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**

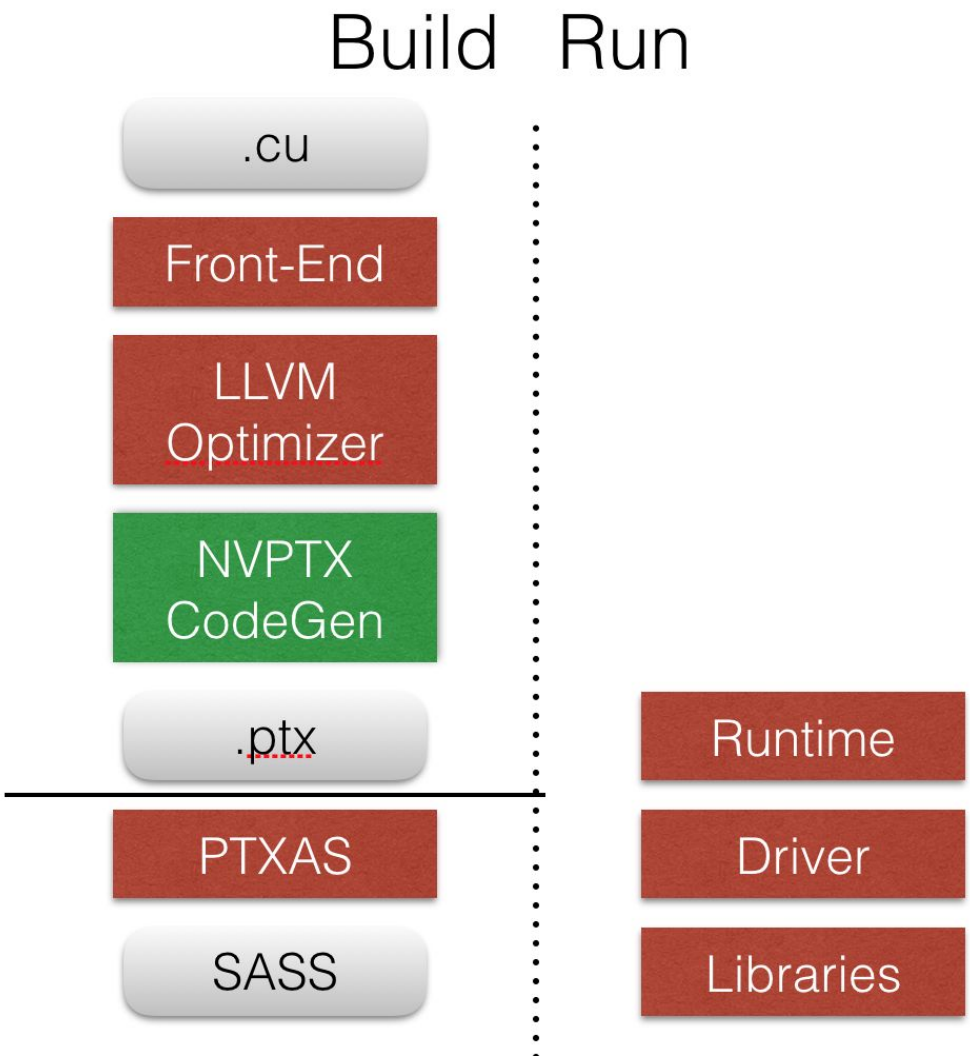
Why Make an Open-Source GPGPU Compiler?

- **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?

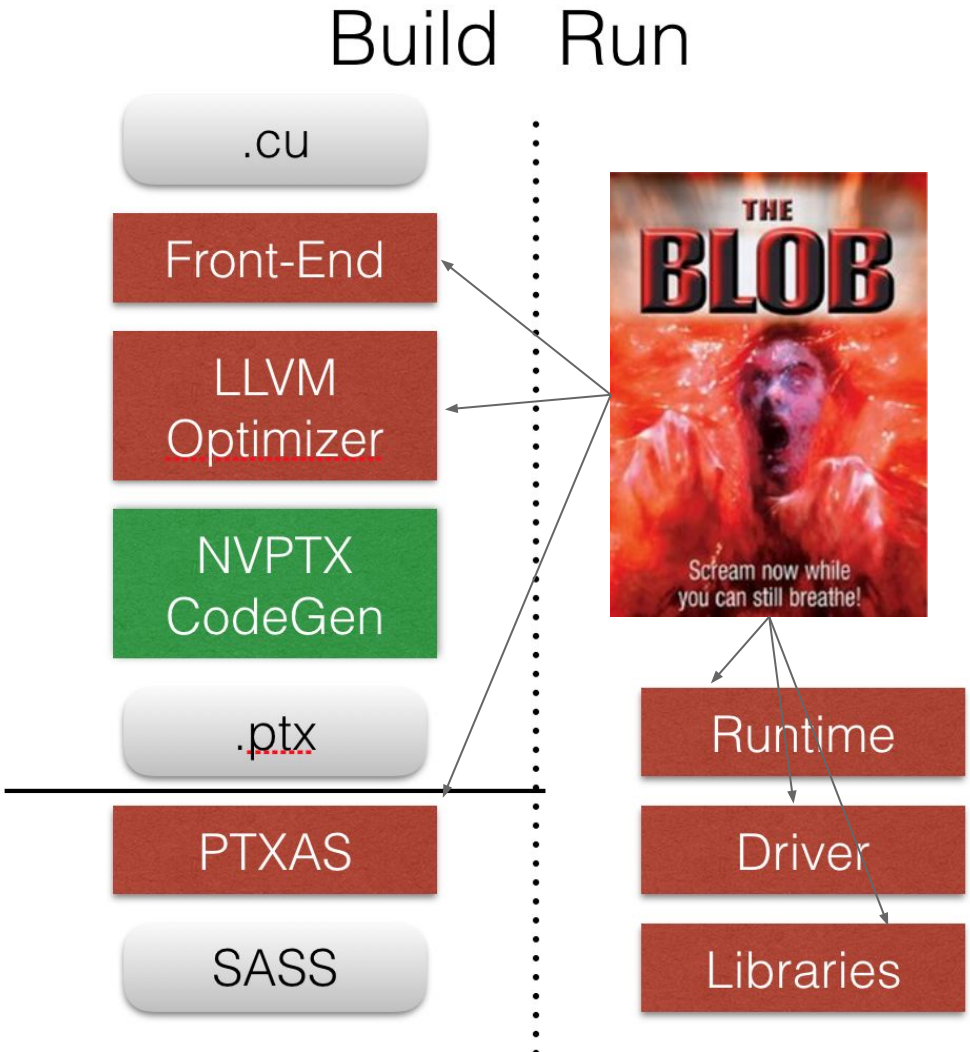
- *Enable compiler research*
- *Enable community contributions*
- *Enable industry breakthroughs*

