Modernizing OpenMP for an Accelerated World

Tom Scogland
Bronis de Supinski

GTC ◆ March 26, 2018

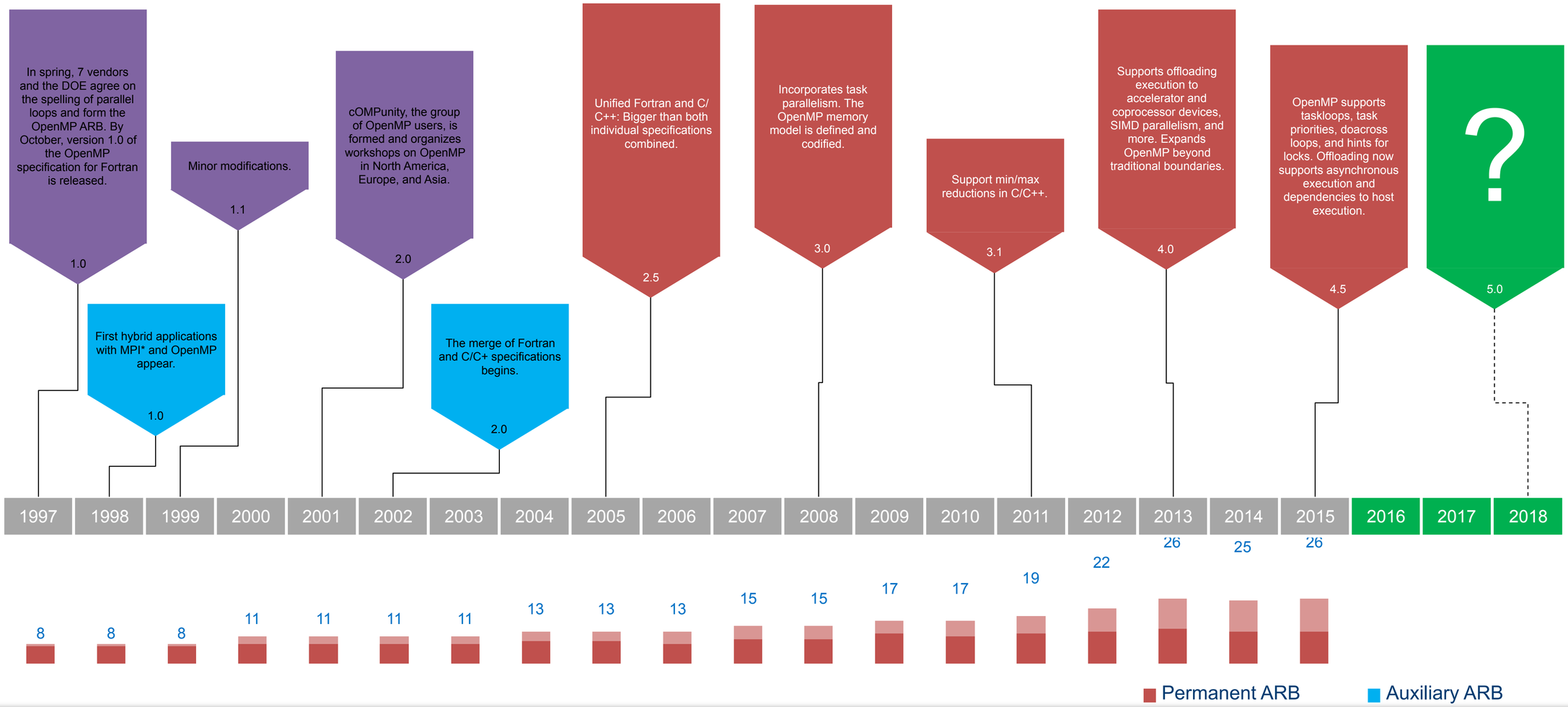
This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

LLNL-PRES-747146

What is OpenMP?

```
#pragma omp parallel for
for(int i=0; i<N; ++i) {
  do_something(i);
}
```

History of OpenMP



Why expand OpenMP target support now?

