Google

gpucc: An Open-Source GPGPU Compiler

Jingyue Wu, Artem Belevich, Eli Bendersky, Mark Heffernan, Chris Leary, Jacques Pienaar, Bjarke Roune, Rob Springer, Xuetian Weng, **Robert Hundt**

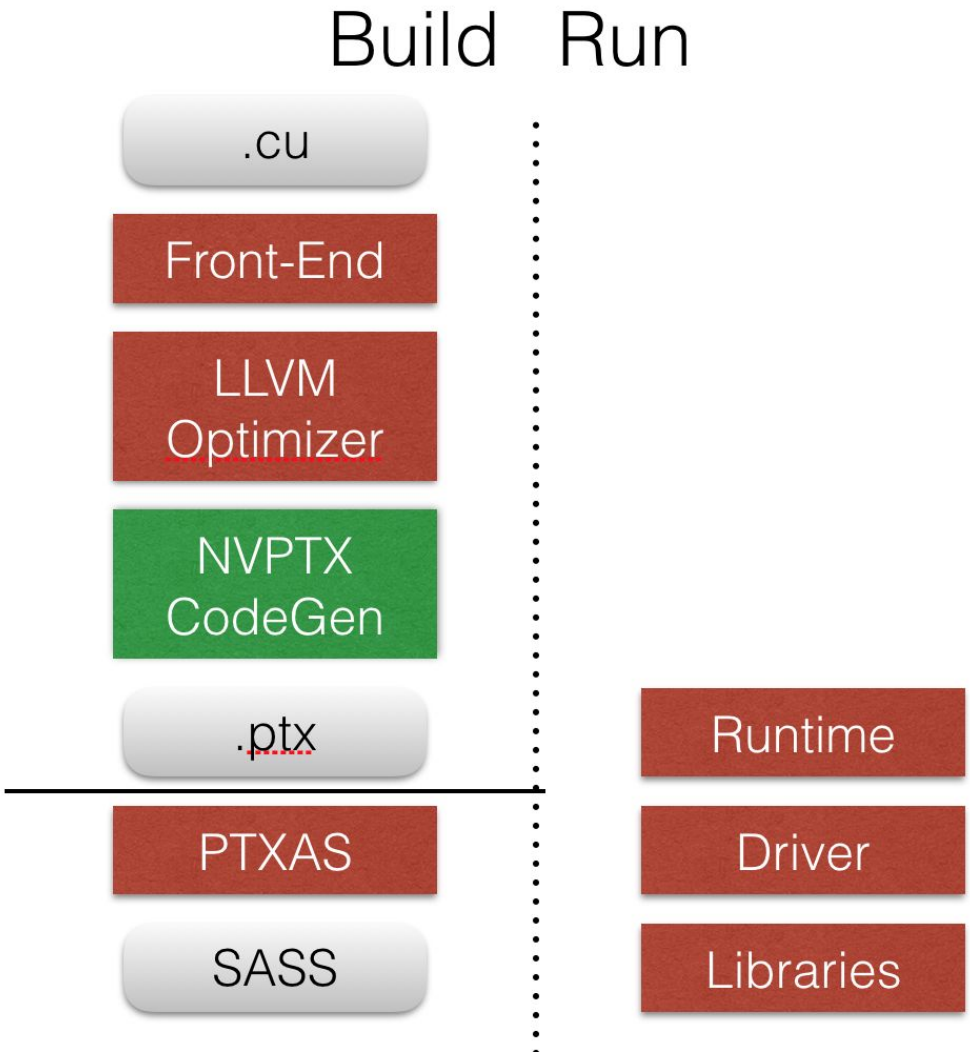
Why Make an Open-Source GPGPU Compiler?

- **Research** - No reproducible research possible
- **Security** - No binary blobs in the datacenter
- **Binary Dependencies** - Software updates become difficult
- **Performance** - We can always do better on *our* benchmarks
- **Bug Fix Time** - We can be much faster than vendors
- **Quality** - Open Source works
- **Language Features** - Incompatible lang environments
- **Lock-In** - Nobody likes that
- **Philosophical** - We just **want** to do this ourselves

Why Make an Open-Source GPGPU Compiler?

