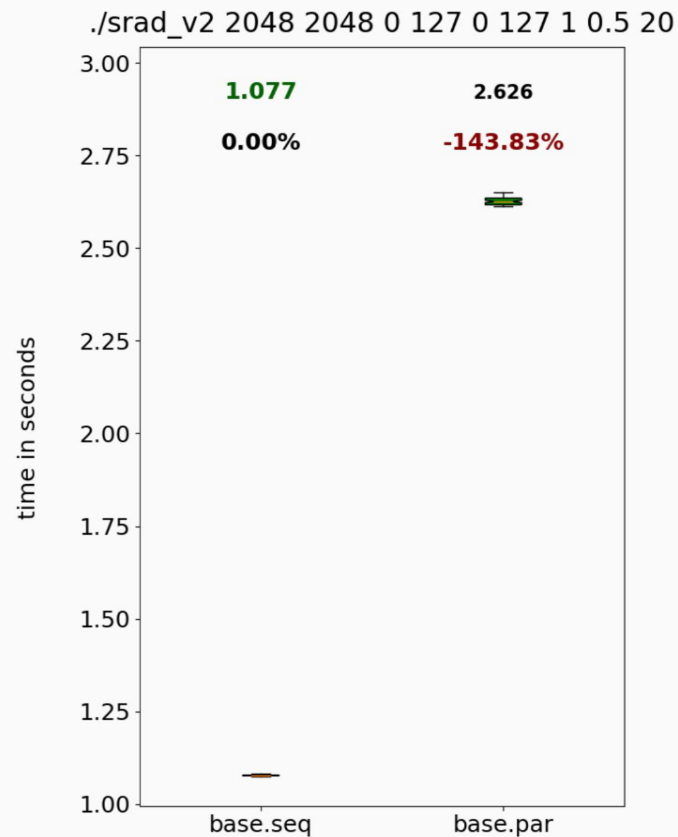
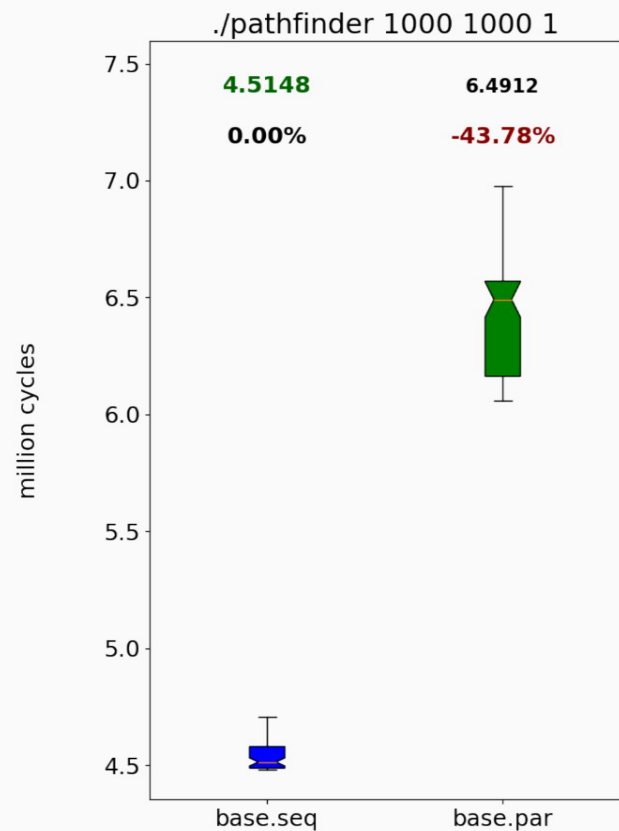
Original Program

```
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?





Optimizing Indirections, or using abstractions without remorse

Presenter: Johannes Doerfert

LLVM/OpenMP

OpenMP in LLVM

<https://openmp.llvm.org/docs>

OpenMP in LLVM

<https://openmp.llvm.org/docs>



Clang

The diagram consists of a light gray rectangular box with a dashed blue border. Inside the box, the word 'Clang' is at the top. Below it, three OpenMP-related components are listed vertically: 'OpenMP Parser', 'OpenMP Sema', and 'OpenMP CodeGen'.

OpenMP
Parser

OpenMP
Sema

OpenMP
CodeGen

OpenMP in LLVM

<https://openmp.llvm.org/docs>

Clang

OpenMP
Parser

OpenMP
Sema

OpenMP
CodeGen

OpenMP runtimes

libomp.so
(classic, host)

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)

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 runtimes

libomp.so
(classic, host)

libomptarget + plugins
(offloading, host)

libomptarget-nvptx
(offloading, device)

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-ng-nvptx
(offloading, device)

LLVM/OpenMP - Learn More

*TAKE A
PICTURE!*

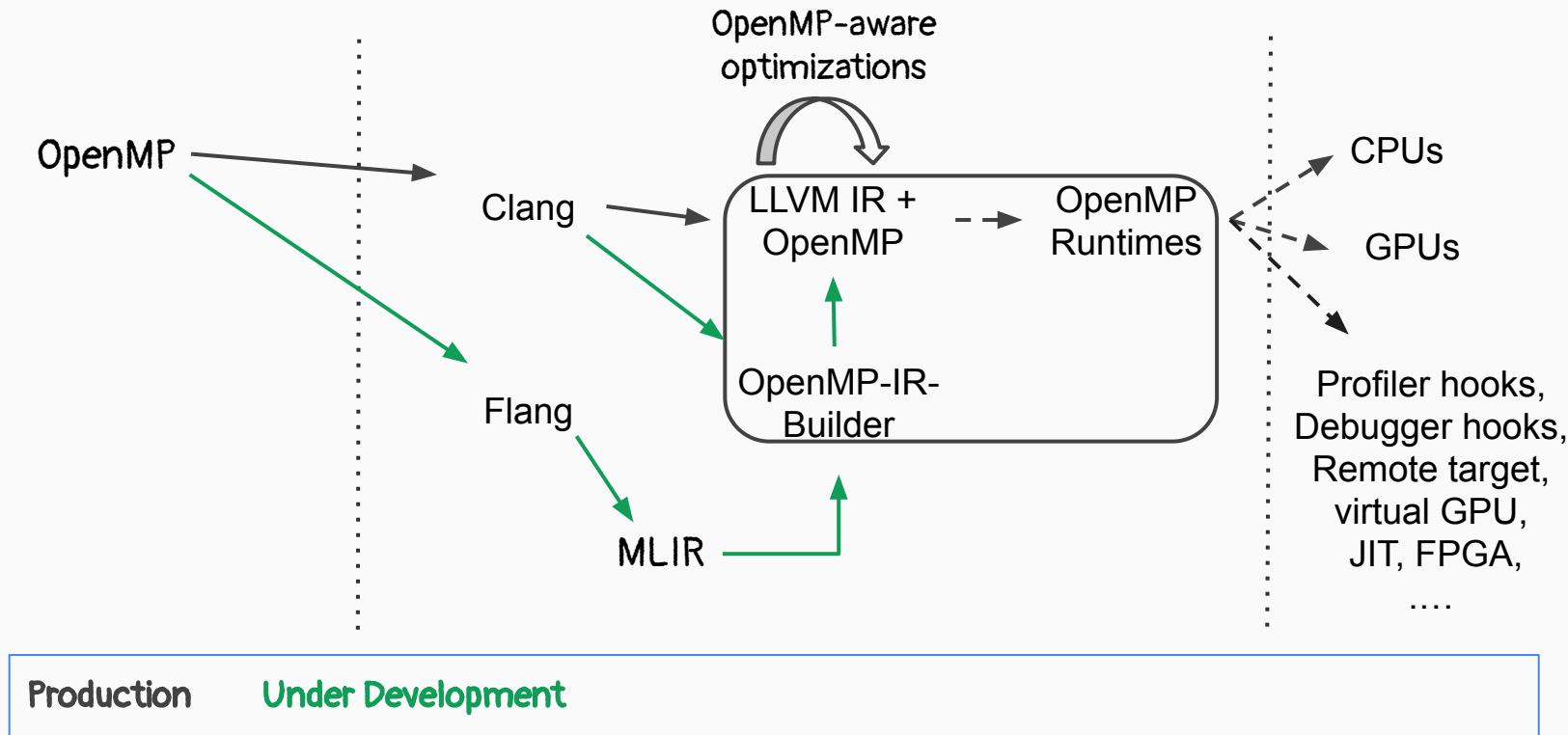
<https://youtu.be/R9PUdx1ya1o>

Full 3h tutorial — lots of details

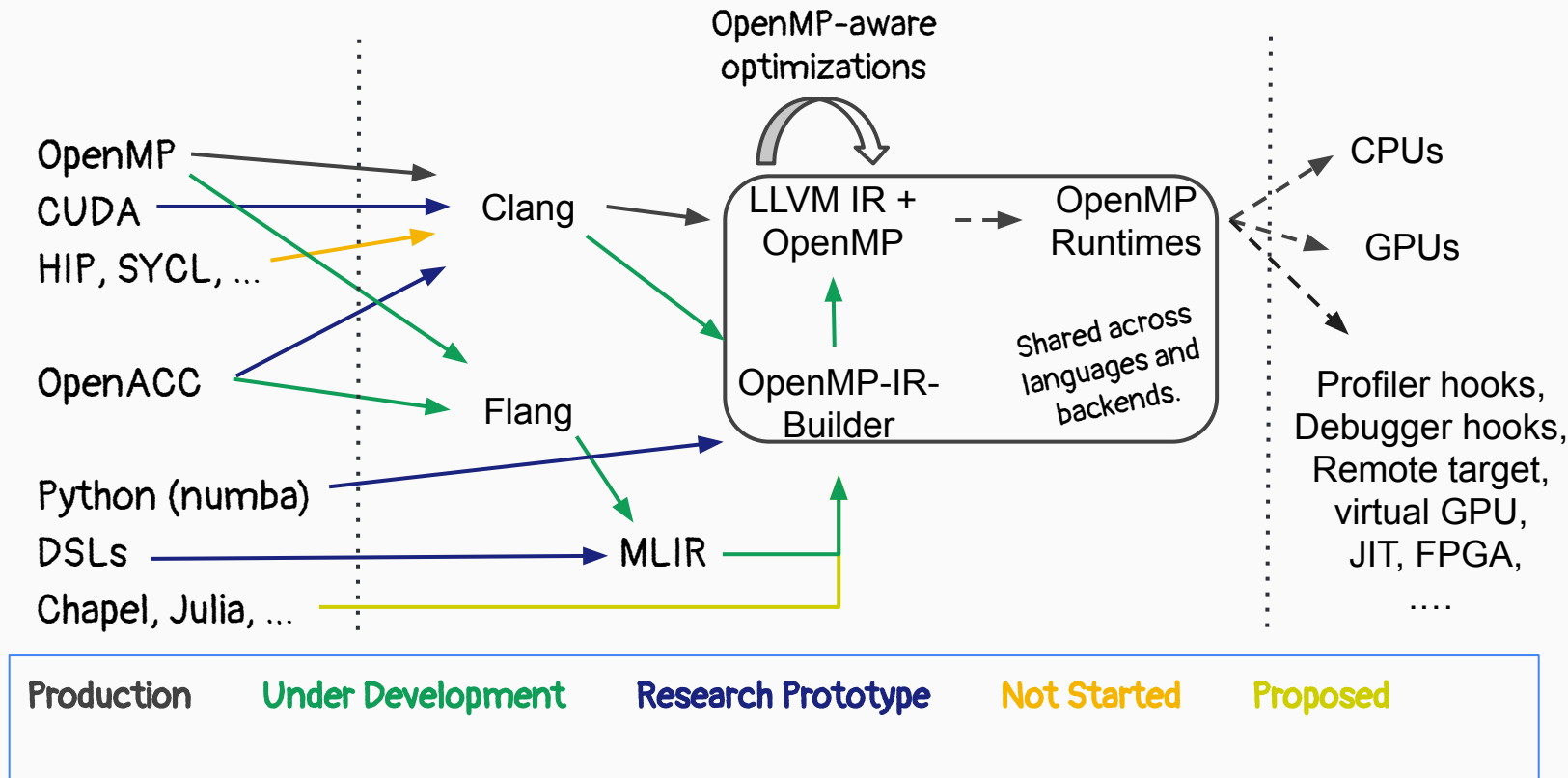


LLVM/OpenMP – Directions

Diverse Inputs/Outputs - Uniform Pipeline



Diverse Inputs/Outputs - Uniform Pipeline



LLVM/OpenMP – Users and Developers

Using LLVM/OpenMP Offload (cheat sheet)

- Use a recent (e.g., nightly) compiler version.
- Enable compilation remarks <https://openmp.llvm.org/remarks/OptimizationRemarks.html>
- Use `LIBOMPTARGET_INFO(=16)` to learn about the GPU execution
<https://openmp.llvm.org/design/Runtimes.html#libomptarget-info>
- Use `LIBOMPTARGET_PROFILE` for built in profiling support.
- Use `LIBOMPTARGET_DEBUG` (and `-fopenmp-target-debug`) for runtime assertions and other opt-in debug features <https://openmp.llvm.org/design/Runtimes.html#debugging>
- Consider assumptions for better performance:
`LIBOMPTARGET_MAP_FORCE_ATOMIC=false`, `-fopenmp-assume-no-thread-state`, ...
- Use device-side LTO `-foffload-lto`

Getting Involved in LLVM/OpenMP

- LLVM/OpenMP webpage (incl. FAQ)

Table I: Performance improvements from different optimizations on OpenMC for the HM-large benchmark problem. All optimizations are added to the ones above them in the table.

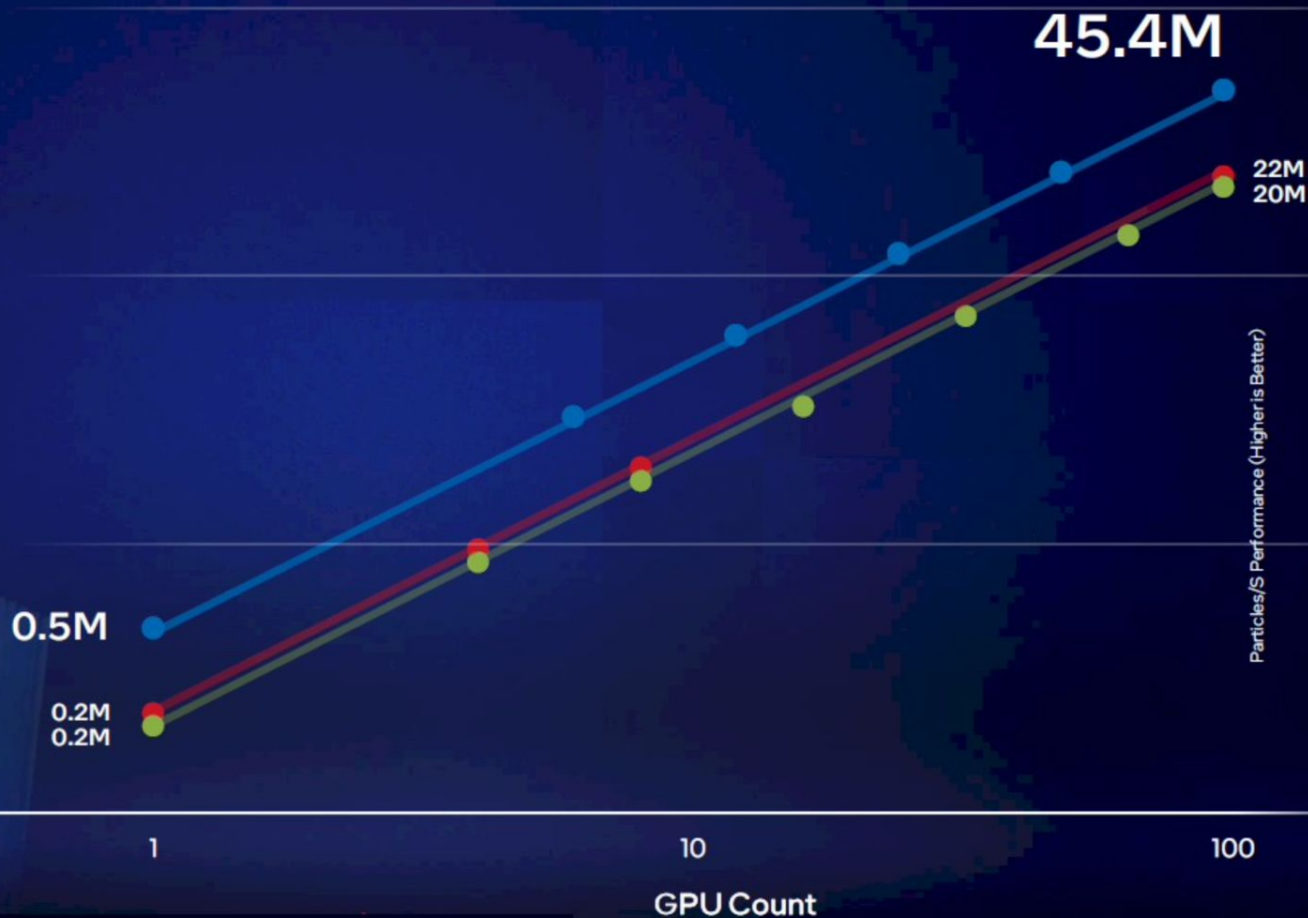
Optimization Description		Inactive Batch Performance [particles/sec]	Additional Speedup
A	CMake unity build	602	-
B	-fopenmp-cuda-mode flag usage	7,714	12.8
C	LLVM update() clause optimization	58,529	7.6
D	OpenMC particle object size reduction	89,732	1.5
E	XS lookup kernel optimizations: inlining + reference removal	117,067	1.3
F	Continuous particle refill	129,345	1.1
G	XS lookup queue sort by energy	164,114	1.3
H	Removal of microscopic XS cache	336,636	2.1
I	Increasing number of particles in-flight to 8 million	349,237	1.1

Also, feel free to reach out at any time!

Monte Carlo Methods Maximized

OpenMC

- Intel Data Center GPU Max Series 1550
- AMD MI250X
- NVIDIA A100 PCIe



Taken from the Intel Presentation @ ISC'23

Papers & Presentations

(selection)

Papers:

- Automatic Asynchronous Execution of Synchronously Offloaded OpenMP Target Regions (LLVM-HPC'22)
- Direct GPU Compilation and Execution for Host Applications with OpenMP Parallelism (LLVM-HPC'22)
- Breaking the Vendor Lock --- Performance Portable Programming Through OpenMP as Target Independent Runtime Layer (PACT'22)
- Just-in-Time Compilation and Link Time Optimization for OpenMP Target Offloading (IWOMP'22)
- 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**)
- Toward Portable GPU Acceleration of the OpenMC Monte Carlo Particle Transport Code (PHYSOR'22)
- 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)

TAKE A
PICTURE!

Presentations:

A Compiler's View of OpenMP <https://www.openmp.org/events/webinar-a-compilers-view-of-the-openmp-api/>

Not OpenMP

Exploration of AI-enhanced Compilers

Current Result

- Relative changes(Instruction counts) to baseline, CTMark
(Trained with test-suite/MultiSource + SingleSource expect for CTMark)

aggressive

	O3	
threshold	compile time	execution time
prob = 0.5	-2.74%	+0.1%
prob = 0.9	-4.93%	+0.51%

AI-based Optimization “skipping”

Hideto Ueno

Tarindu Jayatilaka

Johannes Doerfert

EJ Park

Giorgis Georgakoudis

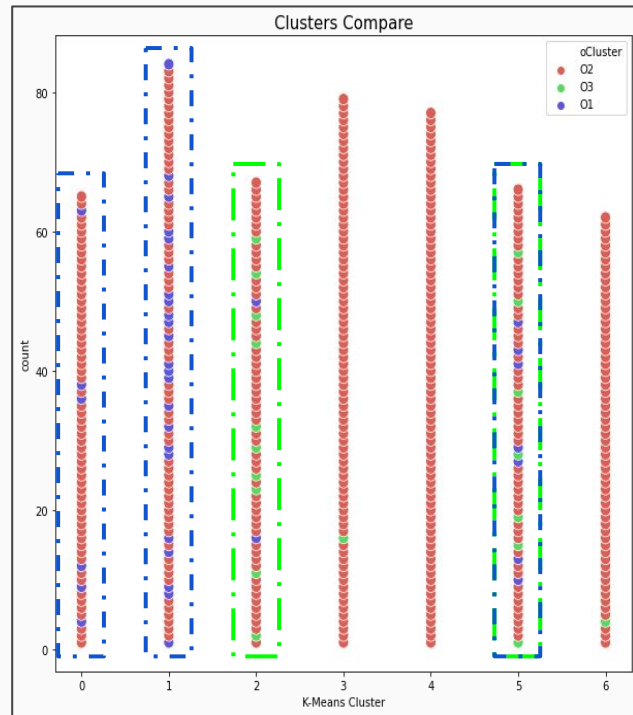
University of Tokyo

University of Moratuwa

Lawrence Livermore National Laboratory

Los Alamos National Laboratory

Lawrence Livermore National Laboratory



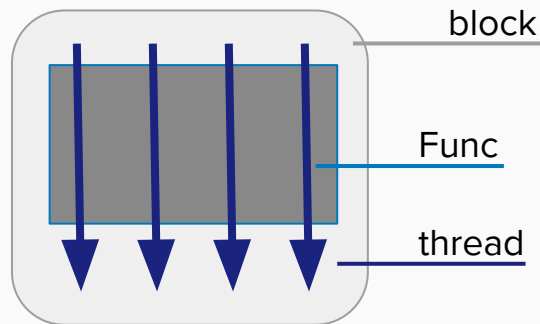
Code Characterization

CPU vs GPU Execution Model

OpenMP Offloading vs Kernel Languages

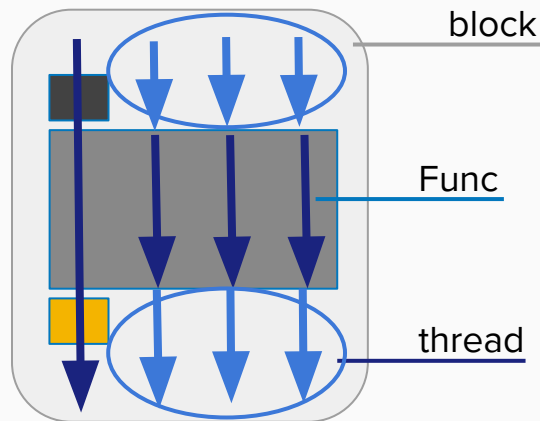
```
Func<<< /* blocks */ 1, /* threads */ 4 >>>(args);
```

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



LLVM/OpenMP

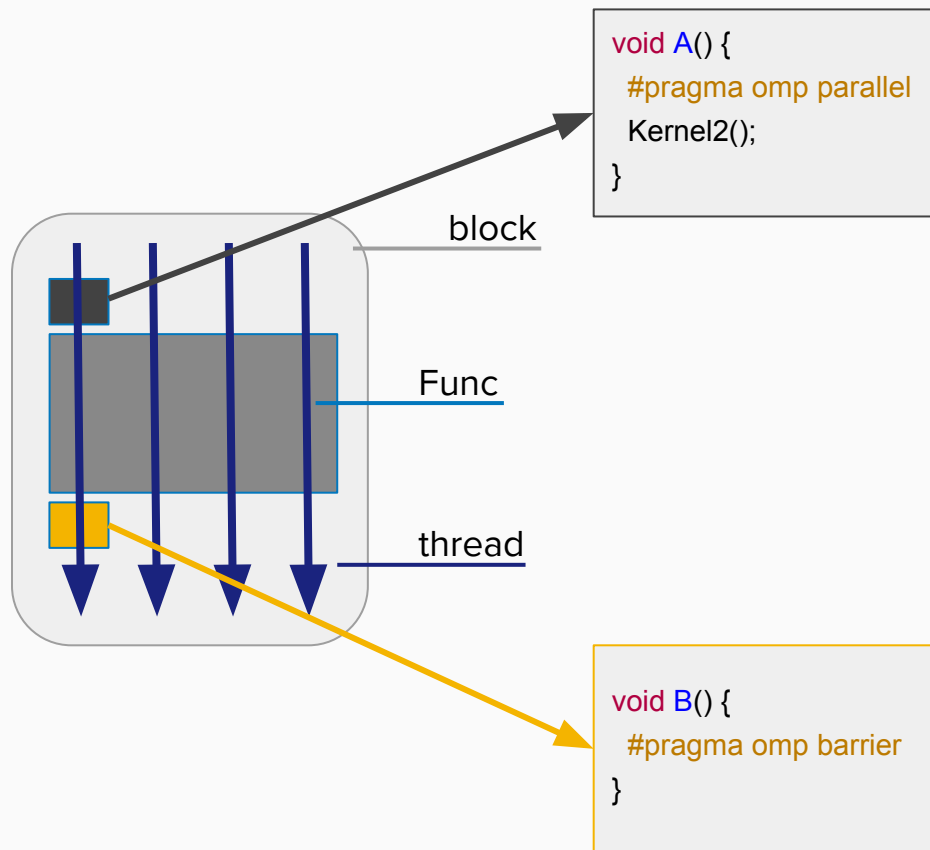
SPMD-mode



Generic-mode

OpenMP Offloading vs Kernel Languages

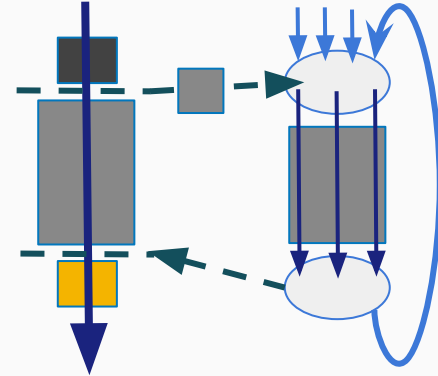
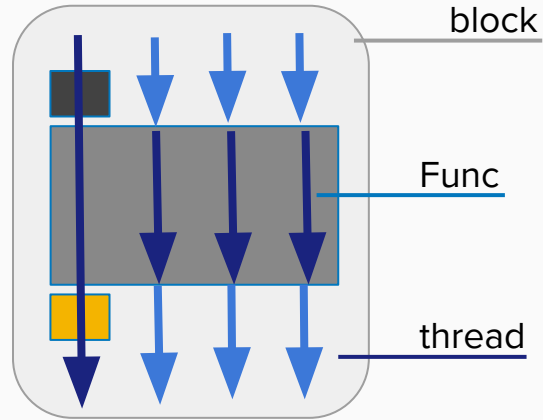
```
#pragma omp target teams num_teams(1)
{
  #pragma omp parallel num_threads(4) default(firstprivate)
  {
    if (omp_get_thread_num() == 0)
      A();
    #pragma omp barrier
    Func(args);
    #pragma omp barrier
    if (omp_get_thread_num() == 0)
      B();
  }
}
```



SPMD-zation, since LLVM 14

OpenMP Offloading vs Kernel Languages (simplified)

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



OpenMP Offloading vs Kernel Languages (simplified)

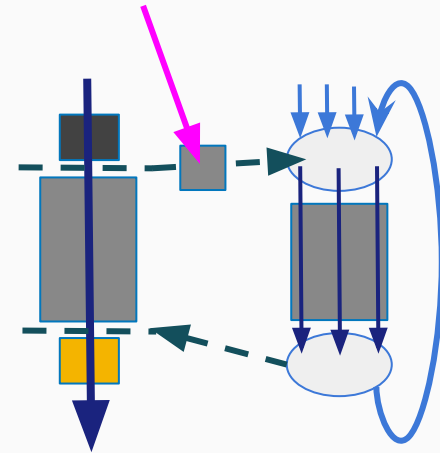
Q: How do you identify a parallel region?

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

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

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

Function Pointer

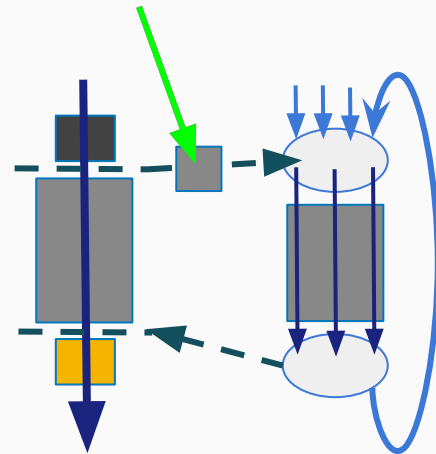


OpenMP Offloading vs Kernel Languages (simplified)

