[llvm-dev] RFC: Supporting the RISC-V vector extension in LLVM

Eric Christopher via llvm-dev llvm-dev at lists.llvm.org
Thu Apr 12 14:27:47 PDT 2018

- Previous message: [llvm-dev] RFC: Supporting the RISC-V vector extension in LLVM
- Next message: [llvm-dev] RFC: Supporting the RISC-V vector extension in LLVM
- **Messages sorted by:** [date] [thread] [subject] [author]

I'm just going to add Kristof here since ARM is looking to add SVE here and this overlaps quite a bit with their goals.

-eric

On Wed, Apr 11, 2018 at 2:45 AM Robin Kruppe via llvm-dev <llvm-dev at lists.llvm.org> wrote:

> *RISC-V is an open and free instruction set architecture (ISA) used in*
> *numerous domains in industry and research. The vector extension (short:*
> *'V') supplements the basic ISA with support for data parallel computations.*
> *This RFC sketches a strategy for targeting this instruction set extension*
> *in LLVM.*
>
> *Some but not all of what is proposed here has already been implemented out*
> *of tree. It is explicitly not proposed to upstream any of this yet: the*
> *vector extension is still evolving (though the core concepts are reasonably*
> *stable), and the implementation is currently very much prototype quality.*
> *Nevertheless, I want to kick off a discussion about this with the LLVM*
> *community now to make sure I'm on the right track and to make the eventual*
> *upstreaming go more smoothly. In particular, a large and potentially*
> *controversial part of the strategy is a proposal for extending LLVM IR with*
> *a new vector type.*
>
> *There is also much to be said about how to structure the code generation*
> *for this ISA. However, since that aspect somewhat simpler, largely*
> *orthogonal and affects a smaller subset of the community, the details will*
> *be left to a future RFC.*
>
> *This RFC is intended to be self-contained, but interested readers can*
> *learn more about the vector extension from Roger Espasa's talk at the 7th*
> *RISC-V workshop (slides [1], recording [2]). The draft specification is*
> *also available as part of the RISC-V Instruction Set Manual [3], but right*
> *now it is unfortunately incomplete and in the process of being updated.*
>
> *I will also be at EuroLLVM with a lightning talk and poster on this*
> *subject, so if you're there as well, we can discuss in person.*
>
> *[1]*
> *https://content.riscv.org/wp-content/uploads/2017/12/Wed-1330-RISCVRogerEspasaVEXT-v4.pdf*
> *[2] https://www.youtube.com/watch?v=GzZ-8bHsD5s*
> *[3] https://github.com/riscv/riscv-isa-manual/*
>
>
> *# Summary*
>
> *First-class support for the RISC-V vector ISA requires representing a*
> *hardware vector length that is not just unknown at compile time, but also*
> *changes during execution. This in turn places some restrictions on code*
> *motion: the vector length must not change while any vector values are live.*
> *This RFC proposes to add a new vector type to LLVM IR for this purpose.*
> *Simply put, its length is tied to the surrounding function and the existing*

> `token` type is leveraged to tell optimization passes that certain vector
> operations must remain together (i.e., in the same function).
>
>
> # Background
>
> The RISC-V vector extension has many interesting properties. This RFC is
> not the right place to talk about it in detail, but this section will
> briefly introduce the aspect that is most difficult to support in LLVM IR,
> and which is consequently the focus of this RFC: the runtime-variable
> vector length.
>
> The number of elements in a vector register is determined by the
> microarchitecture. Software uses strip-mined loops to transparently process
> as many elements per iteration as the hardware can support. But even beyond
> that, the vector length can vary during the execution of a program:
> different kernels may *configure* the vector unit differently depending on
> their needs, leading to different parts of the program having
> differently-sized vector registers.
>
> The vector length being determined by the microarchitecture is similar to
> Arm's Scalable Vector Extension (SVE), for which support is being
> upstreamed at the moment. However, in SVE the vector length is fixed once a
> program starts running, while full use of the RISC-V vector extension will
> lead to the vector length changing regularly during execution. It's
> possible to maintain the same configuration -- and therefore the same
> vector length -- throughout the entire program, but this will often perform
> worse than a tailored configuration.
>
> ## Maximum vs active vs application vector length
>
> The V ISA has two notions of vector length: the *maximum* vector length
> (called MVL), which describes the number of elements in each vector
> register, and the *active* vector length (called vl), which limits how many
> of those elements are actually processed by each vector instruction.
>
> The latter is used to express loops of any application-specified length
> with a single copy of the loop body. Instead of handling the tail
> iterations not divisible by MVL separately with scalar code, the active
> length length is set up so that the last few iterations process as many
> elements as are left to process.
>
> The effect of the active vector length is similar to a mask of the form
> `<true, ..., true, false, ..., false>`, aside from the scalar control logic
> that sets and maintains the active vector length. Thus it can be modeled in
> IR with judicious use of intrinsics and masking. This still allows also
> having a single loop body in IR, without introducing new IR concepts in
> addition to those already needed for the variable MVL.
>
> Thus, the rest of this RFC focuses on handling the MVL: all references to
> "vector length" from this point on should be taken to refer to the MVL, not
> the active vector length.
>
>
> # Scope of the support
>
> To preempt misunderstandings, this section outlines what is meant and not
> meant by "support for the vector extension".
>
> ## Variable vector length
>
> There *is* an option to entirely avoid the concept of the vector length
> changing during execution. Keeping the same vector unit configuration
> throughout the entire program execution also leads to the vector length
> being fixed once the program starts executing. In this case, compiler
> support works out rather rather similar to support for Arm SVE, with the
> biggest difference being that vectors lengths are not multiples of 128 bit
> (which legalization can paper over). Indeed, no IR changes beyond those
> proposed for SVE support appear to be necessary to implement this approach