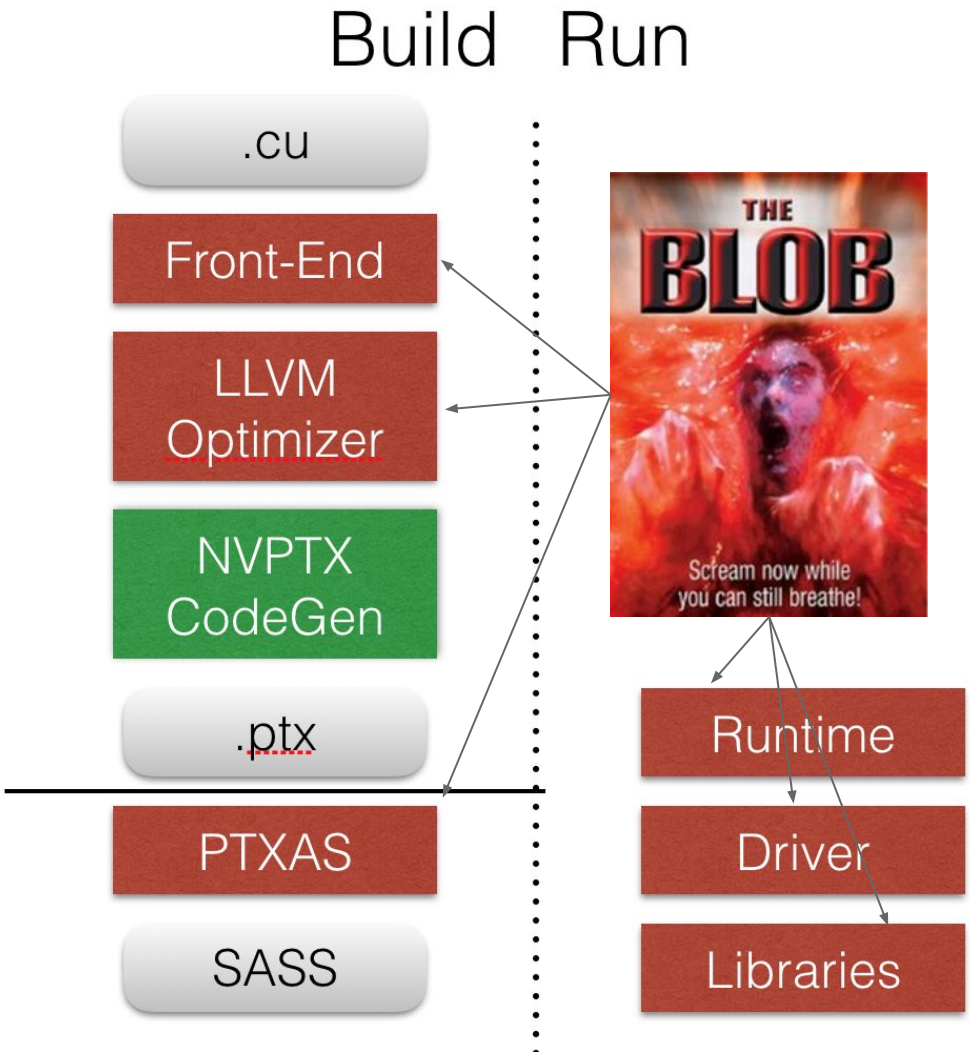
- *Enable compiler research*
- *Enable community contributions*
- *Enable industry breakthroughs*

