Equipping LLVM/OpenMP with Advanced OpenMP Offloading GPU Features

10th LLVM Performance Workshop

January 31, 2026

Kevin Sala | Postdoctoral Researcher
salapenades1@llnl.gov

Chaitanya Sankisa, Krzysztof Parzyszek, Michael Klemm | AMD Collaborators

Prepared by LLNL under Contract DE-AC52-07NA27344.

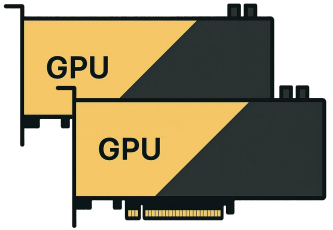
Accelerating with OpenMP

- The *target* model allows **accelerating** regions on devices
  - Most constructs are available within target regions
  - Broad **vendor** support
  - Broad **language** support: C, C++, Fortran, Python (prototype)

OpenMP

```
#pragma omp target teams distribute parallel for \
    num_teams(N/256) thread_limit(256)           \
    map(tofrom: Matrix[0:N*N*N]) collapse(3)
for (int z = 0; z < N; z++)
  for (int y = 0; y < N; y++)
    for (int x = 0; x < N; x++)
      Matrix[x + y*N + z*N*N] *= Val;
```

OpenMP app



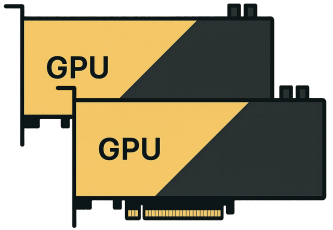
Problems

- The *target* model has **limitations** that hinder broad adoption
  - **Missing** essential GPU-specific features
  - Noticeable **overhead** in target regions (depending on features)
  - Struggle with **hierarchical parallelism**

OpenMP

```
#pragma omp target teams distribute parallel for \
    num_teams(N/256) thread_limit(256)           \
    map(tofrom: Matrix[0:N*N*N]) collapse(3)
for (int z = 0; z < N; z++)
  for (int y = 0; y < N; y++)
    for (int x = 0; x < N; x++)
      Matrix[x + y*N + z*N*N] *= Val;
```

OpenMP app



OpenMP Committee is working on that…

- Recent and upcoming features
  - **Static** *team*-**local** memory → Available OpenMP 6.0
  - **Dynamic** *team*-**local** memory → Approved OpenMP 6.1
  - **Multi-dimensional** *grid* programming → Under discussion
  - Reducing target region overheads → Under discussion
  - *Stream*-like data dependencies → Under discussion
  - Etc.

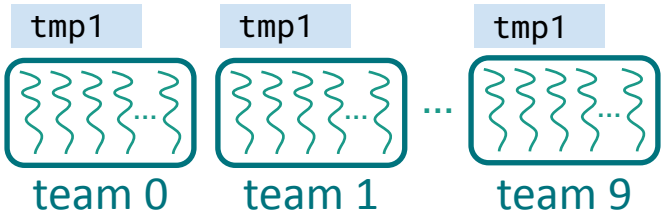
Overview of some recent features

- This talk will cover…
  - **Static** *team*-local memory
  - **Dynamic** *team*-local memory
  - **Multi-dimensional** *grid* programming

- **Overview** of such features
- **Implementation** in LLVM/OpenMP
- Early **evaluation** in two benchmarks

Groupprivate Directive

- Exposes **static shared / local** memory
  - Each *contention group* its **own copy**
  - File-scope, namespace-scope or static block-scope variables



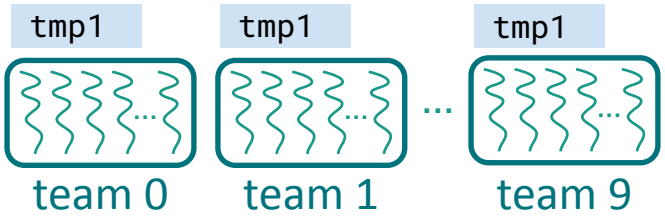
Available OpenMP 6.0

```
void func(int *sum, int tid) {
  static int tmp1[1000];
  #pragma omp groupprivate(tmp1)

  #pragma omp for
  for (int i = 0; i < 1000; i++)
    tmp1[i] = tid + i;

  #pragma omp for reduction(+: sum)
  for (int i = 0; i < 1000; i++)
    sum += tmp1[i];
}

int main() {
  int sums[10];

  #pragma omp target teams num_teams(10) thread_limit(256)
  #pragma omp parallel
  func(&sums[omp_get_team_num()], omp_get_thread_num());
}
```