Binary Blobs Are Everywhere



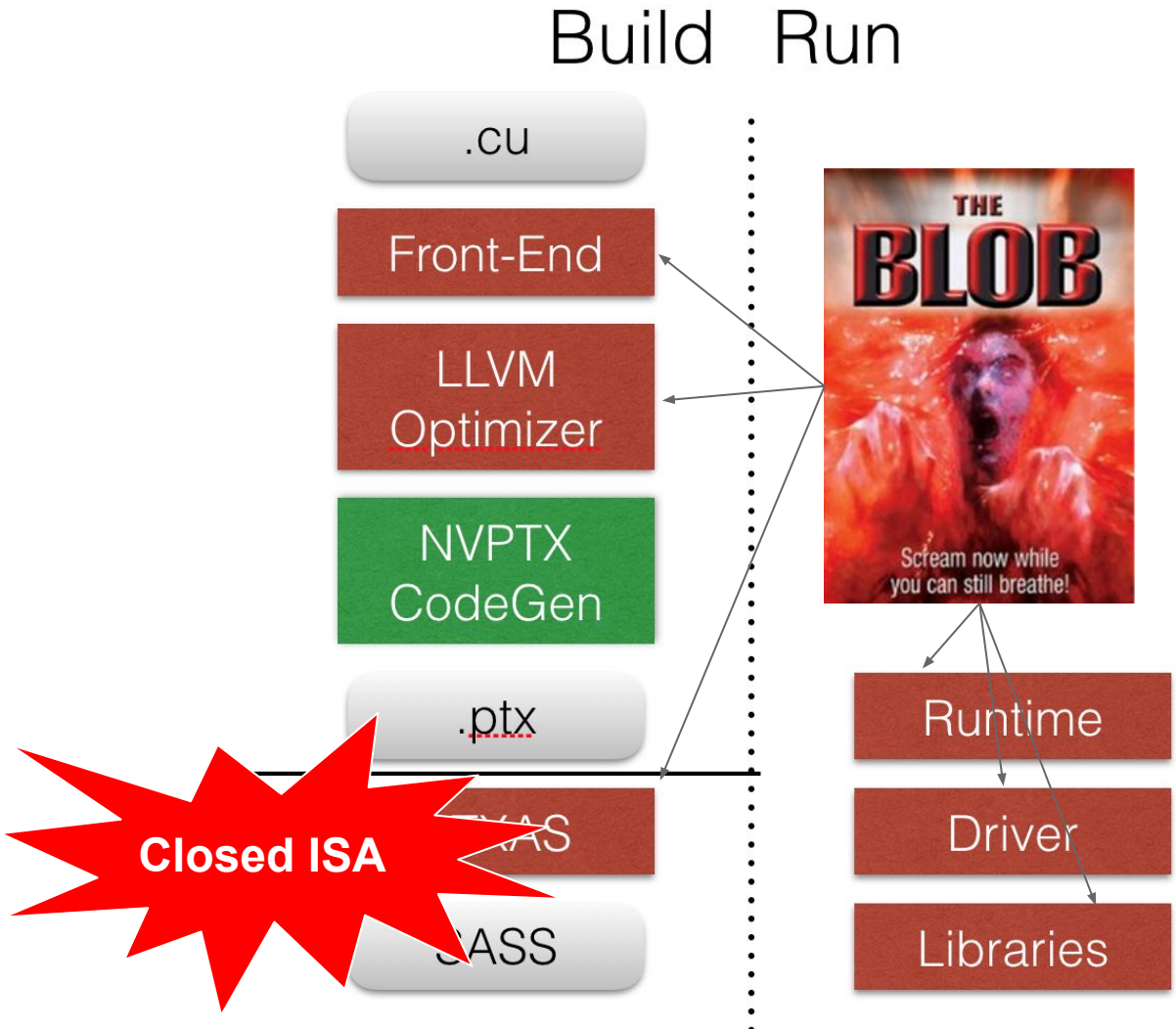
Binary Blobs Are Everywhere

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.

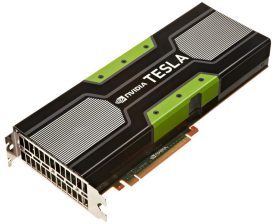
Clang

- 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);
}
```

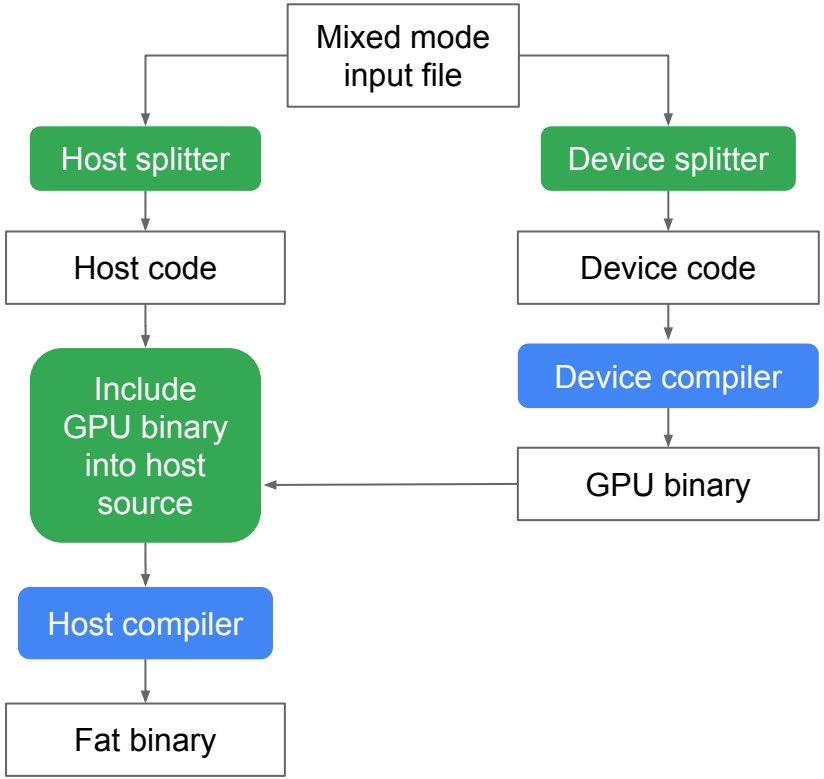
```
__global__ void Write42(float *out) {
  out[threadIdx.x] = 42.0f;
}
```

CPU/host

GPU/device

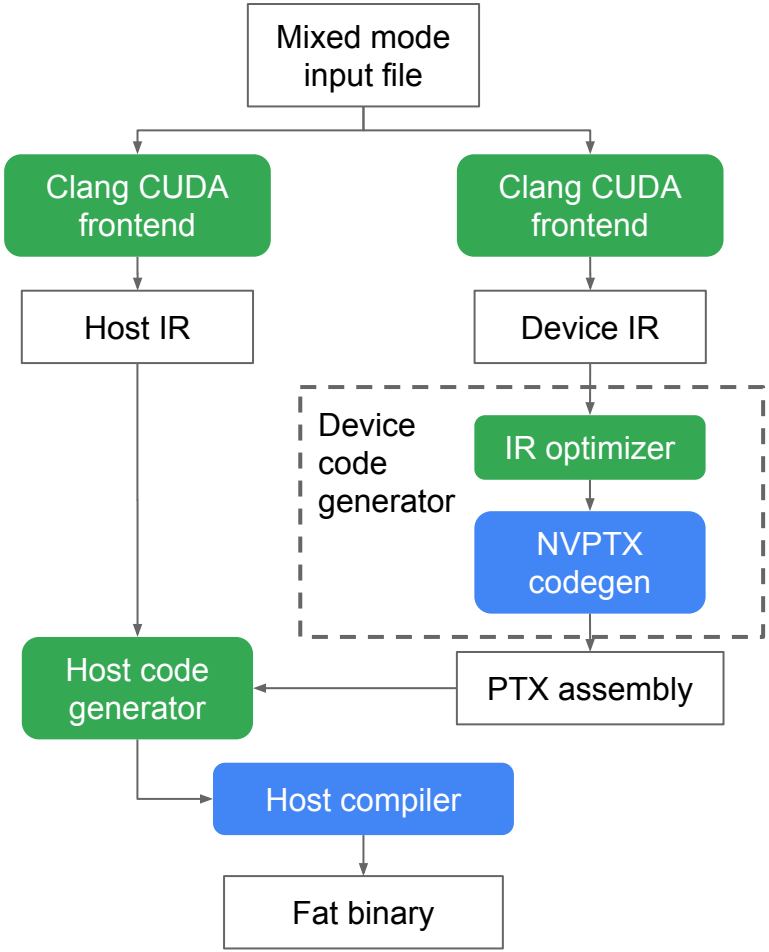
Separate Compilation

- 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



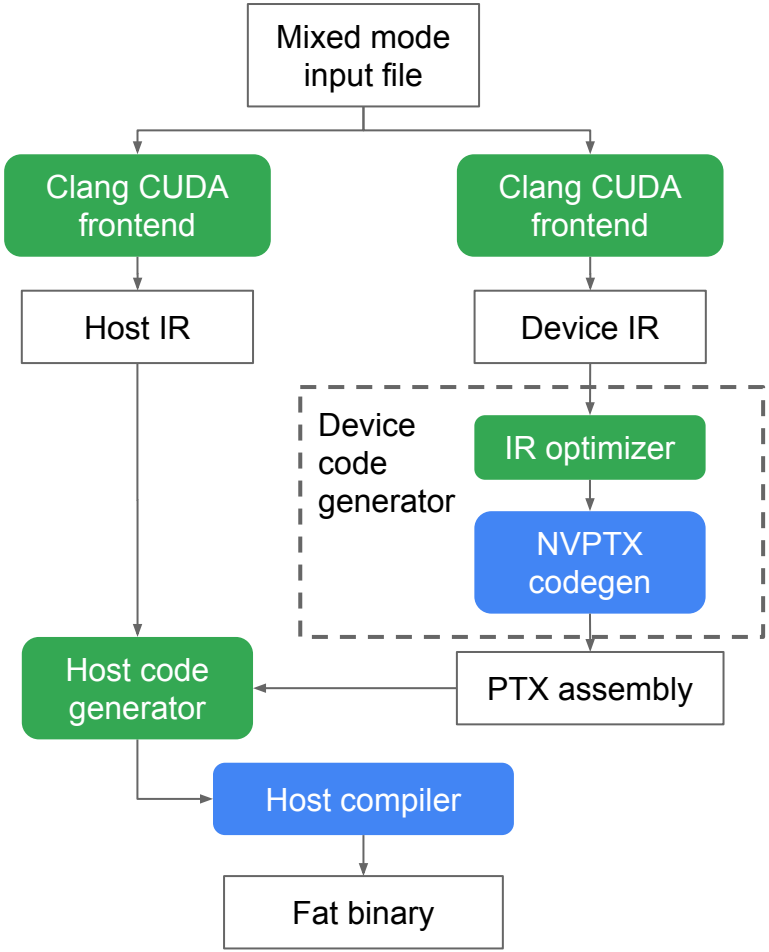
Dual-Mode Compilation

Benefits:

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

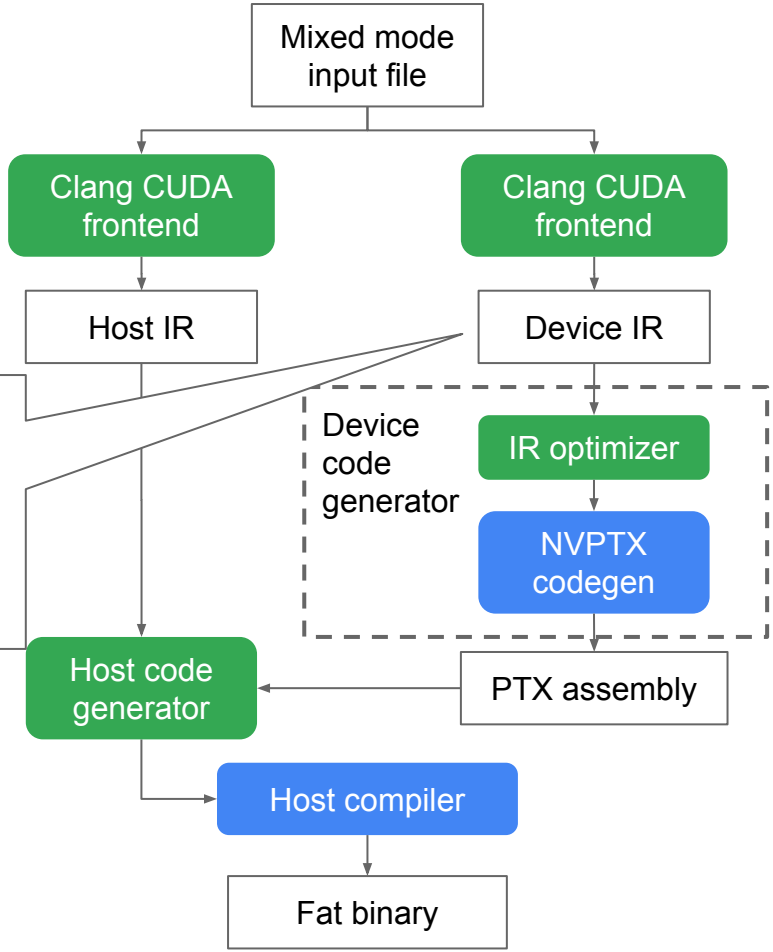
Downsides:

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



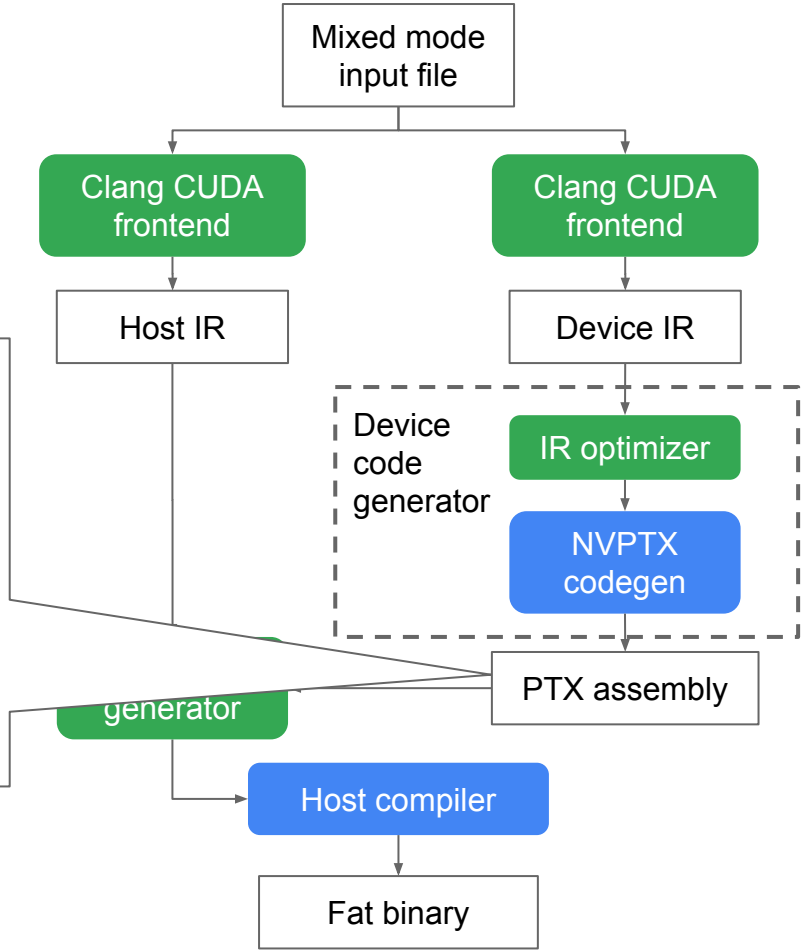
Dual-Mode Compilation

