OPENCL SUPPORT IN CLANG STATUS : OPENCL C 3.0, IMPROVEMENTS AND FUTURE DIRECTIONS

Anton Zabaznov

OPENCL C 3.0 AND OPTIONAL FUNCTIONALITY

- OpenCL is a standard for heterogeneous computing.
- OpenCL C 3.0 specification was released on September 30th 2020.
- OpenCL C 3.0 is backward compatible with 1.2 and introduces optionality of 2.0 features (such as device enqueue, program scope variables etc).
- This brings flexibility for vendors to implement necessary functionality targeting developer requests.

OPENCL C 3.0 AND OPTIONAL FUNCTIONALITY

- Support of various optional features in OpenCL C 3.0 is indicated with a feature test macros within a kernel source (full list of optional features):

Feature macro	Description
__opencl_c_3d_image_writes	Supports of 3D image type and built-in functions for writing to 3D image objects.
__opencl_c_device_enqueue	Supports of necessary types and built-in functions to enqueue additional work from the device.
__opencl_c_generic_address_space	Support of the unnamed generic address space.
__opencl_c_program_scope_global_variables	Support of program scope variables in the global address space.
…	…

OPENCL C 3.0 SUPPORT IN CLANG

- There already exists such concept in OpenCL as *extensions* (see OpenCL extension specification). Handling of OpenCL *extensions* in clang was highly reused to implement OpenCL C 3.0 support.
- Mostly all features are implemented in clang (see OpenCL C 3.0 Implementation Status).
- Existing features are set by the target or can be overridden via existing command line flag `-cl-ext:`

```
$ clang -cl-std=CL3.0 -target r600 -Xclang -cl-ext=+__opencl_c_generic_address_space ...
```

- Some work is still in progress regarding support of device enqueue and adding of built-in functions.

DEVICE ENQUEUE WITHOUT PROGRAM SCOPE VARIABLES FEATURE

- Device enqueue syntax:

```
kernel void kernel foo(uint local_mem_size) {
  void (^block_var)(local void*)                      // block variable
                 = ^void(void local *ptr) {...};      // block literal expression
  enqueue_kernel(..., block_var, ..., local_mem_size) // builtin function
}
```

- Block variable is initialized with block literal expression.
- Block literal expression is a struct, for OpenCL it contains a pointer to function for invocation and some other fields.

DEVICE ENQUEUE WITHOUT PROGRAM SCOPE VARIABLES FEATURE

- Block literal expression for block with no captures is emitted in global scope as optimization according to blocks ABI:

```
$ clang -cl-std=CL3.0 -Xclang
        -cl-ext=+__opencl_c_device_enqueue,+__opencl_c_program_scope_global_variables
```

```
void foo() {
  ...
  int (^b)(int) = ^(int n) {
    return n * n;
  };
  ...
}
```

```
__opencl_block_literal_generic = type {i32, i32, i8 addrspace(4)*}

@__block_literal_global = internal addrspace(1)
                          constant {i32, i32, i8 addrspace(4)*} ...

spir_func void @foo() {
  %1 = alloca __opencl_block_literal_generic addrspace(4)*
  store (cast @__block_literal_global
              to %__opencl_block_literal_generic addrspace(4)*),
          __opencl_block_literal_generic addrspace(4)** %1
  ...
}
```

- Now global address space is used. That’s valid if feature for program scope global variables is supported.

DEVICE ENQUEUE WITHOUT PROGRAM SCOPE VARIABLES FEATURE

- Otherwise, block literal expression is emitted in local scope (in progress, D112230).

```
$ clang -cl-std=CL3.0 -Xclang
        -cl-ext=+__opencl_c_device_enqueue,-__opencl_c_program_scope_global_variables
```

