Moving work into the middle end

Jon Chesterfield

AMD
together we advance_

Context

- Occasionally backend compiler dev
- Looking to spark discussion on staying in IR until the end
- Couple of case studies, interruptions welcome
- Keen to discuss here / email / discourse / whatever

LDS allocation

- Smallish block of memory, amount allocated by GPU kernel launch
- Need to put variables somewhere in it
  - Iff reachable from the corresponding kernel
  - Want to be fast to locate them (ideally immediates in instructions)
  - Handle alignment constraints
  - Ideally reuse blocks where lifetimes are independent
- That's a degenerate regalloc
- Originally done in the backend, append in order of SDag traversal, doesn't work for non-kernel functions

LDS allocation thinking

- Really didn't want to write this for SDag and for GlobalISel
- Going to have to have some fast path optimisations
- Was going to be subtle and I don't like testing ISel via MIR
- Reachable analysis seems easier in IR
- Going to need some lookup tables, can build those in IR

LDS allocation implementation

- Groups variables together in structs, RAUW them
  - Tries to put popular variables in one special struct
  - Put that struct at address zero
- Builds lookup tables in IR, gives kernels an integer id in IR
- Writes addresses into tables and as absolute_symbol metadata
- Backend resolves accesses to said address from the metadata
- Almost all the testing is in IR, SDag/GlobalISel resolution is trivial

Variadic functions

- These are mostly a legacy nuisance from early C
- Seem to invite complicated schemes in the established architectures
- Embedded is prone to special casing printf instead of implementing
- When implementing, tempting to push everything on the stack pointer
- People wanted these on the GPU

Variadic functions thinking

- Seems sad that they block things like inlining
- Look a lot like syntax sugar over pushing a count of things on the stack
- Really didn't want to write this for SDag and for GlobalISe
- Was going to be subtle and I don't like testing call lowering via MIR
- Anticipated a really bad time debugging this on a GPU

Variadic functions implementation

- The va_list iterator abstracts exactly over architecture differences
- Turn ... into a va_list and va_start into a va_copy
- Build an alloca at the call site to initialise a va_list with
- Put the variables at the right place in that alloca
- If ABI preserving, emit a ... thunk that calls the modified function
- Inlining etc now work fine, it's not a variadic call anymore

Lessons learned

- Testing in IR works great
- Debugging IR passes is easy
- No real surprises in either

What’s this backend layer for, really?

- Machine instructions aren't SSA
- Could have an intrinsic per instruction that looks like SSA
- Regalloc means no longer in SSA
  - Must it? %v:rax looks alright here
  - Cranelift seems to stay in SSA, as does libfirm
- Scheduling doesn't care much, could put intrinsics in bundles
- DAGCombines look a lot like instcombine
- Type legalisation could definitely run on IR
- MIR rewrite passes look quite a lot like IR rewrite passes

What if we left it in IR until emit?

- Need an IR intrinsic per instruction
- Need to annotate SSA registers with the machine one
- Better support for marking functions/blocks with calling conventions
  - Blocks are functions anyway
  - Block parameters vs phi nodes, do we have the wrong one?
- Instruction encoding driven by the intrinsic
- Instcombine works harder
- Replaces/ports a whole lot of existing backend code

Conclusions

- LDS lowering in IR works fine
- Variadics lowering in IR works fine
- Other things can probably also move out of the backend

- I suspect we could use LLVM IR instead of MIR etc. Am I wrong?

Disclaimer

The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions, and typographical errors. The information contained herein is subject to change and may be rendered inaccurate for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product releases, product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. Any computer system has risks of security vulnerabilities that cannot be completely prevented or mitigated. AMD assumes no obligation to update or otherwise correct or revise this information. However, AMD reserves the right to revise this information and to make changes from time to time to the content hereof without obligation of AMD to notify any person of such revisions or changes.

THIS INFORMATION IS PROVIDED ‘AS IS.” AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS, OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION. AMD SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY RELIANCE, DIRECT, INDIRECT, SPECIAL, OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF AMD IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.


AMD