```
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
}
```



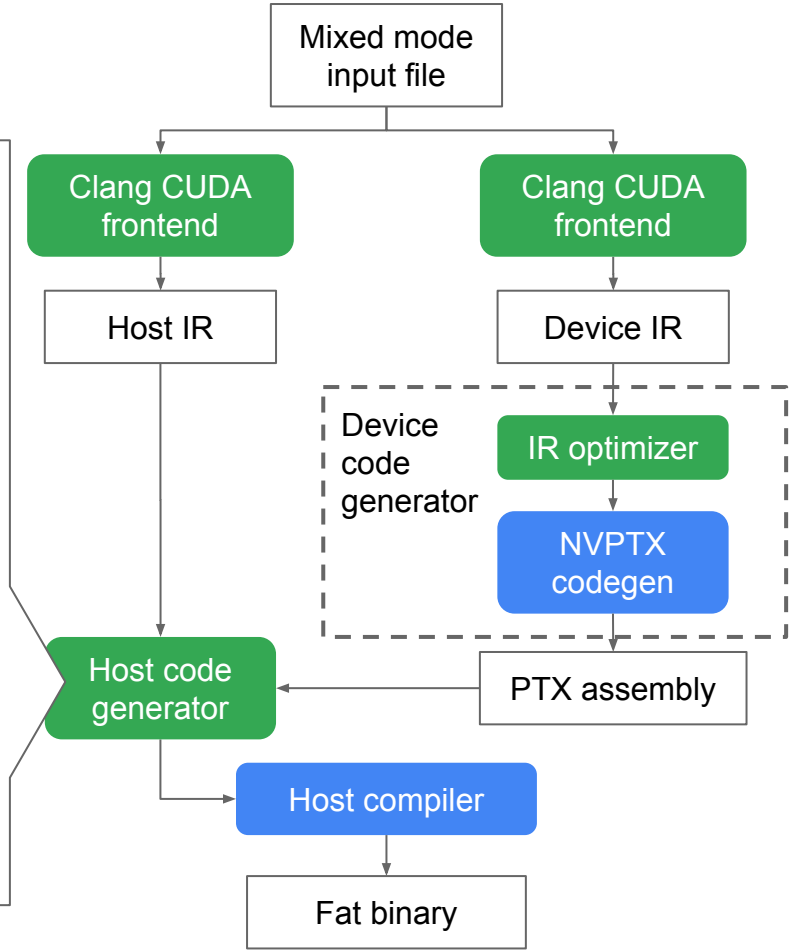
Dual-Mode Compilation