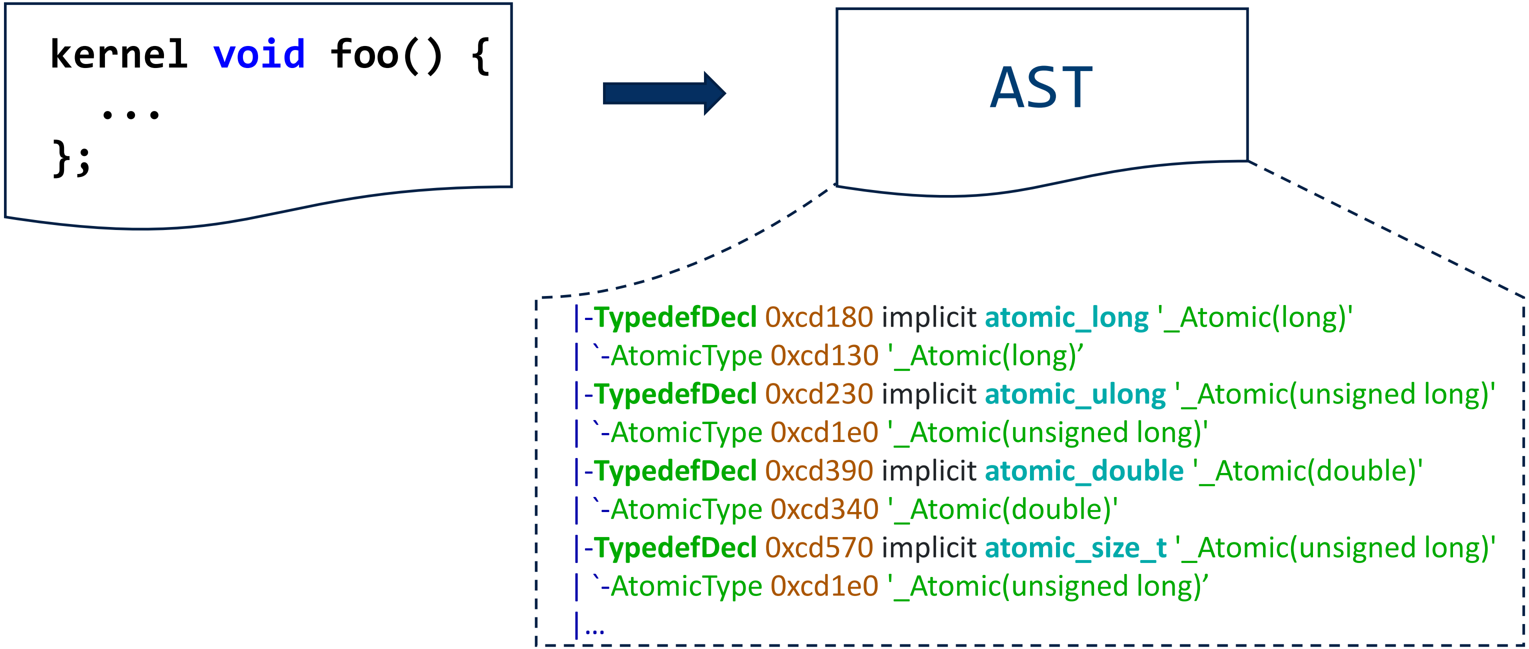
```
void foo() {
  ...
  int (^b)(int) = ^(int n) {
    return n * n;
  };
  ...
}
```

```
__opencl_block_literal_generic = type {i32, i32, i8 addrspace(4)*}


spir_func void @foo() {
  %1 = alloca __opencl_block_literal_generic addrspace(4)*
  %2 = alloca <{ i32, i32, i8 addrspace(4)* }>
  ; local block literal initialization
  %6 = cast %2 to __opencl_block_literal_generic addrspace(4)*
  store __opencl_block_literal_generic addrspace(4)* %6,
                __opencl_block_literal_generic addrspace(4)** %1
  ...
}
```

- Some refactoring of code generation may be needed for global blocks to support constant address space.

REDUNDANT AST NODES FOR IMPLICIT DEFINITIONS

```
$ clang -cl-std=CL2.0 -Xclang -ast-dump
```

