Vectorization Pragmas in LLVM: An RFC

Yashas Andaluri, Happy Mahto, M Sai Praharsh, Bhavya Bagla
IIT Hyderabad
Aug 8th, 2019

[Thanks to feedback from Venugopal Raghavan, Shivarama Rao, Dibyendu Das (AMD) and Michael Kruse & Hal Finkel (ANL).]

Vectorization Pragmas

- ivdep

a. Planned pragma semantics for LLVM:

The pragma will hint to the vectorizer/optimizer to ignore memory dependencies that are not proven but are assumed to exist. Proven dependencies are not ignored.

b. Similar Clang pragma:

Clang has *clang loop vectorize(assume_safety)*
Specifying #pragma clang loop vectorize(assume_safety) on a loop adds the mem.parallel_loop_access metadata to each load/store operation in the loop. This metadata tells loop access analysis (LAA) to skip memory dependency checking.
Reference:http://lists.llvm.org/pipermail/cfe-commits/Week-of-Mon-20150608/130619.html

c. Intel C++ Compiler definition:

#pragma ivdep
The *ivdep* pragma instructs the compiler to ignore assumed vector dependencies. To ensure correct code, the compiler treats an assumed dependence as a proven dependence, which prevents vectorization. This pragma overrides that decision.

The proven dependencies that prevent vectorization are not ignored, only the assumed dependencies are ignored.

d. Intel Fortran Compiler definition:

!DIR$ IVDEP [: option]
where the *option* is LOOP or BACK. This argument is only available on processors using IA-32 architecture.
The IVDEP directive is an assertion to the compiler's optimizer about the order of memory references inside a DO loop.
IVDEP: LOOP implies no loop-carried dependencies. **IVDEP: BACK** implies no backward dependencies.
When no option is specified, the compiler begins dependence analysis by assuming all dependencies occur in the same forward direction as their appearance in the normal scalar execution order. This contrasts with normal compiler behavior, which is for the dependence analysis to make no initial assumptions about the direction of a dependence.
The IVDEP directive is applied to a DO loop in which the user knows that dependencies are in lexical order. For example, if two memory references in the loop touch the same memory location and one of them modifies the memory location, then the first reference to touch the location has to be the one that appears earlier lexically in the program source code. This assumes that the right-hand side of an assignment statement is "earlier" than the left-hand side.
The IVDEP directive informs the compiler that the program would behave correctly if the statements were executed in certain orders other than the sequential execution order, such as executing the first statement or block to completion for all iterations, then the next statement or block for all iterations, and so forth. The optimizer can use this information, along with whatever else it can prove about the dependencies, to choose other execution orders.

- nontemporal

a. Planned Pragma Semantics for LLVM:

This pragma will instruct the compiler to use non-temporal data stores on the system,i.e, indicating that the data will not be used in the near future, does not need to be cached and can be written to memory directly. Optionally it will take variables as arguments for which non-temporal (streaming stores) will be generated. Along with this pragma, store

fences (which impose certain store instruction ordering) must be used by the programmer when required in multi-threaded programs.

b. Intel C++ Compiler definition:

#pragma vector nontemporal[(var1[, var2, ...])]
Instructs the compiler to use non-temporal (that is, streaming) stores on systems based on all supported architectures, unless otherwise specified; optionally takes a comma-separated list of variables.
When this pragma is specified, it is the programmer's responsibility to also insert any store fences as required to ensure correct memory ordering within a thread or across threads.

- vecremainder/novecremainder

a. Planned Pragma Semantics for LLVM:

The pragma will hint to the compiler to attempt to vectorize/not vectorize the remainder loop after the main loop has been vectorized.

b. Intel C++ Compiler definition:

#pragma vector always [no]vecremainder

1. *vecremainder* in presence of *vector always* pragma will vectorize both the main and remainder loops.
2. *novecremainder* in presence of *vector always* pragma will vectorize the main loop, but it does not vectorize the remainder loop.

- mask_readwrite/nomask_readwrite

a. Planned Pragma Semantics for LLVM:

The pragma indicates to the compiler to use masked/unmasked loads and stores for the loop. It will mimic/use the mask_intrinsics available on targets with Intel AVX.
"CodeGenPrepare Pass scalarizes the masked intrinsic if the target does not support it."
Reference:https://llvm.org/devmtg/2015-04/slides/MaskedIntrinsics.pdf

b. Intel C++ Compiler definition:

#pragma vector [no]mask_readwrite
When:

1. *mask_readwrite* is specified as an argument to *vector*, the compiler generates masked loads and stores within all conditions in the loop.

2. *nomask_readwrite* is specified as an argument to *vector*, the compiler generates unmasked loads and stores for increased performance.

- aligned/unaligned

a. Planned pragma semantics for LLVM:

aligned: The pragma indicates to the compiler to align the array references in memory. This helps in increasing fetch rates.
Unaligned: The pragma indicates to the compiler that the array references should not be aligned.

b. Similar OMP pragma:

Clang fully supports OpenMP 4.5.*omp simd aligned* is available.
The aligned clause declares that the object to which each list item points is aligned to the number of bytes expressed in the optional parameter of the aligned clause.
OpenMP 4.5 specification,Page 72:
https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf

c. Intel C++ Compiler definition:

#pragma vector [un]aligned
Instructs the compiler to use [un]aligned data movement instructions for all array references when vectorizing

Questions

- Ivdep: Is *clang loop vectorize(assume_safety)* equivalent to *ivdep*? To what extent do the semantics of ivdep need to be modified for Clang to create an equally "*useful pragma*"? To what extent would it be helpful to have this pragma in Clang?
- Nontemporal:What kind of analysis can we do in LLVM to find where to use nontemporal accesses? Any help would be greatly appreciated.
- vecremainder/novecremainder: Should the pragma simply call the vectorizer to attempt to vectorize the remainder loop, or should the vectorizer use a different method?
- mask_readwrite/nomask_readwrite: Is it a good idea to implement a pragma that will generate mask intrinsics in the IR? What other architectures (except x86) has support for masked read/writes?

  Reference:https://llvm.org/devmtg/2015-04/slides/MaskedIntrinsics.pdf
  LLVM has mask intrinsics for targets with AVX, AVX2, AVX-512.

From Slides: ”Most of the targets do not support masked instructions, optimization of instructions with masks is problematic, avoid introducing new masked instructions into LLVM IR”

- aligned/unaligned: Is it worthwhile to have LLVM specific pragma rather depending on OpenMP?