The Design and Implementation of OpenMP 4.5 and OpenACC Backends for the RAJA C++ Performance Portability Layer

William Killian
Millersville University of Pennsylvania
University of Delaware
Lawrence Livermore National Lab

Tom Scogland, Adam Kunen
Lawrence Livermore National Lab

John Cavazos
University of Delaware

Fourth Workshop on Accelerator Programming Using Directives (WACCPD)

November 13, 2017

Motivation

- Directive-based languages allow for *performance portability***
  - Relatively easy to integrate
  - Minimal SLOC changes
  - Adaptable to many architectures
- Not all Directive-based languages work across all toolchains
  - Need the right compiler + runtime
- Different codes may need to run on a particular platform with a specific toolchain (performance, reproducibility)
  - Cannot always rely on having OpenMP 4.5 or OpenACC everywhere
- Autotuning and optimization space exploration

OpenACC and OpenMP 4.5

- Language + Runtime describing parallelism for GPUs and accelerators

- OpenACC – Implicit + Explicit
- OpenMP – Explicit

- Tradeoffs
  - Do programmers always want full control?
  - Do users always know what's better?

RAJA Performance Portability Abstraction Layer

- Developed for codesign at Lawrence Livermore National Laboratory
- Embedded DSL in C++11 with several backends:
  - Intel TBB, OpenMP, CUDA, SIMD
- Three main components:
  1. Execution Policies
  2. Reductions
  3. Iterables

```
RAJA::forall(ExecPolicy(), iterable, [=] (int i) {
  // loop body
}
```

- This talk focuses on *Execution Policies*

Execution Policies in RAJA

- A C++ type
- Contains any information pertinent toward code generation
    - Sometimes just the type: `RAJA::seq_exec`
    - Sometimes with additional arguments: `RAJA::cuda_exec<128, false>`

- For OpenACC and OpenMP 4.5, we propose defining a set of *Execution Policy* building blocks

- Building blocks are composed to define high-level execution policies.

Adding a Directive-Based Language Backend

1. Define all pragma language grammar tokens as policy tags:

```
namespace omp::tags {
  // region-based tags
  struct Parallel {};
  struct BarrierAfter {};
  // ... BarrierBefore, etc.

  // construct- and clause-based tags
  struct For {};
  struct Static {};
  // ... Guided, Dynamic
}
```

Adding a Directive-Based Language Backend

2. Construct Policies for each high-level grammar rule

```
template <typename Inner>
struct Parallel : policy<tags::Parallel> {
  using inner = Inner;
};
```

```
template <typename... Options>
struct For : policy<tags::For>, Options... {};
```

List of optional clauses

Adding a Directive-Based Language Backend

3. Compose high-level Execution Policies

```
template <unsigned int N>
using omp_parallel_static_exec =
  omp::Parallel<
    omp::BarrierAfter<
      omp::For<omp::Static<N>>>>;
```

```
#pragma omp parallel
{
  {
    #pragma omp for nowait schedule(static, N)
    // loop and body emitted here
  }
  #pragma omp barrier
}
```

