A Journey of OpenCL 2.0 Development in Clang

Anastasia Stulova
anastasia.stulova@arm.com
Media Processing Group
ARM

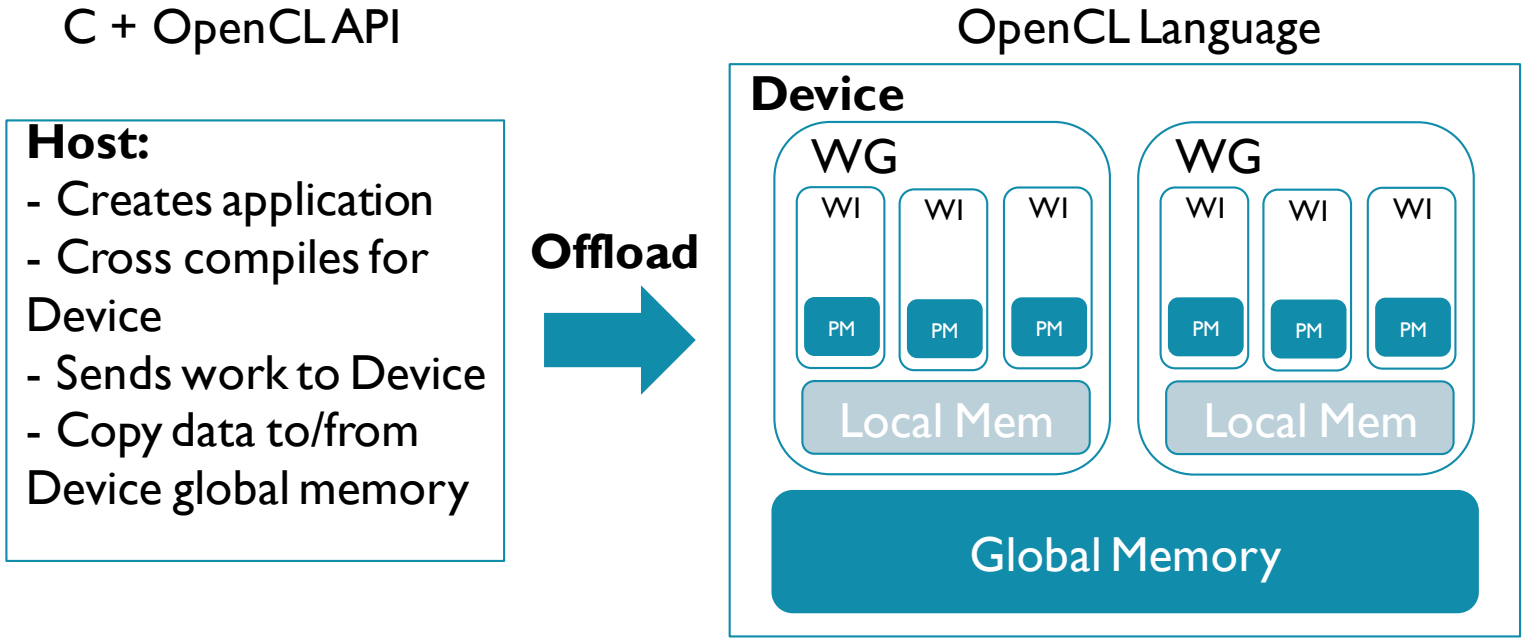
The Architecture for the Digital World®
ARM

Agenda

- OpenCL intro
- OpenCL in Clang
- Overview of OpenCL 2.0
- OpenCL 2.0 implementation
- Summary and discussions

ARM

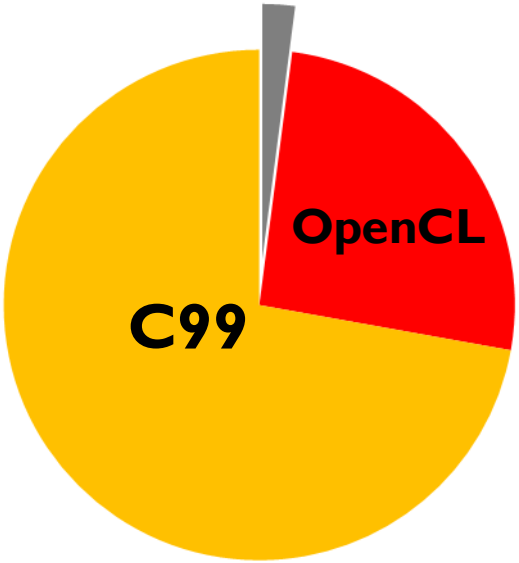
OpenCL programming model and terminology