```
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();  
    }  
}
```

Function Rebinder



Performed since LLVM 12

OpenMP Offloading vs Kernel Languages (simplified)

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

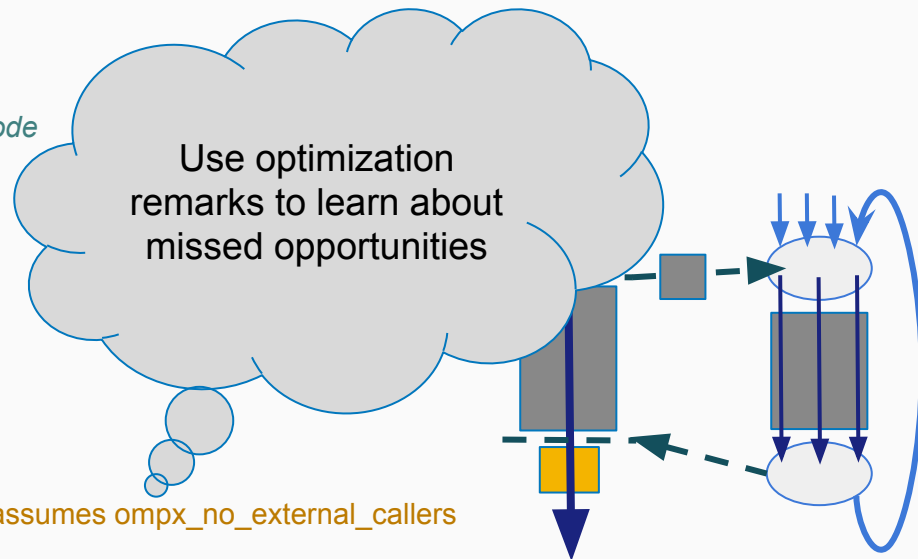
```
void kernel() {  
    if (is_worker()) {  
        // ...  
    } else {  
        visible();  
    }  
}
```

```
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
```



Not OpenMP

The Present and Future of Interprocedural Optimization in LLVM

Stefanos Baziotis
stefanos.baziotis@gmail.com

Kuter Dinel
kuterdinel@gmail.com

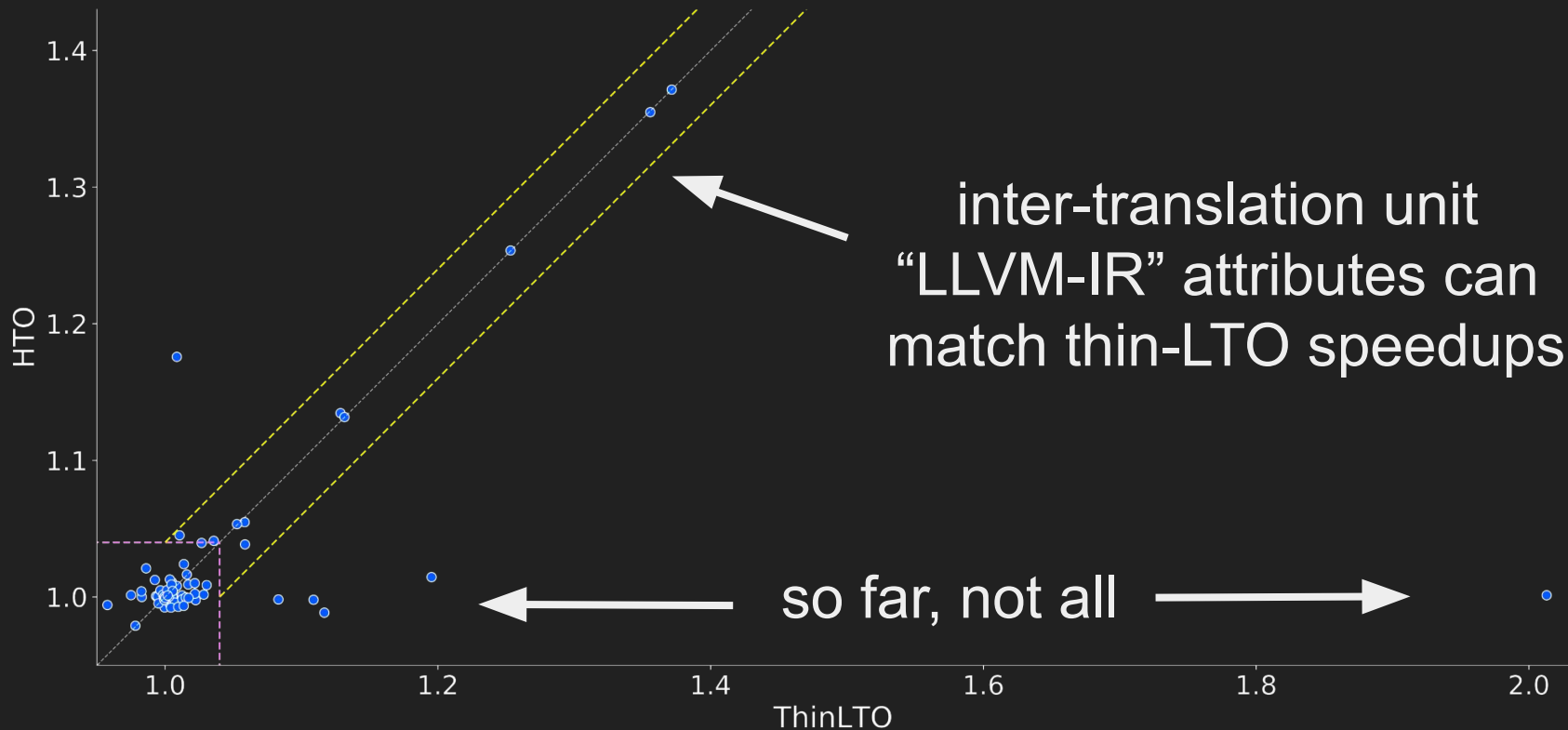
Shinji Okumura
okuraofvegetable@gmail.com

Luofan Chen
clfbbn@gmail.com

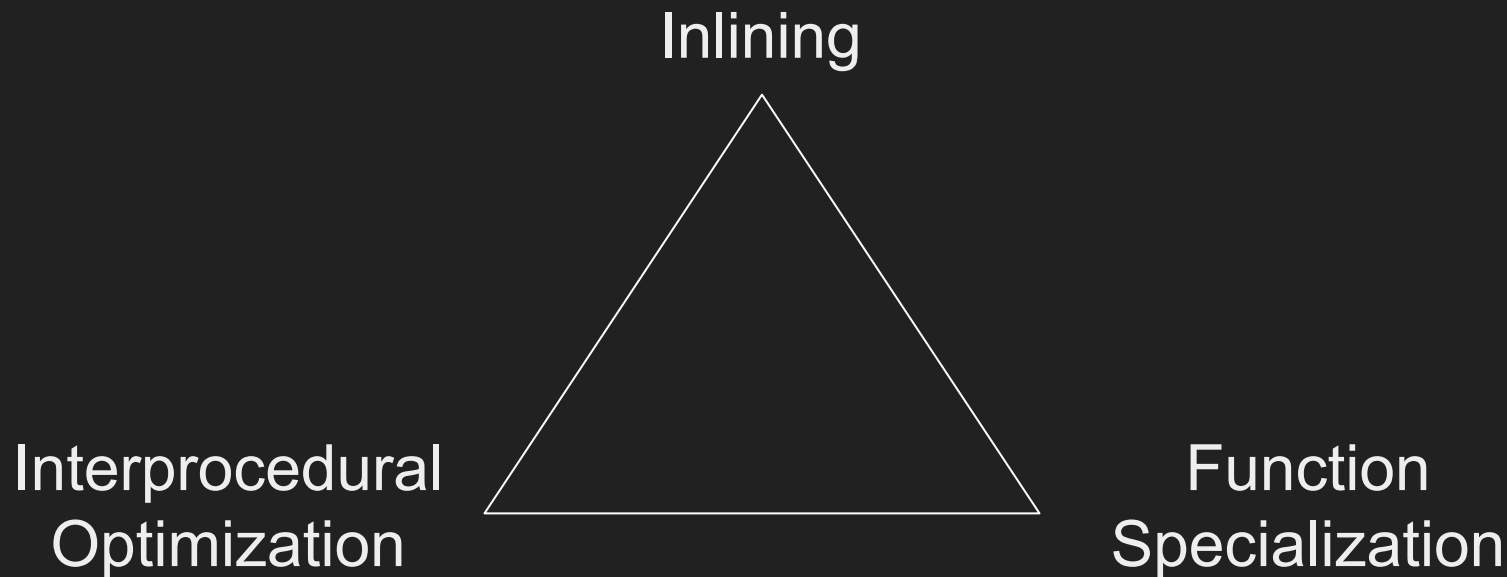
Hideto Ueno
uenoku.tokotoko@gmail.com

Johannes Doerfert
johannesdoerfert@gmail.com

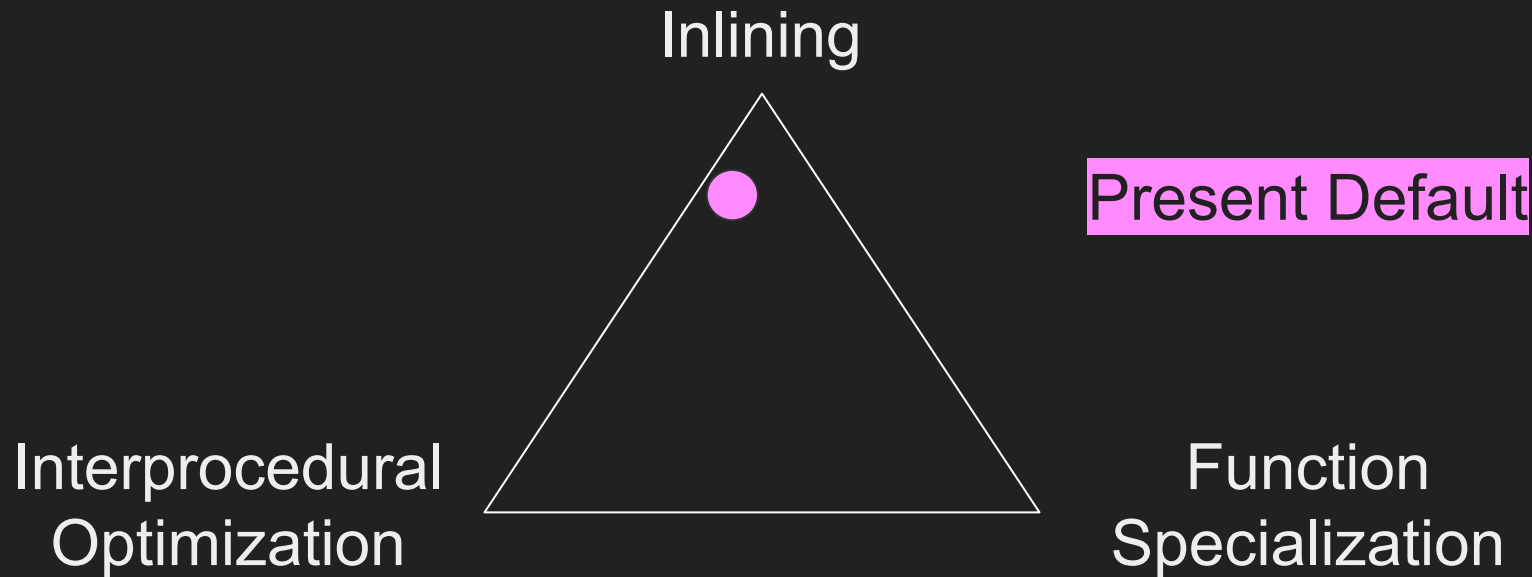
Inlining - Alternatives: thin-LTO^[7] vs HTO^[8]



Design Space



Design Space



Design Space

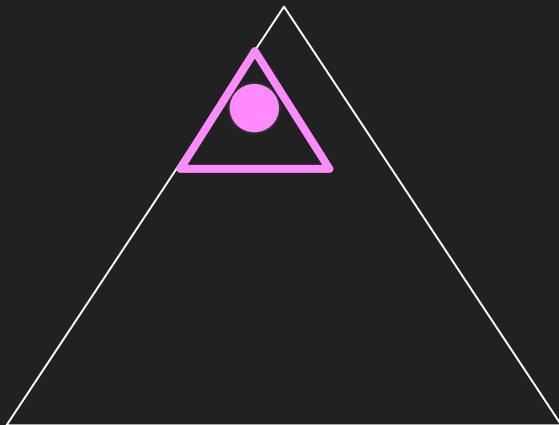
Present Options

Present Default

Interprocedural
Optimization

Inlining

Function
Specialization



Design Space



Design Space

Present Options

Future Options

Interprocedural
Optimization

Inlining



Present Default

Future Default

Function
Specialization

Design Space

Present Options

Future Options

Interprocedural
Optimization

Inlining ✓

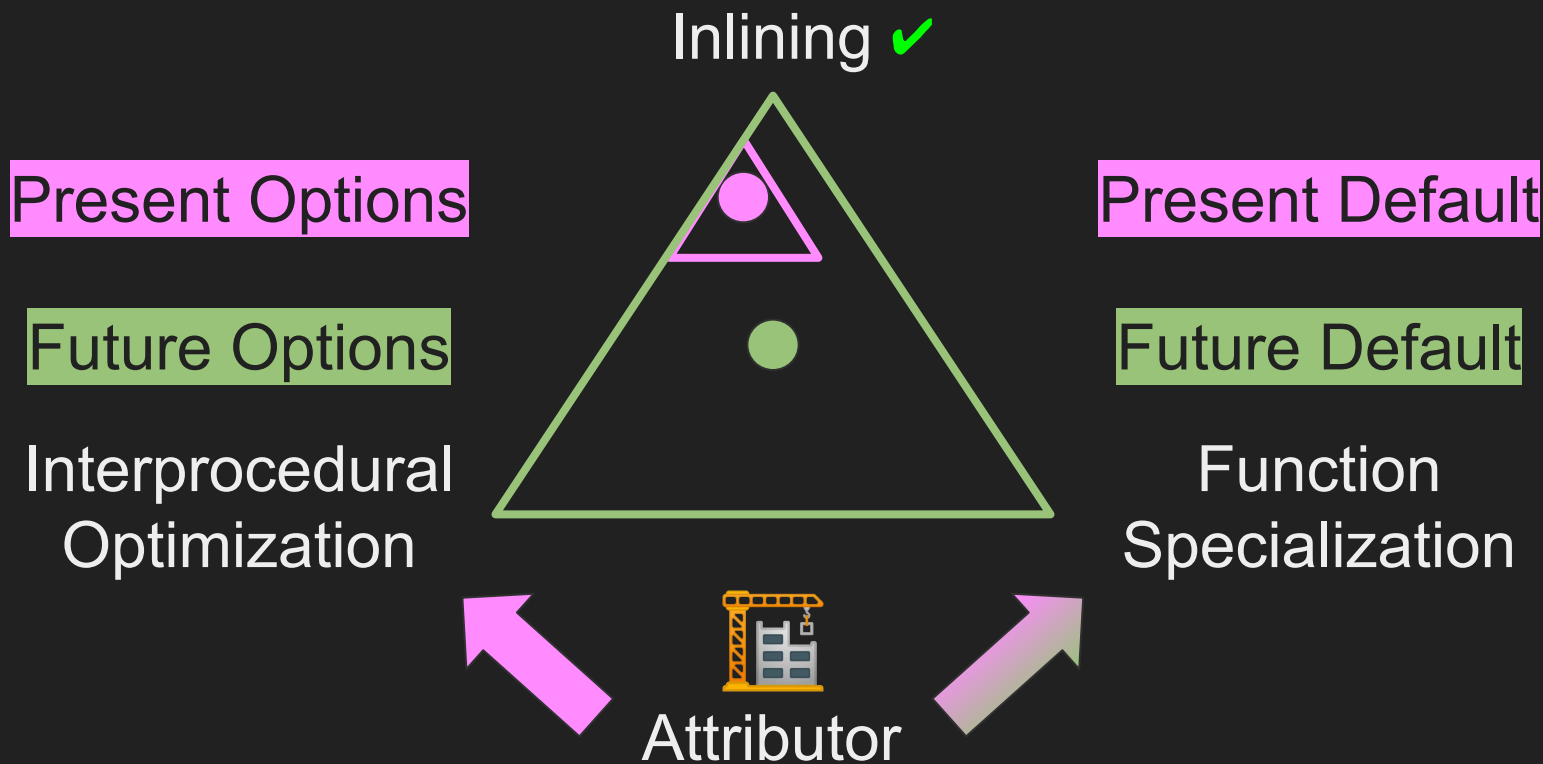


Present Default

Future Default

Function
Specialization

Design Space



Attributor

The Attributor^[1,9] is an *interprocedural fixpoint iteration framework*; with lots of built-in features.

Phase of Attributor

Seeding

Determine which kind of deduction or analysis we try to do

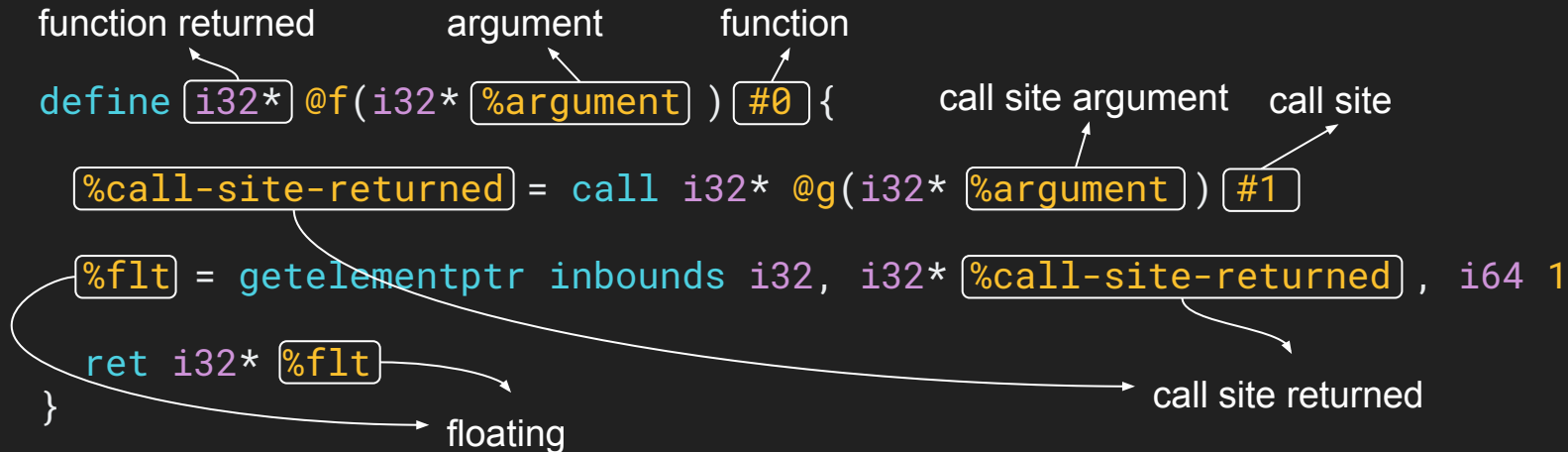
Update

Update states till fixpoint is reached

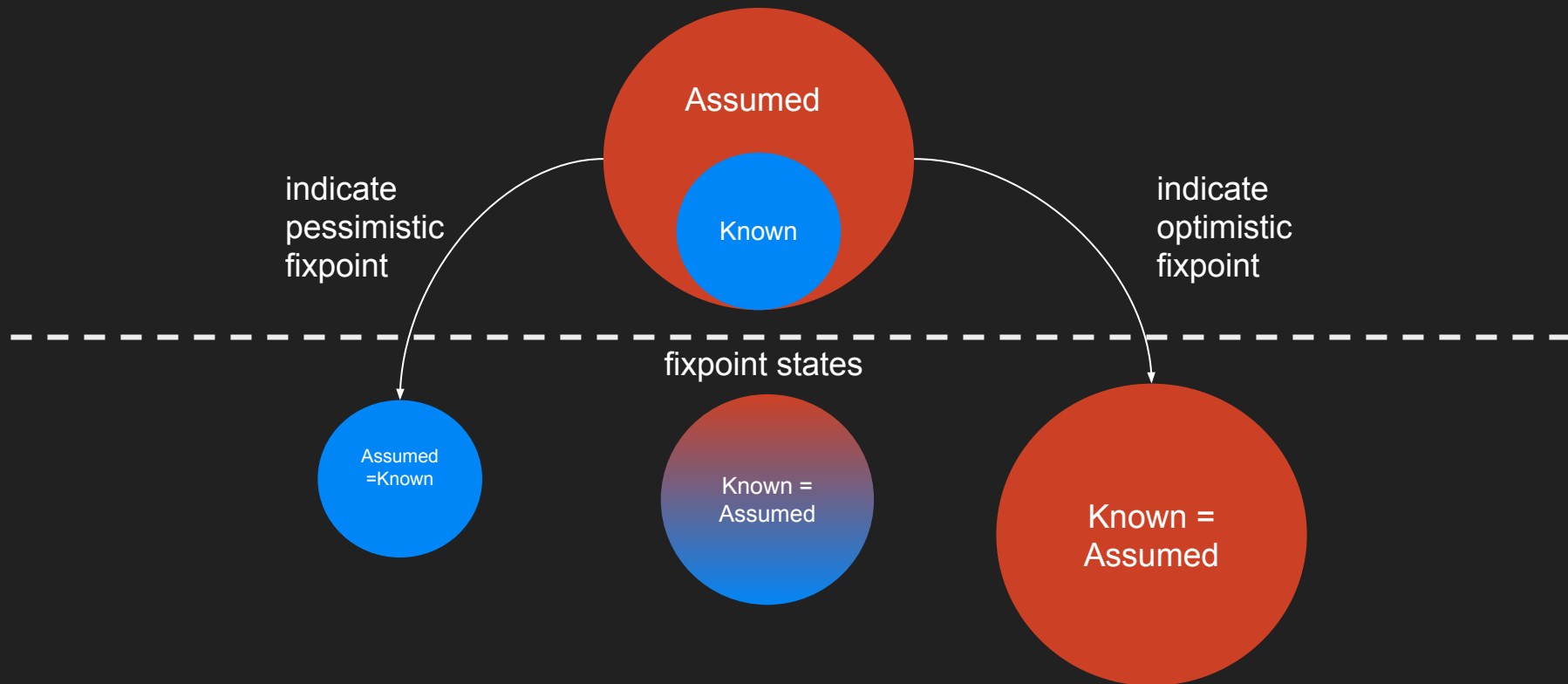
Manifest

Transform IR according to the results

LLVM-IR Positions



Abstract state



The Abstract Attribute (AA) Zoo

AAIsDead	AANoCapture	AANoUndef
AANoUnwind	AAValueSimplify	AACallEdges
AANoSync	AANoFree	AAFunctionReachability
AANoRecurse	AAHeapToStack	AAPointerInfo
AAWillReturn	AAReachability	AAAssumptionInfo
AANoReturn	AAMemoryBehavior	AADominance*
AAReturnedValues	AAMemoryLocation	AACallGraph*
AANonNull	AAPrivatizablePtr	AAExecutionDomain*
AANoAlias	AAUndefinedBehavior	+ AMD GPU Backend
AADereferenceable	AAPotentialValues	+ OpenMP-Opt
AAAlign	AAValueConstantRange	

Attributor Features

- Performance related
 - Dependency types and tracking
 - Dead code “skipping”
 - Selective seeding
- Utility for users
 - Helper classes for generic deduction
 - Helper functions for traversing assumed live uses, basic blocks, call sites, ...
 - Provides a uniform analysis pass query API
 - Time traces, visualization (e.g., dependence graphs)

Transparent Indirections

The Attributor transparently looks through:

- memory (incl. globals, locals, malloced mem, ...)
- calls (incl. indirect* and transitive calls),
- phi-nodes, selects, ...

anything “assumed dead” is ignored/hidden by default.