Groupprivate Directive

- Similar to CUDA and HIP

```
__global__ void func(int *sum, size_t N) {
  __shared__ int tmp1[1000];
  ...
}

func<<<10,256>>>(sum, N);
```

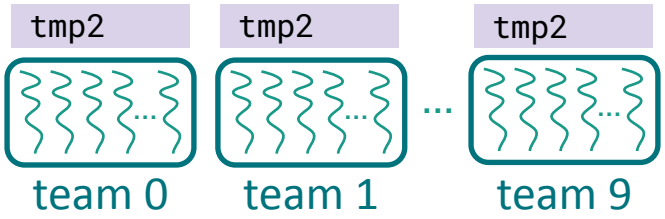


Available OpenMP 6.0

```
void func(int *sum, int tid) {
  static int tmp1[1000];
  #pragma omp groupprivate(tmp1)

  #pragma omp for
  for (int i = 0; i < 1000; i++)
    tmp1[i] = tid + i;

  #pragma omp for reduction(+: sum)
  for (int i = 0; i < 1000; i++)
    sum += tmp1[i];
}

int main() {
  int sums[10];

  #pragma omp target teams num_teams(10) thread_limit(256)
  #pragma omp parallel
  func(&sums[omp_get_team_num()], omp_get_thread_num());
}
```

Dynamic Groupprivate Clause

- Exposes **dynamic shared / local** memory
  - Requests a groupprivate **buffer** with runtime-determined size
  - Each *contention group* its **own buffer copy**



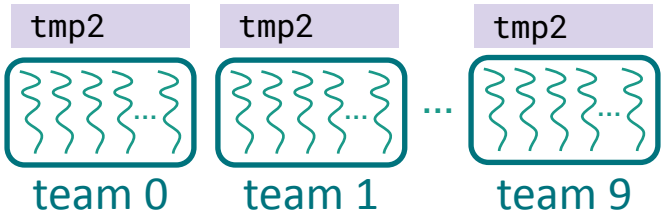
Approved OpenMP 6.1

```
void func(int *sum, int tid, int N) {
  int *tmp = omp_get_dyn_groupprivate_ptr();

  #pragma omp for
  for (int i = 0; i < N; i++)
    tmp[i] = tid + i;

  #pragma omp for reduction(+: sum)
  for (int i = 0; i < N; i++)
    sum += tmp[i];
}

int main(int argc, char **argv) {
  int N = atoi(argv[1]);
  int sums[10];

  #pragma omp target teams num_teams(10) thread_limit(256) \
      dyn_groupprivate(N*sizeof(int))
  #pragma omp parallel
  func(&sums[omp_get_team_num()], omp_get_thread_num(), N);
}
```

Dynamic Groupprivate Clause

- Somewhat similar to CUDA and HIP

```
__global__ void func(int *sum, size_t N) {
  extern __shared__ int tmp2[];
  ...
}

func<<<10,256,N*sizeof(double)>>>(sum, N);
```

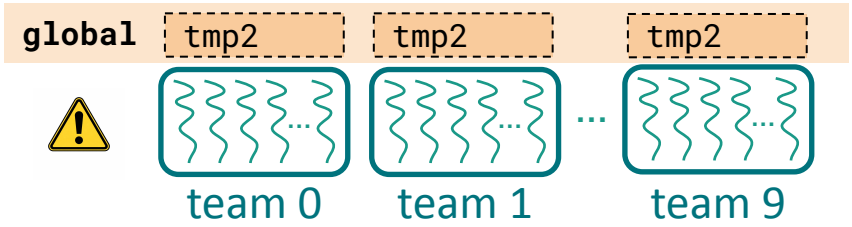


Approved OpenMP 6.1

```
void func(int *sum, int tid, int N) {
  int *tmp2 = omp_get_dyn_groupprivate_ptr();

  #pragma omp for
  for (int i = 0; i < N; i++)
    tmp2[i] = tid + i;

  #pragma omp for reduction(+: sum)
  for (int i = 0; i < N; i++)
    sum += tmp2[i];


int main(int argc, char **argv) {
  int N = atoi(argv[1]);
  int sums[10];

  #pragma omp target teams num_teams(10) thread_limit(256) \
      dyn_groupprivate(N*sizeof(int))
  #pragma omp parallel
  func(&sums[omp_get_team_num()], omp_get_thread_num(), N);
}
```

