Panel Session:

Prioritizing OpenMP Features to Provide for Performance, Portability and Productivity

Oscar Hernandez (ORNL)
Vivek Kale (BNL)
The mission of the OpenMP ARB (Architecture Review Board) is to standardize directive-based multi-language high-level parallelism that is performant, productive and portable.
How OpenMP evolves compared with HPC trends (www.top500.org)

Credit: Jose Monsalve Diaz, at University of Delaware
History of OpenMP: 1997 - 2020

1997
In spring, 7 vendors and the DOE agree on the spelling of parallel loops and form the OpenMP ARB. By October, version 1.0 of the OpenMP specification for Fortran is released.

1998

1999

2000

2001

2002

2003

2004

2005

2006

2007

2008

2009

2010

2011

2012

2013

2014

2015

2016

2017

2018

2019

2020

1.0
Minor modifications

1.1
C/C++ v 1.0. First hybrid applications with MPI* and OpenMP appear.

2.0
Unified Fortran and C/C++: Bigger than both individual specifications combined.

2.5
The merge of Fortran and C/C++ specifications begins.

3.0
Incorporates task parallelism. The OpenMP memory model is defined and codified.

3.1
Supports offloading execution to accelerator and coprocessor devices, SIMD parallelism, and more. Expands OpenMP beyond traditional boundaries.

3.5
OpenMP supports taskloops, task priorities, doacross loops, and hints for locks. Offloading now supports asynchronous execution and dependencies to host execution.

4.0
Incorporates task parallelism. The OpenMP memory model is defined and codified.

4.5
Supports: Memory Management API, Reverse Offload, Loop construct, Detached tasks, Custom Mappers, Tools API

5.0
loop transformation (tiling,...), Improved ‘omp loop’*, variant overloading, runtime variant selection*, compiler agnostic “built-in assume”

5.1
Permanent ARB    Auxiliary ARB
Relevance of OpenMP

OpenMP is about 50%, out of all choices of X

- Programming Accelerators
- Manage memory allocations (High Bandwidth, Low Latency, Accelerator) memories) with traits (pinned memory, etc)
- Data movement of complicated data structures (e.g., deep copy)
- Support for latest C++ and Fortran standards
- Interoperability with libraries
- Performance portable directives
- Task parallelism for asynchronous execution to orchestrate work between CPUs and Accelerators
- SIMD directives (to support SIMD parallelism)
- Focus on continuity of technology and early access to users

Update late 2016: 75% of codes use OpenMP

E3SM  LQCD  CANDLE  QMCPACK  NWChem
OpenMP Offload in QMCPack

**Tests from miniQMC**

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Clang 9</th>
<th>AOMP 0.7-4</th>
<th>XL 16.1.1-3</th>
<th>Cray 9.0</th>
<th>GCC 9.2</th>
<th>GCC 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>device</td>
<td>NV</td>
<td>AMD</td>
<td>NV</td>
<td>NV</td>
<td>NV</td>
<td>AMD</td>
</tr>
<tr>
<td>math header conflict</td>
<td>F</td>
<td>P</td>
<td>P</td>
<td>P</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>math linker error</td>
<td>P</td>
<td>P</td>
<td>P</td>
<td>F</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>declare target static data</td>
<td>P</td>
<td>P</td>
<td>P</td>
<td>F</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>static linking</td>
<td>F</td>
<td>P</td>
<td>P</td>
<td>P</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>Async tasks</td>
<td>F</td>
<td>F</td>
<td>P</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>multiple streams</td>
<td>F</td>
<td>F</td>
<td>P</td>
<td>P</td>
<td>P</td>
<td>F</td>
</tr>
<tr>
<td>check_spo</td>
<td>FR</td>
<td>FW</td>
<td>P</td>
<td>P</td>
<td>P</td>
<td>FL</td>
</tr>
<tr>
<td>check_spo_batched</td>
<td>FR</td>
<td>P</td>
<td>P</td>
<td>P</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>minimqm_sync_move</td>
<td>FR</td>
<td>P</td>
<td>P</td>
<td>F</td>
<td>P</td>
<td>FL</td>
</tr>
</tbody>
</table>

The table indicates that XL is the only survival compiler, while other compilers need further improvements.  