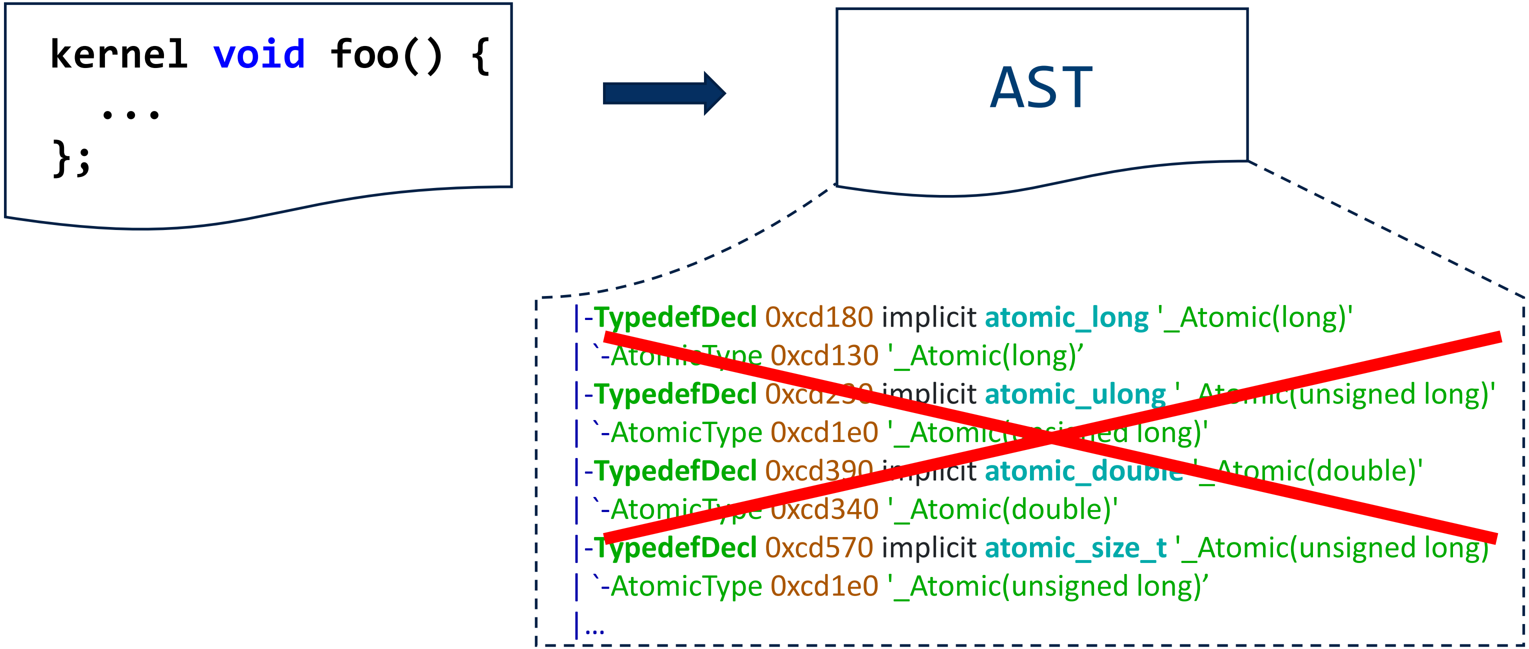


```
kernel void foo() {
  ...
};
```

AST

```
|-TypedefDecl 0xcd180 implicit atomic_long '_Atomic(long)'
| `-AtomicType 0xcd130 '_Atomic(long)’
|-TypedefDecl 0xcd230 implicit atomic_ulong '_Atomic(unsigned long)'
| `-AtomicType 0xcd1e0 '_Atomic(unsigned long)'
|-TypedefDecl 0xcd390 implicit atomic_double '_Atomic(double)'
| `-AtomicType 0xcd340 '_Atomic(double)'
|-TypedefDecl 0xcd570 implicit atomic_size_t '_Atomic(unsigned long)'
| `-AtomicType 0xcd1e0 '_Atomic(unsigned long)’
|…
```

REDUNDANT AST NODES FOR IMPLICIT DEFINITIONS



```
$ clang -cl-std=CL2.0 -Xclang -ast-dump
```