```
.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

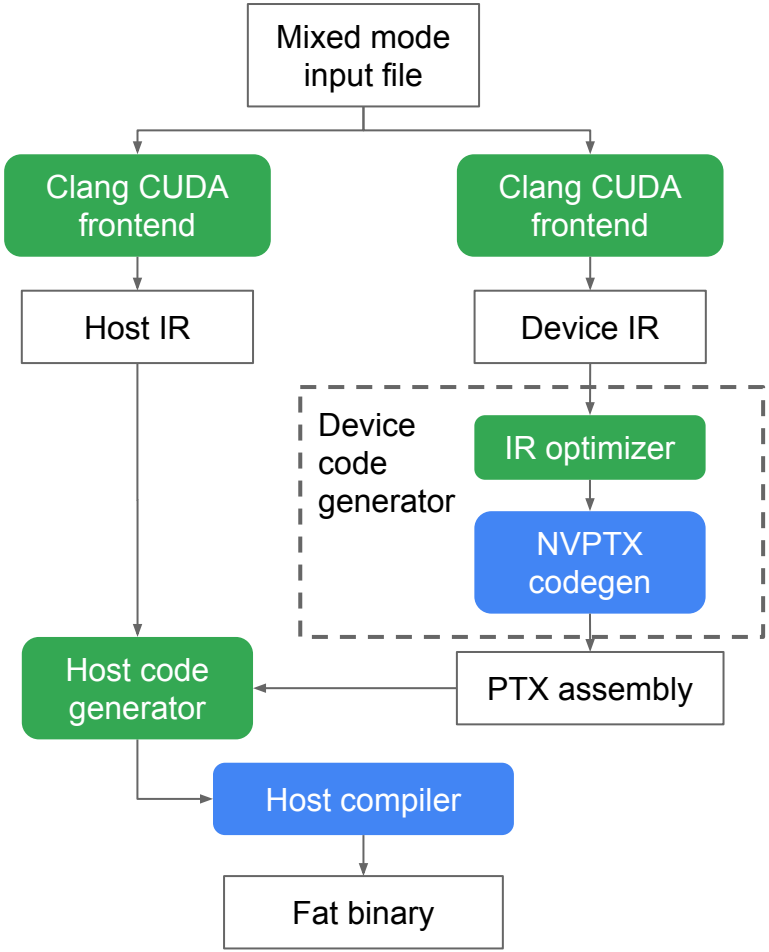
- Written for nvcc which uses separate compilation mode.
- Provides multiple views of CUDA symbols depending on where nvcc includes them from in its pipeline.
- None of the views provide function implementations with __device__ attribute.

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.

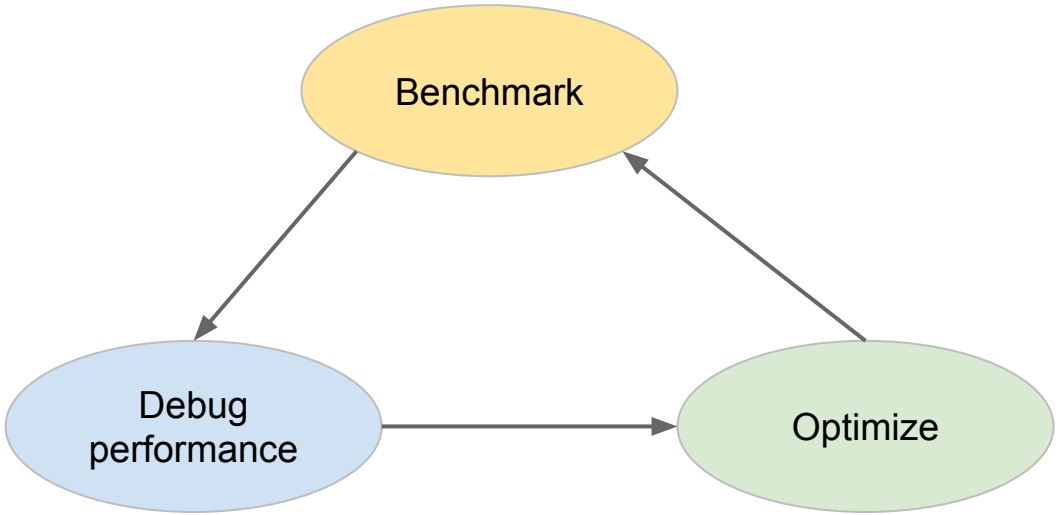
Conclusions

Google

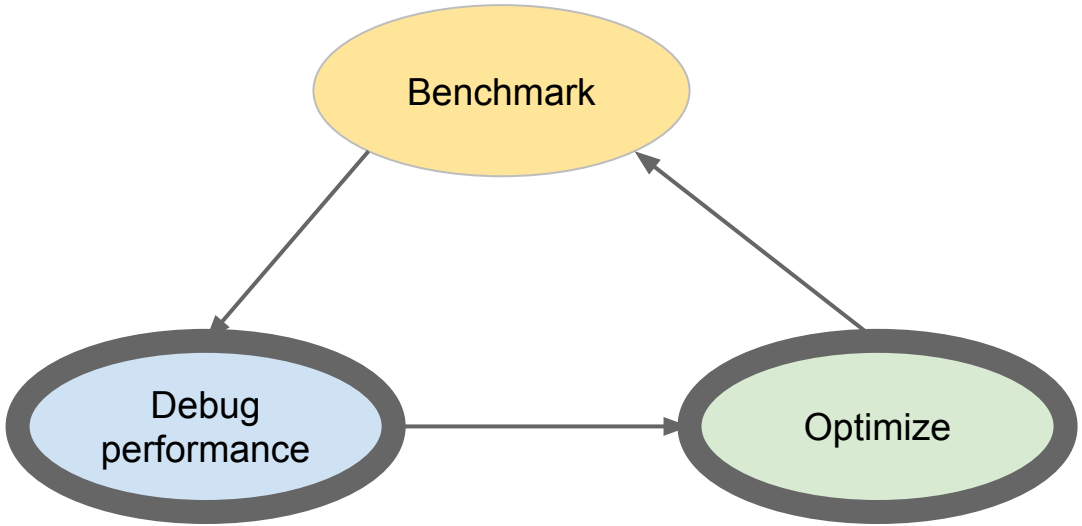
Optimizing gpucc (Part 1)