**Performance with OpenMP Impl.**

- **Recent work (last few months)** with clang to improve it, e.g., on target region-to-stream scheduling, support for std::complex shows promise for performance.
- Still can't show clang result due to unique-to-Summit CUDA driver problem soon to be fixed, but clang OpenMP estimated to have 0.75 performance of IBM XL.
- Also have run with Cray clang and AMD AOMP correctly. These show promise though don't have all feature support of clang.
- Got code to work with oneAPI.  

→ IBM OpenMP is shown reasonably performant though rapid development of LLVM OpenMP has shown significant promise to allow for better performance over IBM offload.
→ QMCPack will continue to track performance of latest OpenMP implementations available on ECP systems.
Functionality Status of Features in OpenMP Implementations

Shows the features that are commonly supported across OpenMP Implementations.

<table>
<thead>
<tr>
<th>Levels of parallelism</th>
<th>LLVM/Clang 10 (2 (teams, parallel) (11: 3 (teams, parallel, simd)))</th>
<th>AMD (mostly tracks LLVM) (2 (teams, parallel))</th>
<th>Cray (CCE 10) (2 (teams, parallel or simd))</th>
<th>IBM (XL V16.1.6) (2 (teams, parallel))</th>
<th>Intel (Approximately 2021 timeframe) (3 (teams, parallel, simd))</th>
<th>NVIDIA/PGI (Early 2021 for a production release) (2 (teams, parallel))</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenMP directive</td>
<td>target</td>
<td>declare target</td>
<td>map</td>
<td>target data</td>
<td>target enter/exit data</td>
<td>target update</td>
</tr>
<tr>
<td></td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td></td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td></td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td></td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td></td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td></td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td></td>
<td>√</td>
<td>√</td>
<td>(may be inactive)</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td></td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td></td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td></td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td></td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td></td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td></td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td></td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td></td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td></td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td></td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td></td>
<td>√</td>
<td>√</td>
<td>(support planned for CCE 11) (X)</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
</tbody>
</table>

Figure 1: Feature support of OpenMP directives in different OpenMP Implementations
Cray compiler has highest performance in 6/7 C benchmarks (unofficial SPEC results)

LLVM/Clang is 39x slower than Cray on 570.pbt!

OpenMP loop construct placement hurts LLVM/Clang performance

From Christopher Daley
NERSC
How is this being addressed?

- LLVM implementations
- OpenMP performance benchmarks
OpenMP Version 5.0

- OpenMP 5.0 introduced powerful features to improve programmability

- Initial C11, C++11, C++14 and C++17 support
- Complete Fortran 2003 Support, Initial Fortran 2008 Support
- Task Reductions
- Memory Allocators
- Detachable Tasks
- Dependence Objects
- Tools
- Unified Shared Memory
- Collapse Non-Rectangular Loops
- Data Serialization for Offload (Deep Copy)
- Meta-Directives
- Task-to-data Affinity
- Improved Affinity Support
- Improved Affinity
- Improved Task Dependences
- Multi-Level Parallelism
- Parallel Scan
- Interoperability and Usability Enhancements
- Function Variants
- Reverse Offload
- Offload
- Improved Task Dependences
- ECP
What are the important features are important to applications?

• Loop construct
• Unified shared memory support
• Accelerator data management
  – Non-contiguous data mappings
• Memory allocators
• Metadirective and variants
• Tasks
  – Detach
  – Reductions
• Deep copy
• C++ virtual methods
loop Construct

• Existing loop constructs are tightly bound to execution model:

  #pragma omp for
  for (i=0; i<N;++i) {...}

  #pragma omp simd
  for (i=0; i<N;++i) {...}

  #pragma omp taskloop
  for (i=0; i<N;++i) {...}

• The loop construct is meant to let the OpenMP implementation pick choose the right parallelization scheme.
How to use OpenMP on Accelerators

```c
#pragma omp target teams
distribute
for (i=0; i<N; ++i) {
    #pragma omp parallel for
distribute
    for (j=0; j<N; ++j) {
        x[j+N*i] *= 2.0;
    }
}
```