```
struct PTy { int *a; int v; };
```

```
void callee(PTy *payload) {  
    *payload->a = payload->v;  
}
```

```
void caller(int * /* align(128) */ A) {  
    PTy payload = {A, 42};  
    callee(&payload);  
}
```

```
struct PTy { int *a; int v; };
```

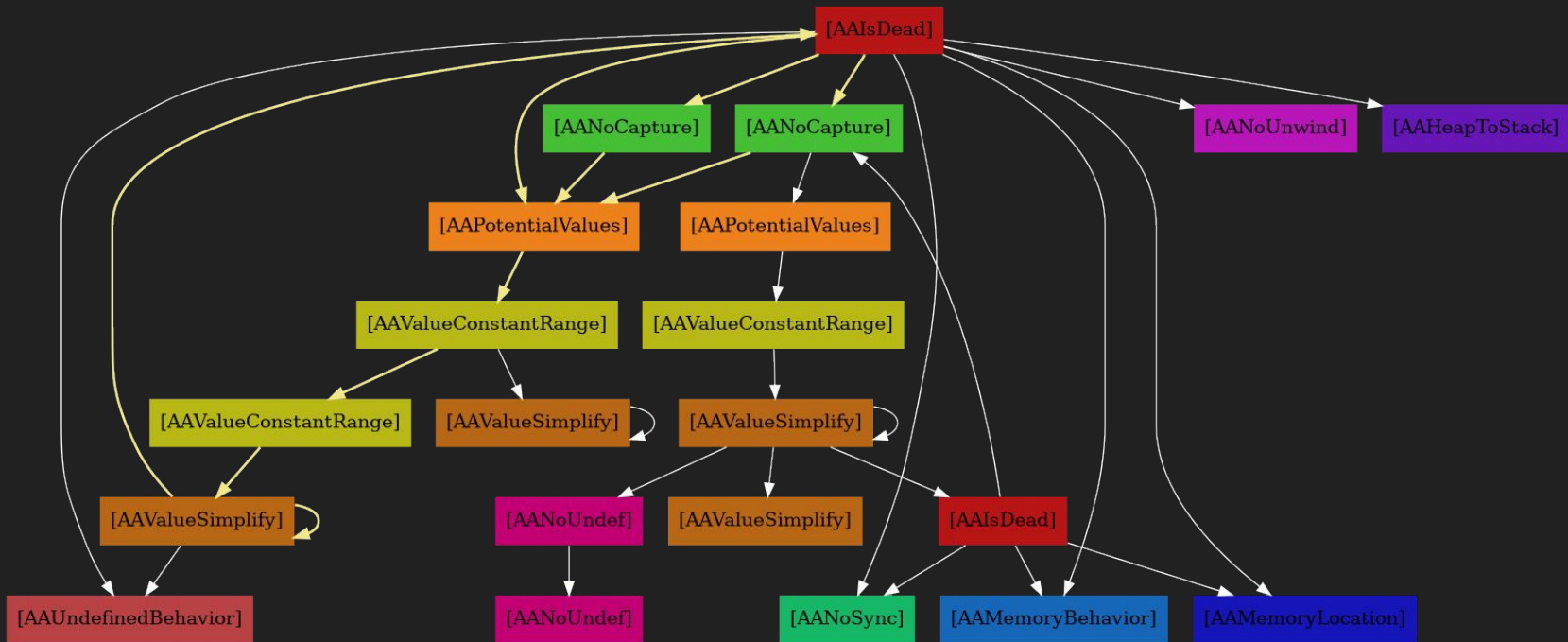
```
void callee(PTy *payload) {  
    /* align(128) */payload->a = 42;  
}
```

```
void caller(int * /* writeonly */ A) {  
    PTy payload = {A};  
    callee(&payload);  
}
```

Dataflow Iterations

```
void unknown(int &x);
static void check_n_inc(int n, int &x, int &y) {
    if (x) unknown(x);
    if (n) check_n_inc(n-1, y, x);
}
```

```
int test(int n) {
    int x = 0, y = 0;
    check_n_inc(n, x, y);
    return x + y;
}
```



OpenMP → CUDA PTX

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

```
__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 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
```

```
        #pragma omp parallel for num_threads(...) shared(L)
```

```
        for (int j = 0; j < BLOCK_SIZE; ++j)
```

```
            if (*L != 0)
```

```
                parallel_work(Buffer, j);
```

```
        #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(...)
```

```
    }
```



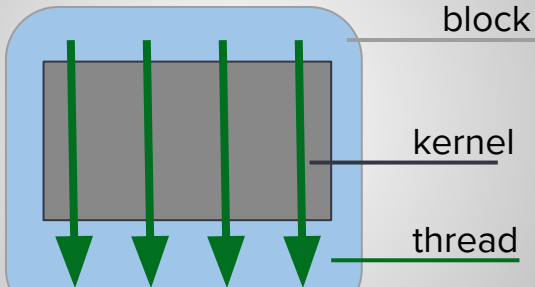
Local/Stack variables cannot be shared across GPU threads.

CUDA vs OpenMP Offload – Execution Mode Mismatch

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

```
shared double Buffer[BLOCK_SIZE]
```

SPMD/GPU Execution Mode

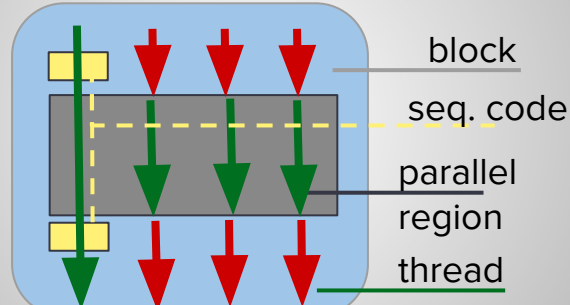


```
if (L != 0)
    parallel_work(Buffer, threadIdx.x);
```

```
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);
```

Generic/CPU Execution Mode



```
    #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

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

```
__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);  
  
    }  
}
```

```
__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  
    }  
}
```

```
__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
    }
}
```

```
__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)

```
__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

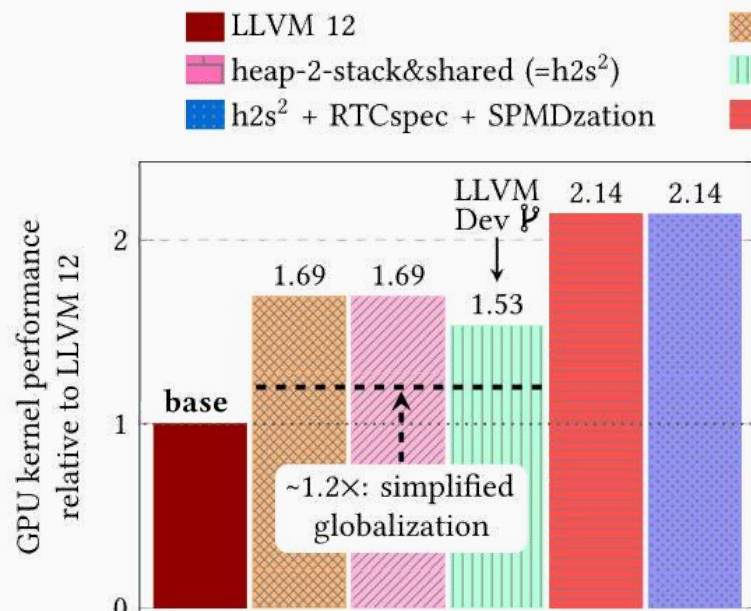
        L = load_data(Buffer, j);
        #pragma omp barrier // aligned

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

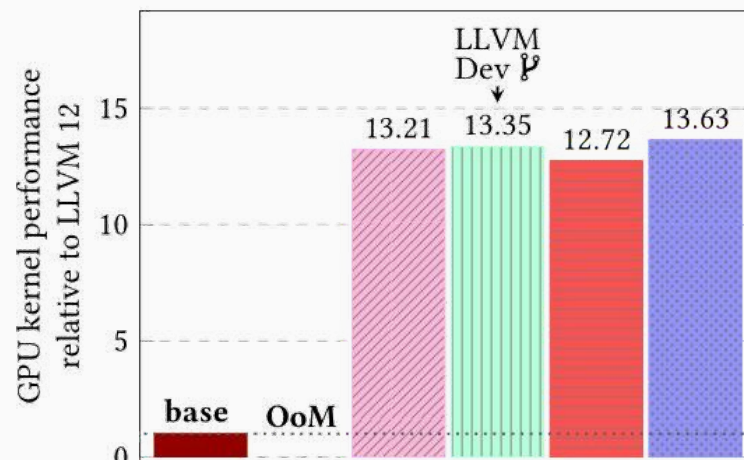
    }
}
```

Optimized OpenMP Offload Performance

See: CGO'22



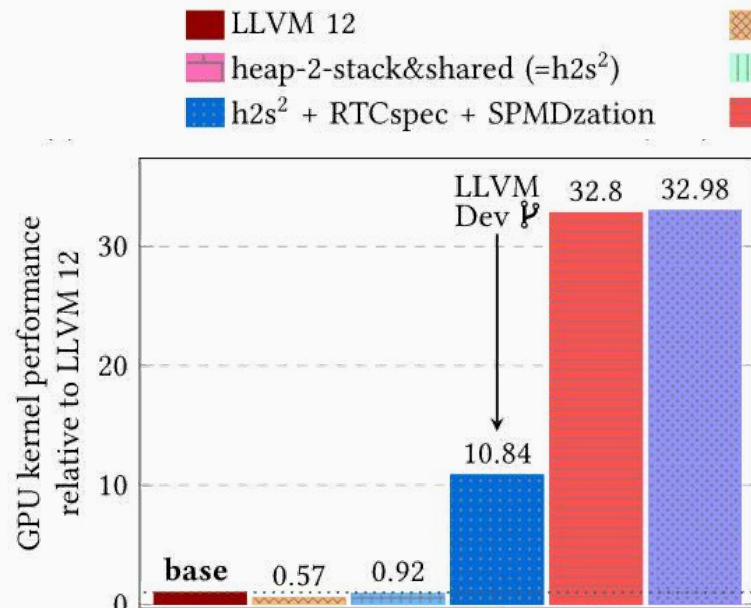
(a) Performance of XSbench relative to LLVM 12 (base).



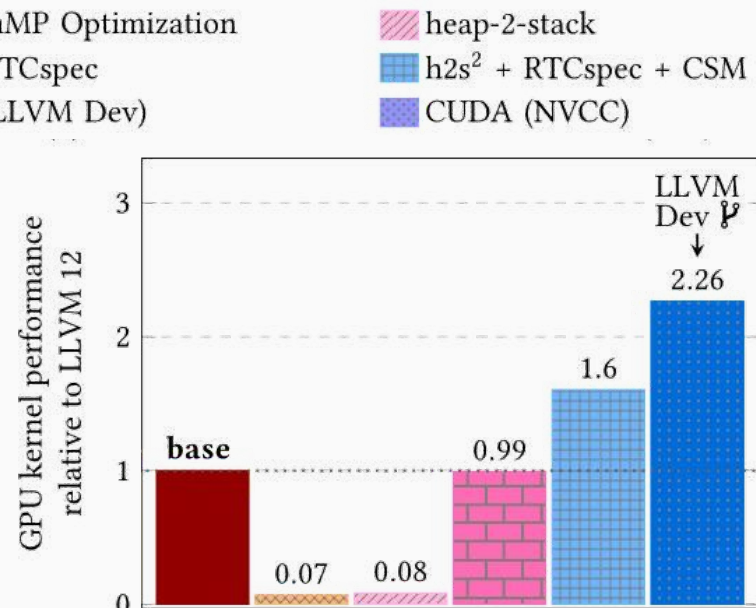
(b) Performance of RSbench relative to LLVM 12 (base).

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

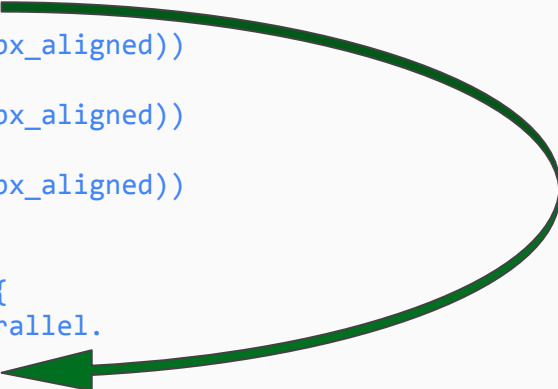
```
__shared__ StateTy State;