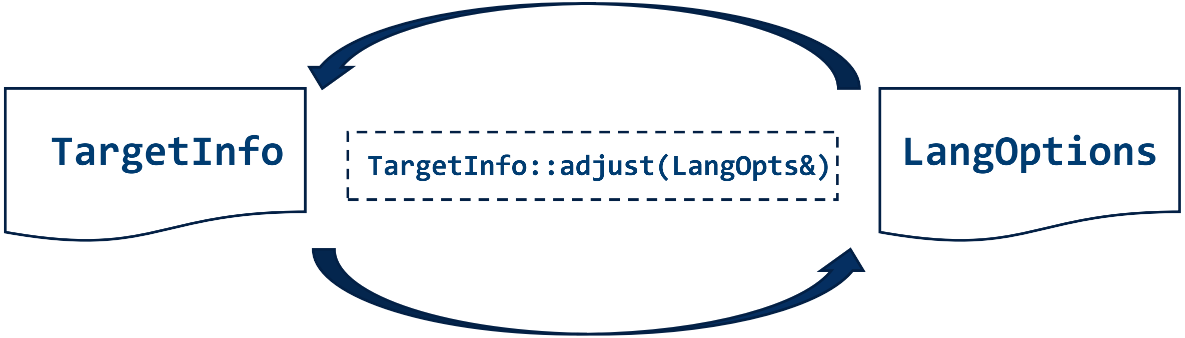
```
kernel void foo() {
  ...
};
```

```
~~|-TypedefDecl 0xcd180 implicit atomic_long '_Atomic(long)'~~
~~| `-AtomicType 0xcd130 '_Atomic(long)’~~
~~|-TypedefDecl 0xcd230 implicit atomic_ulong '_Atomic(unsigned long)'~~
~~| `-AtomicType 0xcd1e0 '_Atomic(unsigned long)'~~
~~|-TypedefDecl 0xcd390 implicit atomic_double '_Atomic(double)'~~
~~| `-AtomicType 0xcd340 '_Atomic(double)'~~
~~|-TypedefDecl 0xcd570 implicit atomic_size_t '_Atomic(unsigned long)'~~
| `-AtomicType 0xcd1e0 '_Atomic(unsigned long)’
|…
```

- Removed AST nodes creation for implicit definitions when unnecessary and simplified diagnostics: D97058, D92244

TARGET OPTIONS AND LANGUAGE OPTIONS SETTING IN CLANG

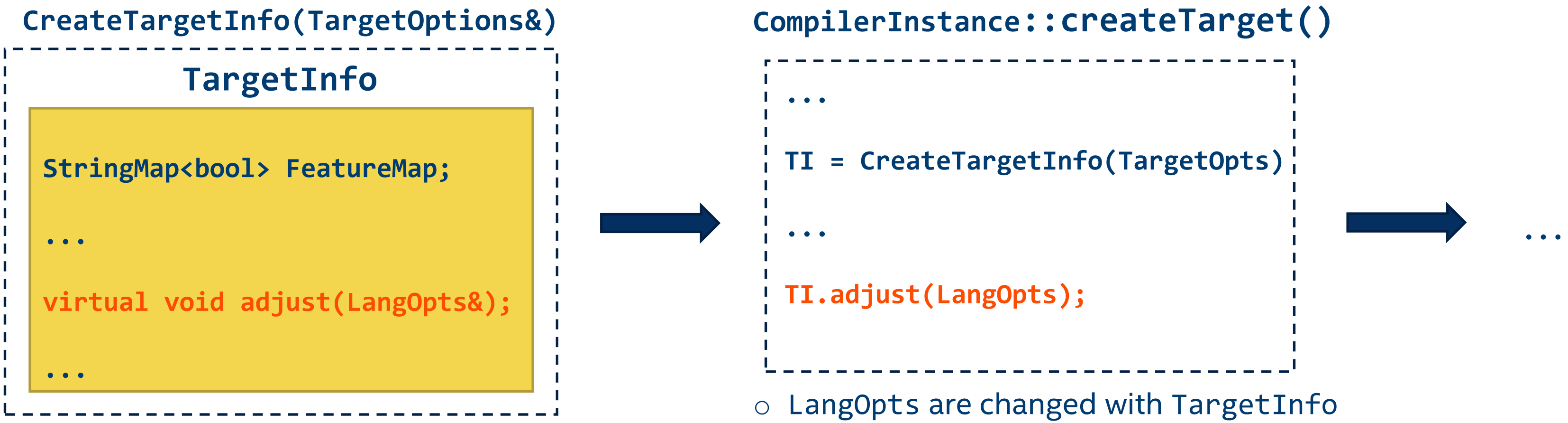
- `TargetInfo` and `LangOptions` augment each other:



- Some target options are set according to language options.
- Some language options are set according to target information, for example arithmetic fences, OpenCL generic address space/pipes, AltiVec extensions in PPC target.

