Portable Performance with OpenMP in a Heterogeneous Era

Tom Scogland

April 25, 2017
Why expand OpenMP to heterogeneous targets?
Why expand OpenMP to heterogeneous targets?

- Heterogeneity and hierarchical complexity is constantly increasing
  - Accelerators add physical heterogeneity and distributed memory
    - GPUs, FPGAs, Co-processors (Intel MIC, Tilera Tile64, etc.)
  - System topology adds heterogeneity through locality imbalance
    - Non-Uniform Memory Access (NUMA) memory systems
    - Operating system imbalance, work unevenly distributed to cores
    - Non-uniform latency to peripheral devices

- We need heterogeneous computing
  - Better energy efficiency
  - More performance without increasing clock speed

- C++ Templated abstractions (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******!
The complexity is no longer avoidable
OpenMP’s evolution so far: From multicore to distributed memory manycore

- **OpenMP 2.5:**
  - Shared memory only
  - Multicore, multithreaded abstractions

- **OpenMP 3.0:**
  - Asynchronous tasking added

- **OpenMP 4.0:**
  - Target constructs add device offload
  - Mapping constructs add structured in-node views on (possibly) distributed memory, simple one-per-device model

- **OpenMP 4.5:**
  - Unstructured data mapping for extra flexibility
  - Better handling of offloaded functions
What are we doing to shape what comes next?

- Compiler-assisted pipelining for streaming computations
- Mapping complex data structures
- Callable tasks: Lambdas with OpenMP for portable abstractions
Pipelining for OpenMP: Why pipelining?

- Default synchronous data motion

- Pipelined data motion
Pipelining and Buffering for OpenMP: 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

- 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

- Optionally we can do more with:
  - Number of concurrent transfers
  - Memory limits
  - Schedulers
  - Etc.
Pipelining in OpenMP:
A non-pipelined stencil example using OpenMP 4.5

```c
#pragma omp target data
   map(to:A0[0:nz-1][0:ny-1][0:nx-1])
   map(to: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 with our proposed extension

```c
#pragma omp target \\
pipeline(\texttt{static}[1,3])\\npipeline\_map(\texttt{to:A0[k-1:3][0:ny-1][0:nx-1]})\\npipeline\_map(\texttt{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[\texttt{Index3D} (i, j, k)] =
        (A0[\texttt{Index3D} (i, j, k + 1)] +
         A0[\texttt{Index3D} (i, j, k - 1)] +
         A0[\texttt{Index3D} (i, j + 1, k)] +
         A0[\texttt{Index3D} (i, j - 1, k)] +
         A0[\texttt{Index3D} (i + 1, j, k)] +
         A0[\texttt{Index3D} (i - 1, j, k)])*c1
        - A0[\texttt{Index3D} (i, j, k)]*c0;
    }
  }
}
```
Pipelining in OpenMP: Pipelined and buffered with our proposed extension

```c
#pragma omp target \
pipeline(static[1,3])\npipeline_map(to:A0[k-1:3][0:ny-1][0:nx-1])\npipeline_map(from:Anext[k:1][0:ny-1][0:nx-1])\npipeline_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[Index3D(i,j,k+1)] =
                (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;
        }
    }
}
```

Replicating this manually requires ~20 more lines of error-prone boilerplate!
Pipelining in OpenMP: Kernel and benchmark performance

Nearly 2x speedup!

Only 1.5x, why?

All results with PGI OpenACC on k40 GPUs, surface cluster
Pipelining in OpenMP: Lattice QCD benchmark memory usage

Buffering reduces memory by 80%
Mapping complex data in OpenMP

- Pipelining is one complex data motion pattern in OpenMP, but it can be generalized into a larger class of problems.

- OpenMP data mapping requires that:
  - Mapped data is made visible to the target device.
  - Data cannot be assumed to have been copied.
  - Data cannot be assumed *not* to have been copied.
  - All copies that take place are *bitwise* in this context.

- Given these restrictions:
  - Plain data works fine.
  - Mapping pointer-based structures and structure members requires explicitly mapping all of the pieces everywhere they’re used.
  - C++, especially the STL, is extremely complex and frequently impossible due to privatization.
Deep Copy:
Potential solutions to map complex data structures

- The OpenACC solution: Allow “pointer attachment” when mapping a piece of data
  - Requires mapping data in a specific order
  - Extremely verbose wherever data mapping is taking place
  - Does not compose

- The annotation solution: Annotate structures to describe how to map their contents entirely with directives
  - Simple cases are very simple to express
  - Allows the compiler to generate optimized copy routines
  - Breaks down, or becomes unwieldy at best, given cyclic data structures

- Our solution: Express data motion for OpenMP as a serialization and deserialization process, much like MPI
  - Routines can be composed to map larger objects from smaller ones
  - Verbosity is kept separate from usage
  - Allows users to re-use existing serialization code
  - General enough to allow data transformation for performance as well
  - Performance may be lower than compiler-generated code
Deep Copy for OpenMP: Why explicit deep copy, array of arrays?

- Default piecewise, or attachment-based, data motion

Diagram showing the process of deep copy for arrays in OpenMP, with arrows indicating copy in and copy out operations, and blocks indicating compute and attach pointers functions.
Deep Copy for OpenMP: Why explicit deep copy, array of arrays?

- Deep-copy marshal-and-fix
Major uses for data serialization and transformation in this model

- User-, or implementer-, defined mappers for complex data types

- Re-structuring of data to better exploit a target platform:
  - Packing pointer-chasing structures
  - Skipping unused data members
  - Transposing or otherwise re-shaping matrices

- Ensuring data is mapped in the right order
  - Building mapping into types without deep-copy support can cause the same data to be re-mapped repeatedly
Enabling RAJA (or Kokkos) for non-C++ languages with OpenMP

- RAJA relies on C++ lambdas to implement loop look-alike abstractions as function calls, for example:
  
  ```
  forall<omp_parallel_for_exec>(0, 100, [=](int i){ a[i] = b[i]; });
  ```

- Lambdas provide powerful abstractions, but with downsides
  - No support in C (clang closure support exists but is not portable)
  - No support in Fortran

- What is a Lambda?
  - A user-controlled mechanism for outlining code and capturing a data environment

- What is an OpenMP task?
  - A user-controlled mechanism for outlining code and capturing a data environment...
Closures with OpenMP:
Extensions required to make tasks reusable and callable

1. Add arguments to the generated function
2. Retrieve a pointer to the outlined function
3. Retrieve a pointer to the captured environment
4. Clean up the captured data
   - Just these are enough to make closure behavior from tasks possible, if not with ideal language-level syntax
   - These extensions have been discussed in OpenMP language committee meetings, to some interest
Could RAJA or Kokkos be used from C or Fortran?

- Short answer: yes, with a but

- Long answer, RAJA could be used by:
  - Explicitly instantiating policies
  - Building a C (or Fortran) interface to call the instantiations
  - Invoking the interface with a built closure

- Efficiency with this approach will require much more optimization work at link time

- Functionally however, C and Fortran could leverage RAJA or a sister library