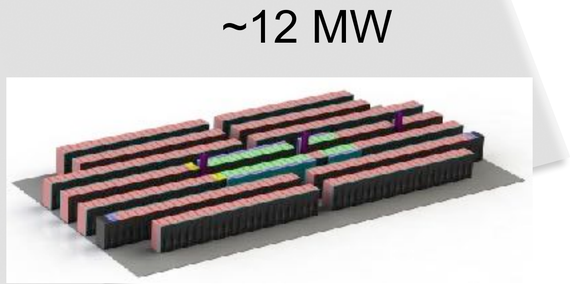
- We need heterogeneous computing
  - Better energy efficiency
  - More performance without increasing clock speed
- C/C++ abstractions (CUDA, Kokkos or RAJA) aren't enough
  - Even the C++ abstractions have to run atop something!
  - Not all codes are written in C++, some are even written in F******!
- New mainstream system architectures require it!

Sierra: The next LLNL Capability System

The Sierra system features a GPU-accelerated architecture

Compute System

4320 nodes
1.29 PB Memory
240 Compute Racks
125 PFLOPS
~12 MW



Compute Rack

Standard 19”
Warm water cooling

Compute Node

2 IBM POWER9 CPUs
4 NVIDIA Volta GPUs
NVMe-compatible PCIe 1.6 TB SSD
256 GiB DDR4
16 GiB Globally addressable HBM2 associated with each GPU
Coherent Shared Memory

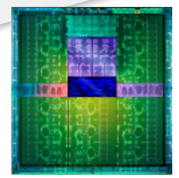
Components

IBM POWER9

- Gen2 NVLink

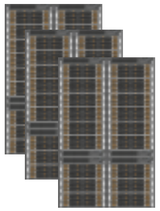
NVIDIA Volta

- 7 TFlop/s
- HBM2
- Gen2 NVLink



Mellanox Interconnect

Single Plane EDR InfiniBand
2 to 1 Tapered Fat Tree



Spectrum Scale File System

154 PB usable storage
1.54 TB/s R/W bandwidth

Sierra stats

	Sierra	uSierra
Nodes	4,320	684
POWER9 processors per node	2	2
GV100 (Volta) GPUs per node	4	4
Node Peak (TFLOP/s)	29.1	29.1
System Peak (PFLOP/s)	125	19.9
Node Memory (GiB)	320	320
System Memory (PiB)	1.29	0.209
Interconnect	2x IB EDR	2x IB EDR
Off-Node Aggregate b/w (GB/s)	45.5	45.5
Compute racks	240	38
Network and Infrastructure racks	13	4
Storage Racks	24	4
Total racks	277	46
Peak Power (MW)	~12	~1.8

Many Updates to Accelerator Support in OpenMP 4.5

- Unstructured data mapping
- Asynchronous execution
- Scalar variables are firstprivate by default
- Improvements for C/C++ array sections
- Device runtime routines: allocation, copy, etc.
- Clauses to support device pointers
- Ability to map structure elements
- New combined constructs
- New way to map global variables (`link`)

Tons of non-accelerator updates for tasking, SIMD and even performance of classic worksharing

Gaps in OpenMP 4.5

- Base language support is out of date
  - C99
  - C++03
  - Fortran 03
- Mapping complex data structures is painful
  - No direct support for unified memory devices
  - No mechanism for deep copying in mappings
- Overlapping data transfers with computation is complex and error prone
- Etc.

Base Language Support in OpenMP 5.0

- C99 -> C11
  - _Atomic still in discussion

- C++03 -> C++17 (yes, 11, 14 and 17 all at once)
  - C++ threads still in discussion
  - Explicit support for mapping lambdas (sanely)
  - Improved support for device code
    - Classes with virtual methods can be mapped (may even be callable)

- Fortran 2008? (in the works, maybe)

Complex Data in OpenMP 5.0: Unified Memory and Deep Copy, Why Both?

1. Mapping provides more information to both the compiler and the runtime
2. Not all hardware has unified memory
3. Not all unified memory is the same
    1. Can all memory be accessed with the same performance from everywhere?
    2. Do atomics work across the full system?
    3. Are flushes required for coherence? How expensive are they?

Specifying unified memory in OpenMP

- OpenMP does not require unified memory
  - Or even a unified address space
- This is not going to change

How do you make non-portable features portable?