Binary Blobs Are Everywhere



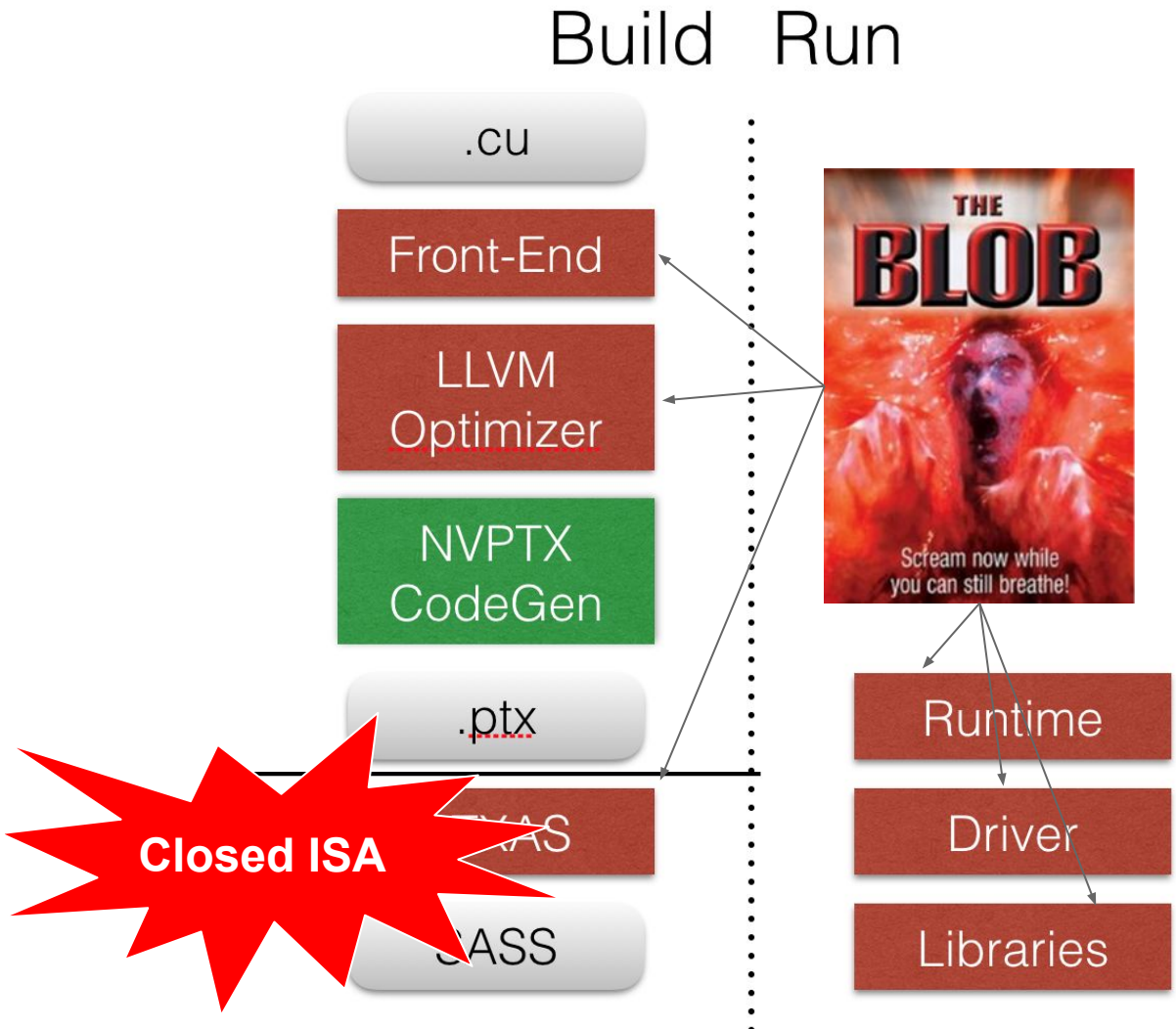
Binary Blobs Are Everywhere

Scream now while you can still breathe



Binary Blobs Are Everywhere

SASS is CLOSED (!)



GPUCC to the Rescue ;-)

Your Fabulous Presenters

Jingyue Wu
jingyue@google.com

Jacques Pienaar
jpienaar@google.com

Artem Belevich
tra@google.com

Agenda

When	Who	What
2:00 - 3:00	Artem Belevich	Overview and user guides
3:00 - 3:30	Jingyue Wu	Optimizing gpucc (part 1)
3:30 - 4:00		Coffee break
4:00 - 4:45	Jacques Pienaar	Optimizing gpucc (part 2)
4:45 - 5:30	Jacques Pienaar Jingyue Wu	Contributing to gpucc: action items & research opportunities

Google

gpucc User Guide and Architecture

Jingyue Wu, **Artem Belevich**, Eli Bendersky, Mark Heffernan, Chris Leary, Jacques Pienaar, Bjarke Roune, Rob Springer, Xuetian Weng, Robert Hundt

LLVM & Clang

Why LLVM/Clang?

- LLVM has NVPTX back-end contributed by NVIDIA.
- Clang and LLVM are easier to understand and modify than gcc.
  - The LLVM Project is a collection of modular and reusable compiler and toolchain technologies.
  - The Clang ASTs and design are intended to be easily understandable by anyone who is familiar with the languages involved.
  - http://clang.llvm.org/comparison.html#gcc
- Active and helpful user community.
- Somewhat more convenient licensing terms:
  - LLVM and Clang are under BSD-like UIUC license.
  - LLVM project does not require copyright assignments.

LLVM Code Generator Highlights

- Approachable C++ code base, modern design, easy to learn
  - Strong and friendly community, good documentation
- Language and target independent code representation
  - Very easy to generate from existing language front-ends
  - Text form allows you to write your front-end in perl if you desire
- Modern code generator:
  - Supports both JIT and static code generation
  - Much easier to retarget to new chips than GCC
- Many popular targets supported:
  - X86, ARM, PowerPC, MIPS, XCore, NVPTX, AMDGPU, etc.

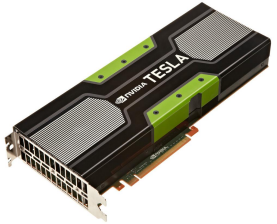
Clang

- Unified parser for C-based languages
  - Language conformance (C, Objective-C, C++)
  - Useful error and warning messages
- Library based architecture with finely crafted API’s
  - Usable and extensible by mere mortals
  - Reentrant, composable, replaceable
- Multi-purpose
  - Indexing, static analysis, code generation
  - Source to source tools, refactoring
- High performance!
  - Low memory footprint, fast compiles

Compiler Architecture

Mixed-Mode CUDA Code

```
__global__ void Write42(float *out) {
  out[threadIdx.x] = 42.0f;
}
```



GPU/device

Mixed-Mode CUDA Code

```
int main() {
  float* arr;
  cudaMalloc(&arr, 128*sizeof(float));
  Write42<<<1, 128>>>(arr);
}
```

```
__global__ void Write42(float *out) {
  out[threadIdx.x] = 42.0f;
}
```

CPU/host

GPU/device

Mixed-Mode CUDA Code

foo.cu

```
int main() {
  float* arr;
  cudaMalloc(&arr, 128*sizeof(float));
  Write42<<<1, 128>>>(arr);
}
```