__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;
```



shared memory lifetime + IP-write-only

```
__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))  
}
```

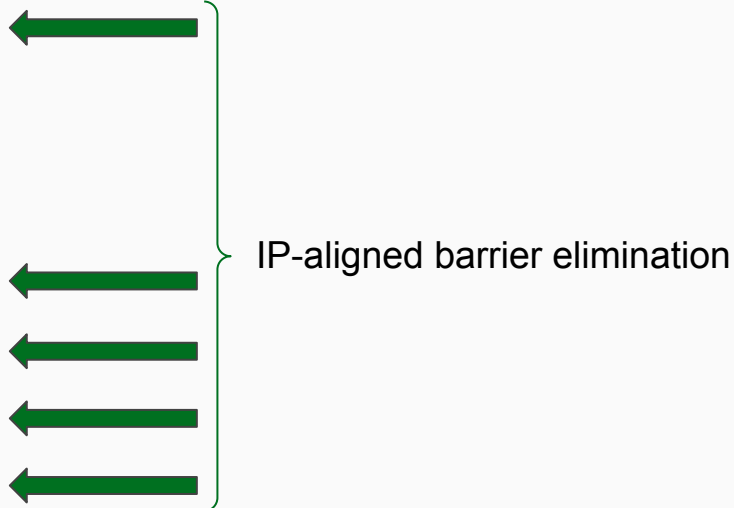


shared memory lifetime + IP-DSE

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

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.

New environment assumptions:

`LIBOMPTARGET_MAP_FORCE_ATOMIC=false`

`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-cuda-mode`

`-fopenmp-assume-no-nested-parallelism`

`-fopenmp-assume-no-thread-state`

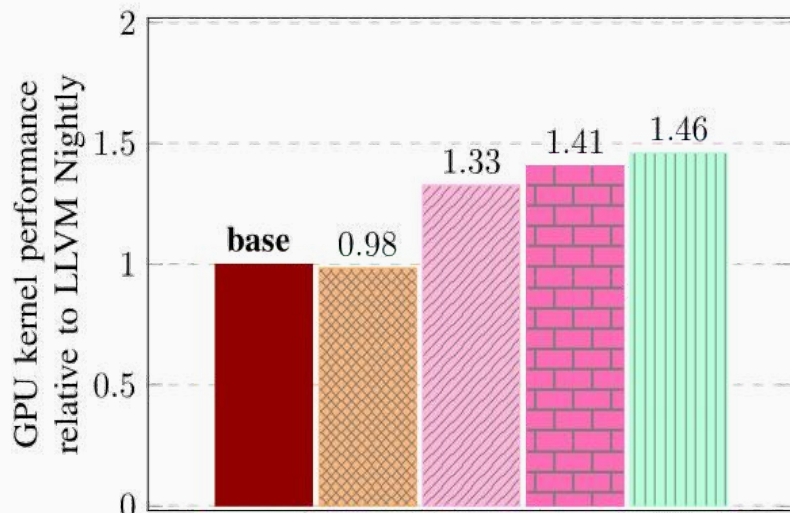
`-fopenmp-assume-teams-oversubscription`

`-fopenmp-assume-threads-oversubscription`

Optimized OpenMP Offload Performance

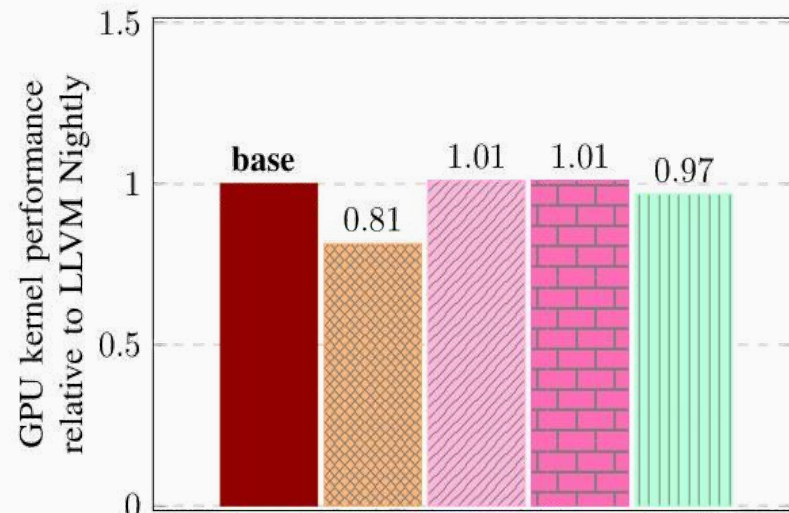
See: IPDPS'22

Old Device RT Nightly
New Device RT $\mathcal{P}$
New Device RT Nightly
CUDA (NVCC)



(a) Performance of XSbench relative to LLVM Nightly.

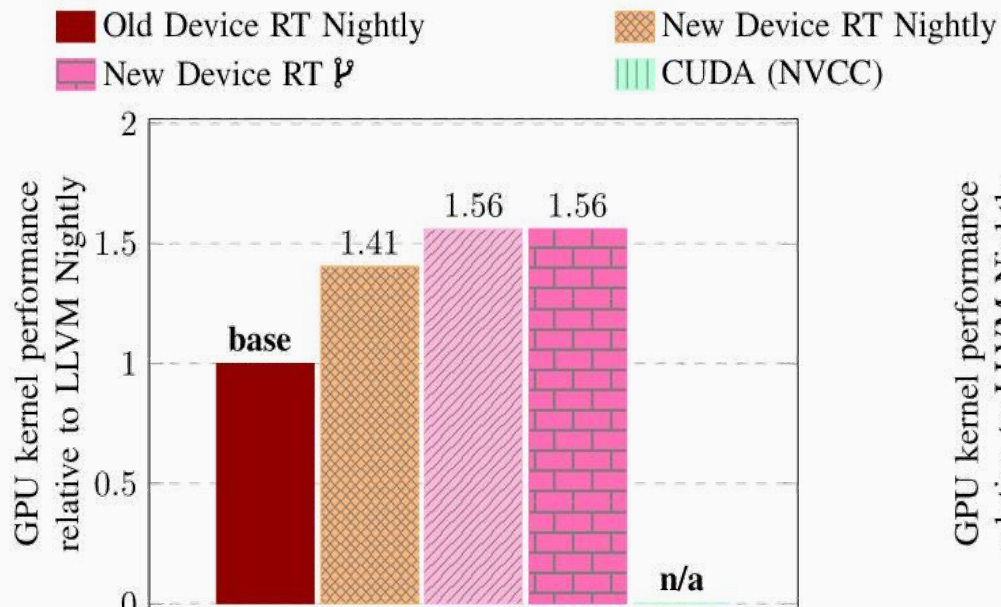
New Device RT w/o Assumptions $\mathcal{P}$



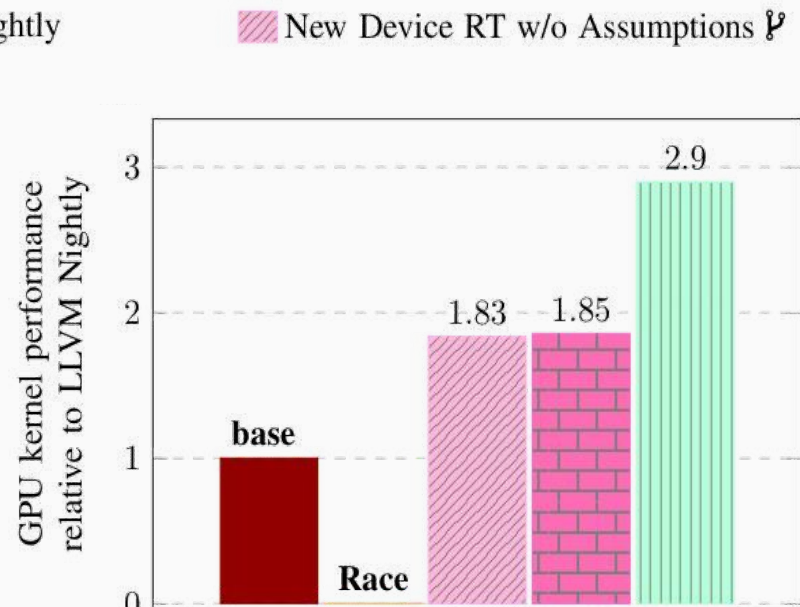
(b) Performance of RSbench relative to LLVM Nightly.

Optimized OpenMP Offload Performance

See: IPDPS'22



(c) Performance of TestSNAP relative to LLVM Nightly.



(d) Performance of MiniFMM relative to LLVM Nightly.

Cuda-like OpenMP

Offload Style

```
int i = blockId.x;  
if (i >= N) return;  
int j = threadId.x;  
if (j >= M) return;  
body1(i, j);  
__syncthreads();  
body2(i, j);  
}
```

Offload Style

```
int i = blockId.x;  
if (i >= N) return;  
int j = threadId.x;  
if (j >= M) return;  
body1(i, j);  
__syncthreads();  
body2(i, j);  
}
```

```
#pragma omp target teams distribute  
for (int i = ...) {  
    #pragma omp parallel for  
    for (int j = ...) {  
        body1(i, j);  
    }  
    #pragma omp parallel for  
    for (int j = ...) {  
        body2(i, j);  
    }  
}
```

Offload Style

```
int i = blockId.x;
if (i >= N) return;
int j = threadId.x;
if (j >= M) return;
body1(i, j);
__syncthreads();
body2(i, j);
}
```

```
#pragma omp target loop
for (int i = ...) {
    #pragma omp loop
    for (int j = ...)
        body1(i, j);
    #pragma omp loop
    for (int j = ...)
        body2(i, j);
}
```

OpenMP Kernel Language

```
_LIBOMPX_NAMESPACE_BEGIN

extern "C" void *ompx_malloc(size_t size) {
    if (size <= 0)
        return nullptr;
    return omp_target_alloc(size, omp_get_default_device());
}

void *ompx::malloc(size_t size) { return ompx_malloc(size); }

_LIBOMPX_NAMESPACE_END
```

libompx – Device Wrappers

```
/// AMDGCN Implementation
///
///{
#pragma omp begin declare variant match(device = {arch(amdgc)})

uint32_t ompx_get_thread_num(int Dim = 0) {
    switch(Dim) {
        case 0:
            return __builtin_amdgcn_workitem_id_x();
        case 1:
            return __builtin_amdgcn_workitem_id_y();
        case 2:
            return __builtin_amdgcn_workitem_id_z();
        default: break;
    }
    __builtin_unreachable();
}

...

#pragma omp end declare variant
```



```
kern<<<nblocks, nthreads, shmem>>>(a1, a2)
```

```
#pragma omp target teams num_teams(nblocks) thread_limit(nthreads) \  
    ompx_cgroup_dyn_mem(shmem) ompx_kernel  
kern(a1, a2)
```

```
__device__ void foo();
```

```
void foo();  
#pragma omp declare target device_type(nohost) to(foo)
```

```
__device__ void foo();
```

```
[[ompx::declare_target(device_type(nohost))]] void foo();
```

Not OpenMP

Performance Exploration Through Optimistic Static Program Annotations

Johannes Doerfert
Argonne National Laboratory

Brian Homerding
Argonne National Laboratory

Hal Finkel
Argonne National Laboratory

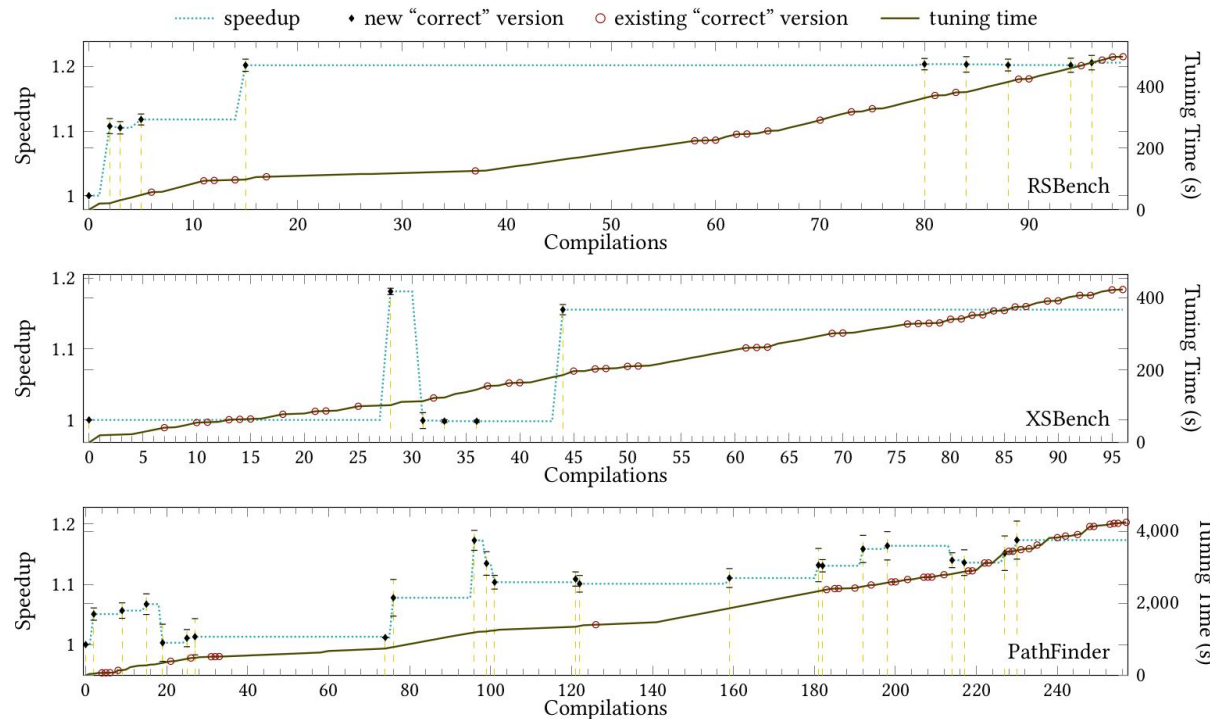


Table 3. Optimistic optimization opportunities exploited and the sections in which they are discussed.

potentially overflowing computations	Section 3.1
potentially parallel loops	Section 3.2
unknown control flow choices	Section 3.3
potential undefined behavior in functions	Section 3.4.1
unknown function side-effects	Section 3.4.2
potential runtime exceptions in functions	Section 3.4.3
unknown function return values	Section 3.4.4
externally visible functions	Section 3.4.5
potentially aliasing pointers	Section 3.5.1
potentially escaping pointers	Section 3.5.2
unknown pointer usage	Section 3.5.3
unknown pointer alignment	Section 3.5.4
potentially non-dereferenceable pointers	Section 3.5.5
potentially invariant memory locations	Section 3.5.6

Sec.	3.1	3.1	3.1	3.2	3.3	3.4.2	3.4.3	3.4.4	3.4.5
Det.	nsw	nuw	gep	function behavior					
(A)	0	12	0	3	1	5/7	2	0	4
(B)	0	16/22	0	3	1	4	0	0	1/3
(C)	0	0	0	4	8/27	15/23	14	0	2
(D)	2	16	0	6	0/2	3/4	1	0	0
(E)	0	18/19	0	0	0	18/37	9/14	8	33/37
(F)	47	132	9	18	3	3	1	0	2

ORAQL - Perfect Alias Analysis Queries

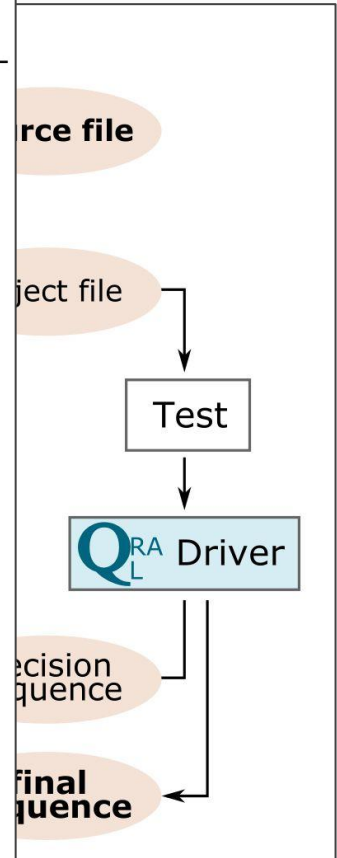
1) Explore a
local opt

2) Determin
and iden

3) Improve
situation

Jan Hueckel
--- See o

Benchmark	Programming Model	ORAQL speedup
TestSNAP	C++	3.6%
TestSNAP	C++, OpenMP	1.0%
TestSNAP	C++, Kokkos, CUDA	0
TestSNAP	Fortran	5%
XSbench	all versions	0
GridMini	C++, OpenMP Offload	-7%
Quicksilver	C++, OpenMP	[inconclusive]
LULESH	C++	0.8%
LULESH	C++, OpenMP	1.4%
LULESH	C++, MPI	-0.2%
MiniFE	C++, OpenMP	0
MiniGMG	C, SSE intrinsics	3.4%
MiniGMG	C, OpenMP tasks	1%
MiniGMG	C, OpenMP	8.3%



LLVM/OpenMP - Misc

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: Co-Designing an OpenMP GPU Runtime and Optimizations (IPDPS'21)

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.

New environment assumptions:

`LIBOMPTARGET_MAP_FORCE_ATOMIC=false`

`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-cuda-mode`

`-fopenmp-assume-no-nested-parallelism`

`-fopenmp-assume-no-thread-state`

`-fopenmp-assume-teams-oversubscription`

`-fopenmp-assume-threads-oversubscription`

Example: LLVM Remarks

```
for (int i = 0; i < j; i++)  
  x_val -= x_ptr[i] * G_ptr[i]
```

```
$ clang++ -fopenmp -Rpass=ana-  
remark: loop not vectorized: 0
```

```
void work(void *);  
void foo() {  
  int local;  
  work(&local);  
}
```

```
#pragma omp declare target(foo)
```

```
$ clang++ -fopenmp -fopenmp-t-  
remark: Could not move global  
parameter as __attribute__((noescape)) void *);
```

OpenMP Optimization Remarks

The [OpenMP-Aware optimization pass](#) is able to generate compiler remarks for performed and missed optimisations. To emit them, pass these options to the Clang invocation: `-Rpass=openmp-opt -Rpass-analysis=openmp-opt -Rpass-missed=openmp-opt`. For more information and features of the remark system, consult the clang documentation:

- [Clang options to emit optimization reports](#)
- [Clang diagnostic and remark flags](#)
- The `-foptimization-record-file` flag and the `-fsave-optimization-record` flag

OpenMP Remarks

Diagnostics Number	Diagnostics Kind	Diagnostics Description
OMP100	Analysis	Potentially unknown OpenMP target region caller.
OMP101	Analysis	Parallel region is used in unknown / unexpected ways. Will not attempt to rewrite the state machine.
OMP102	Analysis	Parallel region is not called from a unique kernel. Will not attempt to rewrite the state machine.
OMP110	Optimization	Moving globalized variable to the stack.
OMP111	Optimization	Replaced globalized variable with X bytes of shared memory.
OMP112	Missed	Found thread data sharing on the GPU. Expect degraded performance due to data globalization.
OMP113	Missed	Could not move globalized variable to the stack. Variable is potentially captured in call. Mark parameter as <code>__attribute__((noescape))</code> to override.
OMP120	Optimization	Transformed generic-mode kernel to SPMD-mode.
OMP121	Analysis	Value has potential side effects preventing SPMD-mode execution. Add <code>__attribute__((assume("omp_spmc_amenable")))</code> to the called function to override.
OMP130	Optimization	Removing unused state machine from generic-mode kernel.
OMP131	Optimization	Rewriting generic-mode kernel with a customized state machine.
OMP132	Analysis	Generic-mode kernel is executed with a customized state machine that requires a fallback.
OMP133	Analysis	Call may contain unknown parallel regions. Use <code>__attribute__((assume("omp_no_parallelism")))</code> to override.
OMP140	Analysis	Could not internalize function. Some optimizations may not be possible.
OMP150	Optimization	Parallel region merged with parallel region at <location>.
OMP160	Optimization	Removing parallel region with no side-effects.
OMP170	Optimization	OpenMP runtime call <call> deduplicated.
OMP180	Optimization	Replacing OpenMP runtime call <call> with <value>.
OMP188	Optimization	Redundant barrier eliminated. (device-opts)

```
p simd  
< j; i++)  
tr[i] * G_ptr[i]
```

operations

```
__attribute__((noescape)) void *);
```

```
};
```

```
declare target(foo)
```

```
omp -O2
```

```
captured in call. Mark
```

Multi-architecture Binaries

- LLVM now supports compiling for many architectures
 - Allows the same binary to run on several machines
- Without `--fopenmp-targets` we will try to infer the triples