- Specify what they provide when they are present
- Give the user a way to assert that they are required
- Give implementers a way to react to that assertion

One solution: Requirement declarations

#pragma omp requires [extension clauses…]

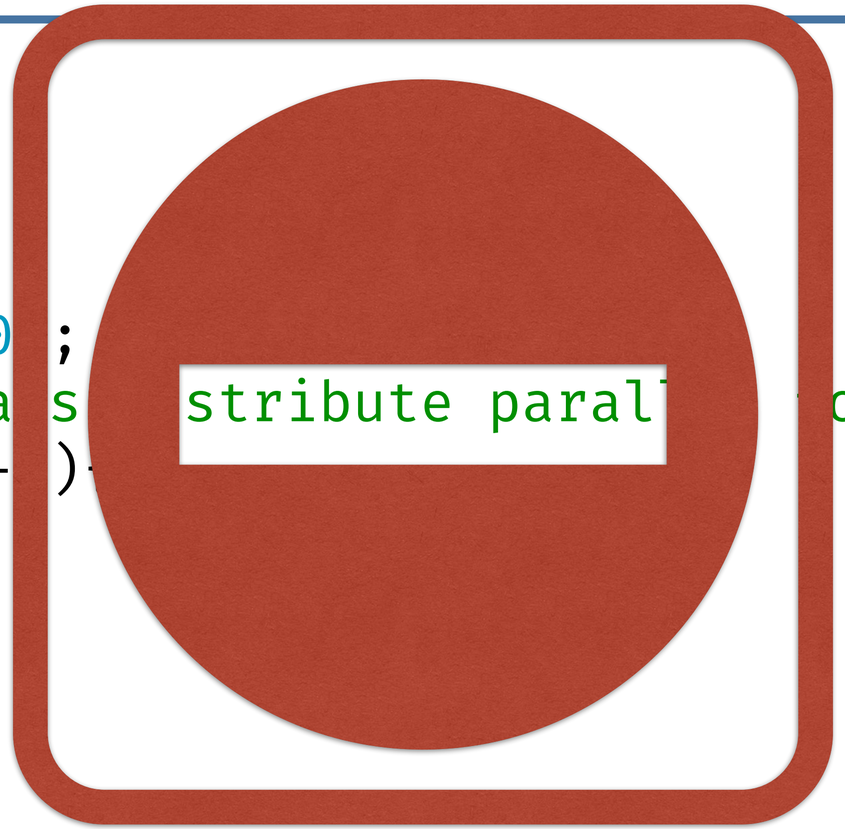
Extension name	Effect
unified_address	Guarantee that device pointers are unique across all devices, is_device_ptr is not required
unified_shared_memory	Host pointers are valid device pointers and considered present by all implicit maps, implies unified_address, memory is synchronized at target task sync

OpenMP unified memory example

```
int * arr = new int[50];
#pragma omp target teams distribute parallel for
for (int i=0; i<50; ++i){
  arr[i] = i;
}
```

OpenMP unified memory example

```
int * arr = new int[50 ;
#pragma omp target tea s    stribute paral     or
for (int i=0; i<50; ++ )
  arr[i] = i;
}
```