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

How we achieved good performance



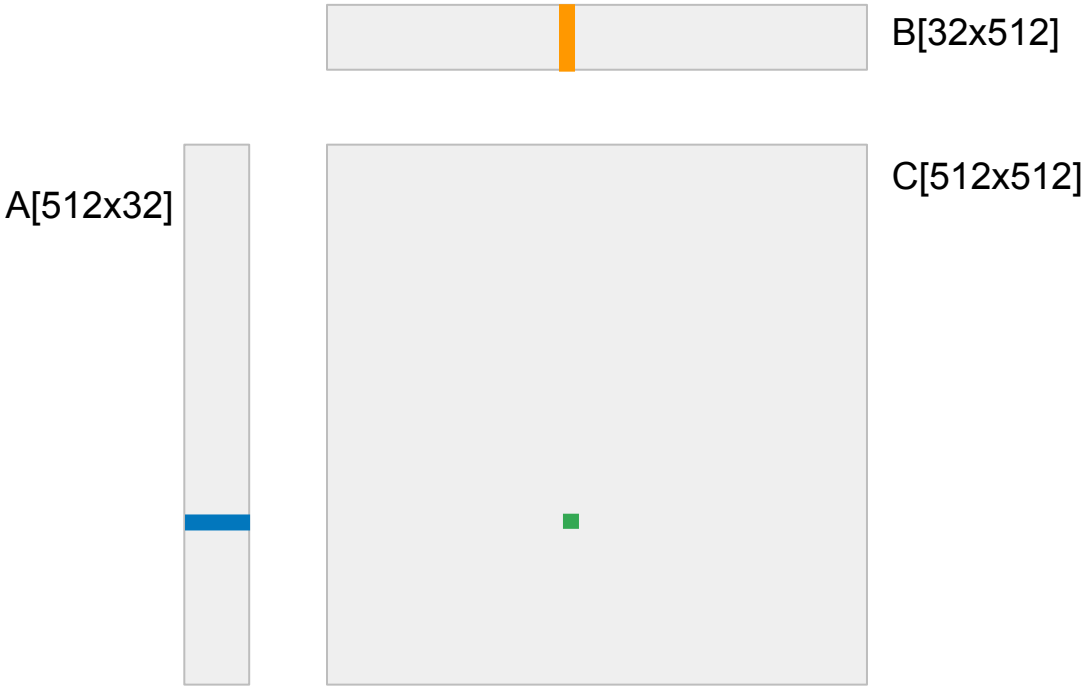
How we achieved good performance



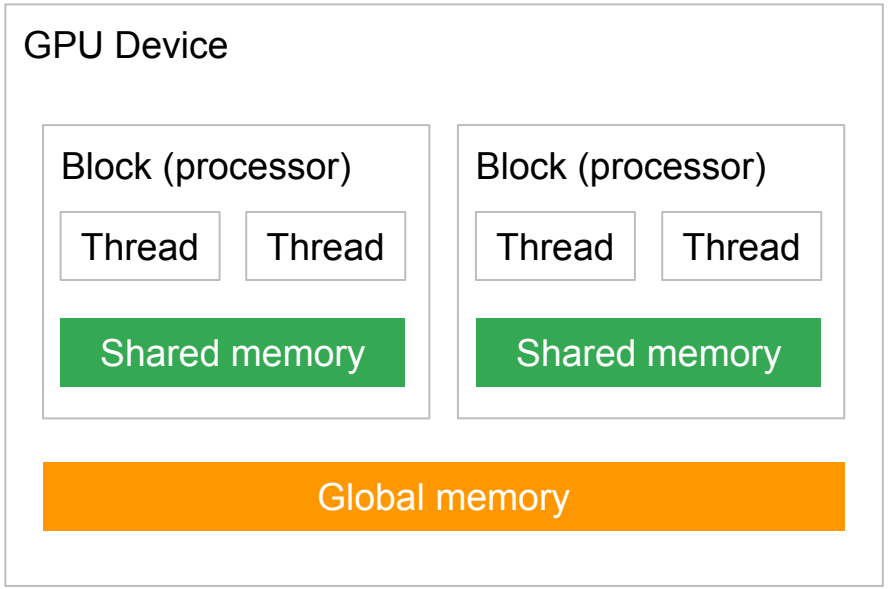
Debug gpucc Performance

- Profiling metrics (`nvprof --metrics`)
  - Instructions executed: load/store, integer, floating point, control flow, etc.
  - Instruction stalls: execution dependency, memory dependency, etc.
  - Occupancy
- Analyzing intermediate files and machine code (`-save-temps`)
  - IR
  - PTX
  - SASS using ptxas and nvdisasm

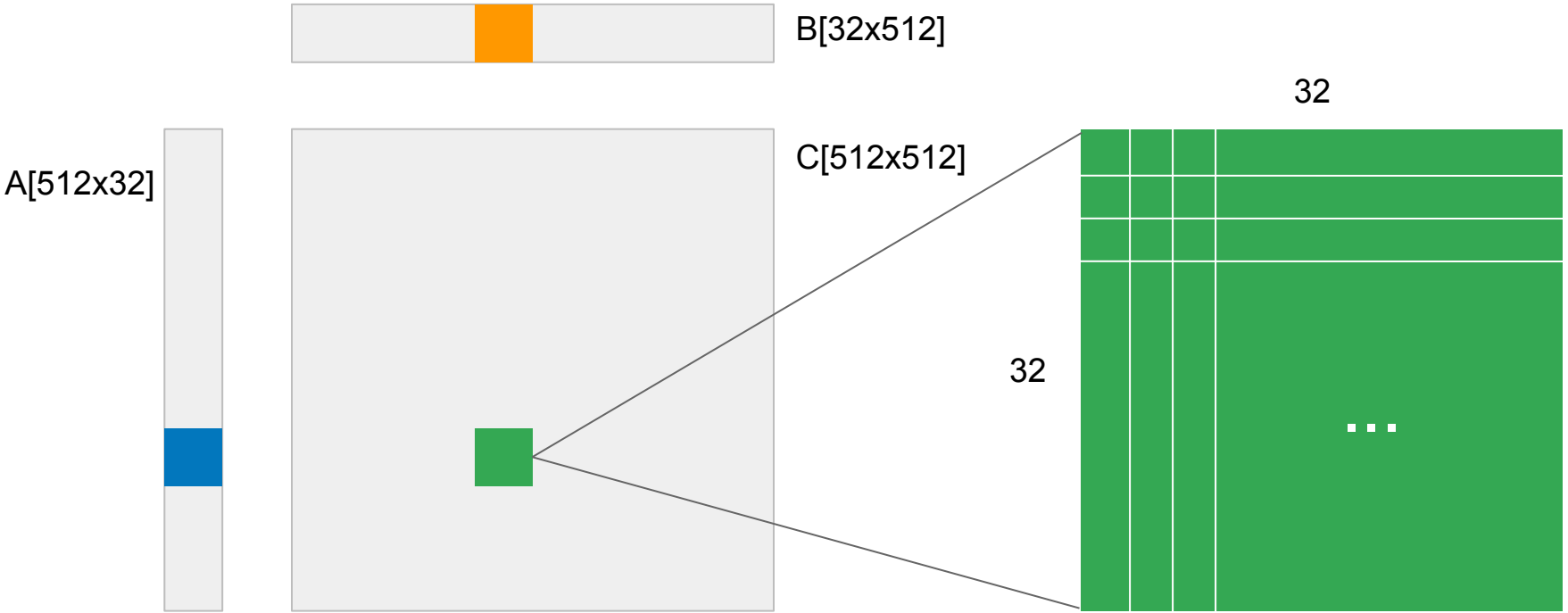
Example: Matrix Multiply



Shared Memory on GPU



Matrix Multiply Using Shared Memory



Matrix Multiply Using Shared Memory

```
__global__ void SharedMultiply(float* a, float* b, float* c) {
  __shared__ float a_tile[kWidth][kWidth], b_tile[kWidth][kWidth];
  int row = blockIdx.y * blockDim.y + threadIdx.y;
  int col = blockIdx.x * blockDim.x + threadIdx.x;
  float sum = 0.0f;
  a_tile[threadIdx.y][threadIdx.x] = a[row * kWidth + threadIdx.x];
  b_tile[threadIdx.y][threadIdx.x] = b[threadIdx.y * kMatrixSize + col];
  __syncthreads();
#pragma unroll 1
  for (int i = 0; i < kWidth; i++) {
    sum += a_tile[threadIdx.y][i] * b_tile[i][threadIdx.x];
  }
  c[row * kMatrixSize + col] = sum;
}
```

Copied and slightly modified from http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/

Full source code can be found at https://gist.github.com/wujingyue/72815bce202841753cbf

Performance Comparison with nvcc

```
$ nvcc matmul.cu -o matmul-nvcc
$ nvprof ./matmul-nvcc

67.106us  SharedMultiply(float*, float*, float*)

$ clang++ matmul.cu -o matmul-gpucc
$ nvprof ./matmul-gpucc

91.042us  SharedMultiply(float*, float*, float*)
```

Profiling Comparison with nvcc

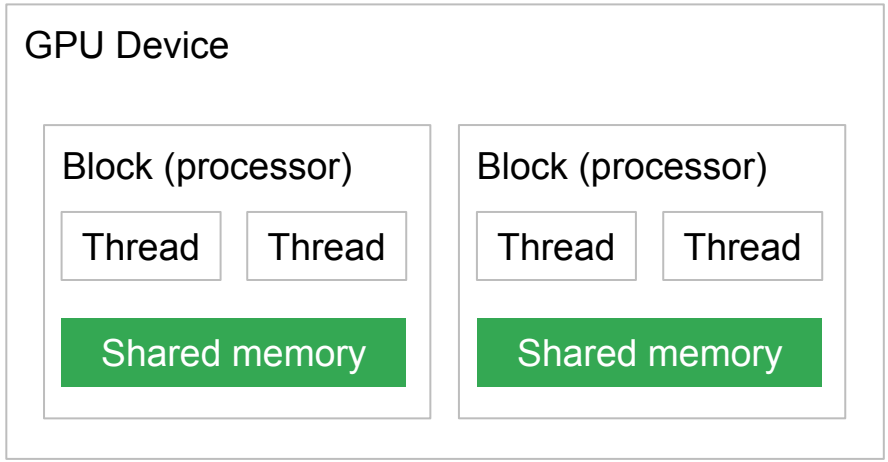
```
$ nvprof --metrics ldst_executed,inst_integer,inst_fp_32,inst_control ./matmul-nvcc

ldst_executed   Executed Load/Store Instructions     565248
inst_integer    Integer Instructions                37486592
inst_fp_32      FP Instructions(Single)              8388608
inst_control    Control-Flow Instructions            8388608

$ nvprof --metrics ldst_executed,inst_integer,inst_fp_32,inst_control ./matmul-gpucc

ldst_executed   Executed Load/Store Instructions     565248
inst_integer    Integer Instructions                82051072
inst_fp_32      FP Instructions(Single)              8388608
inst_control    Control-Flow Instructions            8388608
```