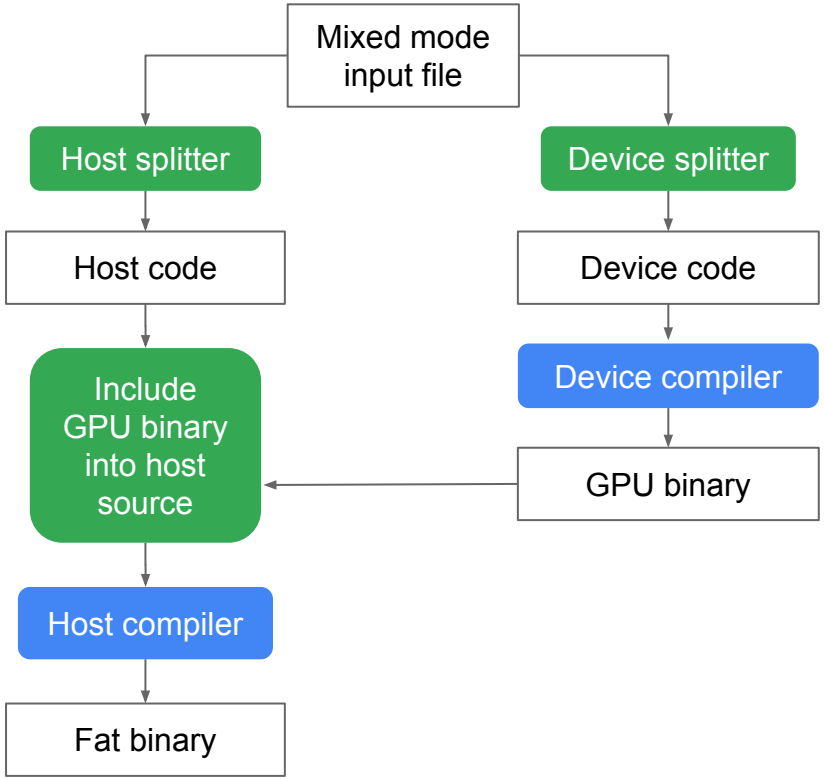
```
__global__ void Write42(float *out) {
  out[threadIdx.x] = 42.0f;
}
```

CPU/host

GPU/device

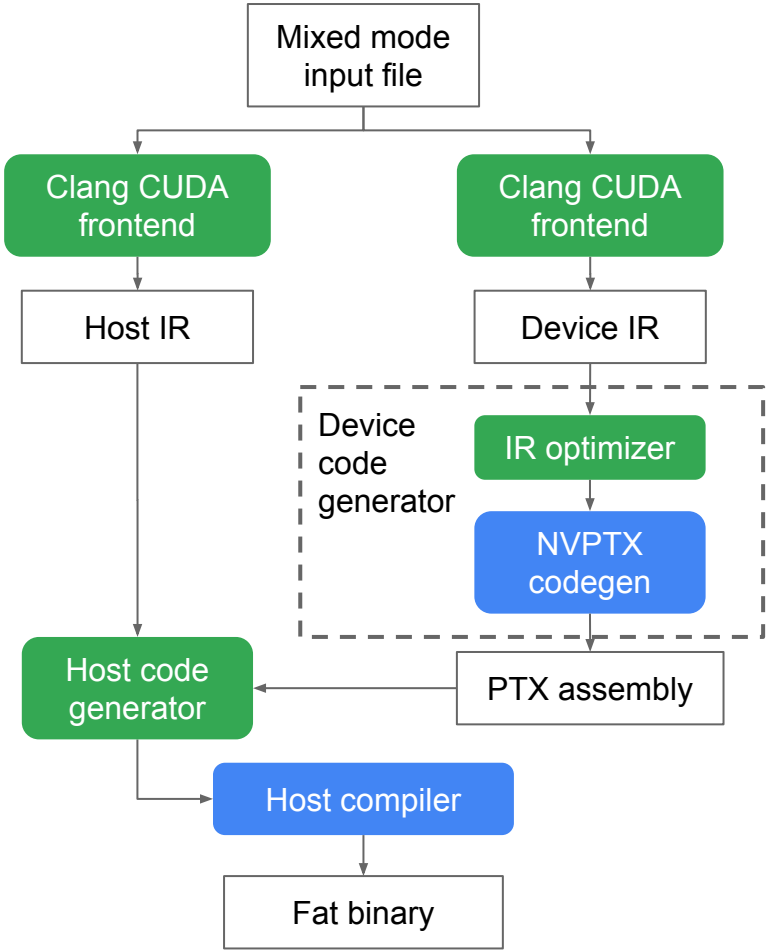
Separate Compilation

- Split host/device into separate files.
- Compile device code.
- Include results into host source.
- Add host-side glue code to register device-side kernels with CUDA runtime.
- Compile host code.



Dual-Mode Compilation

- Compile mixed mode source w/o splitting
- We still have to compile for host and device because CUDA relies on different preprocessor macros for host and device compilation.
- Compiler sees host and device code