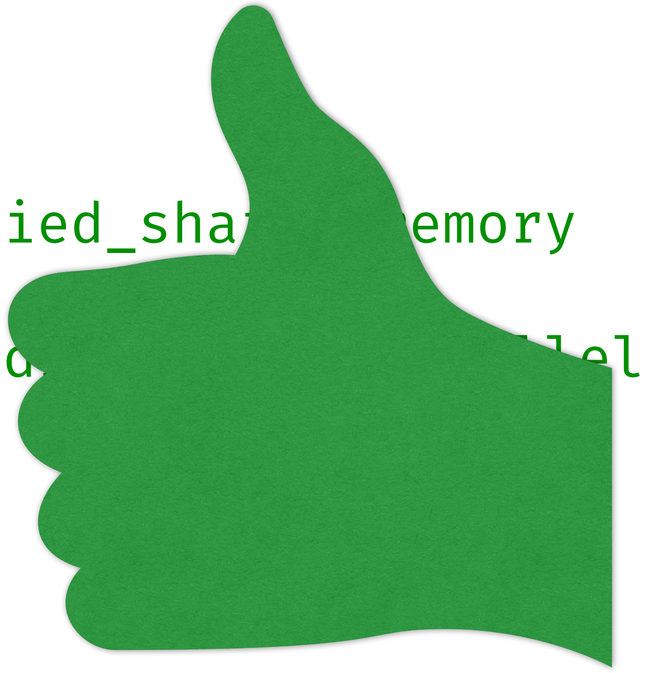


OpenMP unified memory example

```
#pragma omp requires unified_shared_memory
int * arr = new int[50];
#pragma omp target teams distribute parallel for
for (int i=0; i<50; ++i){
  arr[i] = i;
}
```

OpenMP unified memory example

```
#pragma omp requires unified_sha[illegible]emory
int * arr = new int[50];
#pragma omp target teams d[illegible]lel for
for (int i=0; i<50; ++i){
  arr[i] = i;
}
```



Deep copy today

- It **is** possible to use deep copy in OpenMP today
- Manual deep copy works by *pointer attachment*

Pointer attachment

```
typedef struct myvec {
    size_t len;
    double *data;
} myvec_t;
myvec_t v = init_myvec();
#pragma omp target map(v, v.data[:v.len])
{
  do_something_with_v(&v);
}
```

Pointer attachment

Map structure v

```
typedef struct myvec {
    size_t len;
    double *data;
} myvec_t;
myvec_t v = init_myvec();
#pragma omp target map(v, v.data[:v.len])
{
  do_something_with_v(&v);
}
```

Pointer attachment

```
typedef struct myvec {
    size_t len;
    double *data;
} myvec_t;
myvec_t v = init_myvec();
#pragma omp target map(v, v.data[:v.len])
{
  do_something_with_v(&v);
}
```

Map structure v

Map data array and attach to v

What's the downside?

```
typedef struct myvec {
    size_t len;
    double *data;
} myvec_t;
size_t num = 50;
myvec_t *v = alloc_array_of_myvec(num);
#pragma omp target map(v[:50], ??????)
{
  do_something_with_v(&v);
}
```

Map an array of v structures

Now we need a loop, more breaking up the code!

The future of deep copy: User-defined mappers

```
#pragma omp declare mapper(<type> <var>)
    [name(<name>)]
    [use_by_default]
    [map(<list-items>…)…]
```

- Allow users to define mappers in terms of normal map clauses
- Offer extension mechanisms to pack or unpack data that can't be bitwise copied, or expressed as flat maps

Our array example

```
typedef struct myvec {
    size_t len;
    double *data;
} myvec_t;
#pragma omp declare mapper(myvec_t v)\
        use_by_default map(v, v.data[:v.len])
size_t num = 50;
myvec_t *v = alloc_array_of_myvec(num);
#pragma omp target map(v[:50])
{
  do_something_with_v(&v);
}
```

Our array example

```
typedef struct myvec {
    size_t len;
    double *data;
} myvec_t;
#pragma omp declare mapper(myvec_t v)\
        use_by_default map(v, v.data[:v.len])
size_t num = 50;
myvec_t *v = alloc_array_of_myvec(num);
#pragma omp target map(v[:50])
{
  do_something_with_v(&v);
}
```

No loop required, no extra code at usage, just map

Composition of mappers

```
typedef struct myvec {
    size_t len;
    double *data;
} myvec_t;
#pragma omp declare mapper(myvec_t v)\
        use_by_default map(v, v.data[:v.len])
typedef struct mypoints {
    struct myvec * x;
    struct myvec scratch;
    double useless_data[500000];
} mypoints_t;

#pragma omp declare mapper(mypoints_t p) \
    use_by_default \
    map(/* self only partially mapped, useless_data can be ignored */\
        p.x, p.x[:1]) /* map and attach x */ \
    map(alloc:p.scratch) /* never update scratch, including its internal maps */

mypoints_t p = new_mypoints_t();
#pragma omp target
{
  do_something_with_p(&v);
}
```