TARGET OPTIONS AND LANGUAGE OPTIONS SETTING IN CLANG

- As a result, TargetInfo is not immutable after creation:



CreateTargetInfo(TargetOptions&)

TargetInfo

```
StringMap<bool> FeatureMap;

...

virtual void adjust(LangOpts&);

...
```

CompilerInstance::createTarget()

```
...

TI = CreateTargetInfo(TargetOpts)

...

TI.adjust(LangOpts);
```

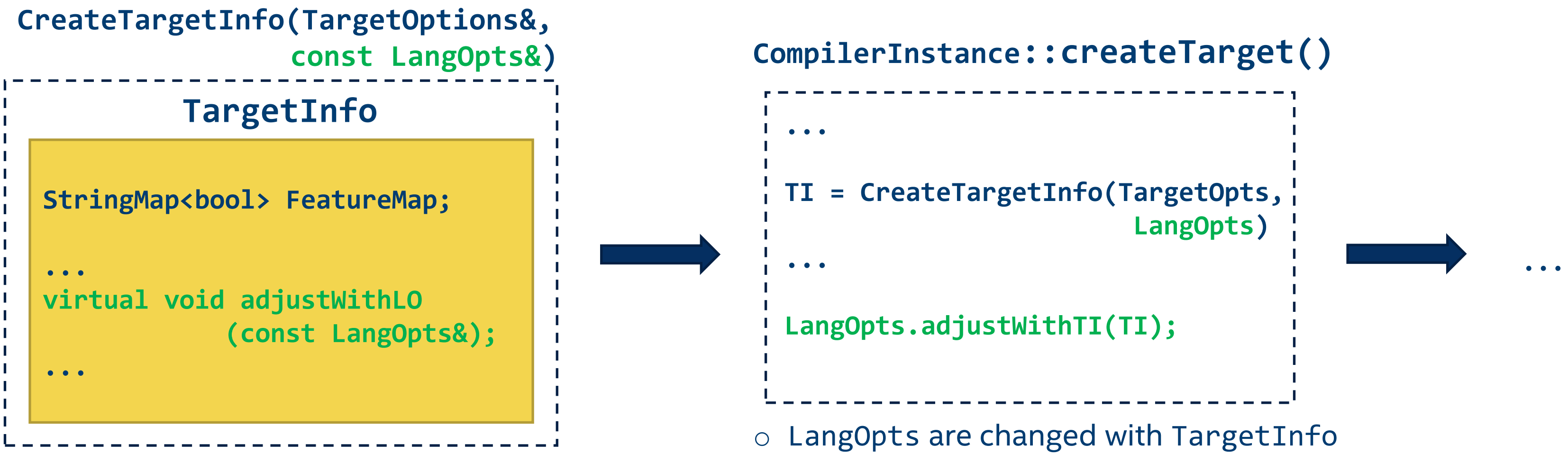
...

- TargetInfo initialized via TargetOptions

- ○ LangOpts are changed with TargetInfo
- ○ TargeInfo is changed with LangOpts

Target is not immutable after creation as it meant to be!

TARGET OPTIONS AND LANGUAGE OPTIONS SETTING IN CLANG



```
CreateTargetInfo(TargetOptions&,
                 const LangOpts&)
```

TargetInfo

```
StringMap<bool> FeatureMap;

...
virtual void adjustWithLO
             (const LangOpts&);
...
```

```
CompilerInstance::createTarget()
```

```
...

TI = CreateTargetInfo(TargetOpts,
                      LangOpts)
...

LangOpts.adjustWithTI(TI);
```

...