- The **target** construct offloads the enclosed code to the accelerator.
- The **teams** construct creates a league of teams.
- The **distribute** construct distributes the outer loop iterations between the league of teams.
- The **parallel for** combined construct creates a thread team for each team and distributes the inner loop iterations to threads.
How to use modern OpenMP – Execution Example

```c
#pragma omp target
#pragma omp loop bind(thread) \ collapse(2)
for (i=0; i<N; ++i) {
    for (j=0; j<N; ++j) {
        x[j+N*i] *= 2.0;
    }
}
```

- The **target** construct offloads the enclosed code to the accelerator.
- The **loop** construct allows concurrent execution of the associated loops.

working soon
#pragma omp target teams
#pragma omp loop bind(teams)
for (i=0; i<N; ++i) {
    #pragma omp loop bind(thread)
    for (j=0; j<N; ++j) {
        x[j+N*i] *= 2.0;
    }
}

- The **target** construct offloads the enclosed code to the accelerator
- The **teams** construct creates a league of teams
- The **loop** construct allows concurrent execution of the associated loops, iterations are "logically" spread across the OpenMP threads in the binding thread set

working very soon
Continuum of Control

Descriptive

- Express "what"
- Ignore implementation
- Rely on quality of implementation

Prescriptive

- Express "how"
- Focus on implementation
- Expose control over execution

OpenMP strives to

- Support a useful subset of this spectrum
- Provide a structured path from descriptive to prescriptive where needed
What are the important features are important to applications?

- Loop construct
- Unified shared memory support
- Accelerator data management
  - Non-contiguous data mappings
- Memory allocators
- Metadirective and variants
- Tasks
  - Detach
  - Reductions
- Deep copy
- C++ virtual methods
Unified Virtual Memory Support

- Single address space over CPU and GPU memories
- Data migrated between CPU and GPU memories transparently to the application - no need to explicitly copy data

```c
#pragma omp requires unified_shared_memory
for (k=0; k < NTIMES; k++)
{
    // No data directive needed for pointers a, b, c
    #pragma omp target teams distribute parallel for
    for (j=0; j<ARRAY_SIZE; j++) {
        a[j] = b[j] + scalar * c[j];
    }
}
```
What are the important features are important to applications?

- Loop construct
- Unified shared memory support
- Accelerator data management
  - Non-contiguous data mappings
- Memory allocators
- Metadirective and variants
- Tasks
  - Detach
  - Reductions
- Deep copy
- C++ virtual methods
Non-contiguous data updates and mappings

allocate( a(nx, ny) )

!$OMP TARGET DATA MAP(to: a(1:nx/2, 1:ny) )

...

!$OMP TARGET TEAMS DISTRIBUTE

  ! a(1:nx/2, 1:ny) = a(1:nx/2, 1:ny)/nx

!$OMP END TARGET TEAMS DISTRIBUTE

...

!$OMP END TARGET DATA
What are the important features are important to applications?

• Loop construct
• Unified shared memory support
• Accelerator data management
  – Non-contiguous data mappings
• **Memory allocators**
• Metadirective and variants
• Tasks
  – Detach
  – Reductions
• Deep copy
• C++ virtual methods
#pragma omp target teams dist...
{
    double Scratchpad[PartitionSize];
    #pragma omp allocate(Scratchpad) \ 
    allocator(omp_pteam_mem_alloc)
}

// OR
double Scratchpad[PartitionSize];
#pragma omp target teams dist... \ 
    private(Scratchpad) \ 
    allocator(omp_pteam_mem_alloc)
{
    // Do stuff
}

- The **allocate** directive allows to place variables in different memory regions, e.g., `omp_pteam_mem_alloc` will put variables into "shared GPU memory"
- The **omp_alloc** runtime call allocates memory dynamically using a specified allocator, e.g., `omp_pteam_mem_alloc`
Example: Using Memory Allocators

```c
void allocator_example(omp_allocator_t *my_allocator) {
    int a[M], b[N], c;
    #pragma omp allocate(a) allocator(omp_high_bw_mem_alloc)
    #pragma omp allocate(b) // controlled by OMP_ALLOCTOR and/or omp_set_default_allocator
double *p = (double *) omp_alloc(N*M*sizeof(*p), my_allocator);

    #pragma omp parallel private(a) allocate(my_allocator:a)
    {
        some_parallel_code();
    }

    #pragma omp target firstprivate(c) allocate(omp_const_mem_alloc:c) // on target; must be compile-time expr
    {
        #pragma omp parallel private(a) allocate(omp_high_bw_mem_alloc:a)
        {
            some_other_parallel_code();
        }
    }

    omp_free(p);
}
```
What are the important features are important to applications?