Composition of mappers

```
typedef struct myvec {
    size_t len;
    double *data;
} myvec_t;
#pragma omp declare mapper(myvec_t v)\
        use_by_default map(v, v.data[:v.len])
typedef struct mypoints {
    struct myvec * x;
    struct myvec scratch;
    double useless_data[500000];
} mypoints_t;

#pragma omp declare mapper(mypoints_t p) \
    use_by_default \
    map(/* self only partially mapped, useless_data can be ignored */\
        p.x, p.x[:1]) /* map and attach x */ \
    map(alloc:p.scratch) /* never update scratch, including its internal maps */

mypoints_t p = new_mypoints_t();
#pragma omp target
{
  do_something_with_p(&v);
}
```

Pick and choose what to map

Composition of mappers

```
typedef struct myvec {
    size_t len;
    double *data;
} myvec_t;
#pragma omp declare mapper(myvec_t v)\
        use_by_default map(v, v.data[:v.len
typedef struct mypoints {
    struct myvec * x;
    struct myvec scratch;
    double useless_data[500000];
} mypoints_t;

#pragma omp declare mapper(mypoints_t p) \
    use_by_default \
    map(/* self only partially mapped, useless_data can be ignored */\
        p.x, p.x[:1]) /* map and attach x */ \
    map(alloc:p.scratch) /* never update scratch, including its internal maps */

mypoints_t p = new_mypoints_t();
#pragma omp target
{
  do_something_with_p(&v);
}
```

Re-use the myvec_t mapper

Pick and choose what to map

Composition of mappers

```
typedef struct myvec {
    size_t len;
    double *data;
} myvec_t;
#pragma omp declare mapper(myvec_t v)\
        use_by_default map(v, v.data[:v.len
typedef struct mypoints {
    struct myvec * x;
    struct myvec scratch;
    double useless_data[500000];
} mypoints_t;

#pragma omp declare mapper(mypoints_t p) \
    use_by_default \
    map(/* self only partially mapped, useless_data can be ignored */\
        p.x, p.x[:1]) /* map and attach x */ \
    map(alloc:p.scratch) /* never update scratch, including its internal maps */

mypoints_t p = new_mypoints_t();
#pragma omp target
{
  do_something_with_p(&v);
}
```

Re-use the myvec_t mapper

Pick and choose what to map

No explicit map required!

Defining mappers from explicit serialization and deserialization (OpenMP 5.1+)

- Declare mappers by stages, all are replaceable
  - alloc
  - pack_to
  - unpack_to
  - pack_from
  - unpack_from
  - release
- Any arbitrary data can be mapped, transformed, or munged how you like!

Dealing with overlapping complexity (OpenMP 5.1+): Automating pipelined data transfers

- Pipelining normally requires users to:
  - Split their work into multiple chunks
  - Add another loop nesting level over the chunks
  - Explicitly copy a subset of their data
  - Transform accesses to reference that subset
  - Ensure all chunks are synchronized

Lots of work for the user!

- Doing this as an extension to OpenMP requires:
  - A data motion direction
  - The portion of data accessed by each iteration
  - Which dimension is being looped over

Simpler, more automatic

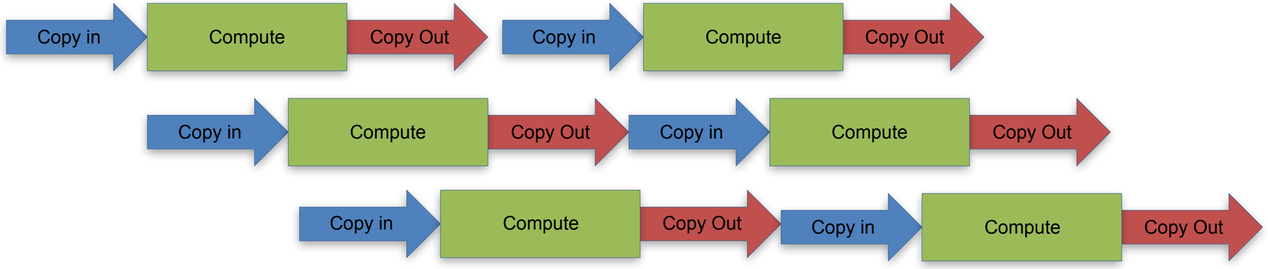
- Optionally we can do more with:
  - Number of concurrent transfers
  - Memory limits
  - Schedulers
  - Etc.