PTX Assembly

```
// matmul-nvcc.ptx

BB0_1:
      .pragma "nounroll";
      ld.shared.f32    %f6, [%rd32];
      ld.shared.f32    %f7, [%rd31];
      fma.rn.f32       %f8, %f7, %f6, %f8;
      add.s64    %rd32, %rd32, 128;
      add.s64    %rd31, %rd31, 4;
      add.s32    %r18, %r18, 1;
      setp.ne.s32      %p1, %r18, 0;
      @%p1 bra   BB0_1;
```

```
// matmul-gpucc.ptx

LBB1_1:
      .pragma "nounroll";
      add.s64    %rd28, %rd2, %rd35;
      cvta.shared.u64 %rd29, %rd28;
      ld.f32     %f4, [%rd29];
      cvta.shared.u64 %rd30, %rd36;
      ld.f32     %f5, [%rd30];
      fma.rn.f32       %f6, %f4, %f5, %f6;
      add.s64    %rd36, %rd36, 128;
      add.s64    %rd35, %rd35, 4;
      cvt.u32.u64      %r15, %rd35;
      setp.eq.s32      %p1, %r15, 128;
      @%p1 bra   LBB1_2;
      bra.uni    LBB1_1;
```

Specific vs Generic Memory Access



- Specific
  - `ld.shared/st.shared`
- Generic
  - `ld/st`
  - Overhead of checking
  - Overhead of conversion
  - Alias analysis suffers

PTX Assembly

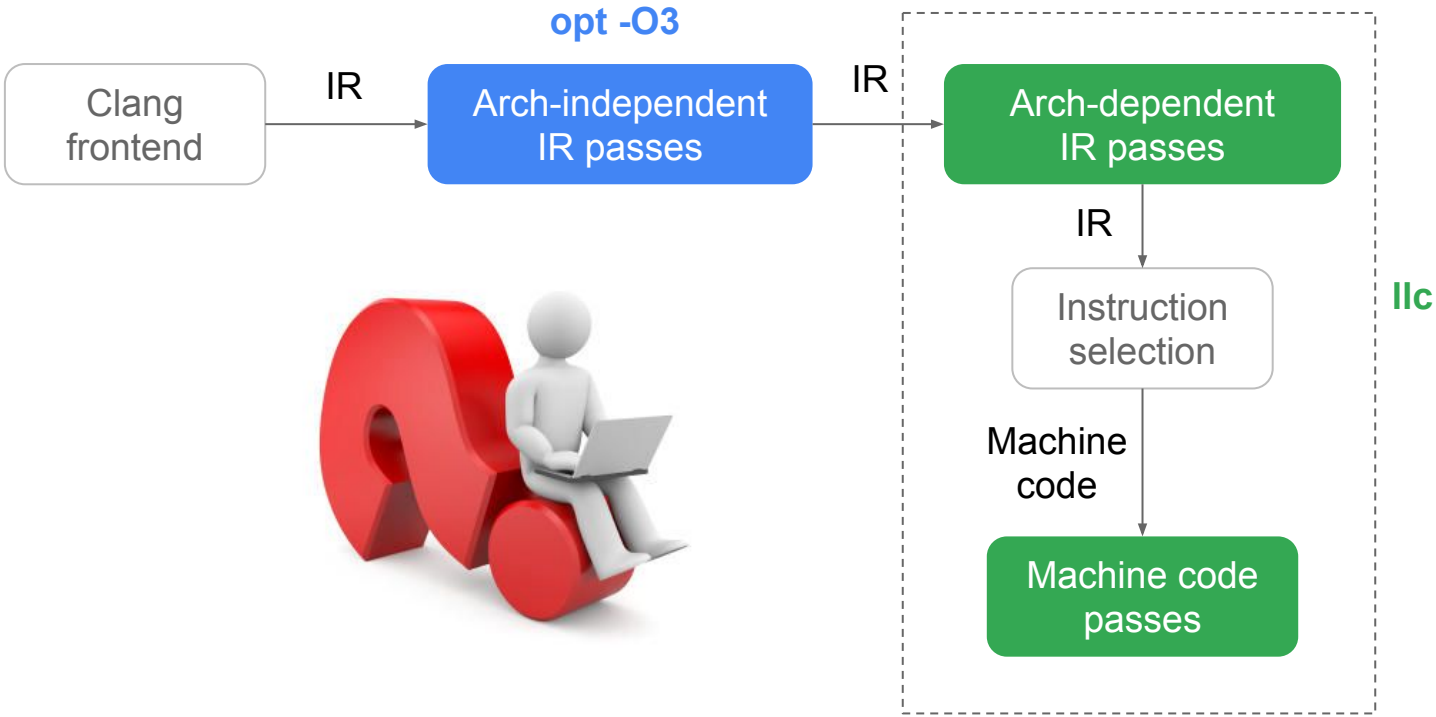
```
// matmul-nvcc.ptx

BB0_1:
      .pragma "nounroll";
      ld.shared.f32    %f6, [%rd32];
      ld.shared.f32    %f7, [%rd31];
      fma.rn.f32       %f8, %f7, %f6, %f8;
      add.s64    %rd32, %rd32, 128;
      add.s64    %rd31, %rd31, 4;
      add.s32    %r18, %r18, 1;
      setp.ne.s32      %p1, %r18, 0;
      @%p1 bra   BB0_1;
```

```
// matmul-gpucc.ptx

LBB1_1:
      .pragma "nounroll";
      add.s64    %rd28, %rd2, %rd35;
      cvta.shared.u64 %rd29, %rd28;
      ld.f32     %f4, [%rd29];
      cvta.shared.u64 %rd30, %rd36;
      ld.f32     %f5, [%rd30];
      fma.rn.f32       %f6, %f4, %f5, %f6;
      add.s64    %rd36, %rd36, 128;
      add.s64    %rd35, %rd35, 4;
      cvt.u32.u64      %r15, %rd35;
      setp.eq.s32      %p1, %r15, 128;
      @%p1 bra   LBB1_2;
      bra.uni    LBB1_1;
```

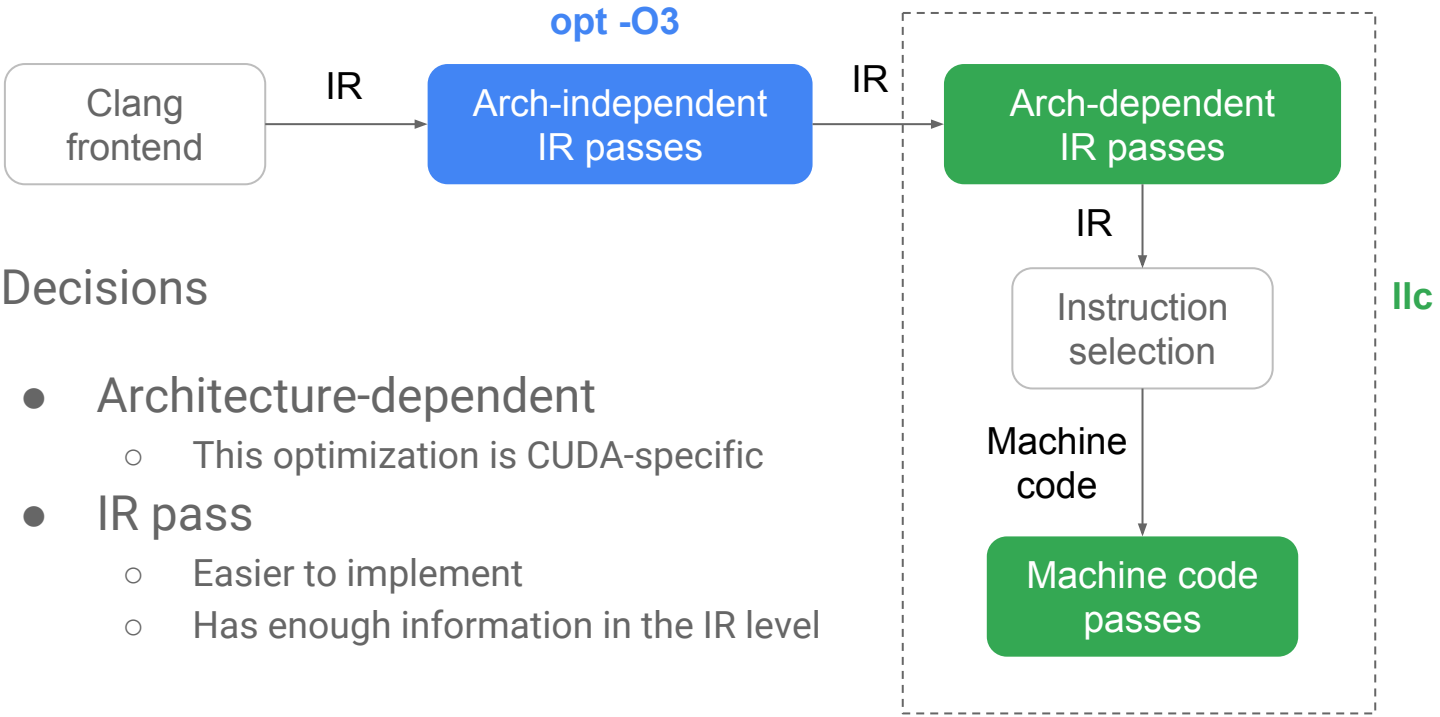
LLVM IR for a_tile[threadIdx.y][i]

```
@_ZZ14SharedMultiplyPfS_S_E6a_tile = internal unnamed_addr addrspace(3) global [32 x
[32 x float]] zeroinitializer, align 4

// float* p = &a_tile[threadIdx.y][i]
%arrayidx32 = getelementptr inbounds [32 x [32 x float]],
    addrspacecast (
        [32 x [32 x float]] addrspace(3)* @_ZZ14SharedMultiplyPfS_S_E6a_tile
        to [32 x [32 x float]]*),
    i64 0, i64 %idxprom14, i64 %idxprom28
// *p
%14 = load float, float* %arrayidx32, align 4, !tbaa !7
```

gpucc Optimization Pipeline



gpucc Optimization Pipeline



Decisions

- Architecture-dependent
  - This optimization is CUDA-specific
- IR pass
  - Easier to implement
  - Has enough information in the IR level