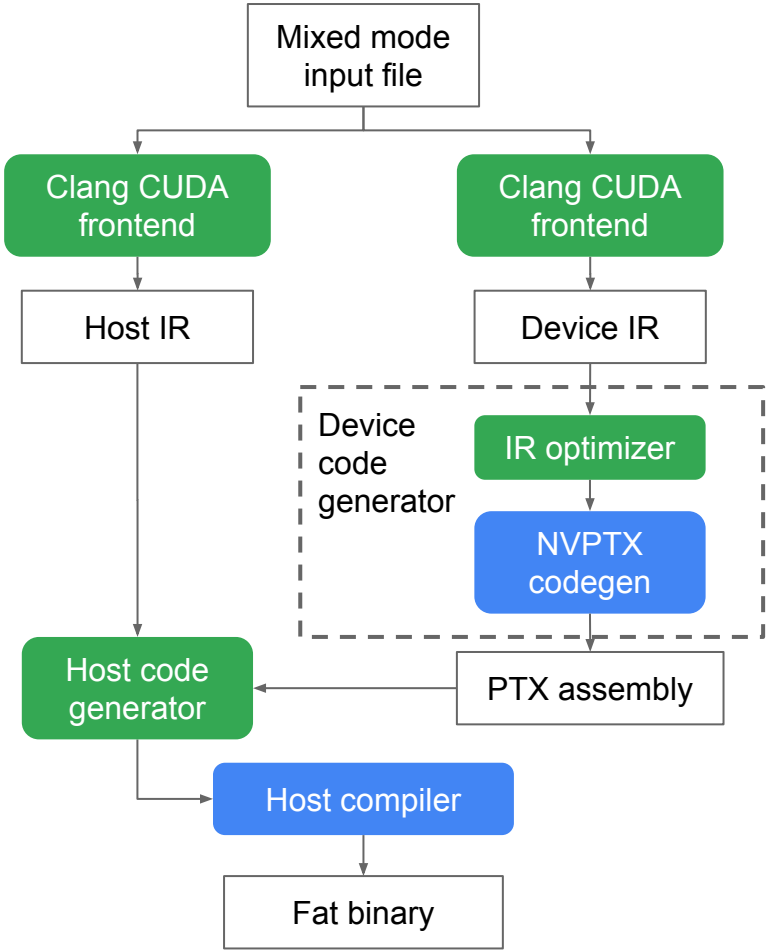
Dual-Mode Compilation

Benefits:

- Faster compilation.
- Compiler sees complete source.
- Same compiler handles host and device compilation.

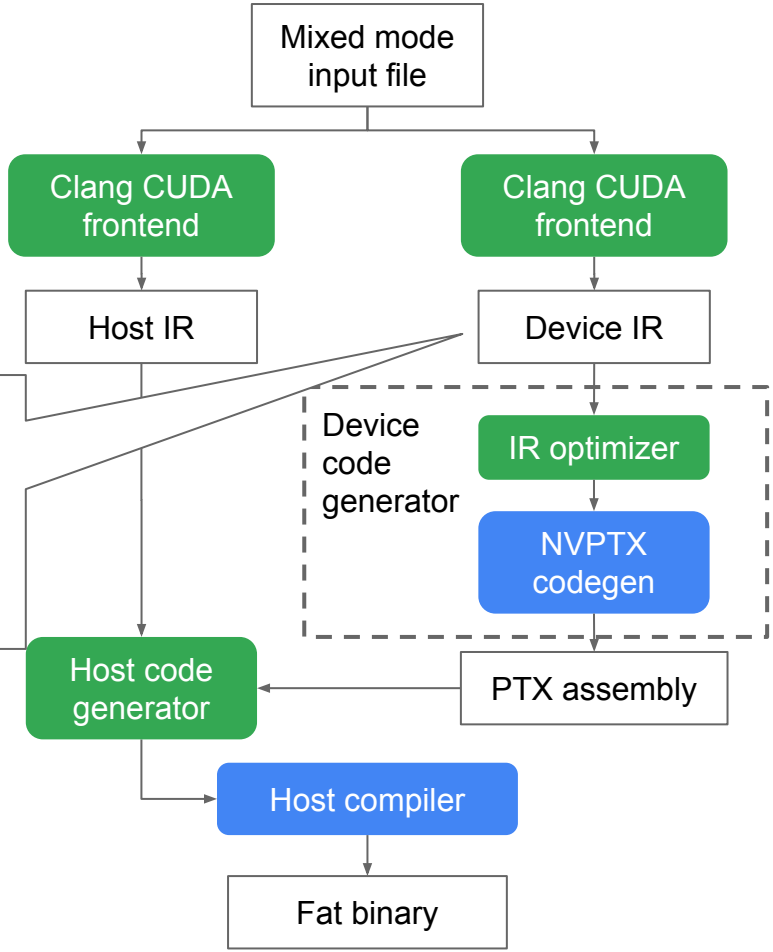
Downsides:

- Have to deal with namespace conflicts between host and device code.
- Deviates from nvcc.



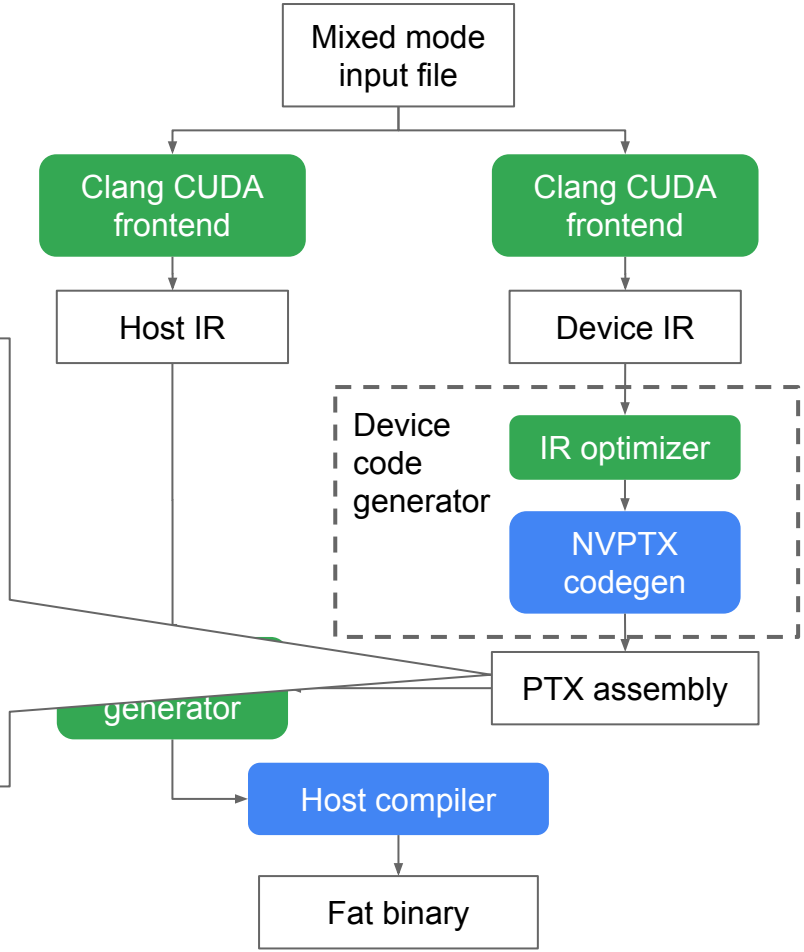
Dual-Mode Compilation

```
define void @kernel(float* %out) {
  %tidx = call i32 @llvm.nvvm.read.ptx.sreg.tid.x();
  %p = getelementptr float, float* %out, i32 %tidx
  store float 42.0f, float* %p
  ret void
}
```