```
$ clang app.c -fopenmp -fopenmp-targets=nvptx64,amdgc -c \  
  -Xopenmp-target=nvptx64 --offload-arch=sm_80          \  
  -Xopenmp-target=amdgc --offload-arch=gfx90a  
$ clang app.c -fopenmp --offload-arch=sm_80 --offload-arch=gfx90a -c  
$ llvm-readelf -S app.o  
Section Headers:  
[Nr] Name                Type                Address            Off    Size   ES Flg Lk Inf Al  
[11] .llvm.offloading        LLVM_OFFLOADING    0000000000002058  002058 0024c0 00 E  0  0  8  
[12] omp_offloading_entries  PROGBITS           0000000000005048  004048 000020 00 A  0  0  8
```

Multi-architecture Binaries

Can inspect the embedded device code with binary utils

```
$ clang app.c -fopenmp --offload-arch=sm_80 --offload-arch=gfx90a -o app
$ llvm-objdump --offloading ./app
OFFLOADING IMAGE [0]:
kind elf
arch gfx90a
triple amdgcn-amd-amdhsa
producer openmp

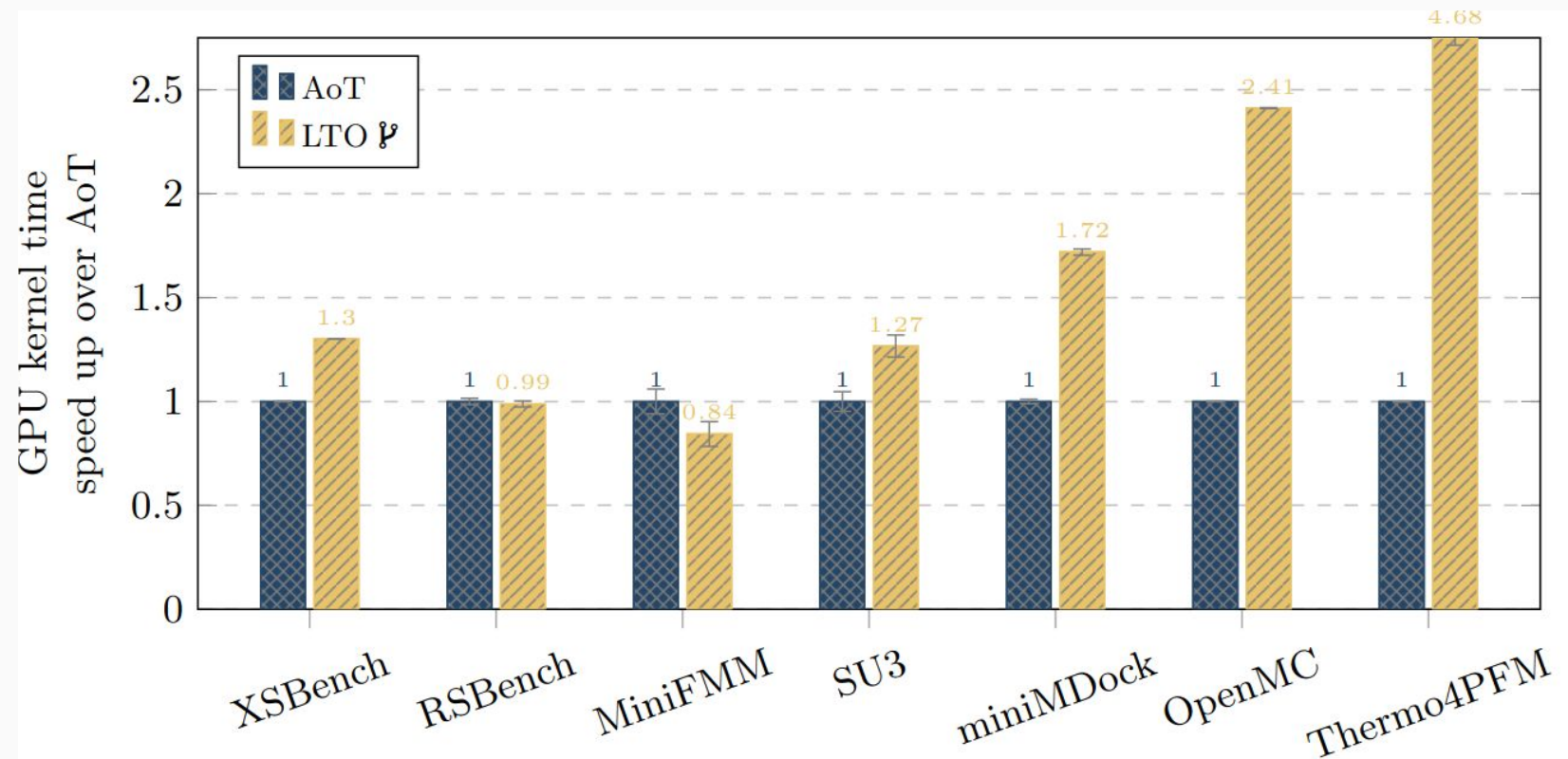
OFFLOADING IMAGE [1]:
kind elf
arch sm_80
triple nvptx64-nvidia-cuda
producer openmp
```

Link Time Optimization (LTO)

- Compilers normally optimize a single translation unit (TU) at a time
 - LTO allows the compiler to optimize the whole program
- LLVM now supports LTO for the device
- Currently needs to be specified for both

```
$ clang app.c -fopenmp -fopenmp-targets=nvptx64 -foffload-lto -O3 -c  
$ clang app.o -fopenmp -fopenmp-targets=nvptx64 -foffload-lto -O3
```

LTO Performance Improvement (A100 Nvidia GPU)



Static Library Support

- LLVM now completely supports static libraries
 - Any method of creating static libraries should work now
- The linker only imports used symbols from static libraries
 - Somewhat inherit this behaviour for multi-architecture binaries

```
$ clang foo.c -fopenmp --offload-arch=sm_70 --offload-arch=sm_80 --offload-arch=gfx908 -c
$ llvm-ar rcs libfoo.a foo.o
$ clang app.c -fopenmp --offload-arch=sm_70 -lfoo -o app
$ llvm-objdump --offloading
OFFLOADING IMAGE [0]:
kind elf
arch sm_70
triple nvptx64-nvidia-cuda
producer openmp
```

Static Library Support

Can use this to create generic libraries,
with LTO -> zero runtime overhead

```
#pragma omp begin declare target device_type(nohost)

#pragma omp begin declare variant match(...)
void foo() {...}
#pragma omp end declare variant

#pragma omp end declare target
```

```
$ clang device.c -c -fopenmp --offload-arch=sm_52,sm_70,sm_80,gfx908,gfx90a,gfx90c -O3 \
    -foffload-lto -fvisibility=hidden -fopenmp-cuda-mode
$ llvm-ar rcs libdevice.a device.o
$ clang app.c -fopenmp --offload-arch=sm_80 -foffload-lto -ldevice
```

CUDA / HIP Interoperability

- The new driver can compile both CUDA and HIP
 - Requires explicitly using the new Driver
- LLVM now supports CUDA compilation in RDC-mode
 - Previously required external build systems

```
$ clang++ cuda.cu util.cu -fgpu-rdc --offload-arch=sm_70 --offload-new-driver -c  
$ clang++ cuda.o util.o --offload-link -lcudart -o app  
$ ./a.out
```

CUDA / HIP Interoperability

OpenMP interoperability with CUDA/HIP

- Caveat: Global state is not yet shared; would require having state registered by OpenMP *or* CUDA

```
void openmp() { printf ("Hello from OpenMP\n"); }  
#pragma omp declare target device_type(nohost) to(openmp)
```

```
__device__ cuda() { printf ("Hello from CUDA\n"); }
```

```
$ clang++ cuda.cu -fgpu-rtc --offload-arch=sm_70 --offload-new-driver -c  
$ clang++ openmp.cpp -fopenmp --offload-arch=sm_70 -c  
$ clang++ cuda.o openmp.o -fopenmp -fopenmp-targets=nvptx64 -lcudart  
./a.out  
Hello from OpenMP  
Hello from CUDA
```

Device Only Compilation

- Device only compilation to output the device code
- Mainly useful for inspecting output

```
$ clang app.c -fopenmp --offload-arch=sm_70 -S -emit-llvm --offload-device-only -o -  
< LLVM IR >
```

Mandatory Offloading

- OpenMP offloading supports host-fallback by default
- This requires emitting each device function on the host
- Can be disabled using a command line flag
 - Makes interoperability with CUDA easier.

```
$ clang app.c -fopenmp --offload-arch=sm_70 -fopenmp-offload-mandatory
```

LLVM 15 has been released and contains various new offloading features, incl.

- A new compiler driver for offloading (OpenMP, CUDA, HIP)
- Multi-Architecture binaries
- Link Time Optimization
- Static Library Support
- OpenMP and CUDA / HIP interoperability
- Extra flags improving offloading performance

Per-Kernel Dynamic Shared Memory

- Dynamic shared memory allocated by the kernel (CUDA/HIP style)
 - Size determined at runtime
 - Works on AMD and NVIDIA GPUs
 - See <https://openmp.llvm.org/design/Runtimes.html#libomptarget-dynamic-shared>

```
#pragma omp target teams ... ompx_dyn_cgroup_mem(4 * N)
{
    float *SharedBuffer = llvm_omp_target_dynamic_shared_alloc()
    compute(A, SharedBuffer, N);
}
```


(Basic) GPU JIT

- Optimize and finalize GPU images (stored as IR) at runtime

```
$ clang app.c -fopenmp --fopenmp-target-jit --offload-arch=native -O3 -foffload-lto  
$ LIBOMPTARGET_JIT_OPT_LEVEL=2 ./a.out
```

- Inspect, modify, and replace the image (IR or object file)
 - See <https://openmp.llvm.org/design/Runtimes.html#libomptarget-jit-replacement-object>

```
$ LIBOMPTARGET_JIT_PRE_OPT_IR_MODULE=pre_jit_opt.ll ./a.out  
$ LIBOMPTARGET_JIT_POST_OPT_IR_MODULE=post_jit_opt.ll ./a.out  
$ LIBOMPTARGET_JIT_REPLACEMENT_MODULE=modified_ir.ll ./a.out  
$ LIBOMPTARGET_JIT_REPLACEMENT_OBJECT=image.o ./a.out
```

Record and Replay for Kernels

- Extract kernels for standalone execution
 - Composes with JIT
 - Allows isolated debugging, profiling, (manual) tuning, ...
 - Certain features are only available after 16, e.g., usage of globals
 - **See:** <https://github.com/llvm/llvm-project/blob/llvmorg-16.0.0-rc1/openmp/docs/ReleaseNotes.rst>

```
$ LIBOMPTARGET_RECORDING=True LIBOMPTARGET_RR_SAVE_OUTPUT=True ./a.out  
$ llvm-omp-kernel-replay <kernel_recording_file.json>
```

LLVM 16 has been released and contains various new offloading features, incl.

- Asynchronous AMD GPU offloading
- JIT support (-fopenmp-target-jit)
- Record and Replay
- --offload-arch=native
- Performance improvements

Work in Progress — Selection

- OpenMP equivalent to “launch bounds” (CUDA) or kernel attributes, e.g., “waves-per-eu”, (AMD)
- Portability layer to access native functionality (e.g., Thrust, or shuffle)
- “Bare-metal” kernels → “CUDA-style kernel programming” (w/o overhead)
- New (=fast) GPU reductions
- Libc and libc++ support for GPUs
- Intel (ARM, Apple) GPU support
- 3-level parallelism (aka. “Proper” SIMD support)
- “omp loop” support

LLVM/OpenMP Device Info

llvm-omp-device-info

A command line utility that, by using `libomptarget`, and the device plugins, list devices information as seen from the OpenMP Runtime.

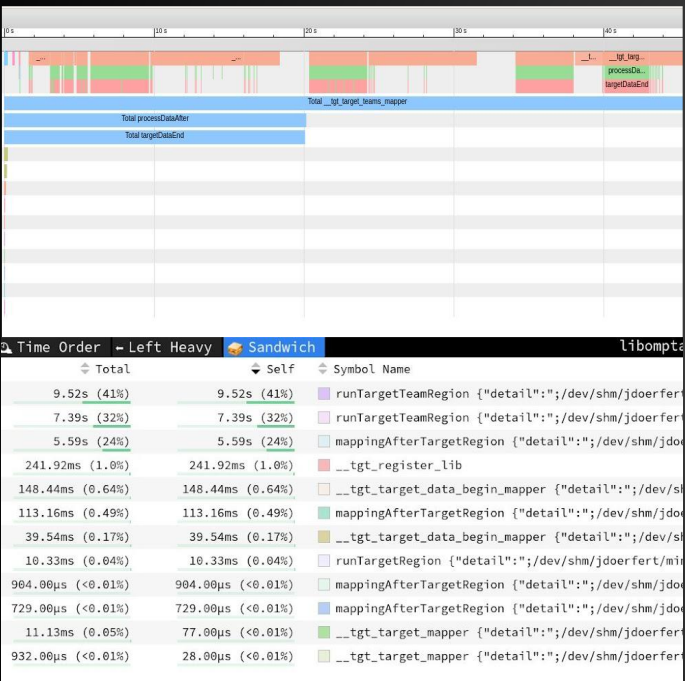
Credit to *Jose Monsalve Diaz* (ANL).

```
→ ./llvm-omp-device-info
Device (0):
  print_device_info not implemented
...

Device (4):
  CUDA Driver Version:      11040
  CUDA Device Number:      0
  Device Name:              NVIDIA GeForce RTX 2080
  Global Memory Size:      4294967295 bytes
  Number of Multiprocessors: 46
  Concurrent Copy and Execution: Yes
  Total Constant Memory:   65536 bytes
  Max Shared Memory per Block: 49152 bytes
  Registers per Block:     65536
  Warp Size:               32 Threads
  Maximum Threads per Block: 1024
  Maximum Block Dimensions: 1024, 1024, 64
  Maximum Grid Dimensions: 2147483647 x 65535 x 65535
  Maximum Memory Pitch:    2147483647 bytes
  Texture Alignment:       512 bytes
  Clock Rate:              1710000 kHz
  Execution Timeout:       No
  Integrated Device:       No
  Can Map Host Memory:     Yes
  Compute Mode:            DEFAULT
  Concurrent Kernels:      Yes
  ECC Enabled:             No
  Memory Clock Rate:       7000000 kHz
  Memory Bus Width:        256 bits
  L2 Cache Size:           4194304 bytes
  Max Threads Per SMP:     1024
  Async Engines:           Yes (3)
  Unified Addressing:      Yes
  Managed Memory:          Yes
  Concurrent Managed Memory: Yes
  Preemption Supported:    Yes
  Cooperative Launch:      Yes
  Multi-Device Boars:      No
  Compute Capabilities:    75
```

LLVM/OpenMP Target Profiling

Chrome Profiling Traces



LLVM 12 introduced

`LIBOMPTARGET_PROFILE=file.json`

to portably track target interaction.

Chrome tracing format, source line information, ...

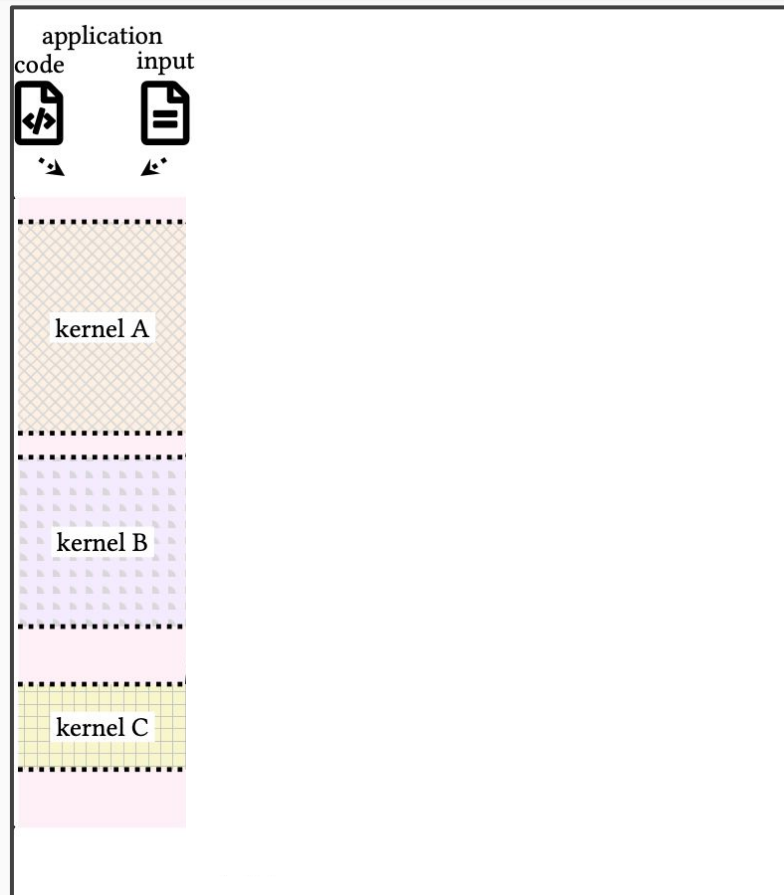
<https://openmp.llvm.org/docs/design/Runtimes.html#libomptarget-profile>

**Basic kernel profiling PoC available,
including user-defined regions!**

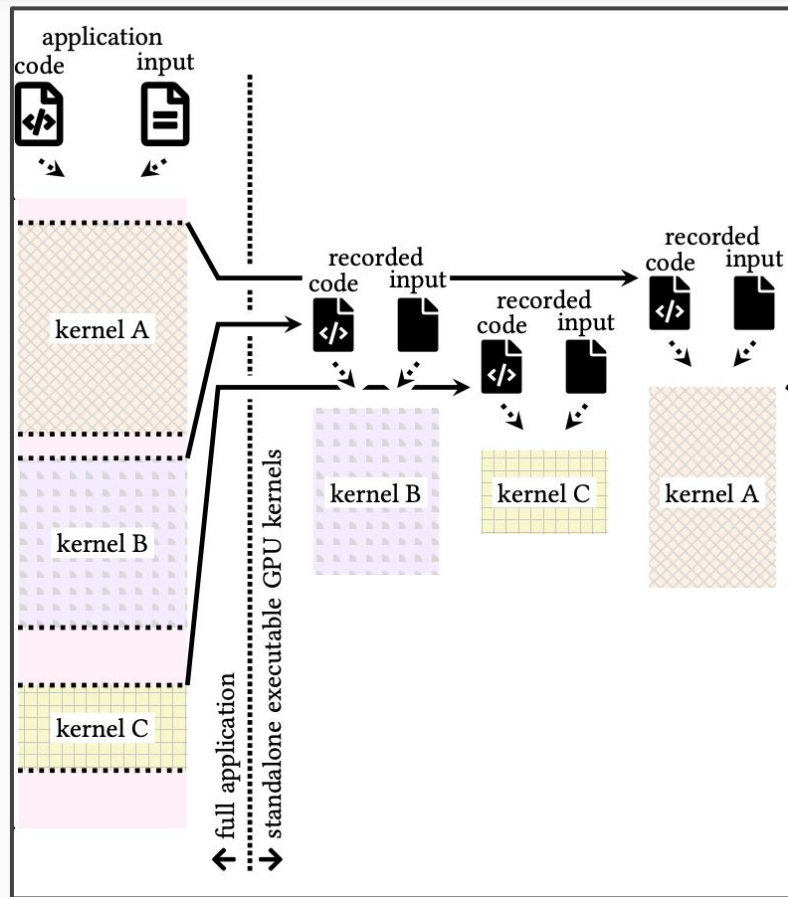
Credit to Giorgis Georgakoudis (LLNL) and Neeraj Rajesh (IIT).

LLVM/OpenMP Record-and-Replay

llvm-omp-kernel-replay

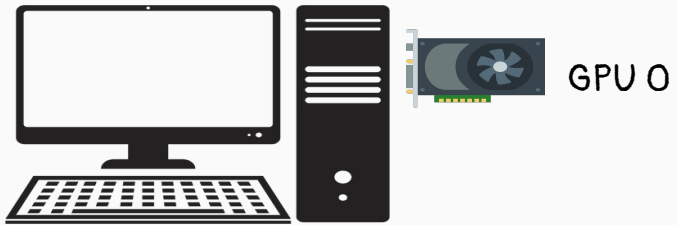


llvm-omp-kernel-replay

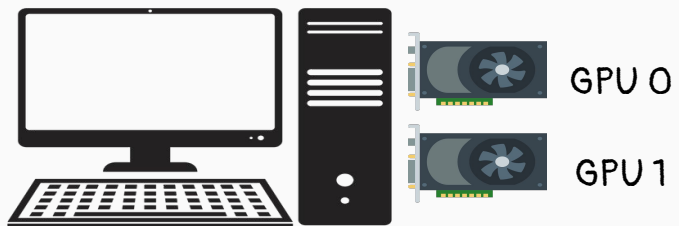


Remote OpenMP Offloading

Remote OpenMP Offloading (Plugin)



Remote OpenMP Offloading (Plugin)

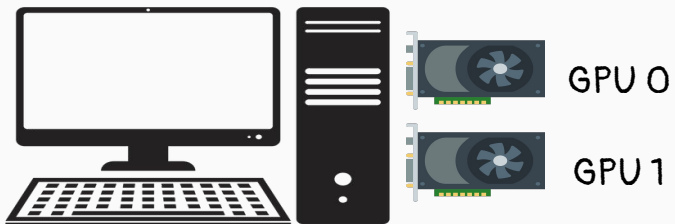


Remote OpenMP Offloading (Plugin)

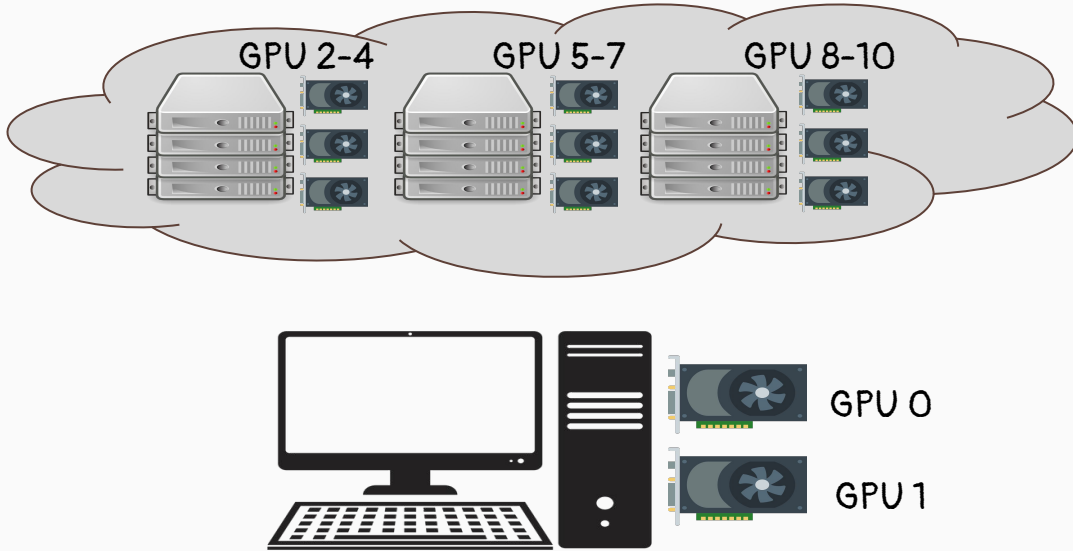
```
#pragma omp parallel for num_threads(num_devices)
for (auto K = 0; K < num_devices; K++) {
    #pragma omp target ... device(K)
    for (auto i = 0; i < lookups_per_device; i++) {

        ...

