OpenACC “kernels” Improvements

Linux Plumbers Conference 21 - GNU Tools Track
Agenda

OpenACC “kernels”
Graphite
Other Improvements
Final Example
OpenACC “kernels”
subroutine row_sum(input, sums)
  integer :: input(:,::)
  integer :: sums(:)
  integer :: i, j
  integer :: sum

  do i = 1, size(input, 1)
    sum = 0
    do j = 1, size(input, 2)
      sum = sum + input(i,j)
    end do
    sums(i) = sum
  end do
end subroutine row_sum

subroutine row_sum(input, sums)
  integer :: input(:,::)
  integer :: sums(:)
  integer :: i, j
  integer :: sum

  !acc kernels
  !acc parallel copyin(input) copyout(sums) private(sum)
  !acc loop independent
  do i = 1, size(input, 1)
    sum = 0
    !pragma acc loop seq
    do j = 1, size(input, 2)
      sum = sum + input(i,j)
    end do
    sums(i) = sum
  end do
  !acc end parallel
end subroutine row_sum

subroutine row_sum(input, sums)
  integer :: input(:,::)
  integer :: sums(:)
  integer :: i, j
  integer :: sum

  !acc kernels
  do i = 1, size(input, 1)
    sum = 0
    do j = 1, size(input, 2)
      sum = sum + input(i,j)
    end do
    sums(i) = sum
  end do
  !acc end kernels
end subroutine row_sum
OpenACC “kernels” in GCC

So far:
- Different internal representation than “acc parallel” regions with many restrictions (e.g. no explicit “reduction” clauses)
- Data-dependence analysis in “parloops” pass
- Restricted assignment of parallel execution dimensions

=> unable to analyze/parallelize real HPC code => bad performance

New:
- Lift restrictions on “acc kernels” regions
  - Allow automatic annotation of inner loops in “kernels” regions
  - Allow calls to builtins and intrinsics
  - Allow more general loop bound expressions in “kernels” loops
- Unify internal representation of “kernels” and “parallel” regions
- Use more powerful data-dependence analysis based on “Graphite”

Status:
- Commit to devel/omp/gcc-11 branch soon
- Submission for mainline soon after
Graphite

- Generic framework for data-dependence analysis and loop-transformations.

- Current uses in GCC:
  - "-floop-paralize-all"
  - "-floop-nest-optimize"

- Based on geometrical “polyhedral compilation” approach:
  - Loops become polyhedra
  - Enables use of mathematical tools on this representation (e.g. integer linear programming) for analysis and transformation
  - Complete representation of the loop structure, can be transformed back to GIMPLE

Pros and Cons:
+ Well-understood approach
+ Can already represent a wide class of loops
+ Quite stable
- Development of Graphite has become stagnant
  - But polyhedral compilation is alive (e.g. LLVM Polly) and we can catch up with recent developments!
- Some restrictions need to be lifted to make it work well on real-world code
Graphite for OpenACC

Rough outline of OpenACC region representation:
- Outline “parallel”, “kernels” etc. regions into a function (“omp_fn”) very early in the pass pipeline
- Represent information about OpenACC loop structure, clauses etc. in internal function calls
- Lower internal function calls in a later step in an offloading device specific way
  - => loop bounds now depend on runtime information!

Some difficulties:
- Graphite runs much later than OpenACC lowering => OpenACC device lowering pass must be moved
- Optimization passes now have to deal with OpenACC’s internal function calls
- Graphite works on CFG loops and does not understand OpenACC’s loop structure
  - OpenACC lowering introduces additional CFG loops and dependences
    - Pretend to Graphite that it analyzes the “original” loop
    - Graphite must know about “private”, “firstprivate” variables and remove fake dependences
  - Some parallelization-enabling transformations have not occurred when Graphite runs
    - Graphite must remove “reduction” dependences
  - => We use Graphite data-dependence analysis only and skip code generation
  - => Future project? Teach Graphite’s code generation to preserve the OpenACC loop structure to enable its use for code transformations
Graphite enhancements

Data-dependence analysis is much more important for OpenACC kernels than for previous use cases.