Implement Address Space Inference

```
// lib/Target/NVPTX/NVPTXInferAddressSpaces.cpp
class NVPTXInferAddressSpaces: public FunctionPass {
public:
  static char ID;
  NVPTXInferAddressSpaces() : FunctionPass(ID) {}
  bool runOnFunction(Function &F) override;
};

INITIALIZE_PASS(NVPTXInferAddressSpaces,
                "nvptx-infer-addrspace",
                "Infer address spaces", false, false)

FunctionPass *llvm::createNVPTXInferAddressSpacesPass()
{
  return new NVPTXInferAddressSpaces();
}
```

```
// lib/Target/NVPTX/NVPTXTargetMachine.cpp
extern "C" void LLVMInitializeNVPTXTarget()
{
  ...
  initializeNVPTXInferAddressSpacesPass(PR);
  ...
}

void NVPTXPassConfig::addIRPasses() {
  ...
  addPass(
      createNVPTXInferAddressSpacesPass());
  ...
}
```

- http://llvm.org/docs/WritingAnLLVMPass.html
- Learn from other passes under llvm/lib/Transforms/

Performance Comparison with nvcc Again

```
$ nvcc matmul.cu -o matmul-nvcc
$ nvprof ./matmul-nvcc

67.106us  SharedMultiply(float*, float*, float*)

$ clang++ matmul.cu -o matmul-gpucc
$ nvprof ./matmul-gpucc

68.273us  SharedMultiply(float*, float*, float*)
```

Profiling Comparison with nvcc

```
$ nvprof --metrics ldst_executed,inst_integer,inst_fp_32,inst_control ./matmul-nvcc

ldst_executed   Executed Load/Store Instructions     565248
inst_integer    Integer Instructions               37486592
inst_fp_32      FP Instructions(Single)             8388608
inst_control    Control-Flow Instructions           8388608

$ nvprof --metrics ldst_executed,inst_integer,inst_fp_32,inst_control ./matmul-gpucc

ldst_executed   Executed Load/Store Instructions     565248
inst_integer    Integer Instructions               37486592
inst_fp_32      FP Instructions(Single)             8388608
inst_control    Control-Flow Instructions           8388608
```

Summary of This Session

- How to debug gpucc performance
- How to add new optimizations
- More optimizations
  - Straight-line strength reduction
  - Bypassing 64-bit divides
  - Speculative execution
  - ...
  - http://bit.ly/llvm-cuda and tomorrow's talk (Session 3: GPU)

Google

Optimizing gpucc (Part 2)

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

Example
"I only need the high-bits of a multiplication"

Where does this arise?

- Input program*

```
__global__ void divide_by_5(unsigned int* x, unsigned int* y) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  y[i] = x[i] / 5;
}
```

- Generated PTX:

```
mul.lo.s64   %rd8, x[i], 3435973837;
shr.u64      %rd9, %rd8, 34;
st.u32       y[i], %rd9;
```


* Simplified from actual kernel

Why?

- Multiply with magic number instead of dividing by integer
  - From Hacker's Delight

$$
\begin{aligned}
\left\lfloor \frac{n}{5} \right\rfloor &= \left\lfloor \frac{n}{5} \times \frac{2^k}{2^k} \right\rfloor \\
&= \left\lfloor m \times \frac{n}{2^k} \right\rfloor \\
&= \left\lfloor \frac{2^k + e}{5} \times \frac{n}{2^k} \right\rfloor \\
&= \left\lfloor \frac{n}{5} + \frac{e}{5} \times \frac{n}{2^k} \right\rfloor
\end{aligned}
$$

Why?

- Multiply with magic number instead of dividing by integer
  - From Hacker's Delight

$$\left\lfloor \frac{n}{5} \right\rfloor = \left\lfloor \frac{n}{5} + \frac{e}{5} \times \frac{n}{2^k} \right\rfloor$$

$$k = 32 + 2$$

$$e = 5 - 2^k \mod 5 = 1$$

$$\frac{e}{5} \times \frac{n}{2^k} = \frac{1}{5} \times \frac{n}{2^k} < \frac{1}{5} \times \frac{2^{32}}{2^{34}} < \frac{1}{5}$$

$$m = \left\lceil \frac{2^k}{5} \right\rceil = 3435973837$$

But why 64-bit mul?

- Multiply by 64-bits and shift right by 34
- Why not: mul hi & shift by 2?

PTX ISA

```
// extended-precision multiply: [r3,r2,r1,r0] = [r5,r4] * [r7,r6]
[ ... ]
mul.hi.u32 r1,r4,r6;         // r1=(r4*r6).[63:32], no carry-out
[ ... ]
```

Finding out what the backend is doing

1. Compile with -v (lots of output)

```
$ ./bin/clang++ div.cu -o div -L<SDK>/lib64
--cuda-gpu-arch=sm_35 -lcudart_static -
lcuda -ldl -lrt -pthread -v
```

2. Compile for device only

```
$ ./bin/clang++ div.cu -o div.o --cuda-gpu-
arch=sm_35 --cuda-device-only -c
```

Finding out what the backend is doing

$ <cmd> -mllvm -help-hidden | grep isel

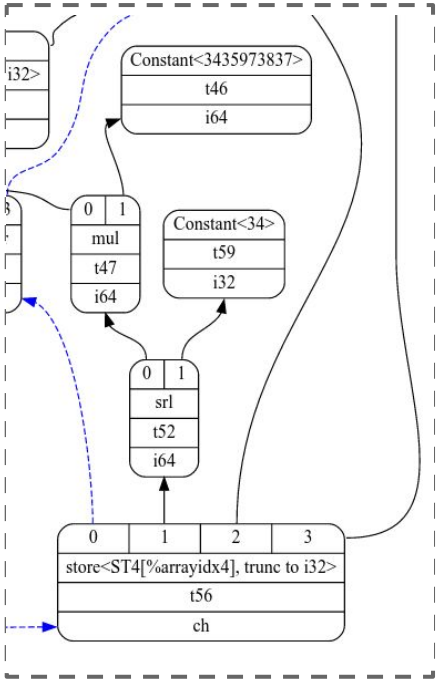
```
  -print-isel-input    - Print LLVM IR input to isel pass
  -view-isel-dags      - Pop up a window to show isel dags
                         as they are selected
```