Dynamic Groupprivate Clause

- Groupprivate memspace is **limited**
- The fallback modifier determines what to do when the limit is reached:
  - **Abort** the execution
  - Return **null** pointer
  - Use memory from a **default memspace** (e.g., global)



Approved OpenMP 6.1

```
#pragma omp target dyn_groupprivate(fallback(abort): N)
{
}
```

```
#pragma omp target dyn_groupprivate(fallback(null): N)
{
}
```

```
#pragma omp target dyn_groupprivate(fallback(default_mem): N)
{
}
```

```
#pragma omp target dyn_groupprivate(N)
{
  // default_mem is the default when not specified
}
```

Multi-dimensional grid programming

- Most accelerated applications use **grid programming**
- Single Program Multiple Data (**SPMD**)

CUDA / HIP

```
__global__ kernel(double *matrix, size_t N) {
  int x = blockIdx.x * blockDim.x + threadIdx.x;
  int y = blockIdx.y * blockDim.y + threadIdx.y;
  int z = blockIdx.z * blockDim.z + threadIdx.z;

  if (x < N && y < N && z < N)
    matrix[x + y * N + z * N * N] = ...;
}

dim3 nblocks(N/256,N/256,N/256);
dim3 nthreads(256,256,256);
kernel<<<nblocks,nthreads>>>(matrix, N);
```

OpenMP 6.0

```
#pragma omp target teams distribute parallel for \
    num_teams(N/256) thread_limit(256) collapse(3)
for (int z = 0; z < N; ++z)
  for (int y = 0; y < N; ++y)
    for (int x = 0; x < N; ++x)
      matrix[x + y * N + z * N * N] = ...;
}
```

Multi-dimensional grid programming

- Most accelerated applications use **grid programming**
- Single Program Multiple Data (**SPMD**)

CUDA / HIP

```
__global__ kernel(double *matrix, size_t N) {
  int x = blockIdx.x * blockDim.x + threadIdx.x;
  int y = blockIdx.y * blockDim.y + threadIdx.y;
  int z = blockIdx.z * blockDim.z + threadIdx.z;

  if (x < N && y < N && z < N)
    matrix[x + y * N + z * N * N] = ...;
}

dim3 nblocks(N/256,N/256,N/256);
dim3 nthreads(256,256,256);
kernel<<<nblocks,nthreads>>>(matrix, N);
```

Future OpenMP

```
#pragma omp target teams num_teams(dims(3): N/256,N/256,N/256) \
    thread_limit(dims(3): 256,256,256)
#pragma omp parallel
{
  int x = omp_get_team_num_dim(1) * omp_get_num_teams_dim(1)
      + omp_get_thread_num_dim(1);
  int y = omp_get_team_num_dim(2) * omp_get_num_teams_dim(2)
      + omp_get_thread_num_dim(2);
  int z = omp_get_team_num_dim(3) * omp_get_num_teams_dim(3)
      + omp_get_thread_num_dim(3);

  matrix[x + y * N + z * N * N] = ...;
}
```

Under Discussion

Multi-dimensional grid programming

- Most accelerated applications use **grid programming**
- Single Program Multiple Data (**SPMD**)

Future OpenMP

```
#pragma omp target teams num_teams(dims(3): N/256,N/256,N/256) \
    thread_limit(dims(3): 256,256,256)
#pragma omp parallel
{
  int x = omp_get_team_num_dim(1) * omp_get_num_teams_dim(1)
      + omp_get_thread_num_dim(1);
  int y = omp_get_team_num_dim(2) * omp_get_num_teams_dim(2)
      + omp_get_thread_num_dim(2);
  int z = omp_get_team_num_dim(3) * omp_get_num_teams_dim(3)
      + omp_get_thread_num_dim(3);

  matrix[x + y * N + z * N * N] = ...;
}
```

Compatibility with **unidimensional** world

```
#pragma omp target teams \
    num_teams(dims(3): N/256,N/256,N/256) \
    thread_limit(dims(3): 256,256,256)
#pragma omp parallel
{
  int team = omp_get_team_num();
  int thread = omp_get_thread_num();
  int nthreads = omp_get_num_threads();

  #pragma omp single
  { ... }
}
```