Greater capabilities!

Pipelining for OpenMP: Why pipelining?

- Default synchronous data motion

- Pipelined data motion



Pipelining in OpenMP: non-pipelined stencil

```
#pragma omp target data \
  map(to:A0[0:nz-1][0:ny-1][0:nx-1]) \
  map(from:Anext[0:nz-1][0:ny-1][0:nx-1])
for(k=1;k<nz-1;k++) {
#pragma omp target teams distribute parallel for
  for(i=1;i<nx-1;i++) {
    for(j=1;j<ny-1;j++) {
      Anext[Index3D (i,     j,     k)] =
        (A0[Index3D (i,     j,     k + 1)] +
         A0[Index3D (i,     j,     k - 1)] +
         A0[Index3D (i,     j + 1, k)] +
         A0[Index3D (i,     j - 1, k)] +
         A0[Index3D (i + 1, j,     k)] +
         A0[Index3D (i - 1, j,     k)])*c1
       - A0[Index3D (i,     j,     k)]*c0;
    } }
}
```

Pipelining in OpenMP: non-pipelined stencil

```
#pragma omp target data \
  map(to:A0[0:nz-1][0:ny-1][0:nx-1]) \
  map(from:Anext[0:nz-1][0:ny-1][0:nx-1])
for(k=1;k<nz-1;k++) {
#pragma omp target teams distribute parallel for
  for(i=1;i<nx-1;i++) {
    for(j=1;j<ny-1;j++) {
      Anext[Index3D (i,     j,     k)] =
        (A0[Index3D (i,     j,     k + 1)] +
         A0[Index3D (i,     j,     k - 1)] +
         A0[Index3D (i,     j + 1, k)] +
         A0[Index3D (i,     j - 1, k)] +
         A0[Index3D (i + 1, j,     k)] +
         A0[Index3D (i - 1, j,     k)])*c1
       - A0[Index3D (i,     j,     k)]*c0;
    } }
}
```