Three instantiations of `RAJA::forall` at compile-time

Adding a Directive-Based Language Backend

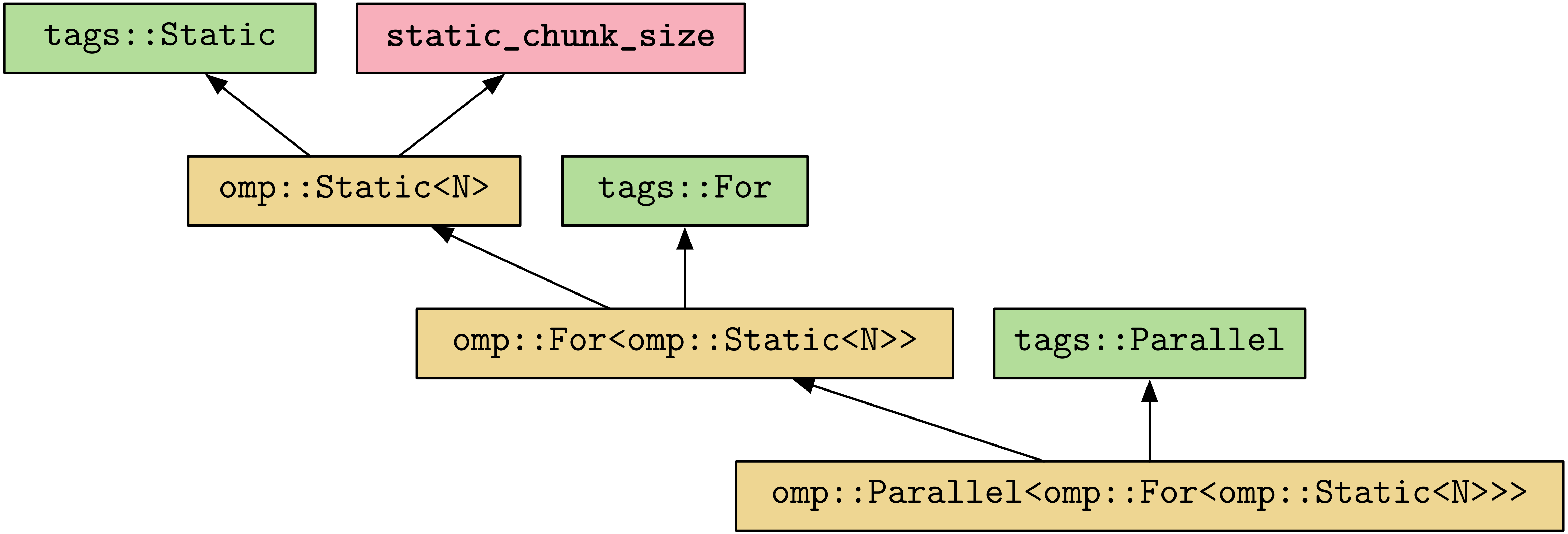
4. Provide Code Specializations for each Execution Policy

```
template <typename Exec, typename Iterable, typename Body>

exact<Exec, omp::tag_list, tags::For, tags::Static>
// does our execution policy exactly match For+Static ?

forall_impl(const Exec &&, Iterable && iter, Body && body) {
  auto size = iter.size();
  #pragma omp for nowait schedule(static, Exec::static_chunk_size)
  for (decltype(size) i = 0; i < size; ++i) {
    body(*(iter.begin() + i));
  }
}
```

Instantiated Type Hierarchy for omp_parallel_for_static<N>



Flat type for omp_parallel_for_static<N>

`tags::Parallel`	`tags::For`	`tags::Static`	`static_chunk_size`

- Code generation must be specialized for each possible permutation
  - `#pragma omp for nowait`
  - `#pragma omp for nowait schedule(static, N)`

- When writing options for OpenACC and OpenMP 4.5, there is state explosion
  - `#pragma acc loop`
  - `#pragma acc loop independent`
  - `#pragma acc loop gang`
  - `#pragma acc loop vector`
  - `#pragma acc loop worker`

OpenMP 4.5 Building Blocks

- All of the OpenMP Building Blocks PLUS:
  - `Target`, `Teams`, and `Distribute` tags
- Aggregate Policy definitions for:
  - `TargetTeamsDistribute`
  - `TargetTeamsDistributeParallelFor`
  - `TargetTeamsDistributeParallelStatic`
- Define a dispatch overload for forall with all OpenMP 4.5 policies
- Define built-in policies for some OpenMP 4.5 policies:
  - `OMP_Target`, `omp_target_teams_distribute_parallel_for`, and a few others.

OpenACC Building Blocks

- All OpenACC clause and directive names as tags:
  - `Parallel, Kernels, Loop, Independent, Gang, Worker, Vector, NumGangs, NumWorkers, VectorLength`
- Aggregate Policy definitions for each clause
  - Parallel, Loop, Independent, etc.
- Dispatch overload for OpenACC policies
- RAJA::forall specializations for:
  - `acc (parallel|kernels) (num_gangs)? (num_workers)? (vector_length)?`
    - 16 possible versions
  - `acc loop (independent)? (gang)? (worker)? (vector)?`
    - 16 possible versions

Evaluation Machine + Toolchains

- IBM POWER 8 + NVIDIA P100
  - CORAL EA System @ LLNL
  - 2x 10-core @ 4.0GHz
  - 256GB DDR3 RAM
  - NVLINK
- IBM Clang w/ OpenMP 4.5 support
- PGI Compiler 17.7 w/ OpenACC (nocopy lambda support)
- CUDA 8.0.61

Evaluation Benchmarks

- 9 Synthetic Kernels:
  - Jacobi-1D, Jacobi-2D, Heat-3D
  - Matrix-Matrix Multiplication
  - Matrix-Vector Multiplication
  - Vector Addition
  - Tensor-2D, Tensor-3D, Tensor-4D
- A C++ view class was used to represent multi-dimensional arrays
- No Reductions were used – emphasis on execution policies