Transparent **flattening** of identifiers/sizes

Implementation in LLVM/Clang

- **Multi-dimensional** grid
  - Support **dims** modifier in *num_teams* and *thread_limit* clauses
  - Support in LLVM/OpenMP DeviceRTL
  - **Linearization** implemented in DeviceRTL
  - Some cases not supported yet
- Full implementation of **Dynamic Groupprivate**
- **Groupprivate** not exposed as `#pragma` yet, using instead:

```
static __attribute((address_space(3),aligned(16),loader_uninitialized)) double tmp[1024];
```

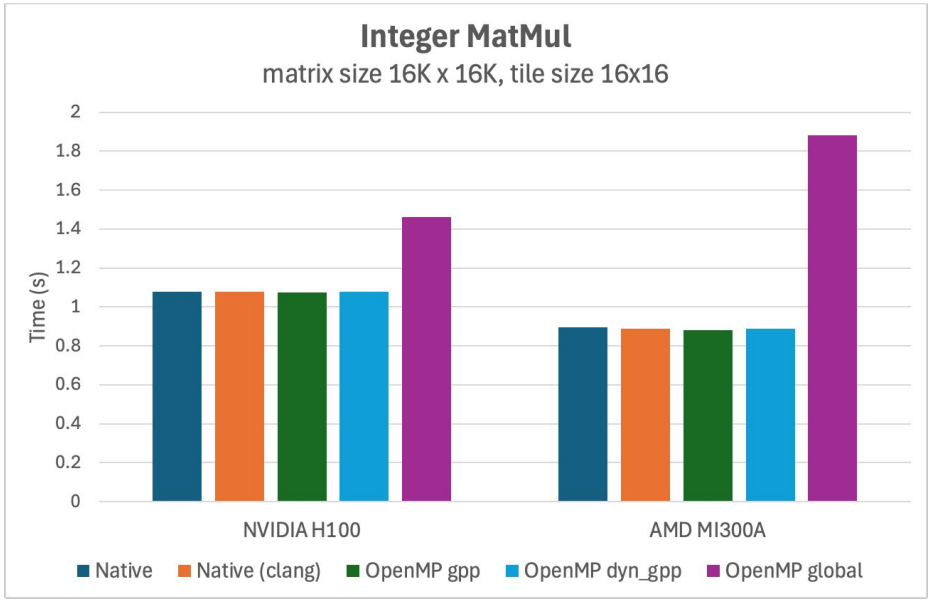
  - PR #69018 not evaluated yet (author @Ritanya-B-Bharadwaj)

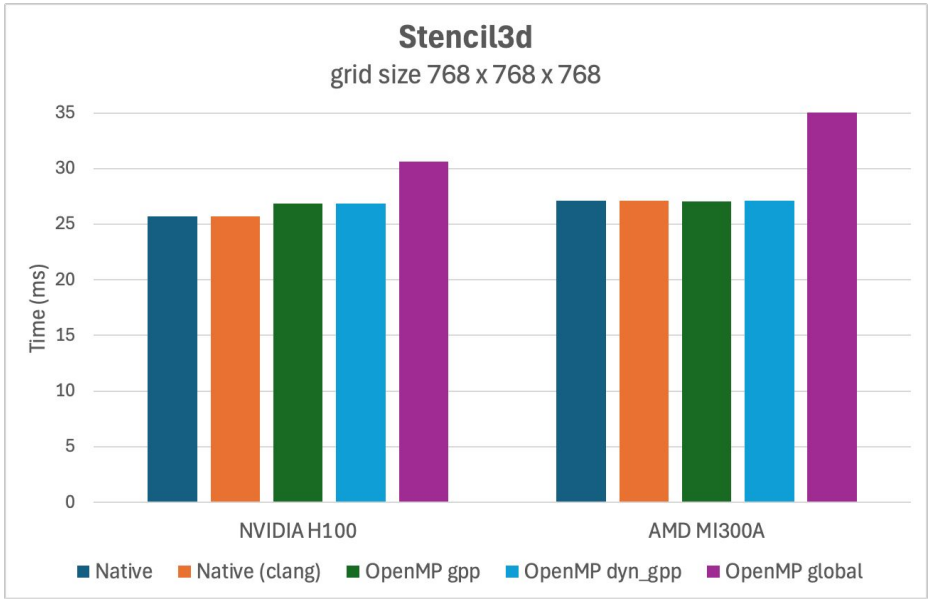
Evaluation

- Two benchmarks
  - **Matmul** (manual implementation)
  - **Stencil3d** (extracted from HeCBench)
- Common characteristics
  - **Multi-dimensional grid** programming
  - Heavy use of **shared/local** memory