    }
}
```

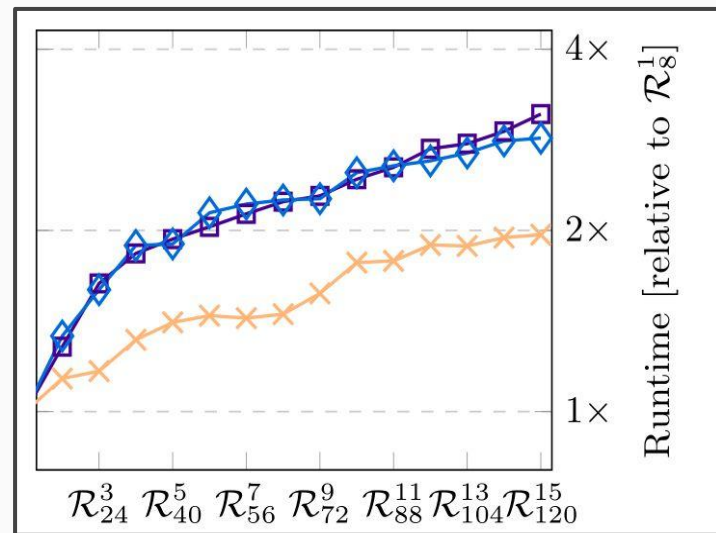
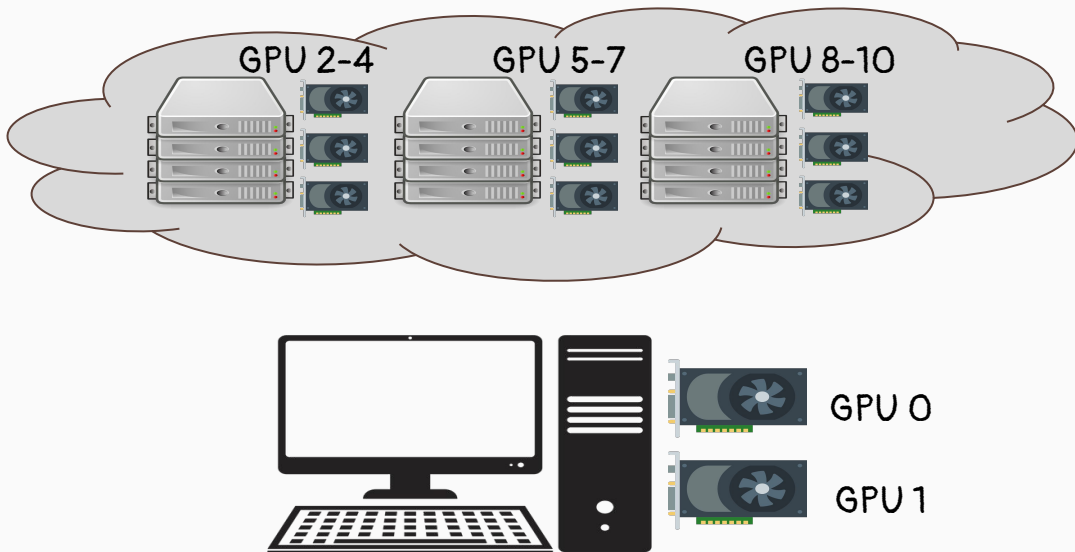


Remote OpenMP Offloading (Plugin)



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

Remote OpenMP Offloading (Plugin)



XSbench - scaling up to
15x8 A100 GPUs compared to 1x8

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

Not OpenMP

ORAQL - Perfect Alias Analysis Queries

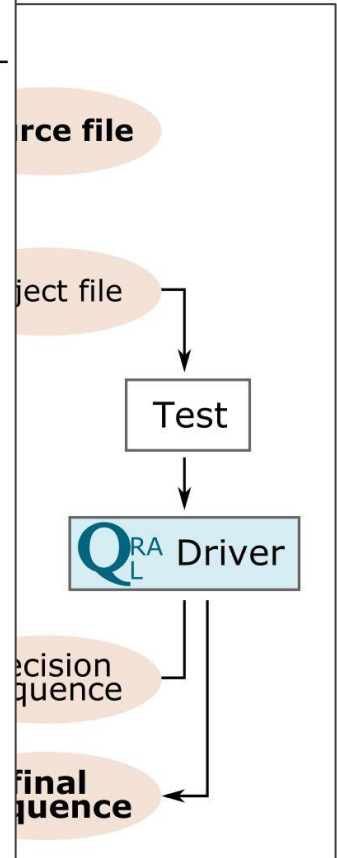
1) Explore a
local opt

2) Determin
and ident

3) Improve
situation

Jan Hueckelh
--- See o

Benchmark	Programming Model	ORAQL speedup
TestSNAP	C++	3.6%
TestSNAP	C++, OpenMP	1.0%
TestSNAP	C++, Kokkos, CUDA	0
TestSNAP	Fortran	5%
XSbench	all versions	0
GridMini	C++, OpenMP Offload	-7%
Quicksilver	C++, OpenMP	[inconclusive]
LULESH	C++	0.8%
LULESH	C++, OpenMP	1.4%
LULESH	C++, MPI	-0.2%
MiniFE	C++, OpenMP	0
MiniGMG	C, SSE intrinsics	3.4%
MiniGMG	C, OpenMP tasks	1%
MiniGMG	C, OpenMP	8.3%



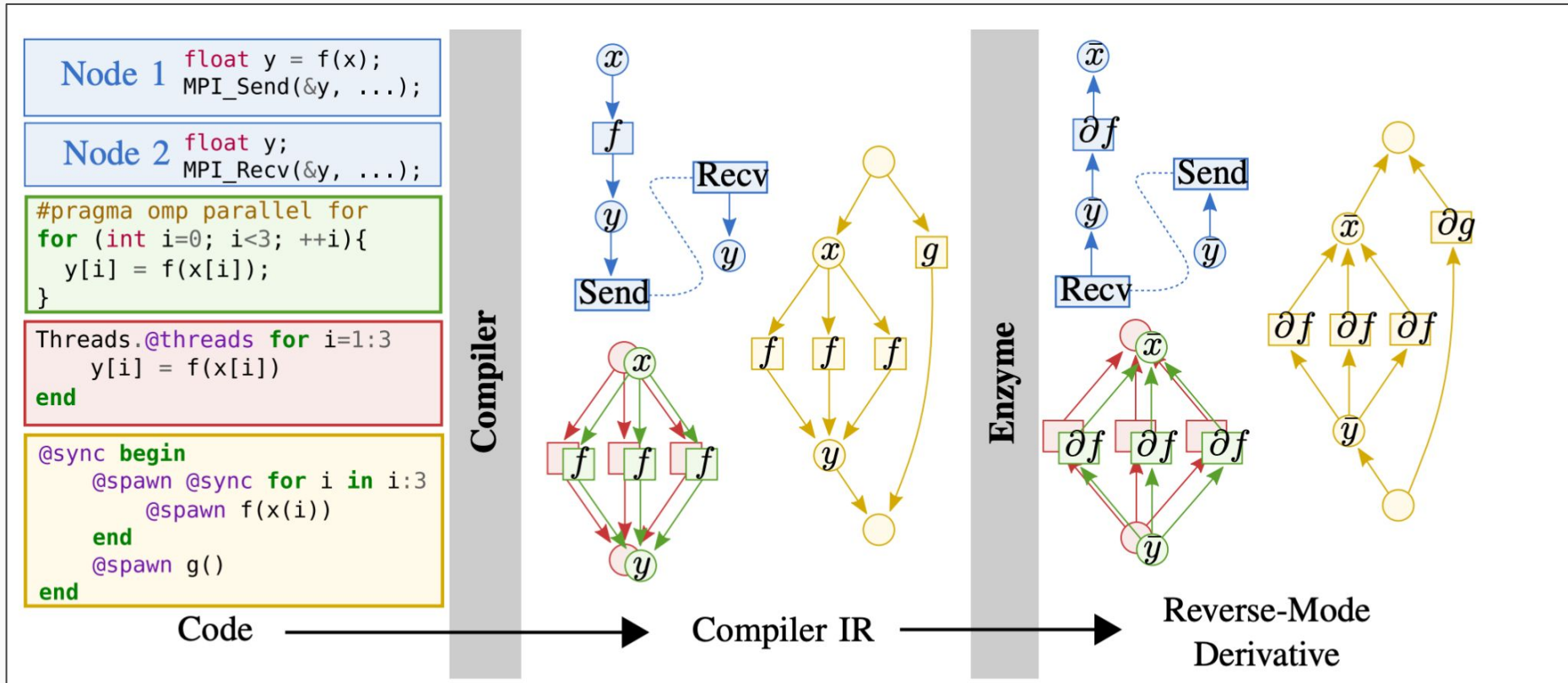
Automatic Differentiation (AD) without Python/ML/JAX*

* Or other domain specific languages (DSLs).



Enzyme AD

Automatic Compiler-Based Differentiation of HPC Codes

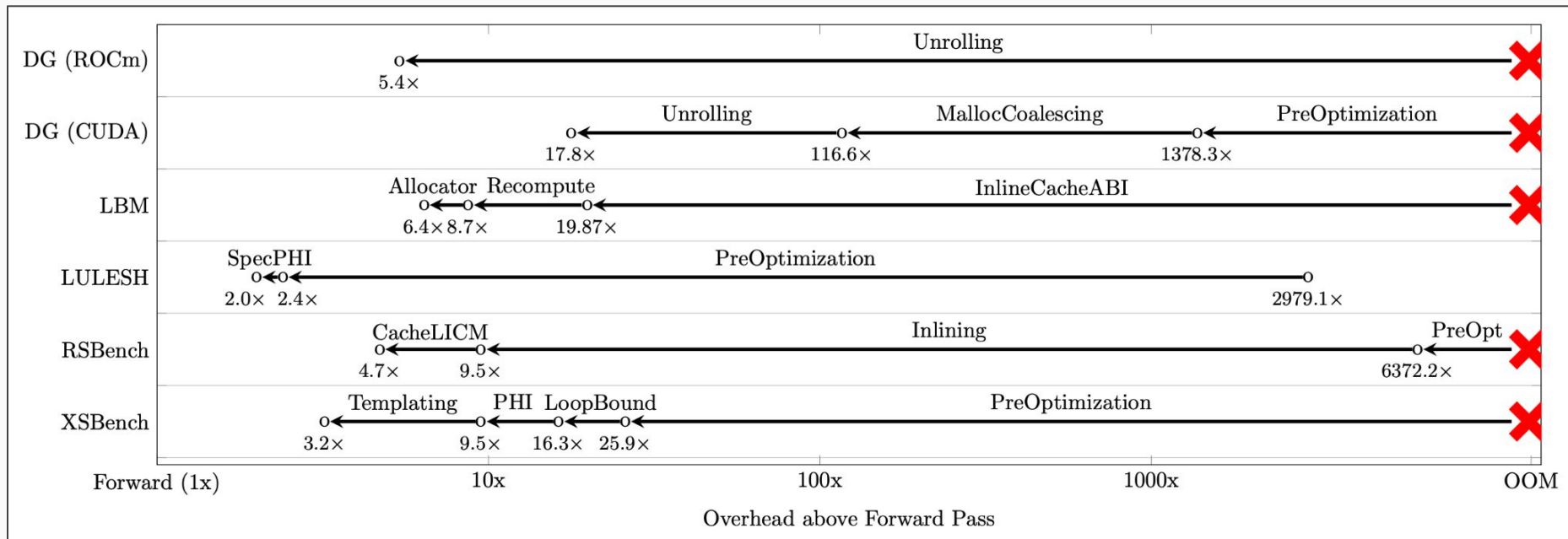


Moses, William, and Valentin Churavy. "Instead of rewriting foreign code for machine learning, automatically synthesize fast gradients." Advances in Neural Information Processing Systems (NeurIPS). 2020



Enzyme AD

Automatic Compiler-Based Differentiation of HPC Codes

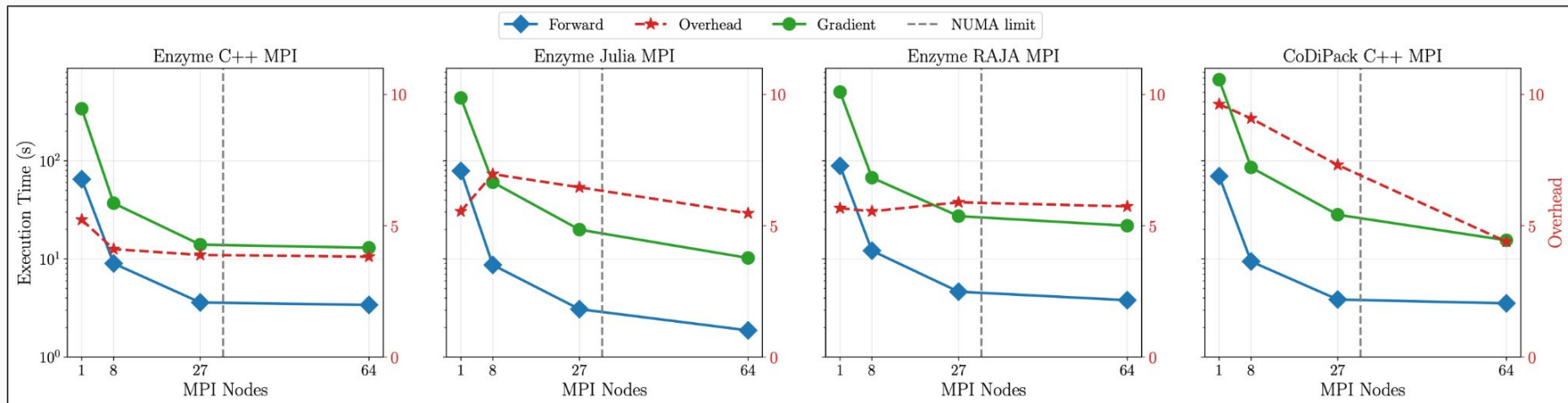


Moses, William S., et al. "Reverse-mode automatic differentiation and optimization of GPU kernels via Enzyme." High Performance Computing, Networking, Storage and Analysis (SC). 2021



Enzyme AD

Automatic Compiler-Based Differentiation of HPC Codes



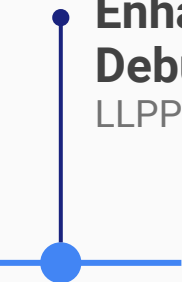
Moses, William S., et al. "Scalable automatic differentiation of multiple parallel paradigms through compiler augmentation." High Performance Computing, Networking, Storage and Analysis (SC). 2022

AMA: Ask Me Anything



Brief Recap & Outlook

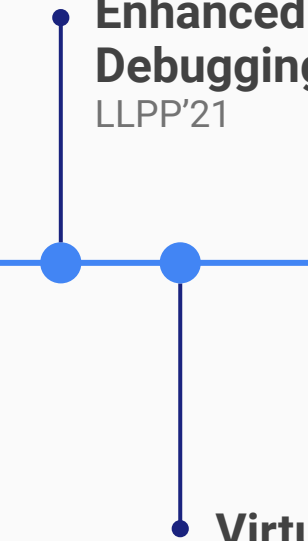
Brief Recap



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

**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

**Virtual and
Remote GPU Offloading**

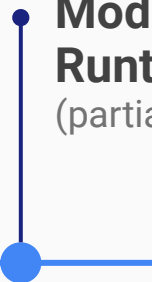
LLPP'21, ISC'22 (best paper)

**Co-Designed Opt. &
Portable GPU Runtime**

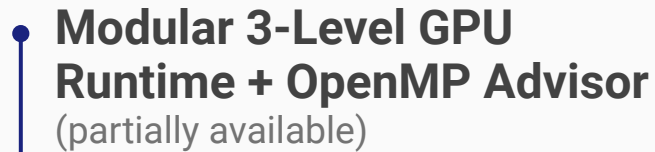
IPDPS'22

**Fully Portable and Inter-
operable GPU Codes**

PACT'22

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

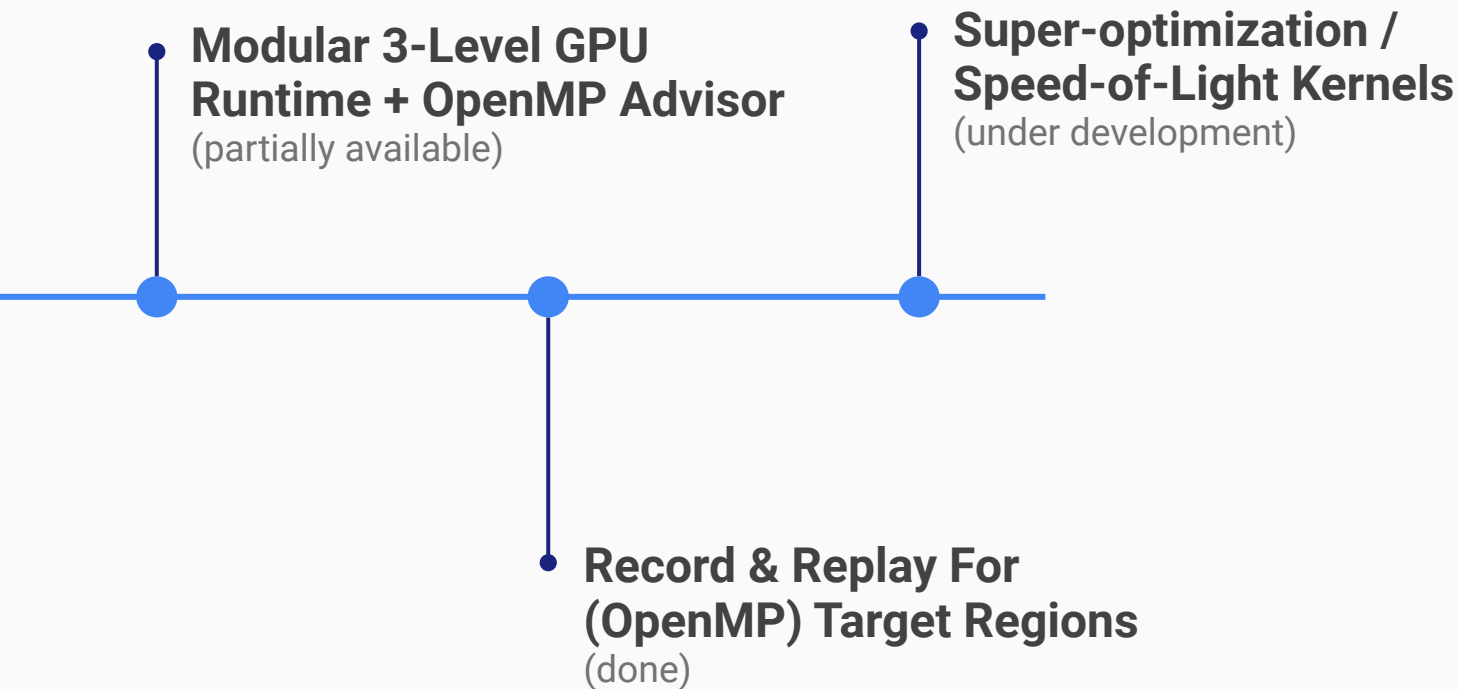
Brief Outlook



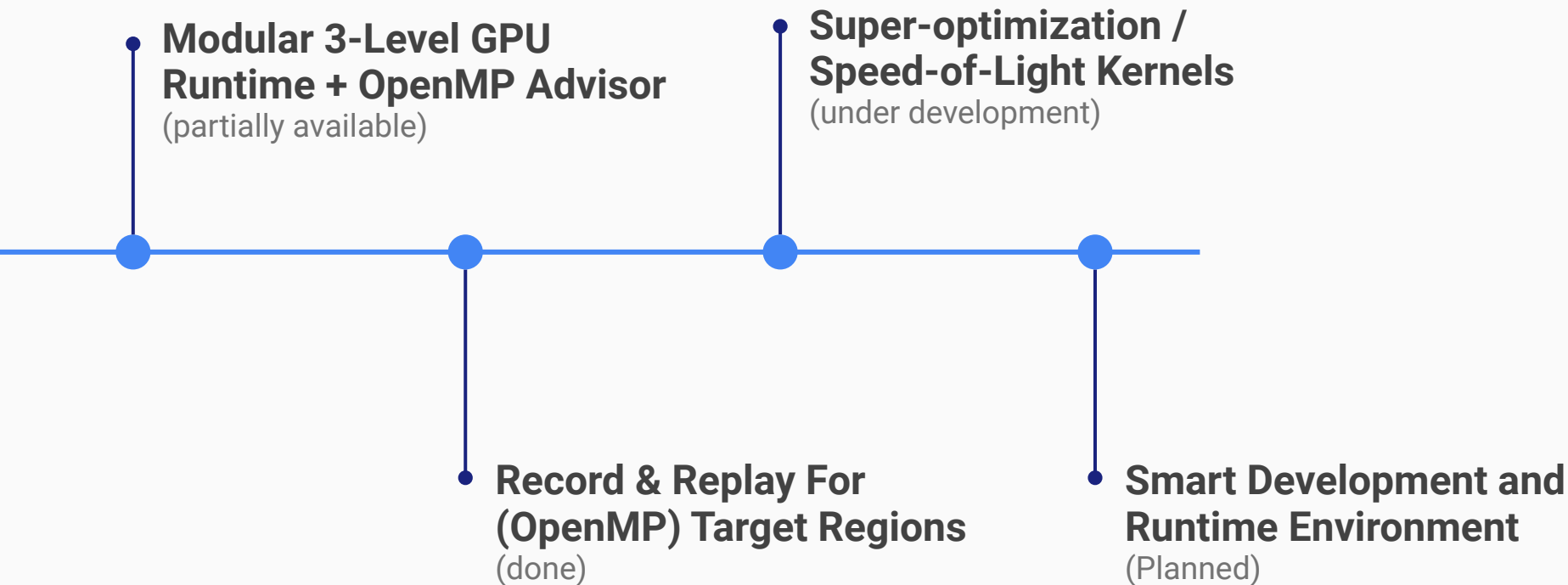
**Modular 3-Level GPU
Runtime + OpenMP Advisor**
(partially available)

**Record & Replay For
(OpenMP) Target Regions**
(done)

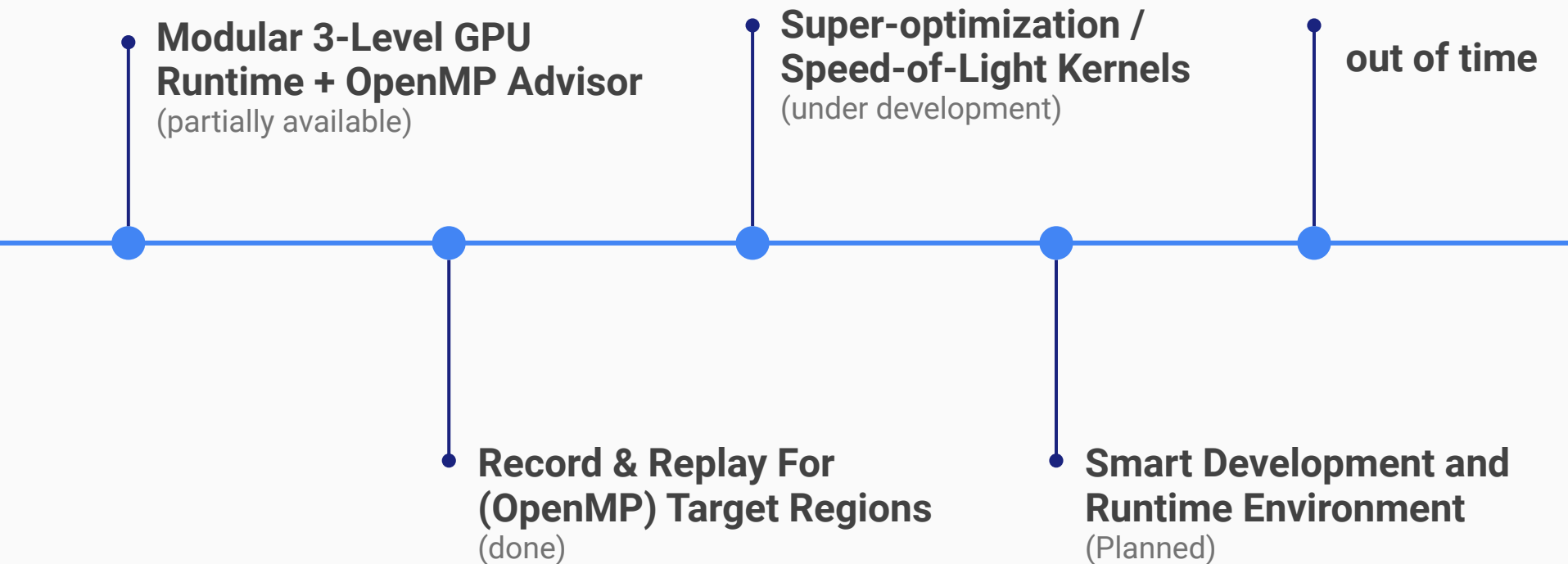
Brief Outlook



Brief Outlook

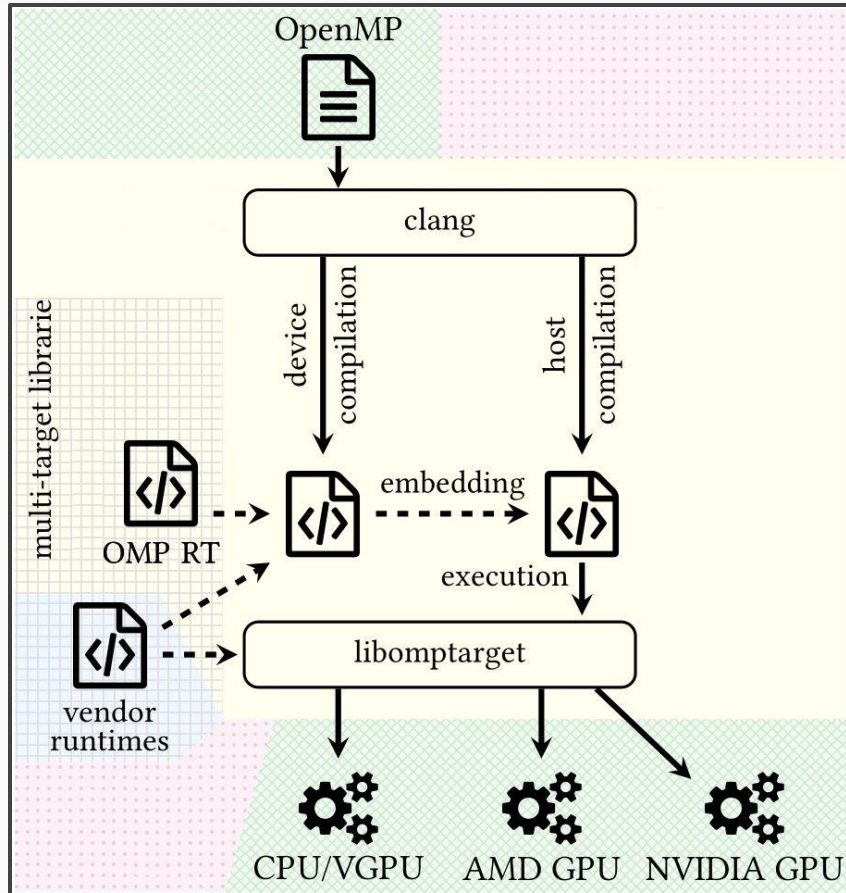


Brief Outlook



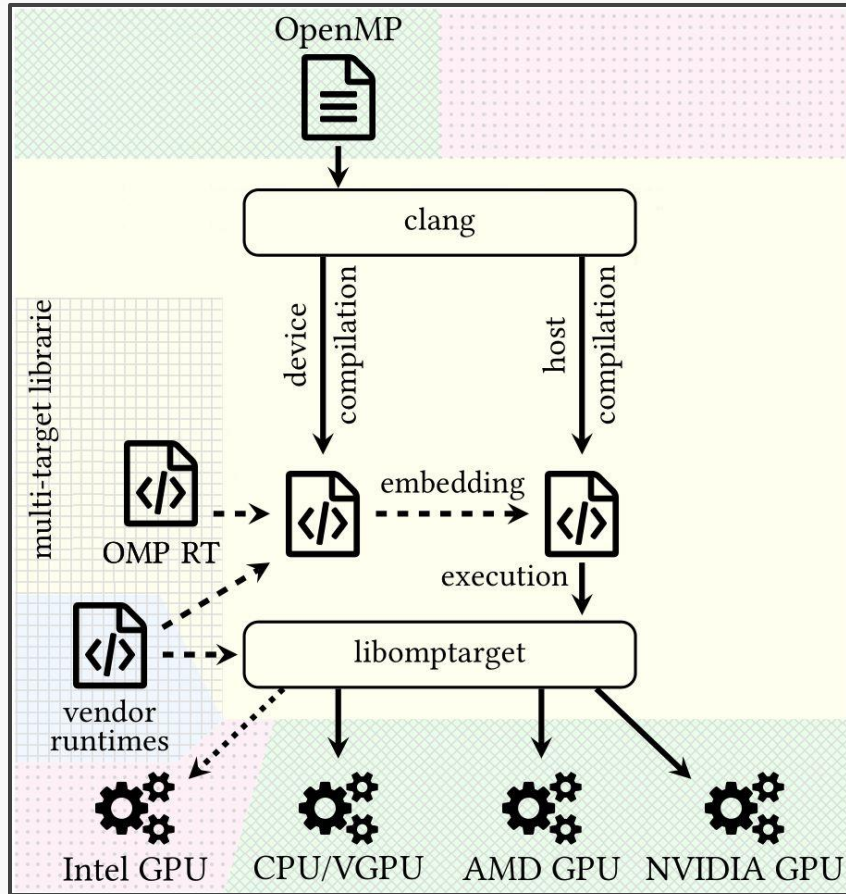