- Loop construct
- Unified shared memory support
- Accelerator data management
  - Non-contiguous data mappings
- Memory allocators
- **Metadirective and variants**
- Tasks
  - Detach
  - Reductions
- Deep copy
- C++ virtual methods
Metadirective

```c
#pragma omp target teams
{
  #pragma omp metadirective
  when(device={kind(nohost)}: distribute parallel for) \
  default (parallel for)
  for(int i=0; i<N; i++)
    C[i] = A[i]+B[i];
}
```
Begin declare variant

// Nvidia
#pragma omp begin declare variant match(device={arch(nvptx)}, \ 
               implementation={score(1):vendor(llvm,ibm)})
float fast_sqrt(float __x) { return __nv_sqrt(__x); }  
#pragma omp end declare variant

// Intel
#pragma omp begin declare variant match(device={arch(haswell)}, \ 
               implementation={score(1):vendor(intel)})
float fast_sqrt(float __x) { return intel_asm_sqrt(__x); }  
#pragma omp end declare variant

// Default
float fast_sqrt(float __x) { return slow_sqrt(__x); }
What are the important features are important to applications?

- Loop construct
- Unified shared memory support
- Accelerator data management
  - Non-contiguous data mappings
- Memory allocators
- Metadirective and variants
- Tasks
  - Detach
  - Reductions
- Deep copy
- C++ virtual methods
What are the important features are important to applications?

- Loop construct
- Unified shared memory support
- Accelerator data management
  - Non-contiguous data mappings
- Memory allocators
- Metadirective and variants
- Tasks
  - Detach
  - Reductions
- Deep copy
- C++ virtual methods
Task Reductions

- Task reductions extend traditional reductions to arbitrary task graphs
- Extend the existing task and taskgroup constructs
- Also work with the taskloop construct

```c
int res = 0;
node_t* node = NULL;
...
#pragma omp parallel
{
    #pragma omp single
    {
        #pragma omp taskgroup task_reduction(+: res)
        {
            while (node) {
                #pragma omp task in_reduction(+: res) firstprivate(node)
                {
                    res += node->value;
                }
                node = node->next;
            }
        }
    }
}
```
What are the important features are important to applications?

- Loop construct
- Unified shared memory support
- Accelerator data management
  - Non-contiguous data mappings
- Memory allocators
- Metadirective and variants
- Tasks
  - Detach
  - Reductions
- **Deep copy**
- C++ virtual methods
- Interoperability with GPU streams
OpenMP 5.0 Improves Using Devices: Deep Copy Support

- Not all devices support shared memory so requiring it makes a program less portable

- Painstaking care was required to map complex data before 5.0

- OpenMP 5.0 adds deep copy support so that programmer can ensure that compiler correctly maps complex (pointer-based) data

```c
typedef struct mypoints {
    int len;
    double *needed_data;
    double useless_data[500000];
} mypoints_t;

// no declare target needed
int do_something_with_p(mypoints_t *p);

#pragma omp declare mapper(mypoints_t v)
    map(v.len, v.needed_data,
        v.needed_data[0:v.len])

mypoints_t * p = create_array_of_mypoints_t(N);

#pragma omp target map(p[:N])
{ do_something_with_p(p);
}
```
What are the important features are important to applications?