Pipelining in OpenMP: pipelined stencil

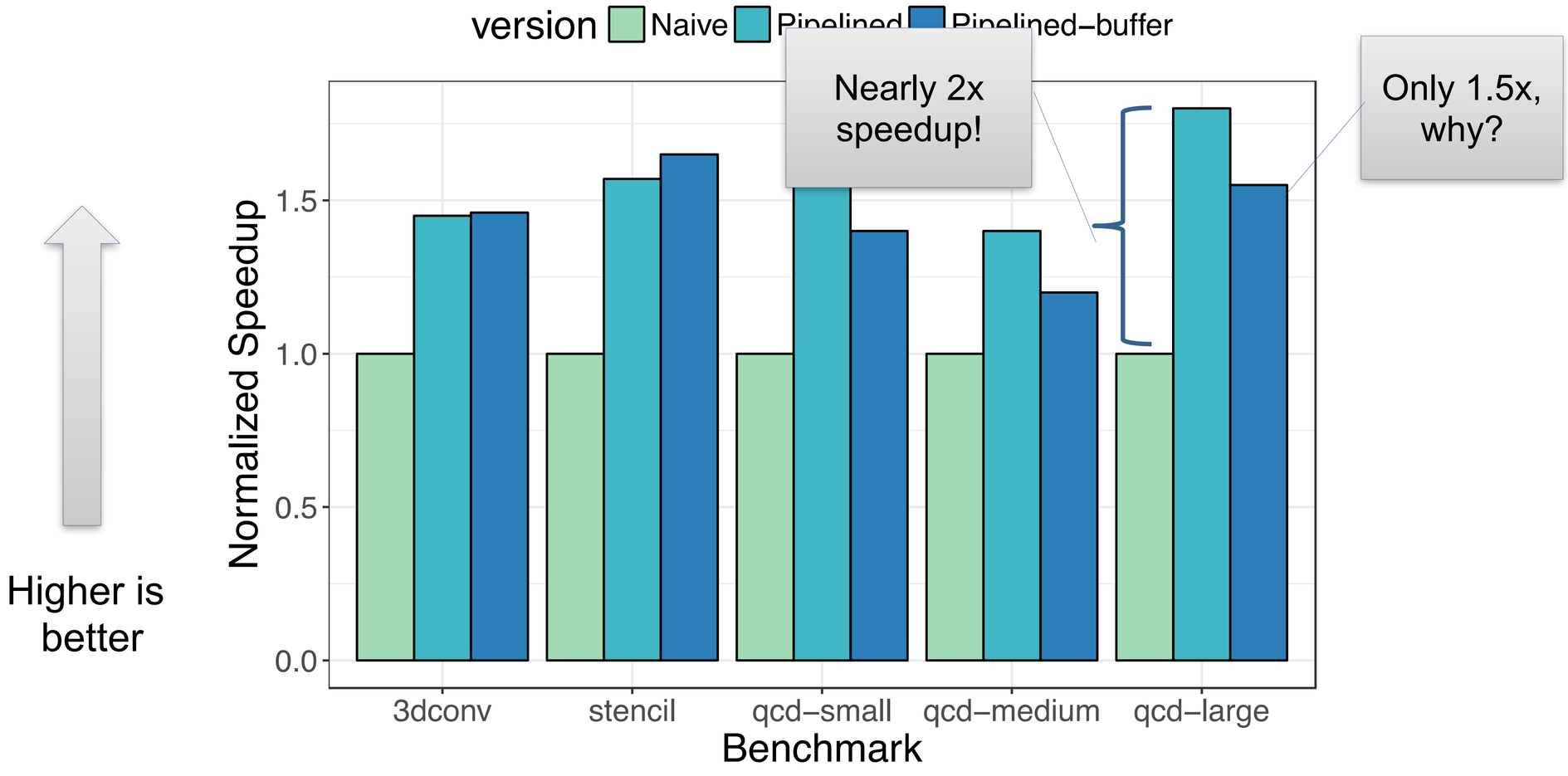
```
#pragma omp target \
  pipeline(static[1,3])\
  pipeline_map(to:A0[k-1:3][0:ny-1][0:nx-1])\
  pipeline_map(from:Anext[k:1][0:ny-1][0:nx-1])\
for(k=1;k<nz-1;k++) {
#pragma omp target teams distribute parallel for
  for(i=1;i<nx-1;i++) {
    for(j=1;j<ny-1;j++) {
      Anext[Index3D (i,     j,     k)] =
        (A0[Index3D (i,     j,     k + 1)] +
         A0[Index3D (i,     j,     k - 1)] +
         A0[Index3D (i,     j + 1, k)] +
         A0[Index3D (i,     j - 1, k)] +
         A0[Index3D (i + 1, j,     k)] +
         A0[Index3D (i - 1, j,     k)])*c1
       - A0[Index3D (i,     j,     k)]*c0;
    } }
}
```

Pipelining in OpenMP: pipelined *and buffered* with our proposed extension

```
#pragma omp target \
  pipeline(static[1,3])\
  pipeline_map(to:A0[k-1:3][0:ny-1][0:nx-1])\
  pipeline_map(from:Anext[k:1][0:ny-1][0:nx-1])\
  pipeline_mem_limit(MB_256)
for(k=1;k<nz-1;k++) {
#pragma omp target teams distribute parallel for
  for(i=1;i<nx-1;i++) {
    for(j=1;j<ny-1;j++) {
      Anext[
        (A
          A
          A
          A
          A
          A
        - A0[Index3D (1,     j,     k)]*c0;
    } }
}
```