Dual-Mode Compilation

```
.visible .entry kernel(
    .param .u64 kernel_param_0) {
  ld.param.u64        %rd1, [kernel_param_0];
  cvta.to.global.u64  %rd2, %rd1;
  mov.u32             %r1, %tid.x;
  mul.wide.u32        %rd3, %r1, 4;
  add.s64             %rd4, %rd2, %rd3;
  mov.u32             %r2, 1109917696;
  st.global.u32       [%rd4], %r2;
  ret;
}
```



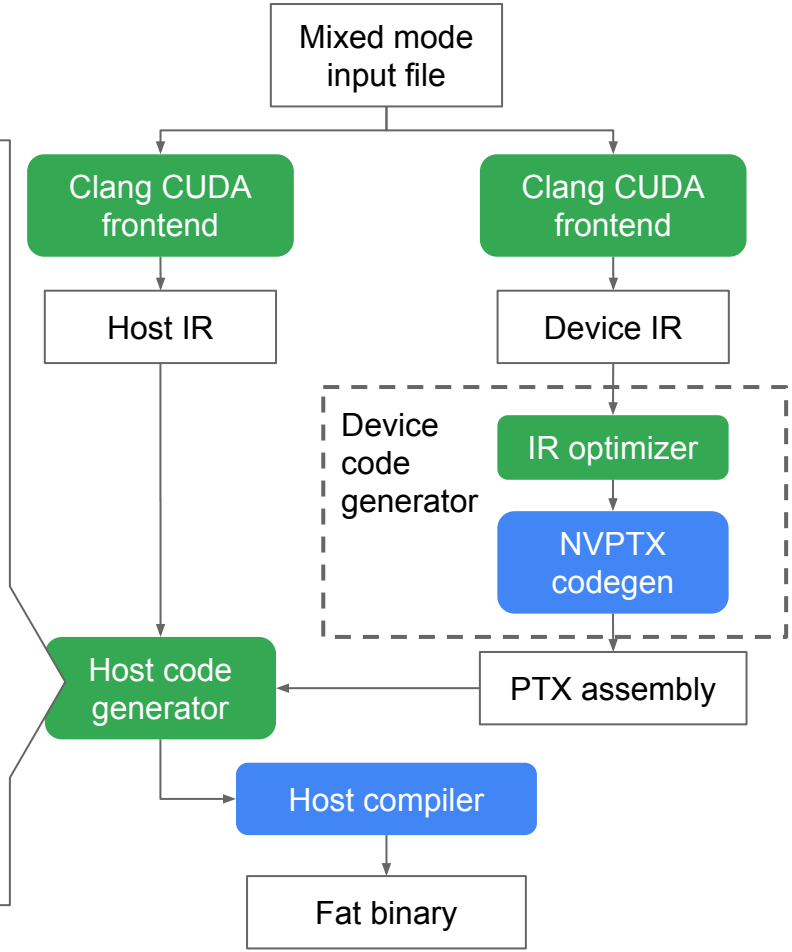
Dual-Mode Compilation

```
const char* __ptx = “<GPU binary goes here>”;

void __cuda_module_ctor() {
  __cudaRegisterFatBinary(..., __ptx, ...);
  __cudaRegisterFunctions(kernel, "kernel");
}

void kernel(float* out) {
  cudaSetupArgument(out, ...);
  cudaLaunch(kernel);
}

int main() {
  float *arr;
  cudaMalloc(&arr, 128*sizeof(float));
  cudaConfigureCall(1, 128, 0, nullptr);
  kernel_stub(arr);
  return 0;
}
```



Attribute-based function overloading

- Consider __host__/__device__ attribute during function overload resolution.
- Host and device functions can coexist in AST now.
- Paves way for using overloads (instead of preprocessor with __CUDA_ARCH__ macro) to provide device-specific functions.
- .. and, eventually, compiling and generating code for both host and device within a single compiler invocation.