- Loop construct
- Unified shared memory support
- Accelerator data management
  - Non-contiguous data mappings
- Memory allocators
- Metadirective and variants
- Tasks
  - Detach
  - Reductions
- Deep copy
- C++ virtual methods
- Interoperability with GPU streams
Classes with virtual methods

class Base {
    virtual void something() = 0;
    virtual void mapSelf() = 0;
}
class Derived : public Base {
    void something() override { /* do logic */ }
    void mapSelf() override {
        #pragma omp target enter data map(to:this[0])
    }
}

void foo() {
    Derived d;
    d.mapSelf();
    bar(&d);
}
void bar(Base *b) {
    #pragma omp target
    b->something();
}
OpenMP 5.0 will support other C++ accelerator frameworks

- Number of related technologies: Kokkos, RAJA, OpenACC, CUDA/HIP, SYCL
- Goal is to deliver enhanced OpenMP to address increasing heterogeneity and complexity of systems (e.g. accelerator offloading, tasks)

<table>
<thead>
<tr>
<th>Languages</th>
<th>CUDA / HIP</th>
<th>Kokkos</th>
<th>OpenACC</th>
<th>OpenMP 5.0</th>
<th>RAJA</th>
<th>SYCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>C/C++</td>
<td>C/C++</td>
<td>C/C++/ Fortran</td>
<td>C/C++/ Fortran</td>
<td>C/C++</td>
<td>C/C++</td>
<td>C/C++</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Prog. Style</th>
<th>Kokkos</th>
<th>RAJA</th>
<th>Service Layers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Template Meta-programming, C++11 lambdas</td>
<td>C++11 lambdas</td>
<td>OpenCL</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parallelism</th>
<th>OpenMP, Pthreads, CUDA, HIP</th>
<th>SIMD, Fork-Join, CUDA, HIP</th>
<th>SPMD, SIMD, Tasks, Fork-Join, CUDA, HIP</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIMT</td>
<td>SIMT</td>
<td>SIMT</td>
<td>SPMD</td>
</tr>
<tr>
<td>CUDA, HIP</td>
<td>CUDA, HIP</td>
<td>CUDA, HIP</td>
<td>OpenMP, CUDA, HIP, HIP</td>
</tr>
<tr>
<td>HIP</td>
<td>HIP</td>
<td>HIP</td>
<td>HIP</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Licensing/Accessibility</th>
<th>Proprietary</th>
<th>Open-sourced</th>
<th>Few compilers</th>
<th>Not on all arch.</th>
<th>Open-sourced</th>
<th>Open-sourced</th>
<th>Royalty-Free</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstraction Level</td>
<td>Low</td>
<td>Medium</td>
<td>High</td>
<td>High</td>
<td>Medium</td>
<td>Medium</td>
<td>Medium</td>
</tr>
</tbody>
</table>

HIP targets are work-in-progress activities
What are the important features are important to applications?

- Loop construct
- Unified shared memory support
- Accelerator data management
  - Non-contiguous data mappings
- Memory allocators
- Metadirective and variants
- Tasks
  - Detach
  - Reductions
- Deep copy
- C++ virtual methods
- Interoperability with GPU streams
Interop: get stream/queue/etc.

omp_interop_t o = OMP_INTEROP_NONE; intptr_t type;

#pragma omp interop tasksync init obj(o) depend(inout: a)
omp_get_interop_property(o, OMP_INTEROP_TYPE, &type);
if (type == OMP_INTERFACE_CUDA) {
    cudaStream_t s;
    omp_get_interop_property(o, OMP_INTEROP_TASKSYNC, &s);
    cublasSetStream(s);
    call_cublas_async_stuff();
} else {
    // handle other cases
}

#pragma omp interop tasksync destroy obj(o) depend(inout: a)
Tuning OpenMP target: Thread Blocking Effects

```c
#pragma omp target
#pragma omp teams distribute num_teams(nblocks) thread_limit(nthreads)
for(int ss=0; ss<nblocks; ss++) {
    #pragma omp parallel for
    for(int tt=0; tt<nthreads; tt++) {
        auto tmp = eval(ss*nthreads+tt, expr);
        vstream(me[ss*nthreads+tt], tmp);
    }
}
```

<table>
<thead>
<tr>
<th>nblocks</th>
<th>nthreads</th>
<th>GB/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default</td>
<td>Default</td>
<td>240</td>
</tr>
<tr>
<td>65536</td>
<td>8</td>
<td>162</td>
</tr>
<tr>
<td>32768</td>
<td>32</td>
<td>252</td>
</tr>
<tr>
<td>640</td>
<td>128</td>
<td>289</td>
</tr>
<tr>
<td>4096</td>
<td>256</td>
<td>306</td>
</tr>
</tbody>
</table>