- WI – work-item is a single sequential unit of work with its private memory (PM)
- WG – work-group is a collection of WIs that can run in parallel and access local memory shared among all WIs in the same WG

ARM

OpenCL language intro



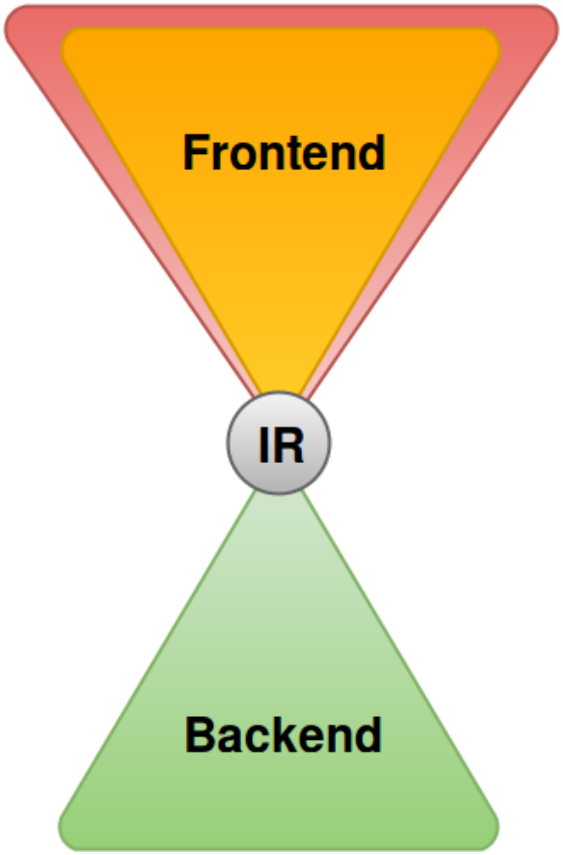
- C99 based
- Parallel units of work – kernels
- Explicitly assign object to memory using address space qualifier with each type
- Special types: images, events, pipes, …
- Access qualifiers - read/write only applies to some types
- No standard C includes or libs, but defines its own libs