- <math.h>

```
extern double
exp (double __x) throw ();
```

- CUDA

```
static __device__ __inline__ double
exp(double __x) { return __nv_exp(__x); }
```

Differences vs. nvcc

- Target attribute based function overloading instead of separate compilation.
  - Clang will compile some code that nvcc will not.
- Built-in variables are not supported by compiler
  - .. implemented in a header file instead.
  - `extern const dim3 blockDim;` // will fail
- Limited CUDA runtime support
  - Runtime is not documented by NVidia.
  - Kernel launching works, but using other CUDA runtime APIs may or may not work (yet).
- Number of features are not supported (yet):
  - Textures/surface lookup
  - Managed variables.
  - Device-side lambda functions.

Differences in supported features

- Device-side compilation does not support everything available on host
- E.g. TLS, exceptions, inline assembly syntax is different, different set of compiler builtins

Solution:

- Filter out ‘unsupported’ errors in host-only code during device compilation and vice versa.
- Make both host and device-specific builtins available and add appropriate __host__ or __device__ attributes, so compiler can control their use.

CUDA headers

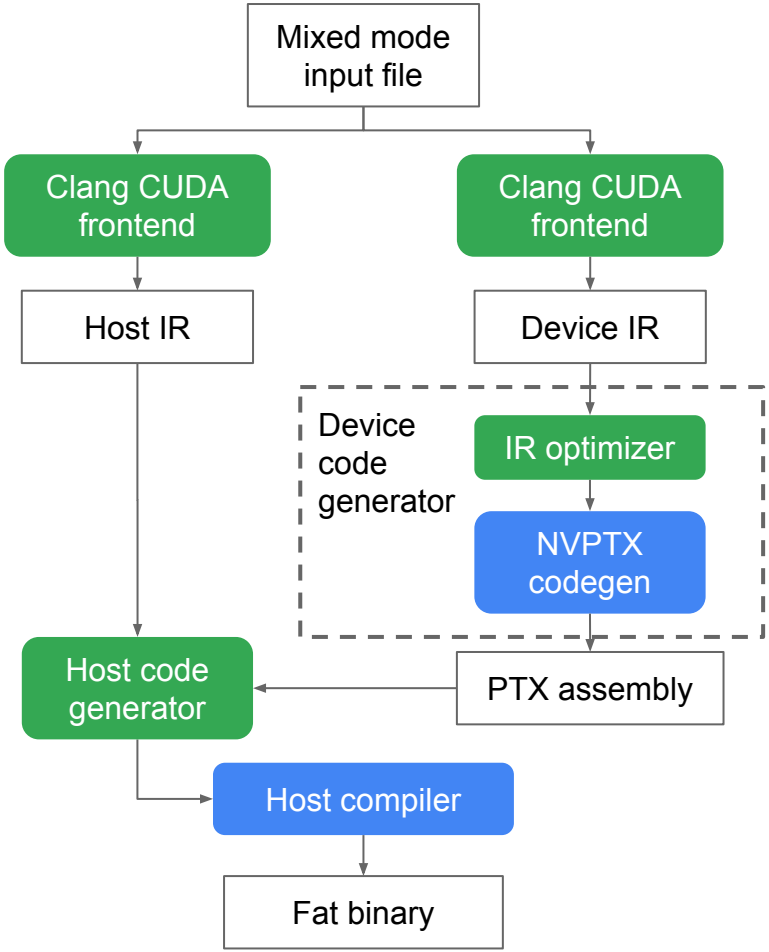
- Written for nvcc which uses separate compilation mode.
- Provides multiple views of CUDA symbols depending on where nvcc includes them from in its pipeline.
- None of the views provide function implementations with __device__ attribute.

Solution (for now):

- Preprocessor magic.
- Pre-include CUDA headers in a way usable by clang.
- Works for particular CUDA versions only.

Dual-Mode Compilation

```
$ clang++ foo.cu -o foo \
      -lcudart_static -lcuda -ldl -lrt -pthread
$ ./foo
```



Using Clang

Installation

- Prerequisites
  - Linux machine (Ubuntu 14.04/x86_64)
  - Reasonably modern GPU (Fermi/GTX4xx or newer)
- CUDA
  - Supported versions: 7.0 or 7.5
- Clang
  - V3.8.0 or newer

CUDA installation

- Download CUDA 7.5
  - https://developer.nvidia.com/cuda-toolkit
- Install according to CUDA instructions
  - http://docs.nvidia.com/cuda/cuda-getting-started-guide-for-linux/
- Install CUDA examples
  - http://docs.nvidia.com/cuda/cuda-samples/#getting-cuda-samples

LLVM & Clang Installation

- Download v3.8.0 from llvm.org
  - http://llvm.org/releases/download.html#3.8.0
- .. or use prebuilt nightly packages
  - http://llvm.org/apt/
- Installation on Ubuntu (VERSION=precise/trusty/willy):
  - `sudo add-apt-repository deb "http://llvm.org/apt/$VERSION/ llvm-toolchain-$VERSION main"`
  - `wget -O - http://llvm.org/apt/llvm-snapshot.gpg.key | sudo apt-key add -`
  - `sudo apt-get update`
  - `sudo apt-get install clang-3.9`
- .. or build from sources
  - http://clang.llvm.org/get_started.html