Code from GridMini in ECP's Lattice QCD
## Functionality of OpenMP C Implementations Based on SOLLVE’s V&V

<table>
<thead>
<tr>
<th>Application Kernels</th>
<th>SUMMIT</th>
<th>OBVIAN*</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Compiler result</td>
<td>Runtime result</td>
</tr>
<tr>
<td>linked list</td>
<td>PASS</td>
<td>PASS</td>
</tr>
<tr>
<td>mmm target</td>
<td>PASS</td>
<td>PASS</td>
</tr>
<tr>
<td>mmm target parallel for simd</td>
<td>PASS</td>
<td>PASS</td>
</tr>
<tr>
<td>gmpc&amp;k target static lib</td>
<td>PASS</td>
<td>PASS</td>
</tr>
<tr>
<td>reduction separated directives</td>
<td>PASS</td>
<td>PASS</td>
</tr>
<tr>
<td>nested target simd</td>
<td>PASS</td>
<td>PASS</td>
</tr>
<tr>
<td>target data</td>
<td>PASS</td>
<td>PASS</td>
</tr>
<tr>
<td>target enter data</td>
<td>PASS</td>
<td>PASS</td>
</tr>
<tr>
<td>target enter exit data</td>
<td>PASS</td>
<td>PASS</td>
</tr>
<tr>
<td>target parallel</td>
<td>PASS</td>
<td>PASS</td>
</tr>
<tr>
<td>target private</td>
<td>PASS</td>
<td>PASS</td>
</tr>
<tr>
<td>target simd</td>
<td>PASS</td>
<td>PASS</td>
</tr>
<tr>
<td>target teams distribute</td>
<td>PASS</td>
<td>PASS</td>
</tr>
<tr>
<td>target teams distribute parallel for</td>
<td>PASS</td>
<td>PASS</td>
</tr>
<tr>
<td>target teams distribute parallel for devices</td>
<td>PASS</td>
<td>PASS</td>
</tr>
<tr>
<td>target update</td>
<td>PASS</td>
<td>PASS</td>
</tr>
<tr>
<td>task target</td>
<td>PASS</td>
<td>PASS</td>
</tr>
</tbody>
</table>

**Figure 1:** Table of test results for OpenMP C Implementations

For further information, e.g., understanding of failures, visit: [https://crpl.cis.udel.edu/ompvvssollve/results/](https://crpl.cis.udel.edu/ompvvssollve/results/)

*Obivan is a HPC cluster @ UDel*
Functionality of OpenMP C++ Implementations

Tests based on GEMV from QMCPack.

<table>
<thead>
<tr>
<th>Application Kernels</th>
<th>SUMMIT Compiler result</th>
<th>RUNTIME result</th>
<th>OBVIAN Compiler result</th>
<th>RUNTIME result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alpaka - complex template</td>
<td>PASS</td>
<td>PASS</td>
<td>PASS</td>
<td>PASS</td>
</tr>
<tr>
<td>GEMV - target</td>
<td>PASS</td>
<td>PASS</td>
<td>PASS</td>
<td>PASS</td>
</tr>
<tr>
<td>GEMV - target many matrices</td>
<td>PASS</td>
<td>PASS</td>
<td>PASS</td>
<td>PASS</td>
</tr>
<tr>
<td>GEMV - target reduction</td>
<td>PASS</td>
<td>PASS</td>
<td>PASS</td>
<td>PASS</td>
</tr>
<tr>
<td>GEMV - target teams dist par for</td>
<td>PASS</td>
<td>PASS</td>
<td>PASS</td>
<td>PASS</td>
</tr>
<tr>
<td>reduction separated directives</td>
<td>PASS</td>
<td>PASS</td>
<td>PASS</td>
<td>PASS</td>
</tr>
<tr>
<td>target map classes default</td>
<td>PASS</td>
<td>PASS</td>
<td>PASS</td>
<td>PASS</td>
</tr>
<tr>
<td>target data map classes</td>
<td>PASS</td>
<td>PASS</td>
<td>PASS</td>
<td>PASS</td>
</tr>
<tr>
<td>target enter data classes inheritance</td>
<td>PASS</td>
<td>PASS</td>
<td>PASS</td>
<td>PASS</td>
</tr>
<tr>
<td>target enter data classes simple</td>
<td>PASS</td>
<td>PASS</td>
<td>PASS</td>
<td>PASS</td>
</tr>
<tr>
<td>target enter exit data classes</td>
<td>PASS</td>
<td>FAIL</td>
<td>FAIL</td>
<td>FAIL</td>
</tr>
</tbody>
</table>