ARM

OpenCL for compiler writer

Ideal

How we imagined it to be



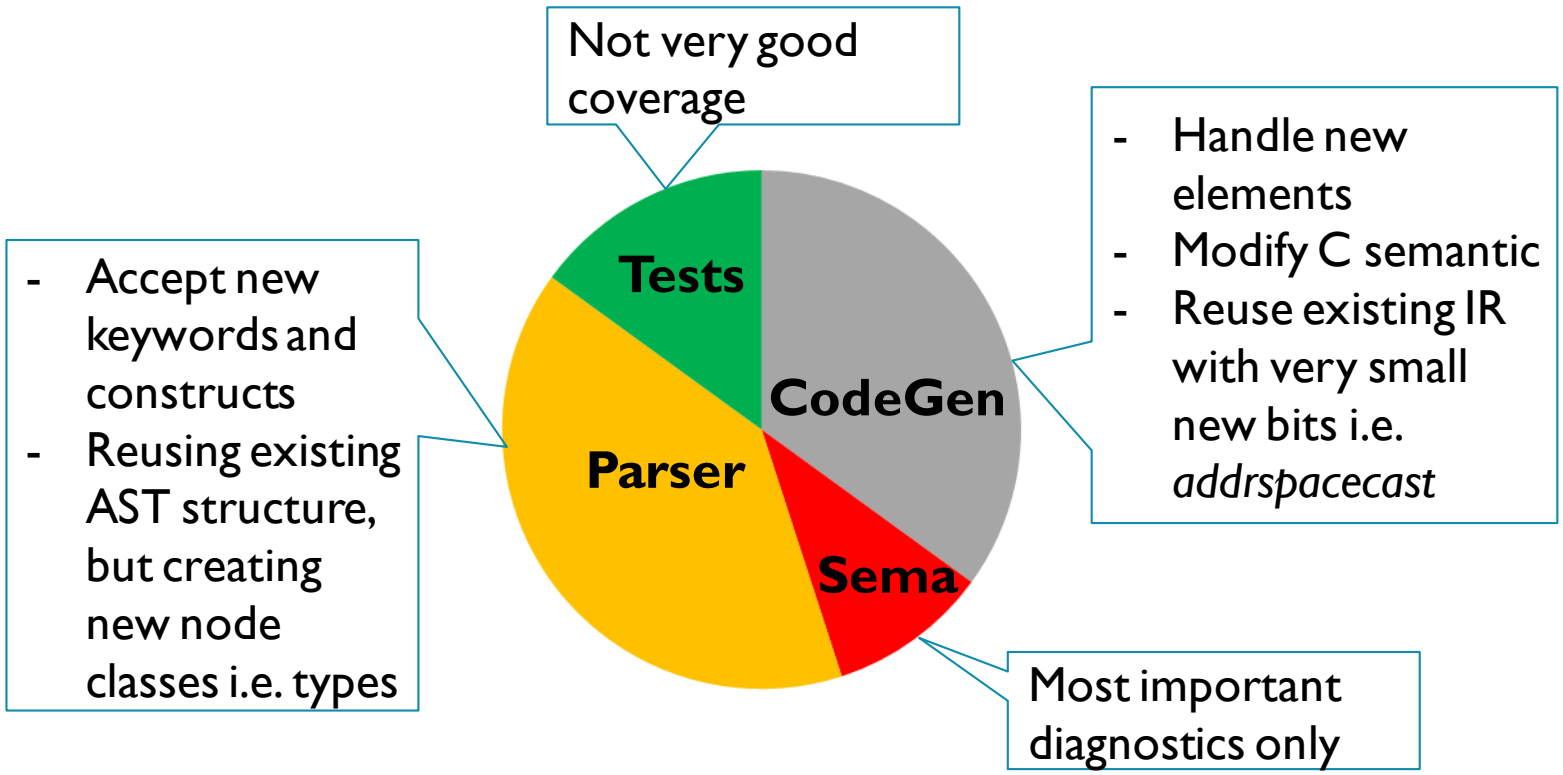
Reality

What we missed

- How to handle invalid targets?
- Conflicts between C and OpenCL unforeseen by Spec (especially in undefined behaviour)!
- How to generate IR generically with absence of enough info on various backends?
- Missing explicit IR constructs are substituted with metadata and intrinsics!

ARM

First implementation in Clang (OpenCL 1.1/1.2)



ARM

OpenCL 2.0 feature overview

- Hierarchical/Dynamic parallelism - device side enqueue (work creation bypassing host) using ObjC blocks
- Reduce difficulty of writing code with address spaces (abstract away from memory model as much as possible, late binding)
- Simplify communications among kernels (avoid going outside of device via host)
  - Program scope variables persist across kernel invocations
  - Pipe communication using streaming pattern
- C11 atomics with memory visibility scope
- New image types and access qualifier