- TargetInfo initialized via:
  - TargetOptions
  - LangOptions

- LangOpts are changed with TargetInfo

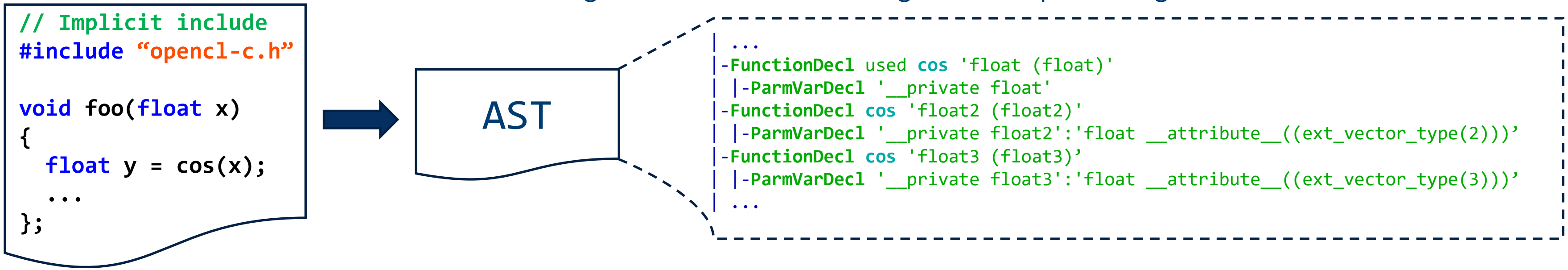
Target is immutable after creation

- Proposed design solution to extend interfaces when creating a target info: D110036

OTHER IMPROVEMENTS AND FUTURE DIRECTIONS: BUILT-INS

- Old behaviour:

```
$ clang -cl-std=CL2.0 -Xclang -ast-dump -Xclang -finclude-default-header:
```



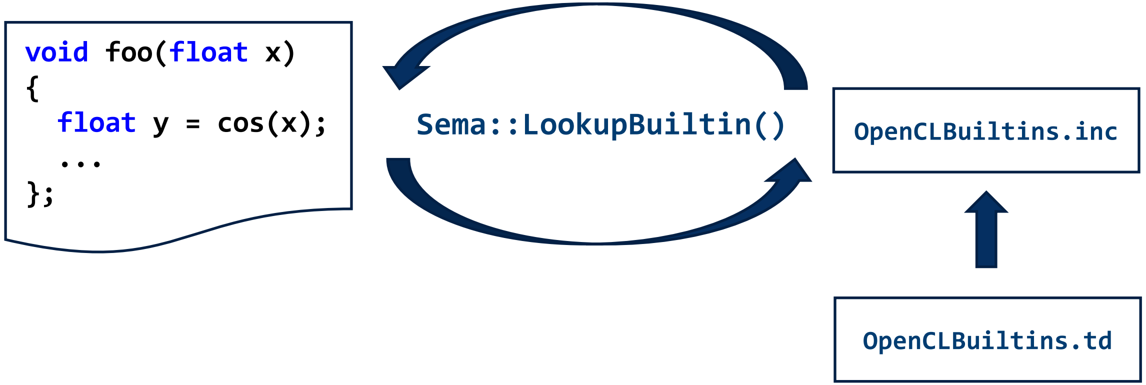
```
// Implicit include
#include “opencl-c.h”

void foo(float x)
{
  float y = cos(x);
  ...
};
```

```
| ...
|-FunctionDecl used cos 'float (float)'
| |-ParmVarDecl '__private float'
|-FunctionDecl cos 'float2 (float2)'
| |-ParmVarDecl '__private float2':'float __attribute__((ext_vector_type(2)))’
|-FunctionDecl cos 'float3 (float3)’
| |-ParmVarDecl '__private float3':'float __attribute__((ext_vector_type(3)))’
| ...
```

- To use OpenCL built-in functions, user had to explicitly specify `–finclude-default-header` flag, which includes a file with all OpenCL built-in functions declarations.
- As a result, increase in compilation time when parsing header with declarations.

OTHER IMPROVEMENTS AND FUTURE DIRECTIONS: BUILT-INS

- New behaviour:



```
void foo(float x)
{
  float y = cos(x);
  ...
};
```

Sema::LookupBuiltin()

OpenCLBuiltins.inc

OpenCLBuiltins.td

- Dynamically insert built-in function declaration from table if needed. Table is auto-generated from TableGen file.
- Implemented by Sven Van Haastregt, see his talk.

OTHER IMPROVEMENTS AND FUTURE DIRECTIONS

- Dynamic lookup is enabled as default behaviour: D96515.
  - Option `-cl-no-stdinc` can be used to use the old approach (via header).
- Planning to deprecate usage of `opencl-c.h` and always use dynamic lookup.
  - Testing scheme for soft switching is proposed: D99577
  - Make sure that TableGen definitions provide on-to-one functionality with `opencl-c.h` via diffing.
- Planning to support device enqueue for OpenCL C 3.0.
- Planning to support the rest of OpenCL C 3.0 built-ins.

CONTRIBUTORS

- Anastasia Stulova, Arm
- Sven van Haastregt, Arm
- Anton Zabaznov, Intel
- Dave Airlie, Red Hat