Example: Heat-3D (OpenACC)

```
#pragma acc parallel num_gangs(16), vector_length(128)
{
  #pragma acc loop gang
  for (int i = 1; i < n - 1; ++i)
    #pragma acc loop worker
    for (int j = 1; j < n - 1; ++j)
      #pragma acc loop vector
      for (int k = 1; k < n - 1; ++k)
        B(i,j,k) = 0.125 * (A(i + 1, j, k) - 2.0 * A(i, j, k) + A(i - 1, j, k))
                 + 0.125 * (A(i, j + 1, k) - 2.0 * A(i, j, k) + A(i, j - 1, k))
                 + 0.125 * (A(i, j, k + 1) - 2.0 * A(i, j, k) + A(i, j, k - 1))
                 + A(i,j,k);
}
```

Example: Heat-3D (RAJA+OpenACC)

```
using ExecPol = RAJA::NestedPolicy<
  RAJA::ExecList<
    RAJA::acc::Loop<RAJA::acc::Gang>,
    RAJA::acc::Loop<RAJA::acc::Worker>,
    RAJA::acc::Loop<RAJA::acc::Vector>>,
  RAJA::acc::Parallel<RAJA::acc::NumGangs<16>, RAJA::acc::VectorLength<128>>>;
const RAJA::RangeSegment r (1, n - 1);
RAJA::forallN<ExecPol>(r, r , r, [=] (int i, int j, int k) {
  B(i,j,k) = 0.125 * (A(i + 1, j, k) - 2.0 * A(i, j, k) + A(i - 1, j, k))
           + 0.125 * (A(i, j + 1, k) - 2.0 * A(i, j, k) + A(i, j - 1, k))
           + 0.125 * (A(i, j, k + 1) - 2.0 * A(i, j, k) + A(i, j, k - 1))
           + A(i,j,k);
});
```

Evaluation Criteria

- Compilation Overhead (%)
  - RAJA + OpenACC backend vs. OpenACC
  - RAJA + OpenMP 4.5 backend vs. OpenMP 4.5
- Execution Overhead (%)
  - RAJA + OpenACC backend vs. OpenACC
  - RAJA + OpenMP 4.5 backend vs. OpenMP 4.5

$$\frac{(\mathrm{time}_{\mathrm{RAJA}} - \mathrm{time}_{\mathrm{Directive}})}{\mathrm{time}_{\mathrm{Directive}}}$$

Compilation Overhead

Kernel	OpenACC	OpenMP 4.5
Jacobi 1D	**17.50%**	**8.75%**
Jacobi 2D	50.24%	20.42%
Heat 3D	74.40%	**30.91%**
Matrix Matrix Multiplication	**80.28%**	31.24%
Matrix Vector Multiplication	45.41%	16.47%
Vector Addition	**15.20%**	**6.24%**
Tensor 1D	48.94%	17.57%
Tensor 2D	72.85%	27.53%
Tensor 3D	**120.74%**	**59.29%**
Average	**95.07%**	**38.78%**

Compilation Overhead

- Additional template instantiations (3 per `RAJA::forall`)
- Overload resolution goes from none to at least 8 per `forall` call
  - For OpenMP 4.5, 7 additional resolutions per `forall`
  - For OpenACC, 32 additional resolutions per `forall`
- 5 additional types created per nest level with `forallN`

- With a 3-nested loop with the OpenACC backend enabled:
  - > 18 type constructions
  - > 110 overload resolution attempts

Execution Overhead

Kernel	OpenACC	OpenMP 4.5
Jacobi 1D	**2.52%**	**1.94%**
Jacobi 2D	1.25%	1.14%
Heat 3D	1.08%	1.19%
Matrix Matrix Multiplication	**0.96%**	**1.01%**
Matrix Vector Multiplication	1.13%	1.38%
Vector Addition	**0.21%**	**0.38%**
Tensor 1D	0.98%	1.21%
Tensor 2D	1.34%	1.44%
Tensor 3D	**2.18%**	**2.14%**
Average	**1.66%**	**1.69%**

Conclusion and Future Work

- Proposed Execution Policy constructs for OpenACC and OpenMP 4.5
- Implemented OpenACC and OpenMP 4.5 backends for RAJA
- Showed significant compilation overhead when using RAJA
- Showed minor execution overhead when using RAJA

- Leverage/Propose conditional clause execution with directives
  - Avoids switchyard of SFINAE during compilation
- Add full reduction support to the proposed backends

http://www.github.com/LLNL/RAJA