ARM

Generic address space

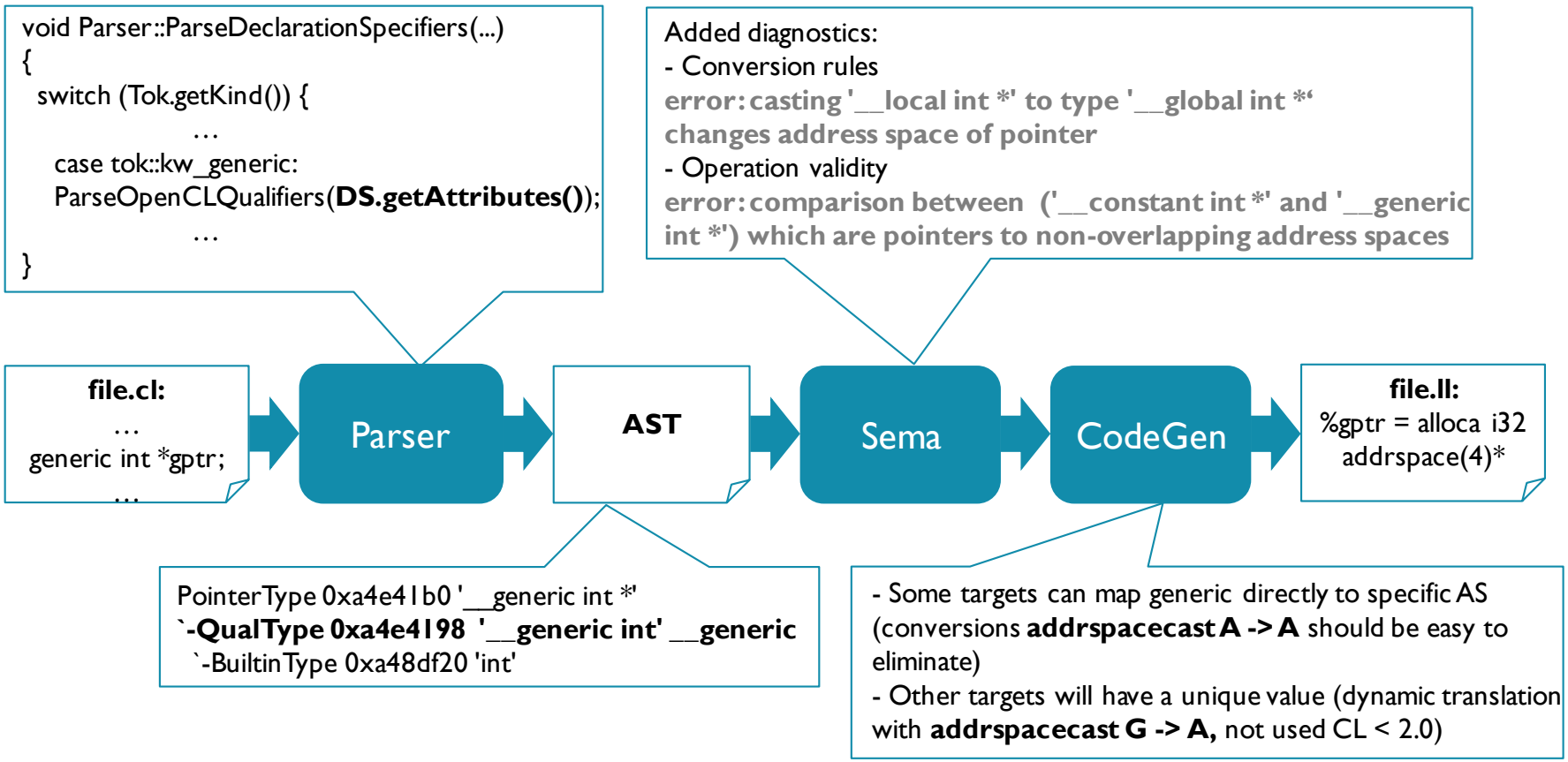
```
void foo(local int *lptr) {...}
void foo(global int *gptr) {...}

kernel void bar(local int *lptr, global int *gptr){
  foo(lptr);
  foo(gptr);
}
```


```
void foo(int *gen) {...} // only one foo is needed, use late binding

kernel void bar(local int *lptr, global int *gptr){
  foo(lptr);  // local to generic AS conversion
  foo(gptr); // global to generic AS conversion
}
```

- Address Space (AS) in OpenCL is almost a part of a type
- Nothing is allowed with objects of distinct ASes including casting, operations etc.
- One of the largest changes affected Parser, Sema and CodeGen of many C paths
- Generic helps writing code more conveniently
- Easy to support in Clang reusing existing AS functionality

ARM

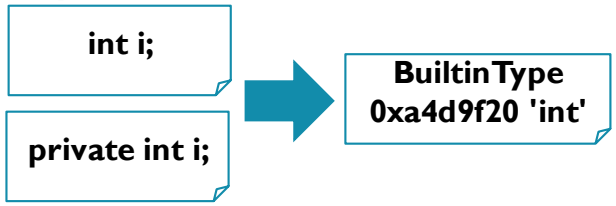
Generic address space in Clang



ARM

Default address space

OpenCL <2.0:



OpenCL >=2.0:

Scope Type	global	local
pointer	*LangAS::generic*	*LangAS::generic*
scalar	*LangAS::global*	*LangAS::private*