Clang/LLVM installation

- Verify that it detects CUDA installation:

  ```
  $ clang++ -v -x cuda /dev/null -### 2>&1 | grep CUDA
  Found CUDA installation: /usr/local/cuda
  ```

- If CUDA is installed in non-default location, add
  - --cuda-path=/your/cuda/install/path

Basic compilation

```
$ wget -O axpy.cu http://pastebin.com/raw/EwmXchJ0

$ clang++ axpy.cu -o axpy \
    -L<CUDA install path>/<lib64 or lib> \
    -lcudart_static -ldl -lrt -pthread
$ ./axpy
y[0] = 2
y[1] = 4
y[2] = 6
y[3] = 8
```

Key CUDA compilation parameters

- Specify GPU architecture(s) to compile for:
  - --cuda-gpu-arch=sm_CC
  - .. where CC is compute capability: 20, 21, 30, 32, 35, 50, 52, 53
  - Can be used multiple times.
  - Each unique --cuda-gpu-arch flag creates device compilation pass.
  - Default is --cuda-gpu-arch=sm_20
  - Compute capabilities for specific cards are listed here:
    - https://developer.nvidia.com/cuda-gpus
  - Compute Capability features are listed here:
    - http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capabilities

Useful optimization options

- Enable FMA
  - -ffp-contract=fast
  - Default is to fuse mul+add->fma according to explicit FP_CONTRACT pragma only.
- Fast math
  - -ffast-math
  - Does not affect optimizations in clang, but defines __FAST_MATH__ macro with will affect some math functions provided by CUDA headers.

Partial host/device compilation

- For device side only compilation, use
  - --cuda-device-only
  - Implies ‘-c’ so compiles to object file only
- If you want host-only compilation, use
  - --cuda-host-only
  - Will also compile to .o, but without including device-side GPU objects
  - .. which will **not** work if you link it into an executable and run it.

Pass parameters to ptxas

- -Xcuda-ptxas <ptxas option>
  - Similar to nvcc's --ptxas-options=...
  - Use another -Xcuda-ptxas <arg> for ptxas option argument, if needed.
- Use cases:
  - -v -- Print verbose info on register and memory use by kernels.
  - --maxrregcount -- Specify the maximum amount of registers that GPU functions can use
  - --def-load-cache/--def-store-cache -- Specify default caching behavior on loads/stores.
  - http://docs.nvidia.com/cuda/parallel-thread-execution/#cache-operators

Accessing intermediate compiler files

- Easy way
  - Add ‘-save-temps’ option.

```
$ clang++ -c axpy.cu -save-temps
$ ls axpy*
axpy.bc axpy.cu axpy.cu.fatbin axpy.cui axpy.o axpy.s
axpy-sm_20.bc axpy-sm_20.cui axpy-sm_20.o axpy-sm_20.s
```

What’s in those files?

Device files have ‘`-sm-CC`’ suffix. Host files have no additional suffix.

.cui	Preprocessed source
.bc	LLVM IR bitcode. Can be converted to readable text with ‘llvm-dis’
.s	Assembly/PTX
.o	Host: object file. Device: cubin file
.fatbin	Binary that contains combined cubin files for all device compilations.

Cookbook

- Device PTX on standard output:
  - $ clang++ axpy.cu --cuda-device-only -S -o -
- Device-side IR (as text) on standard output:
  - $ clang++ axpy.cu --cuda-device-only -S -Xclang -emit-llvm -o -
- Host-side is similar
  - Just use --cuda-host-only
- See what compiler driver does under-the-hood
  - $ clang++ -c axpy.cu -###
  - .. add -save-temps and see even more.
- Print device AST:
  - $ clang++ axpy.cu --cuda-device-only -fsyntax-only -Xclang -ast-dump
  - Add "-Xclang -ast-dump-filter -Xclang <substring>" to limit AST output.

cuobjdump

- Objdump for CUDA executables
- Works on final exe or on .fatbin files.
- Dump PTX:
  - `$ cuobjdump --dump-ptx axpy.cu.fatbin`
- Dump SASS:
  - `$ cuobjdump --dump-sass axpy.cu.fatbin`
- Limit output to particular GPU arch:
  - Add ‘-arch sm_CC’

nvdisasm

- SASS disassembler with control flow analysis and advanced display options.
- Works on cubin files (device-side .o) only.
- Disassemble axpy code:
  - nvdisasm axpy-sm_20.o
- Generate control flow graph (only for sm_30 or higher)
  - nvdisasm -cfg axpy-sm_30.o | xdot
- Print register life info (sm30+):
  - nvdisasm --print-life-ranges axpy-sm_30.o
- Annotate disassembly with source line information
  - nvdisasm --print-line-info axpy-sm_30.o

cuda-memcheck

Functional correctness checking suite included in the CUDA toolkit.

- detects out of bounds and misaligned memory access errors.
- reports hardware exceptions encountered by the GPU.
- can report shared memory data access hazards that can cause data races.

http://docs.nvidia.com/cuda/cuda-memcheck/index.html

$ cuda-memcheck --tool {memcheck, racecheck, initcheck, synccheck} ./axpy

Debugging

- NVPTX only emits line info at the moment.
- .. so no easy access to variables.
- Use '-g --cuda-noopt-device-debug' to enable debugging on device-side.
- Note: disables PTX-to-SASS optimizations. Expect lower performance.

Conclusions