- Lift simple restrictions for OpenACC outlined functions:
  - Increase parameter values meant to restrict resource use
    - “kernels” regions are usually small and Graphite’s heavy resource use is not a major problem
  - Operate on otherwise “unprofitable” loop-nests (loop-nests oft depth 1, not iterating loops etc.)

- Support runtime alias checking
  - Graphite must know which data-references might alias
  - Old approach: Bail out if aliasing cannot be analyzed statically
    - Not acceptable: rules out most non-trivial C code, a lot of Fortran code
  - New approach:
    - Continue Graphite execution if aliasing cannot be analyzed statically
    - Remember unanalyzed data-reference pairs
    - Create runtime alias check expression for all such data-references in a SCoP
    - Fallback to sequential execution of all loops in SCoP if aliasing is detected at runtime
Other Improvements
Supporting enhancements: Delinearized Array Accesses

- **Delinearization of array accesses** in the Fortran frontend
  - C-style dynamical multidimensional array access is linearized:
    - \( A[i][j] \) becomes \(*((int*)A + i*n + j)\)
    - Not an **affine** expression of the variables => cannot be represented by Graphite
    - Fortran has proper multidimensional arrays, but uses the **same kind of representation internally**
- **Solution**: Change Fortran frontend to emit nested ARRAY_REFs for the individual dimensions instead of a linearized expression
- **Status**: Working; some case are not covered yet (e.g. scalarized array accesses)

**Possible improvement**: Middle-end delinearization

- Delinearization at the GIMPLE level or at the data reference level (tree-data-ref.c)
- All languages could benefit from this
- See e.g. “Optimistic Delinearization of Parametrically Sized Arrays” [Grosser, Ramanujam, Pouchet, Sadayappan, Pop ICS 15]
Supporting enhancements: OpenACC synthetic “private” clauses

- Automatically add “private” clauses to “kernels” regions
- New pass `pass_omp_data_optimize`
- Runs before `pass_lower_omp`
- Adds “private” to whole regions only

Ideas for improvements:
- Synthetic “reduction” clauses
- Synthetic clauses on loops
  - Run later as loop optimization pass?
  - Would have to repeat “private” clause lowering
Final Example
This code was not parallelized by the old "kernels" implementation:

```
subroutine row_sum(input, sums)
  integer :: input(:,::)
  integer :: sums(:)
  integer :: i,j
  integer :: sum
  !acc kernels
  do i = 1, size(input, 1)
    sum = 0
    do j = 1, size(input, 2)
      sum = sum + input(i,j)
    end do
    sums(i) = sum
  end do
  !acc end kernels
end subroutine row_sum
```

Now it is essentially equivalent to the following explicitly parallelized code:

```
subroutine row_sum(input, sums)
  integer :: input(:,::)
  integer :: sums(:)
  integer :: i,j
  integer :: sum
  !acc kernels
  !pragma acc parallel copyin(input) copyout(sums) private(sum)
  !acc loop independent
  do i = 1, size(input, 1)
    sum = 0
    !pragma acc loop seq
    do j = 1, size(input, 2)
      sum = sum + input(i,j)
    end do
    sums(i) = sum
  end do
  !acc end parallel
end subroutine row_sum
```
Acknowledgement

This research used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725

Disclaimer

© Siemens 2021

Subject to changes and errors. The information given in this document only contains general descriptions and/or performance features which may not always specifically reflect those described, or which may undergo modification in the course of further development of the products. The requested performance features are binding only when they are expressly agreed upon in the concluded contract.

All product designations may be trademarks or other rights of Siemens AG, its affiliated companies or other companies whose use by third parties for their own purposes could violate the rights of the respective owner.