- Workable solution in order not to modify previous scheme:
  - AS is handled as a type attribute while parsing a type
  - If absent look at scope and type being parsed
  - But too early to be able to consider object kind: ***NULL*** **- (void*)0** no AS
- We could introduce private AS explicitly as unique qualifier
  - Affects how AS is represented by previous standards
- Type printing issue (difference with the original type)

int x = &f; // warning: incompatible pointer to integer conversion initializing '__global int' with an expression of type …

ARM

Atomic types

- Map CL to C11 atomic types in Clang:

  Sema.cpp - Sema::Initialize():

  ```
  // typedef _Atomic int atomic_int
  addImplicitTypedef("atomic_int", Context.getAtomicType(Context.IntTy));
  ```

- Only subset of types are allowed
- Added Sema checks to restrict operations (only allowed through builtin functions):

  ```
  atomic_int a, b;     a+b;   // disallowed in CL
  _Atomic int a, b;    a+b;   // allowed in C11
  ```

- Use C11 builtin functions in Clang to implement CL2.0 functions
  - Missing memory visibility scope as LLVM doesn't have this construct

  ```
  C atomic_exchange_explicit(volatile A *obj, C desired, memory_order order, memory_scope scope); // CL
  C atomic_exchange_explicit(volatile A *obj, C desired, memory_order order); // C11
  ```

  - Can be added as metadata or IR extension

ARM

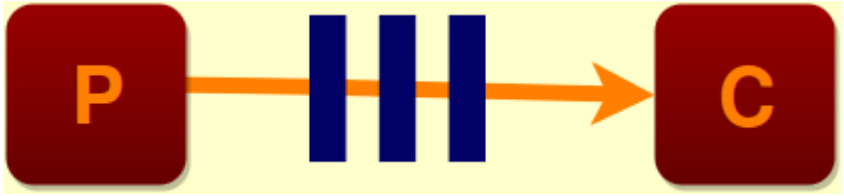
Program scope variable

- Syntax like a global variable in C, but its value persists among different kernel executions
- Disallowed in earlier standards => Sema modification to allow

- In earlier standards we added implicit *local WG storage class* for local AS variables:
  - ***local int x;*** => Clang added *local WG storage class*
  - ***static local x;*** => Results in 2 storage classes but C allows only one
  - Removed *local WG storage* as this can be checked by an AS qualifier

ARM

Pipe



Device

```
kernel void producer(write_only pipe int p) {
  int i = …;
  write_pipe(p, &i);
}

kernel void consumer(read_only pipe int p) {
  int i ;
  read_pipe(p, &i);
}
```

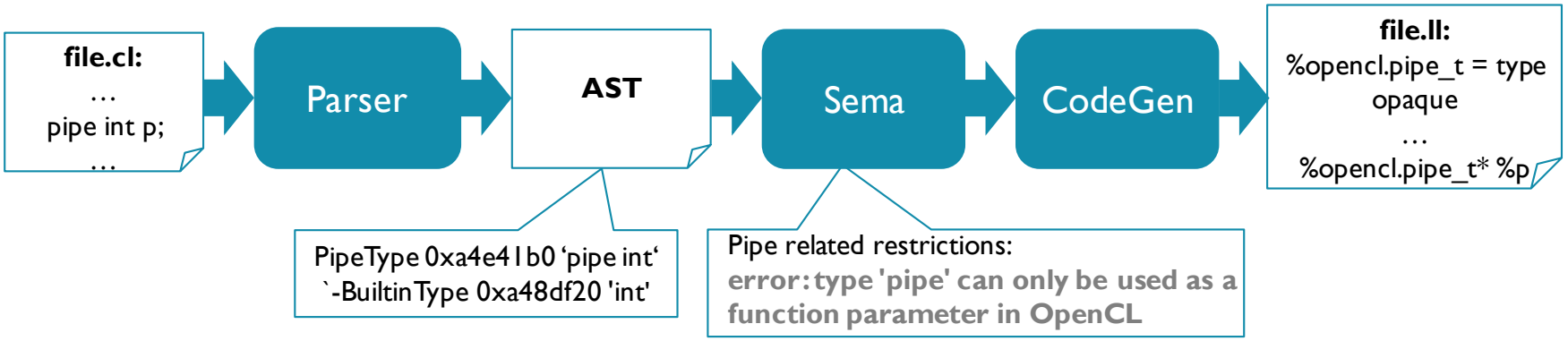
Host

```
pipe = clCreatePipe(context, 0, sizeof(int), 10 /* # packets*/…);

producer  = clCreateKernel(program, "producer", &err);
consumer = clCreateKernel(program, "consumer", &err);

err = clSetKernelArg (producer,  0, sizeof(pipe), pipe);
err = clSetKernelArg (consumer,  0, sizeof(pipe), pipe);

err = clEnqueueNDRangeKernel(queue, producer, …);
err = clEnqueueNDRangeKernel(queue, consumer, …);
```