- Two platforms in LLNL
  - NVIDIA H100
  - AMD MI300A

Evaluation


Integer MatMul
matrix size 16K x 16K, tile size 16x16
Time (s)
2
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
NVIDIA H100
AMD MI300A
Native
Native (clang)
OpenMP gpp
OpenMP dyn_gpp
OpenMP global


Stencil3d
grid size 768 x 768 x 768
Time (ms)
35
30
25
20
15
10
5
0
NVIDIA H100
AMD MI300A
Native
Native (clang)
OpenMP gpp
OpenMP dyn_gpp
OpenMP global

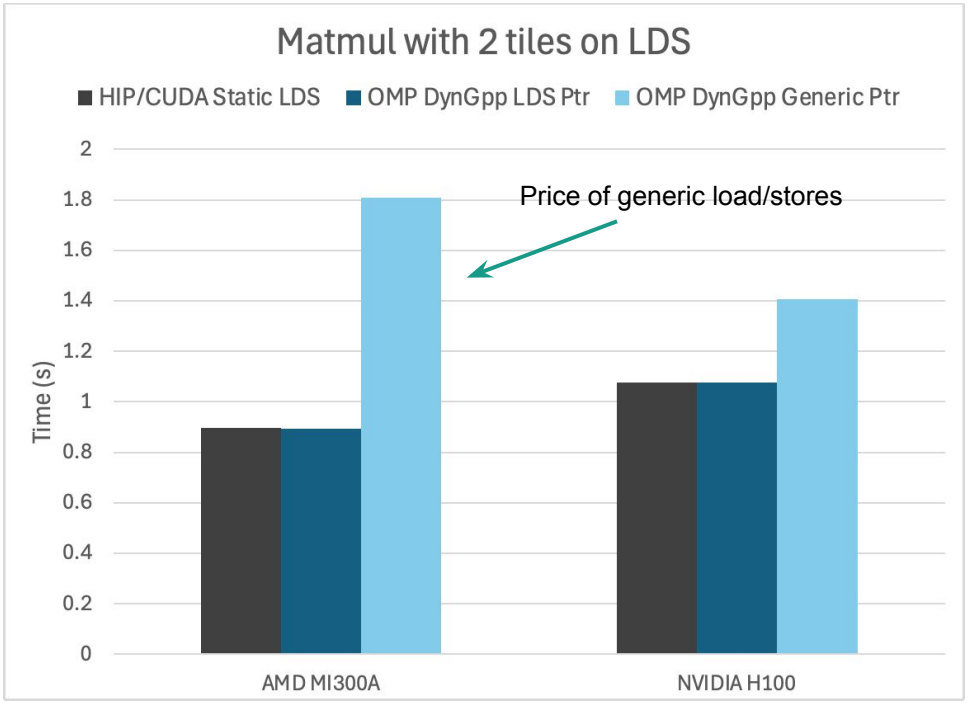
Lessons learned from the implementation

- Make sure the compiler can determine the correct address space in the user code
  - **addrspace(3)** in nvidia and amdgpu

- This is problematic:

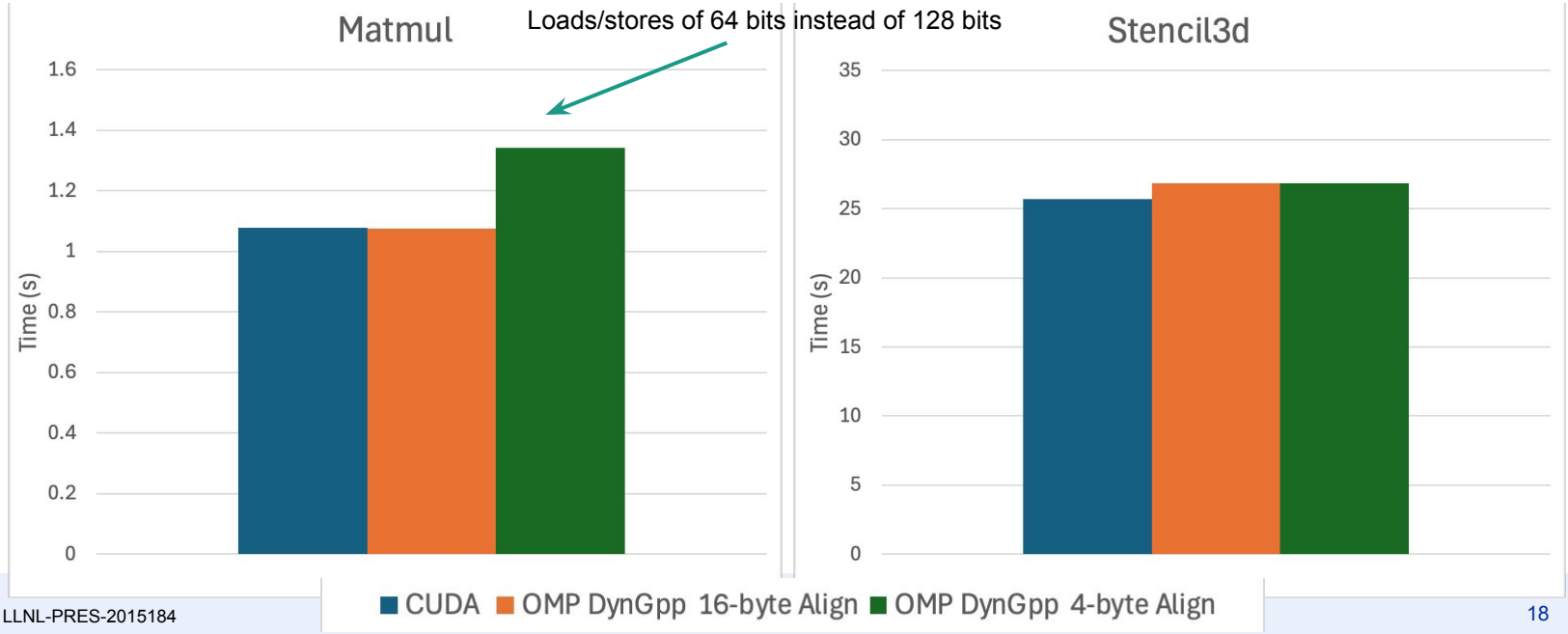
```
void *omp_get_dyn_groupprivate_ptr(...) {
  return (!RT.was_fallback) ? RT.shmem_ptr
                            : RT.fallback_glb_ptr;
}
```

addrspace(3) vs. addrspace(1) at runtime => generic addrspace(0)



Lessons learned from the implementation

- Make sure to forward shared memory pointer alignment to user code
  - e.g., **16-byte** alignment in nvidia



Fortran

- AMD folks are working on the LLVM Flang implementation
  - Krishna Chaitanya Sankisa
  - Krzysztof Parzyszek
  - Michael Klemm

- More results coming soon!

Conclusions

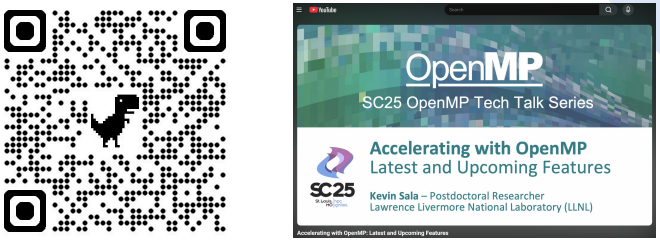
- OpenMP *target* provides
  - **Standard** for acceleration with **multi-vendor** support
  - **Language portability**: C, C++, Fortran, Python (prototype)
  - **Abstraction** over complex hardware and software
- Recent and upcoming OpenMP features
  - **Get** the same **performance** with native offloading APIs
  - **Performance portability**
  - **Reduce** porting **effort**
- Positive results for **LLVM/OpenMP** in C/C++

- Future work: full evaluation in larger applications

Thank you!

Kevin Sala
salapenades1@llnl.gov

Watch the OpenMP tech talk on YouTube!



https://www.youtube.com/watch?v=xOsoajfdIww

Thank you for participating in LLVM Performance!

- Invite speakers to send copy of their slides
- Feedback to improve the workshop?