> to RISC-V vector extension support.
>
> However, this approach is wasteful, as a tailored configuration can
> improve performance and energy efficiency significantly. As one data point,
> the Hwacha project reported [4] up to 9.5% fewer cycles taken and up to 11%
> less energy consumed on a microarchitecture built to exploit narrower bit
> widths of vector elements (comparing "mvp, packed: yes" to "mvp, packed:
> no"). Besides such microarchitectural optimizations, enabling fewer
> registers can improve context switch times because fewer registers need to
> be saved, and being more flexible in how registers can be used (in
> particular, how many are reserved for scalar values) aids register
> allocation.
>
> Thus, restricting programs to a single configuration may be a good first
> step to get things up and running, but ultimately support for
> runtime-varying vector lengths is desired to make the most of hardware
> capabilities.
>
> [4] "A Case for MVPs: Mixed-Precision Vector Processors", Albert Ou, Quan
> Nguyen, Yunsup Lee, Krste Asanović,
> http://hwacha.org/papers/hwacha-mvp-prism2014.pdf
>
>
> ## Producing vector code
>
> It is intended that vector code is primarily produced via loop
> vectorization and other IR-level auto-vectorizers (e.g., the region
> vectorizer), not written by hand. Supporting loop vectorization is of
> highest priority. The groundwork for loop vectorization should be useful
> for other kinds of automatic vectorization as well, but loop vectorization
> will be implemented first.
>
> It's not required or expected that the stock loop vectorizer can generate
> RISC-V vector code from the start. Considering the many significant
> differences to the packed-SIMD architectures the loop vectorizer is
> tailored to, it's quite likely that some experimentation in this space is
> required (e.g., building on VPlan and writing custom recipes). Of course,
> in the long term there should be as much code sharing as possible.
>
> Support for hand-written vector code va source-language-level intrinsics
> (as opposed to inline assembly) would be nice to have and probably falls
> out for free, but is rather low priority.
>
>
> ## No vector unit configuration in IR
>
> While configuring the vector unit is an essential part of compiling for
> the V ISA, it has no place in LLVM IR. Vectors should be regular SSA values
> that don't reference any extra state other than (by necessity) the vector
> length. Deciding how to configure the vector unit for a given piece of code
> is target-dependent and intertwined with register allocation, and will
> therefore be left to the backend.
>
> # Challenge: Code motion around vector length changes
>
> When the vector length can change during execution, there are implicit
> dependencies between vector operations and points in the program where the
> vector length may change. These dependencies must be taken into account
> somehow, or else code motion passes could move vector operations across
> vector length changes, effectively changing program semantics. For example,
> it's nonsensical to compute a vector value `%v1` with one vector length,
> change the vector length, and then compute another vector `%v2` with the
> new vector length and add it to `%v1`. This makes no more sense than adding
> `<4 x i32>` to `<8 x i32>`, yet it could happen if an input program has a
> vector length change *after* these vector calculations and optimization
> passes are not aware of the impact of the vector length change on those
> calculations.
>
> Crucially, the vector length changes when calling and returning from