- Classical streaming pattern
- OpenCL code specifies how elements are written/read
- Host (C/C++) code sets up pipe and connections to kernels

ARM

Pipe type



- Code repetition in Clang wrapper style types (i.e. AtomicTypes, PointerTypes, etc) and factory creation code in ASTContext
  - refactoring needed!
- Pipe builtin functions:

  CL: *int read_pipe (read_only pipe gentype p, gentype *ptr)*
- ***gentype*** is any builtin or user defined type
- Generic programming style behaviour in C99
- Implemented as Clang builtin function with custom check

  Buildins.def: *LANGBUILTIN(read_pipe, "i.", "tn", OCLC_LANG)*
- CodeGen to *call i32 @__read_pipe(%opencl.pipe_t* %p, i8* %ptr)*

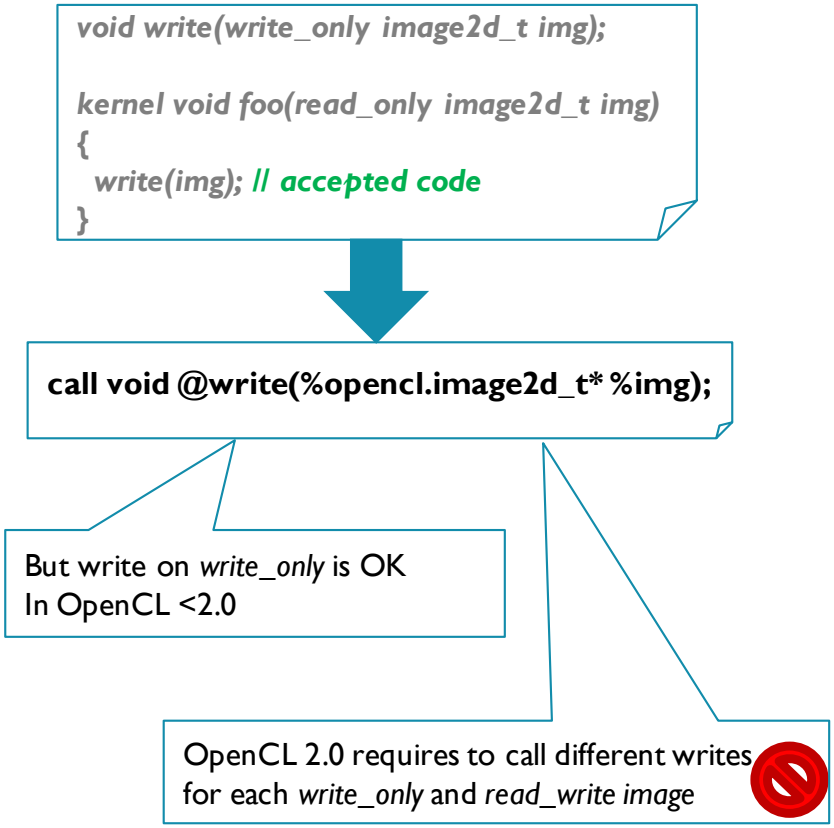
ARM

Images

- All images are special Clang builtin types
- Handled in a similar way => a lot of copy/paste code
- OpenCL <2.0: 6 different types
  - *image1d_t, image1d_array_t, image1d_buffer_t, image2d_t, image2d_array_t, image3d_t*
- OpenCL >=2.0: 6 new types:
  - *image_2d_depth_t, image_2d_array_depth_t, image_2d_msaa_t, image_2d_array_msaa_t, image_2d_msaa_depth_t, image_2d_array_msaa_depth_t*
- Access qualifier:
  - OpenCL <2.0: ***read_only/write_only***
  - OpenCL >=2.0 adds ***read_write***
- Access qualifier + image type = unique type