OpenMP as Intermediate Layer

LLVM/OpenMP Target Offloading



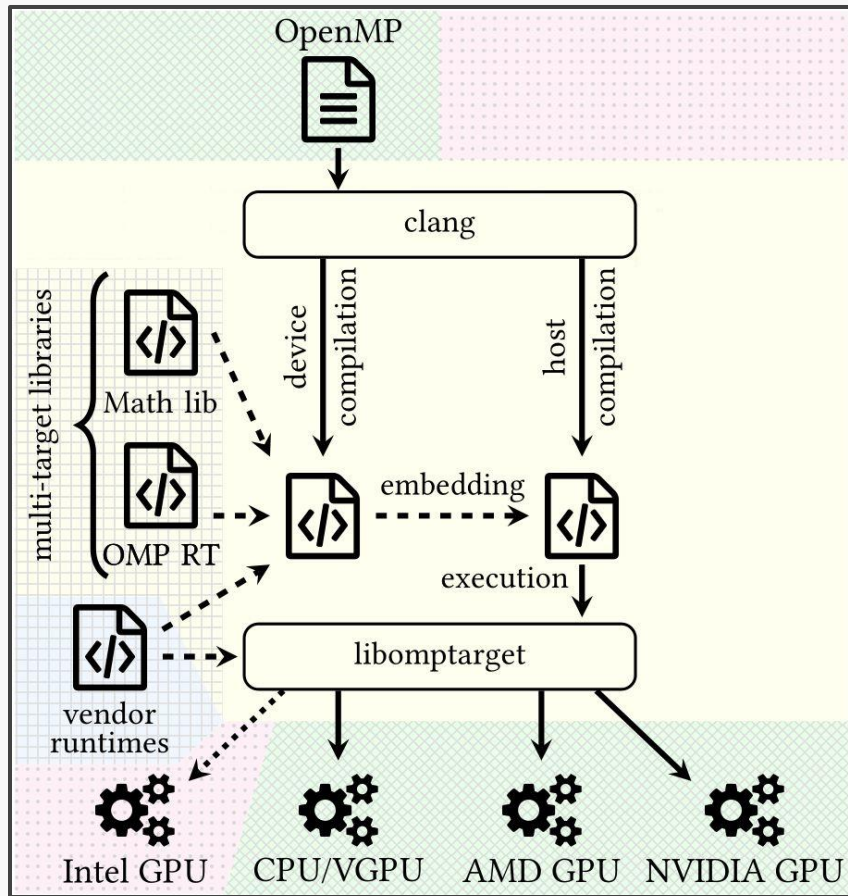
- OpenMP offload code compilation for CPUs, virtual GPU (VGPU), AMD and NVIDIA GPUs

LLVM/OpenMP Target Offloading



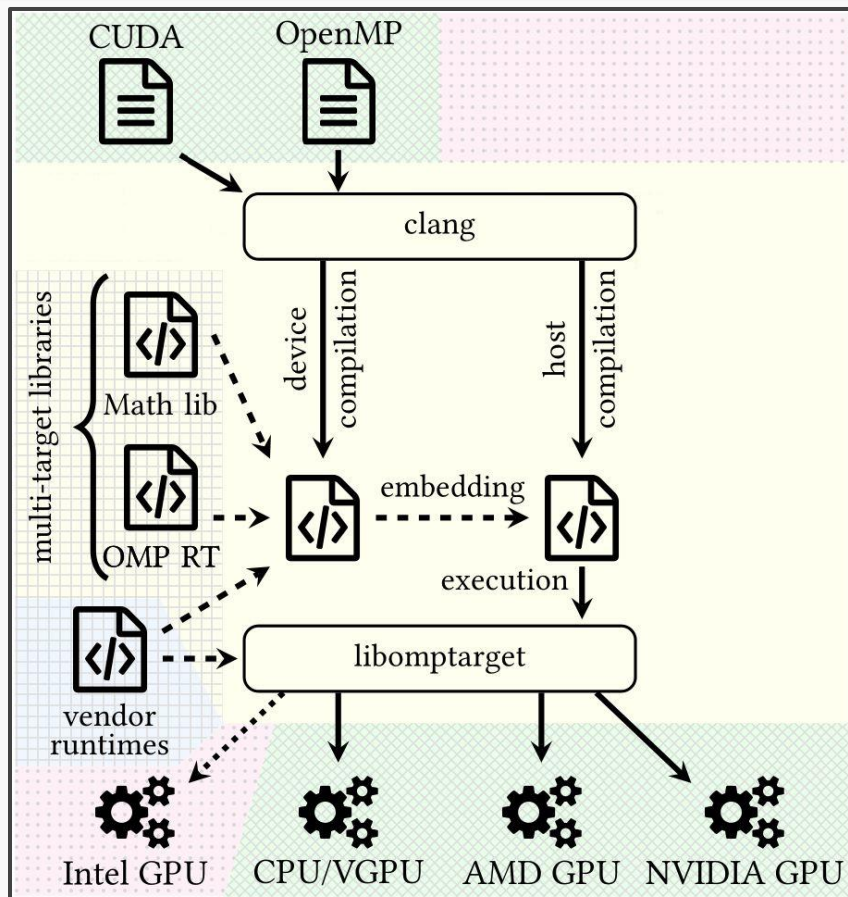
- OpenMP offload code compilation for CPUs, virtual GPU (VGPU), AMD and NVIDIA GPUs
- Intel GPU support is WIP

LLVM/OpenMP Target Offloading + Math Runtimes



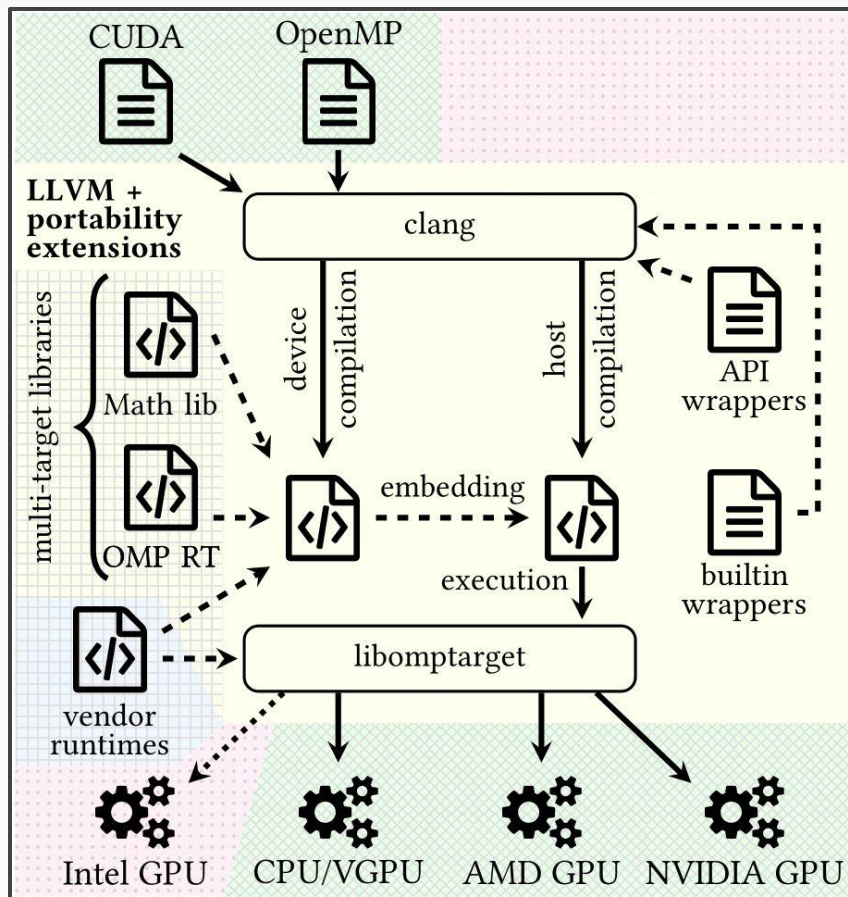
- OpenMP offload code compilation for CPUs, virtual GPU (VGPU), AMD and NVIDIA GPUs
- Intel GPU support is WIP
- Target independent math library (libm.a) for all supported architectures. Defines `sin(...)`, etc.

LLVM/OpenMP Target Offloading + CUDA Device Compilation



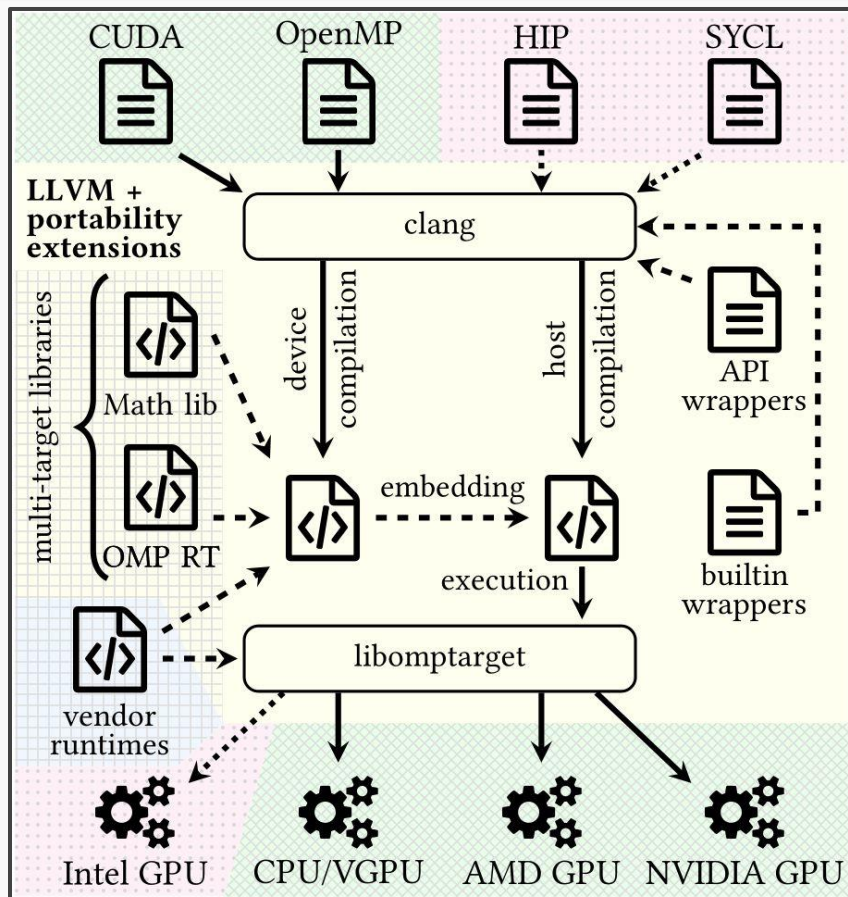
- OpenMP offload code compilation for CPUs, virtual GPU (VGPU), AMD and NVIDIA GPUs
- Intel GPU support is WIP
- Target independent math library (libm.a) for all supported architectures. Defines `sin(...)`, etc.
- CUDA device code interoperability with OpenMP target. Link in CUDA device runtimes e.g., Thrust.

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



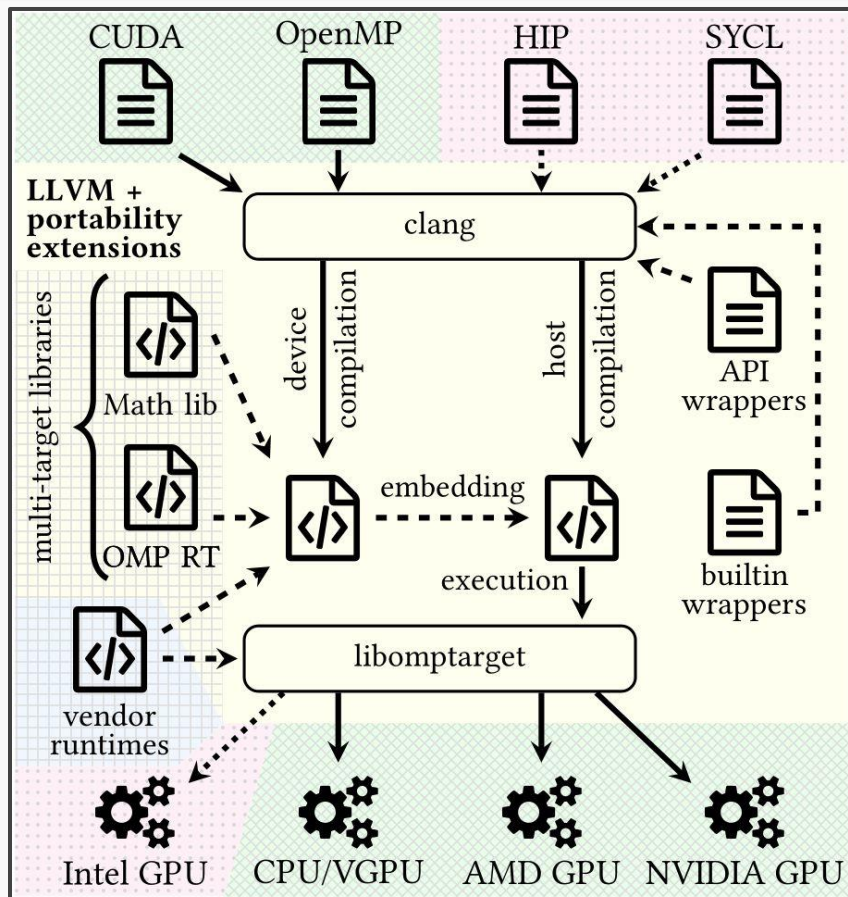
- OpenMP offload code compilation for CPUs, virtual GPU (VGPU), AMD and NVIDIA GPUs
- Intel GPU support is WIP
- Target independent math library (libm.a) for all supported architectures. Defines `sin(...)`, etc.
- CUDA device code interoperability with OpenMP target. Link in CUDA device runtimes e.g., Thrust.
- Define CUDA API and builtins through OpenMP runtime functions. Allow to retarget CUDA codes.

LLVM/OpenMP as Target Independent Runtime Layer (WIP)



- OpenMP offload code compilation for CPUs, virtual GPU (VGPU), AMD and NVIDIA GPUs
- Intel GPU support is WIP
- Target independent math library (libm.a) for all supported architectures. Defines `sin(...)`, etc.
- CUDA device code interoperability with OpenMP target. Link in CUDA device runtimes e.g., Thrust.
- Define CUDA API and builtins through OpenMP runtime functions. Allow to retarget CUDA codes.
- HIP, SYCL, and other languages can be added as needed. Full interoperability and portability.

LLVM/OpenMP as Target Independent Runtime Layer (WIP)



- OpenMP offload code compilation for CPUs, virtual GPU (VGPU), AMD and NVIDIA GPUs
- Intel GPU support is WIP
- Target independent math library (libm.a) for all supported architectures. Defines `sin(...)`, etc.
- CUDA device code interoperability with OpenMP target. Link in CUDA device runtimes e.g., Thrust.
- Define CUDA API and builtins through OpenMP runtime functions. Allow to retarget CUDA codes.
- HIP, SYCL, and other languages can be added as needed. Full interoperability and portability.
- Overall WIP but proof-of-concept is ready and under review (PACT) right now. Parts have been upstreamed (incl. Driver) or are prepared to be.

LLVM/OpenMP as Target Independent Runtime Layer (WIP)