> functions in most calling conventions. Functions that don't specifically
> use a vectorcall ABI configure the vector unit for their own use when
> called, rather than using configuration set up by the caller. Therefore,
> caller and callee will generally have different vector lengths, and moving
> vector operations from the caller into the callee or vice versa tends to
> break programs.
>
> However, note that the precise value of the vector length doesn't really
> matter -- software is supposed to be *vector length agnostic*. Completely
> inlining a function is perfectly fine, for example. What matters is that
> the vector length doesn't change *during* vector computations, i.e., while
> any vector values are live (either as SSA values, or in memory!). Thus,
> there is no need to support and track vector length changes at instruction
> granularity. It's enough to coarsely enforce that the vector length remain
> constant throughout a larger code region (say, a loop nest, or a function).
>
>
> # Runtime-varying vector length in the IR
>
> This is achieved by simply declaring "by fiat" that the vector length is
> determined on function entry and remain constant for the rest of the
> function execution. Other functions and other calls to the same function
> may observe a different vector length, but within one call to a given
> function, the vector length is fixed. That is not precisely how the
> hardware works, but it is a contract the backend can uphold easily (more on
> this later) and it allows piggy-backing on existing IR constructs (the
> `token` type) to communicate restrictions to optimization passes.
> Nevertheless, some additions to IR are required: a new first class type, a
> new instruction, and a new operand for some existing instructions.
>
> ## IR semantics
>
> Every time a function is called, a positive integer called the *dynamic
> vector length* is determined in an unspecified way. The dynamic vector
> length can differ not only between different functions, but also between
> different calls to the same function. The exception is that functions with
> the `inherits_vlen` attribute get the same dynamic vector length as their
> caller (Note: this attribute is a straw man, target-specific calling
> conventions may work better for this purpose).
>
> A new instruction `vlentoken` is added, which has no operands and is of
> type `token`. This token represents the dynamic vector length of the
> function execution it is in. There can only be one such instruction per
> function (this is inconsequential to the operational semantics, but it
> simplifies IR passes).
>
> A new kind of type is added, the *dynamic-length vector*, written `< vlen
> x <element type> >`. It represents a vector with a number of elements equal
> to the dynamic vector length. Like fixed-length and scalable vectors, these
> vectors can only contain integer, float and pointer elements.
>
> A use of a `vlentoken` (representing a dynamic vector length) is attached
> to all operations that care for the dynamic vector length. That is, *every*
> instruction that handles dynamic-length vectors or is impacted by their
> length receives the respective function's `vlentoken` as extra operand, and
> operates on a number of elements equal to the corresponding dynamic vector
> length.
>
> `< vlen x <element type> >` is a first-class type and supports most common
> instructions (details below), but cannot be used as function argument or
> function return type unless the `inherits_vlen` attribute is applied to the
> callee.
>
> ## Rationale / "why this works"
>
> Although the vector length is a property of vector *values*, tracking the
> dynamic vector length at the type level would require a "separate type" for
> each call to any function. It's much more feasible to attach the vector
> length to the *operations* instead. This works out because SSA values are

> function-local (so all operation on them agree on the vector length by
> definition) with the exception of function arguments and return values.
> Consequently, dynamic-length vectors in function signatures are disallowed
> unless the `inherits_vlen` attribute ensured caller and callee have the
> same dynamic vector length.
>
> The `vlentoken` token ensures that all operations that start out in the
> same function must remain in the same function while the code is
> transformed (recall that tokens cannot be passed to or returned from
> non-intrinsic functions). That's why it is important that `vlentoken` is a
> token, not simply an integer as one might expect. In other words,
> `vlentoken` does not give the program access to the dynamic vector length,
> it communicates a restriction to the optimizer.
>
> ## More details
>
> The `< vlen x <element type> >` type is separate from the existing vector
> types. Instructions for fixed-length vectors (elementwise arithmetic,
> `insertelement`, `select` with a vector of `i1`s, etc.) are not extended to
> this new type, at least not in this RFC. It's a possible future extension,
> but for now, target-specific intrinsics work fine for those operations.
>
> The following operations on dynamic-length vectors *are* supported:
>
> - `phi`
> - `load` and `store`
> - `alloca` (at least of a single vector; the `alloca <ty>, <ty>
> <NumElements>` form ties into the open question about aggregates and GEPs
> below)
> - `select` (with `i1` condition)
> - Argument passing and return values (`call`, `invoke`, `ret`) for
> functions with `inherits_vlen`
>
> All of these instructions (including phi) have an additional operand of
> type `token` if and only if they operate on a vector of dynamic length. In
> textual IR, one appends `, vlen <the token>` to the instruction, for
> example:
>
>   %0 = vlentoken
>   %ptr = alloca <vlen x i32>, vlen %0
>   %v = call <vlen x i32> @foo(), vlen %0
>   store <vlen x i32> %v, <vlen x i32>* %ptr, vlen %0
>
> Open question: should GEPs and aggregates involving dynamic-length vectors
> work? This RFC errs on the side of simplicity and excludes them (they're
> non-trivial to implement and not needed for strip-mined loops) but if
> desired, they could be supported.
>
> There are no constants of dynamic-length-vector type except
> `zeroinitializer` and `undef` (resp. `poison` once that is adopted). In
> particular, there is no equivalent to fixed-length vector constants (`<ty
> elem1, ty elem2, ...>`). Dynamic-length vectors also cannot be stored in
> globals.
>
> ## Impact on optimizations
>
> The semantics imply several restrictions on optimizations, but these are
> mostly encoded with existing IR constructs -- chiefly, the `token` type
> that ties all vector operations to a `vlentoken`. For example, because
> token values cannot be passed to (non-intrinsic) functions or returned from
> them, no special pleading is needed to keep an outliner or partial inliner
> from spreading vector operations across multiple functions -- correct
> passes already don't do that when tokens are involved. Passes do, however,
> need to be updated in two respects.
>
> First, the new token operand needs to be respected when comparing two
> instructions, creating new instructions, etc. -- this is an inherent
> downside of adding new operands, but also rather mechanical. The rule that
> there is only one `vlentoken` per function makes this even more mechanical