Figure 3: Table of test results for OpenMP C++ Implementations.

→ Most OpenMP offload features in all OpenMP implementations work.

→ target enter exit data isn’t supported properly across any OpenMP implementations.

For further information, e.g., understanding of failures, visit: https://crpl.cis.udel.edu/ompvsollve/results/

Courtesy Swaroop Pophale (ORNL) and David Bernholdt (ORNL)
OpenMP Offload in HPGMG

- HPGMG is a DOE benchmark which may be included in the SPEC HPC 2020 benchmark suite
- The plot compares CUDA, OpenACC, and OpenMP performance on 1 socket of the Summit supercomputer using 3 MPI ranks and 3 GPUs for the Unified Memory version of HPGMG run on Summit.
- Results for the explicit data management version of HPGMG will be shown at a later date when the IBM compiler fixes a reported bug and the CCE compiler supports OpenMP pointer attachment

Figure 5: Performance of HPGMG with different Implementations.

Higher is better

IBM xl's OpenMP offload performance shown with HPGMG is encouraging
Performance of SU3 LQCD Benchmark with OpenMP Libraries

- Developed benchmark code representative of applications in ECP Application Project LQCD. The code is at https://bitbucket.org/dwdoerf/su3_bench.
- Ran with three different OpenMP libraries, with CUDA and with PGI’s OpenACC.
- Note that the peak GF/s in plots refers to the theoretical floating point performance based on the Arithmetic Intensity of the offloaded kernel. A Volta GPU has a peak GF/s of 7800 GF/s for kernels which are not bound by memory bandwidth.

```c
#pragma omp target teams distribute
for(int i=0; i<1048576; ++i) {
    #pragma omp parallel for collapse(3)
    for(int j=0; j<4; ++j) {
        for(int k=0; k<3; k++) {
            for(int l=0; l<3; l++) {
                Complex cc;
                for(int m=0; m<3; m++) {
                    cc += d_a[i].link[j].e[k][m] * d_b[j].e[m][l];
                }
                d_c[i].link[j].e[k][l] = cc;
            }
        }
    }
}
```

Listing 1: GPU Computation region of SU3 benchmark

- Results for SU3 benchmark run on NVIDIA Tesla V100 with different OpenMP libraries (left plot) shows how clang provides best performance of 640 GFLOP/s.
- The performance of clang OpenMP is 3% of peak and is very low compared to other Compilers. However, manual SPMDization of code can reduce implicit memory flushes and increases performance to 401 GFLOP/s.
- Ongoing changes in clang OpenMP can provide better performance over the other OpenMP vendor libraries.
Conclusions

- On ECP Systems (particularly Summit) compilers are ready for device offload. Fundamental features are available. Still, tests could be improved to handle real-world data structures with pointers.
- Applications can move towards OpenMP offload using clang/LLVM OpenMP as it has support for many new OpenMP 5.0 offload features.
- IBM’s support for OpenMP offload for C is mature. Could be improved for Fortran.
- Performance of IBM OpenMP offload is 70% of CUDA performance in HPGMG.
- While QMCPack currently relies on IBM xl OpenMP for offload, it’s recently (a) shown to potentially have good performance from using LLVM clang OpenMP offload support and (b) works with other vendor compilers
- The SPEC HPC 2020 benchmark suite is under active development. The benchmarks will be pruned over the next few months based on benchmark readiness and formally meeting benchmark suite requirements (no new benchmarks will be considered at this stage).