ARM

Image problem



- Not implemented correctly
- Access qualifiers are ignored after parsing:
  - No diagnostics wrt image access
  - No access qualifiers in IR
- Several attempts to correct
- Current review setup to correct functionality:

  http://reviews.llvm.org/D17821

ARM

Device side enqueue

- OpenCL builtin function

*enqueue_kernel(…,void (^block)(local void *, …))*

  - ***block*** has an ObjC syntax
  - ***block*** can have any number of ***local void**** arguments

- Kind of variadic prototype
  - No standard compilation approach
  - To diagnose correctly needs to be added as Clang builtin function with a custom check

ARM

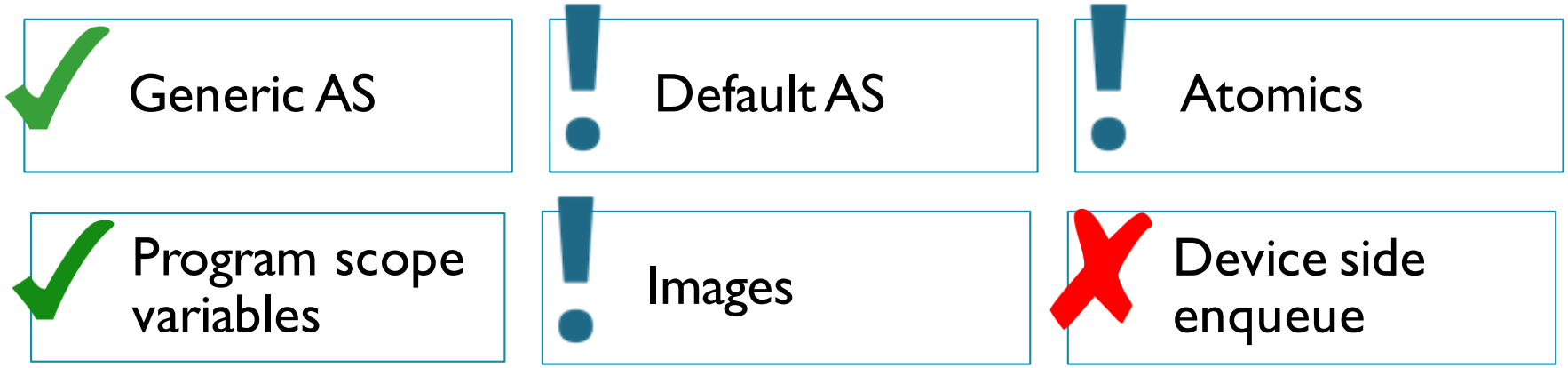
Misc features

- Loop unroll hint attribute added
  - Diagnostics and CodeGen code shared with pragma C loop hint implementation
- NOSVM attribute (but ignored)
- Still fixing AS issues in CodeGen and Sema
- Added ObjC blocks restrictions in OpenCL

```
int ^bl(int,...) = ^int(int I,...) // error: invalid block prototype, variadic arguments are not allowed in OpenCL
```


ARM

OpenCL 2.0 current state & future work



- Finalise remaining issues: default AS, atomics, images
- Add support for missing device side enqueue and other misc
- Improve tests and diagnostics for previous standards
- Refactoring of problematic parts

ARM

Summary

- Good progress on OpenCL2.0 (completion planned in rel3.9)
- Beneficial to derive from production quality C frontend
  - Some parts are difficult as there is no standard mechanism in Clang
  - Best use of existing C/OpenCL functionality but not affecting old functionality much
- Clang AST and internals are tailored quite well to OpenCL but IR is still very ad-hoc
  - Would it make sense to add more constructs to LLVM IR or improve support for alternative formats such as SPIR-V?

ARM

Contributors:

- ARM: Anastasia Stulova
- Intel: Alexey Bader, Xiuli PAN
- AMD: Liu Yaxun
- Tampere University of Technology: Pekka Jääskeläinen
- Others: Pedro Ferreira

ARM