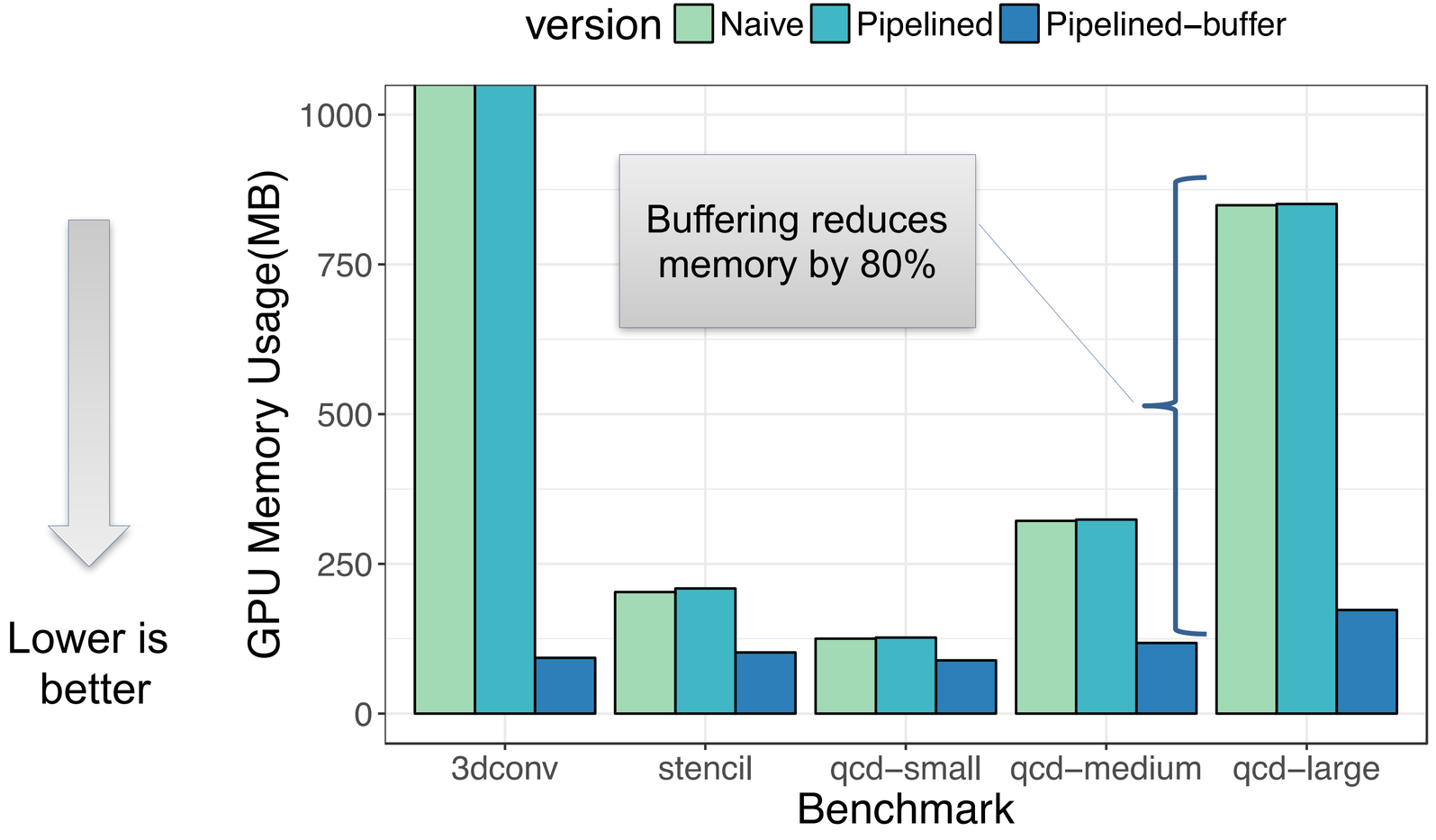
Replicating this manually requires ~20 more lines of error-prone boilerplate ***per loop***!

Pipelining in OpenMP: Kernel and benchmark performance (Sierra EA, P100)



All results with PGI OpenACC on k40 GPUs, LLNL surface cluster

Pipelining in OpenMP: Lattice QCD benchmark memory usage



OpenMP into the Future: What's next?

- Descriptive loop constructs
- Automated pipelining
- Arbitrarily complex data transformation and deep copy
- Memory affinity
- Multi-target worksharing
- Support for complex hierarchical memories
- Better task dependencies and taskloop support
- Free-agent threads, possibly even detachable teams

References

- X. Cui, T. R. Scogland, B. R. de Supinski, and W.-c. Feng. Directive-based partitioning and pipelining for graphics processing units. In International Parallel and Distributed Processing Symposium, pages 575–584. IEEE, 2017.
- Scogland T., Earl C., de Supinski B. (2017) Custom Data Mapping for Composable Data Management. In: Scaling OpenMP for Exascale Performance and Portability. IWOMP 2017.