```
Thread 17 "su3" hit Breakpoint 1, k_mat_nn (a=0x[...]8b10, b=0x[...]cb20, c=0x[...]cc50, total_sites=256) at ./mat_nn_cuda.hpp:22
```

```
22 int myThread = blockDim.x * blockIdx.x + threadIdx.x;
```

```
(gdb) bt
```

```
#0 k_mat_nn (a=0x[...]8b10, b=0x[...]cb20, c=0x[...]cc50, total_sites=256) at ./mat_nn_cuda.hpp:22
```

```
#1 0x[...]d6dd in ?? () from /usr/lib64/libffi.so.7
```

```
#2 0x[...]9a69 in VGPUty::VGPUty()::{lambda()#2}::operator()() () from [...]lib/libomptarget.rtl.vgpu.so
```

```
(gdb) print myThread
```

```
$2 = 15
```

```
(gdb) next
```

```
25 if (mySite < total_sites) {
```

```
(gdb) cont
```

```
Continuing.
```

```
Thread 17 "su3" hit Breakpoint 2, k_mat_nn (a=<opt out>, b=<opt out>, c=0x[...]cc50, total_sites=256) at ./mat_nn_cuda.hpp:32
```

```
32 CMULSUM(a[mySite].link[j].e[k][m], b[j].e[m][l], cc);
```

```
(gdb) next
```

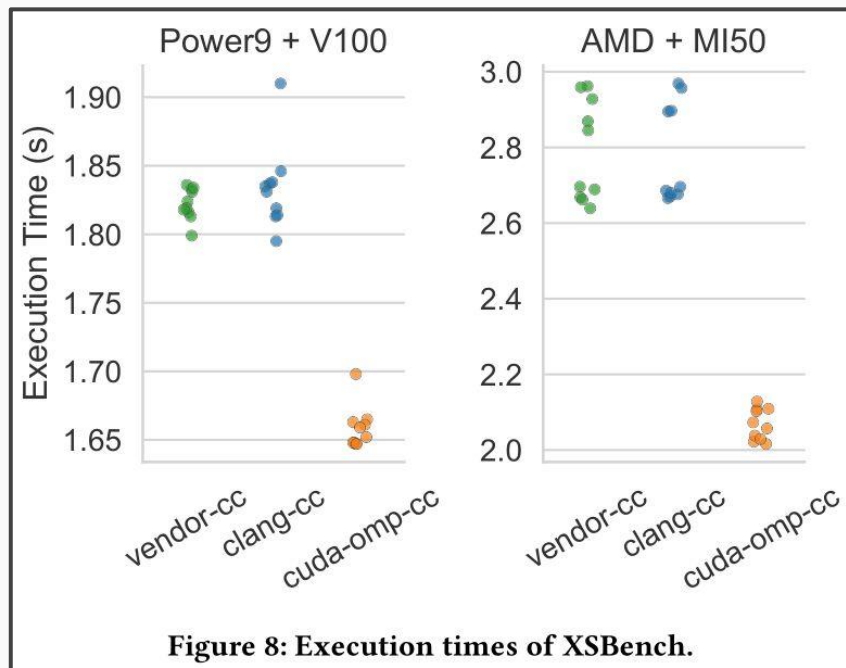
```
36 c[mySite].link[j].e[k][l] = cc;
```

```
(gdb) print cc
```

```
$3 = {real = 1, imag = 0}
```

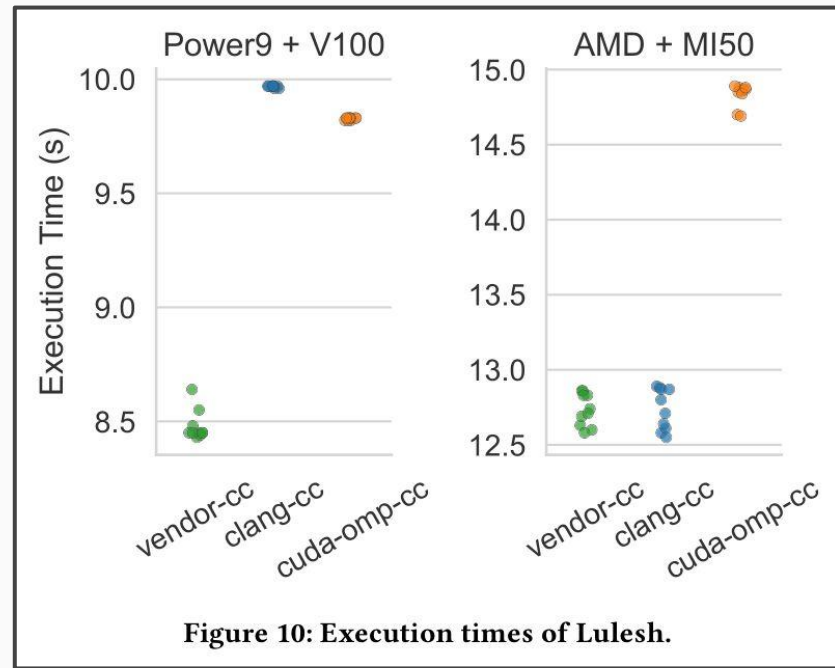
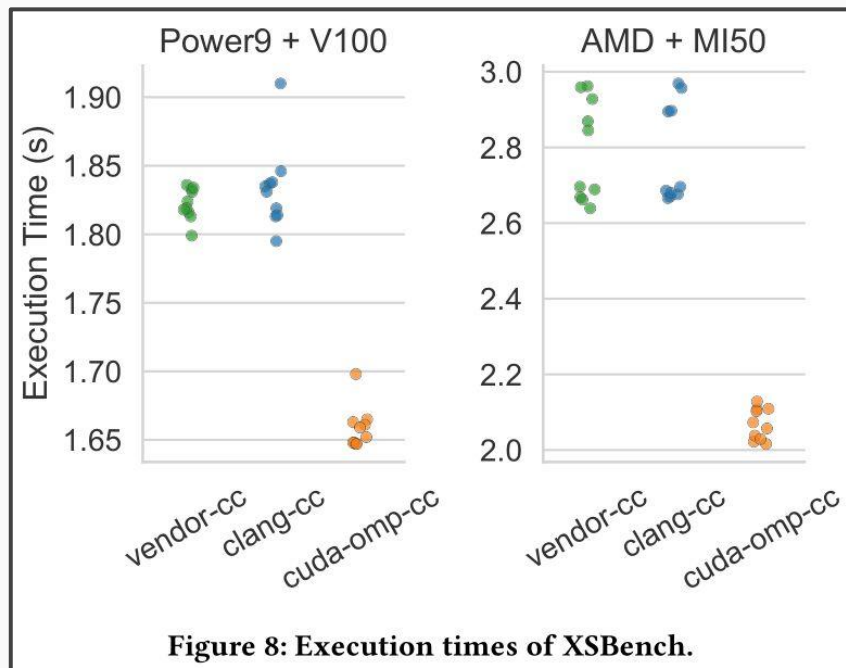
Host GDB running the SU3 bench CUDA code via the OpenMP layer on the virtual GPU.

LLVM/OpenMP as Target Independent Runtime Layer (WIP)



Breaking the Vendor Lock — Performance Portable Programming Through OpenMP as Target Independent Runtime Layer (PACT'22, accepted)

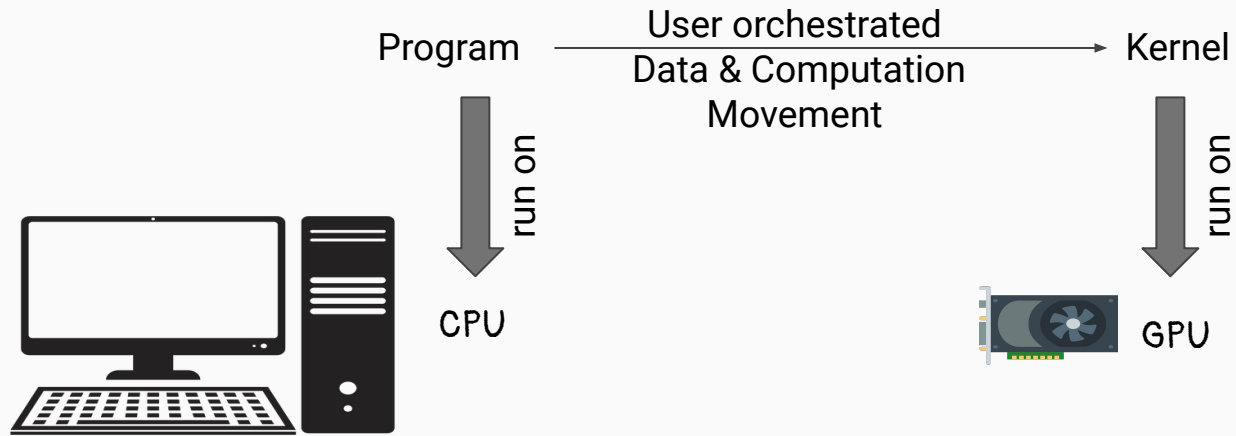
LLVM/OpenMP as Target Independent Runtime Layer (WIP)



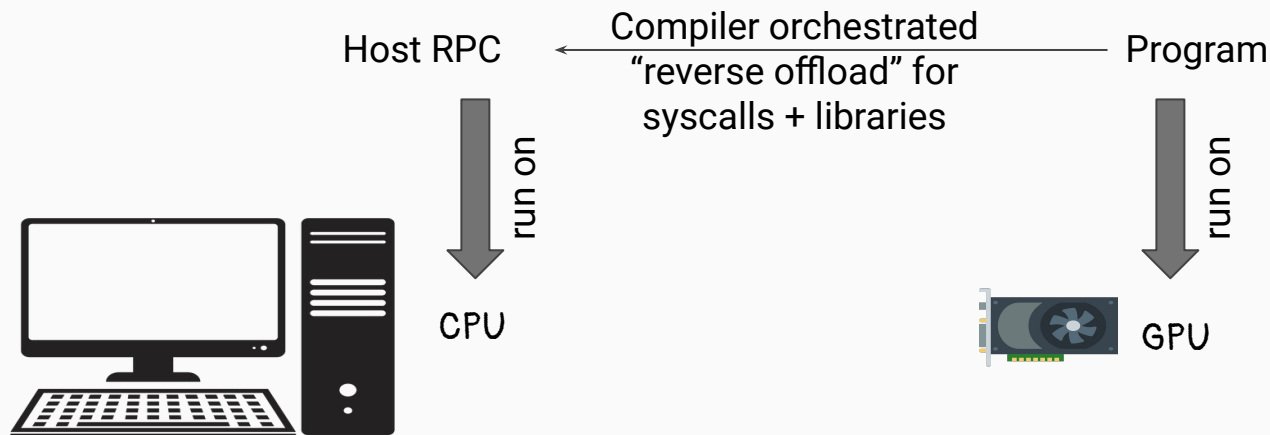
Breaking the Vendor Lock — Performance Portable Programming Through OpenMP as Target Independent Runtime Layer (PACT'22, accepted)

Direct GPU Compilation

Direct GPU offloading



Direct GPU offloading

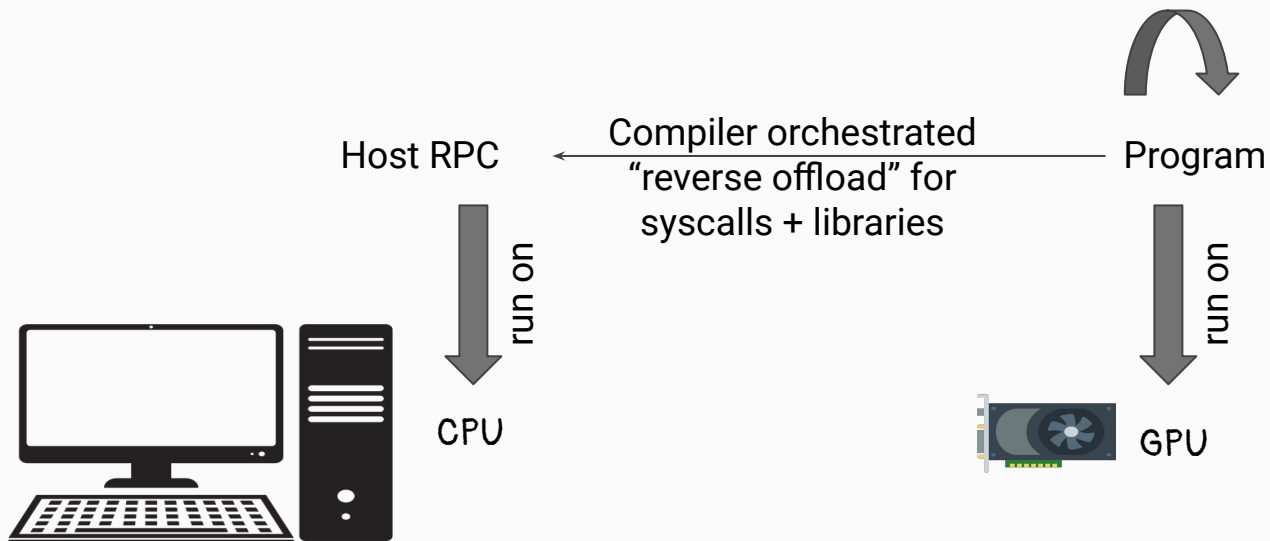


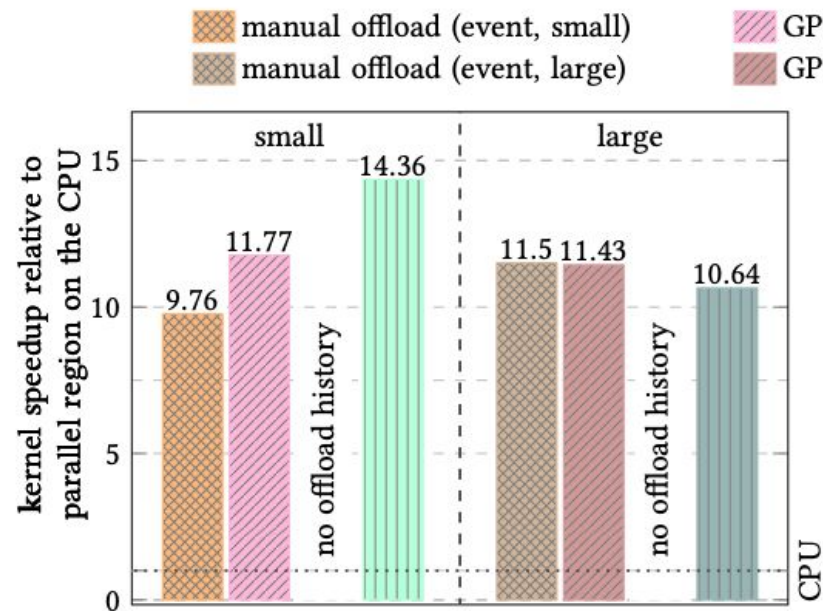
Direct GPU Compilation and Execution for Host Applications with OpenMP Parallelism
(LLVM-HPC'22, accepted)

Direct GPU offloading

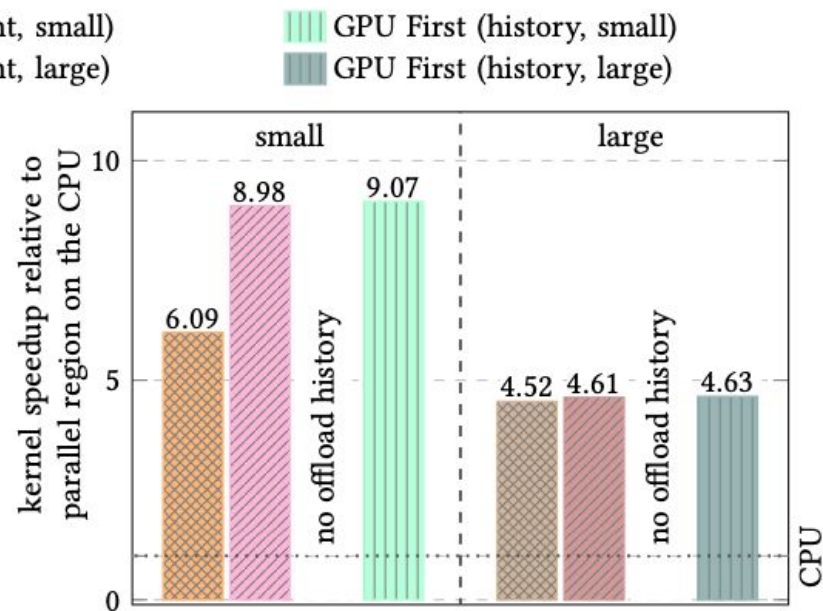
`#omp parallel for` →

`#omp target teams distribute parallel for`





(a) Performance of the compute kernel of XSbench relative to the CPU version.



(b) Performance of the compute kernel of RSbench relative to the CPU version.

Figure 8: Performance of different GPU versions of the OpenMC proxy applications XSbench and RSbench compared to their respective CPU counterpart.