> than usual, because all instruction within one function have the same
> vector length token. This means that one does not need to consider them
> when comparing instructions from the same function, and it's always clear
> which token should be used when creating a new instruction.
>
> Second, the very same rule of only one `vlentoken` per function must be
> respected during interprocedural code motion. For example, inlining can't
> just copy the `vlentoken` from the callee into the caller.
>
> However, note that it's valid to *merge* the caller's and callee's
> `vlentoken` instructions. Because the semantics state that each call to a
> function can get a different dynamic vector length, merging `vlentoken`s
> *refines* the program's behavior by picking the possible execution where
> the callee "happens to" get the same vector length as the caller in the
> inlined calls. So inlining can simply replace all `vlentoken` tokens in the
> inlined code with the `vlentoken` token of the callee. Other passes are
> likely similarly easy to update (and in the worst case, they can just bail
> out when seeing dynamic-length vectors).
>
> ## Impact on backends
>
> Unsurprisingly, the IR types `<vlen x <element type> >` come with
> associated MVTs. There's also a new SelectionDAG node `VLENTOKEN` to mirror
> the `vlentoken` IR instruction (and presumably `G_VLENTOKEN` in GMIR for
> GlobalISel).
>
> Backends other than RISC-V can legalize these MVTs and the `VLENTOKEN`
> node very easily, even if in practice there currently aren't many useful
> operations on these vectors without target-specific intrinsics.
> Specifically, `< vlen x <element type> >` can be legalized as `< n x
> <element type> >` or even `< scalable n x <element type> >` (the vector
> type for Arm SVE) for any fixed `n`. All the complications stemming from
> the runtime-varying vector length go away, and the `vlentoken` node can
> simply be dropped on the floor.
>
> That leaves targets with an actual runtime-varying vector length in
> hardware, i.e., RISCV with the V feature enabled. As stated in the
> introduction, this RFC does not cover the backend changes in detail, but to
> give you a rough idea, here's a sketch. Keep in mind (especially if you're
> familiar with V) that this is glossing over everything not directly related
> to the proposed IR type (particularly the "polymorphic instruction set"
> aspect of the register configuration).
>
> As described earlier, the vector length in RISC-V is completely determined
> by the vector unit configuration. Therefore, vector operations in Machine
> IR have an implicit use of the configuration registers. This is the moral
> equivalent of the `vlentoken` token operand, but more precise (and MIR
> doesn't have an equivalent of the token type anyway). To complete the
> picture, all operations that change the configuration are made explicit.
>
> Because only virtual registers can be live across basic block boundaries
> before register allocation, this may require a dummy register class with
> only a single physical register, or something similarly inelegant.
>
> With a way to precisely represent vector length changes in hand, the
> backend just needs to ensure it implements the semantics of `< vlen x
> <element type> >` described earlier. This is achieved by configuring the
> vector unit "in the prologue", and then not doing anything that might
> change the vector length inside the function. This setup is effectively in
> the prologue (i.e., before any user code) but not actually inserted during
> the "prologue/epilogue insertion" pass, which runs far too late for this
> purpose.
>
> For scenarios like two entirely separate vectorized loops within one
> function, it might be useful to drastically change the vector unit
> configuration in the middle of a function. This could be implemented as an
> optimization (e.g., a pre-RA machine function), but it's all hypothetical
> so far.
>

> _____
> *LLVM Developers mailing list*
> *llvm-dev at lists.llvm.org*
> *http://lists.llvm.org/cgi-bin/mailman/listinfo/llvm-dev*
>

-------------- next part --------------
An HTML attachment was scrubbed...
URL: <http://lists.llvm.org/pipermail/llvm-dev/attachments/20180412/de147bde/attachment.html>

- Previous message: [llvm-dev] RFC: Supporting the RISC-V vector extension in LLVM
- Next message: [llvm-dev] RFC: Supporting the RISC-V vector extension in LLVM
- **Messages sorted by:** [date] [thread] [subject] [author]

More information about the llvm-dev mailing list