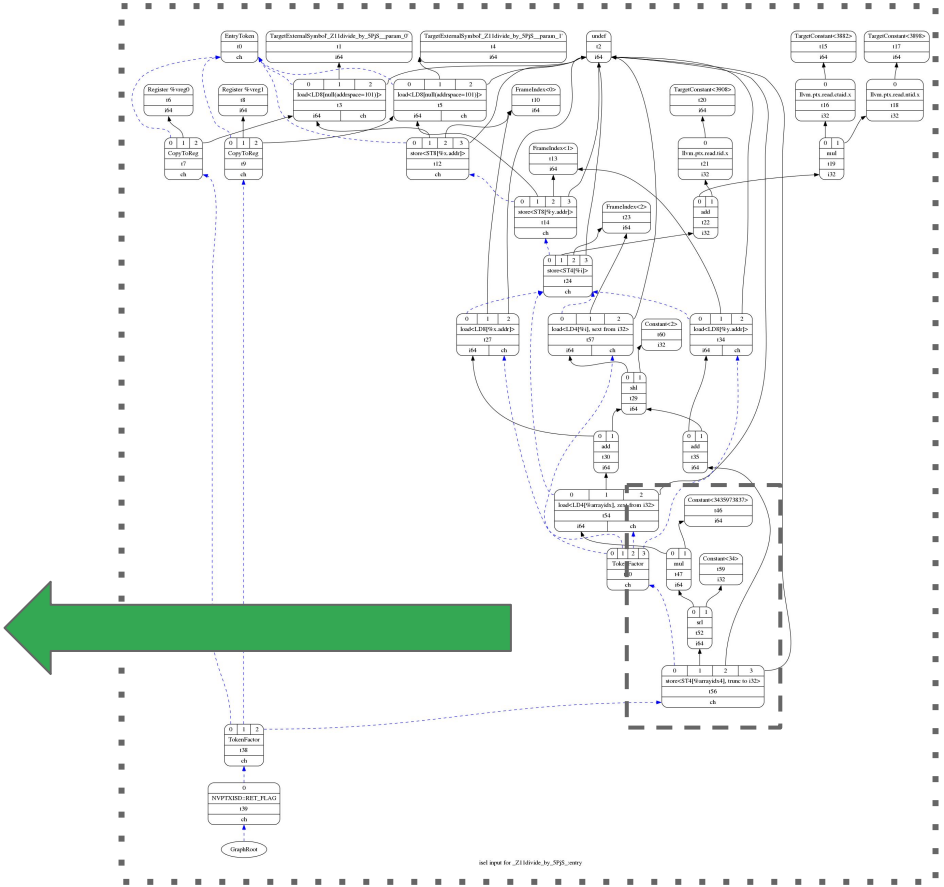
Print ISEL input

```
__global__ void divide_by_5(
 unsigned int* x, unsigned int* y) {

  int i = blockIdx.x * blockDim.x +
          threadIdx.x;


   y[i] = x[i] / 5;
}
```

```
define void @_Z11divide_by_5PjS_(i32* %x, i32* %y) #2 {
entry:
  %x.addr = alloca i32*, align 8
  %y.addr = alloca i32*, align 8
  %i = alloca i32, align 4
  store i32* %x, i32** %x.addr, align 8
  store i32* %y, i32** %y.addr, align 8
  %0 = call i32 @llvm.ptx.read.ctaid.x() #4
  %1 = call i32 @llvm.ptx.read.ntid.x() #4
  %mul = mul i32 %0, %1
  %2 = call i32 @llvm.ptx.read.tid.x() #4
  %add = add i32 %mul, %2
  store i32 %add, i32* %i, align 4
  %3 = load i32, i32* %i, align 4
  %idxprom = sext i32 %3 to i64
  %4 = load i32*, i32** %x.addr, align 8
  %arrayidx = getelementptr inbounds i32, i32* %4, i64 %idxprom
  %5 = load i32, i32* %arrayidx, align 4
  %div = udiv i32 %5, 5
  %6 = load i32, i32* %i, align 4
  %idxprom3 = sext i32 %6 to i64
  %7 = load i32*, i32** %y.addr, align 8
  %arrayidx4 = getelementptr inbounds i32, i32* %7, i64 %idxprom3
  store i32 %div, i32* %arrayidx4, align 4
  ret void
}
```

ISEL DAGs





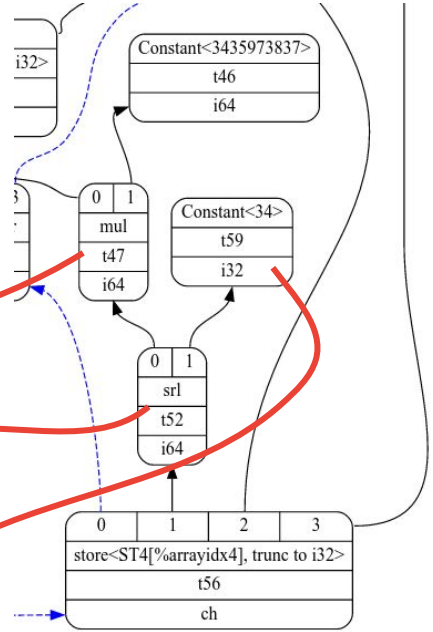
NVPTX backend

- NVPTX : PTX backend originally contributed by NVIDIA
- NVPTXInstrInfo.td : TableGen file with instruction information

```
defm SRL    : RSHIFT_FORMAT<"shr.u", srl>;
defm MULTHS : I3<"mul.hi.s", mulhs>;
multiclass I3<string OpcStr, SDNode OpNode> {
  [...]
  def i32ri : NVPTXInst<(outs Int32Regs:$dst), (ins Int32Regs:$a, i32imm:$b),
                        !strconcat(OpcStr, "32 \t$dst, $a, $b;"),
                        [(set Int32Regs:$dst, (OpNode Int32Regs:$a, imm:$b))]>;
  [...]
}
```

Aha, add special matcher

- No magic: look and match
- One (very small) peephole optimization




```
def : Pat<(srl (mul Int64Regs:$a, (i64 imm:$b)), (i32 34))
          (SRLi32ri (MULTHUi32ri Int32Regs:$a, imm:$b), 2)>;
```

Not so fast ...

```
Anonymous_516: (SRLi32ri:i32 (MULTHUi32ri:i32 Int32Regs:
<empty>:$a, (imm:i32):$b), 2:i32)
Included from cgo-tut/llvm/lib/Target/NVPTX/NVPTX.td:19:
cgo-tut/llvm/lib/Target/NVPTX/NVPTXInstrInfo.td:1091:1: error: In
anonymous_516: Type inference contradiction found, merging 'i64'
into 'i32'
def : Pat<(srl (mul Int64Regs:$a, (i64 imm:$b)), (i32 34)),
^
```

```
def : Pat<(srl (mul Int64Regs:$a, (i64 imm:$b)), (i32 34)),
          (SRLi32ri (MULTHUi32ri Int32Regs:$a, imm:$b), 2)>;
```

Debug output to the rescue

$ <cmd> -mllvm -debug -mllvm -print-after-all

- Find the last udiv:

```
Combining: t33: i32 = udiv t31, Constant:i32<5>
... into: t44: i32 = srl t42, Constant:i32<2>
[...]
Combining: t42: i32 = mulhu t31, Constant:i32<-858993459>
... into: t50: i32 = truncate t49
```

3435973837

Trace back change

- Search through LLVM files for "Combining:"
  - Leads to "lib/CodeGen/SelectionDAG/DAGCombiner.cpp"
- Change happens in mulhu, so look at visitMULHU:

```
SDValue DAGCombiner::visitMULHU(SDNode *N) {
 [...]
 // If the type twice as wide is legal, transform the mulhu to a wider multiply
 // plus a shift.
 if (VT.isSimple() && !VT.isVector()) {
   [...]
 }

 return SDValue();
}
```

Trace back change

- Search through LLVM files for "Combining:"
  - Leads to "lib/CodeGen/SelectionDAG/DAGCombiner.cpp"
- Change happens in mulhu, so look at visitMULHU:

```
SDValue DAGCombiner::visitMULHU(SDNode *N) {
 [...]
 // If the type twice as wide is legal and not targeting NVPTX, transform the
 // mulhu to a wider multiply plus a shift.
 Triple TT = DAG.getTarget().getTargetTriple();
 if (!TT.isNVPTX() && VT.isSimple() && !VT.isVector()) {
   [...]
 }

 return SDValue();
}
```

Difference in resultant output

old.s

```
+-- 25 lines: // Generated by LLVM NVPTX Back-End
      .param .u64 _Z11divide_by_5PjS__param_1
)
{
      .local .align 8 .b8     __local_depot1[24];
      .reg .b64         %SP;
      .reg .b64         %SPL;
      .reg .s32         %r<6>;
      .reg .s64         %rd<12>;

+--  15 lines
      add.s64         %rd6, %rd4, %rd5;
      ld.u32  %rd7, [%rd6];
      mul.lo.s64      %rd8, %rd7, 3435973837;
      shr.u64         %rd9, %rd8, 34;
      ld.u64  %rd10, [%SP+8];
      add.s64         %rd11, %rd10, %rd5;
      st.u32  [%rd11], %rd9;
      ret;
}
```

new.s

```
+-- 25 lines: // Generated by LLVM NVPTX Back-End
      .param .u64 _Z11divide_by_5PjS__param_1
)
{
      .local .align 8 .b8     __local_depot1[24];
      .reg .b64         %SP;
      .reg .b64         %SPL;
      .reg .s32         %r<9>;
      .reg .s64         %rd<9>;

+--  15 lines
      add.s64         %rd6, %rd4, %rd5;
      ld.u32  %r6, [%rd6];
      mul.hi.u32      %r7, %r6, -858993459;
      shr.u32         %r8, %r7, 2;
      ld.u64  %rd7, [%SP+8];
      add.s64         %rd8, %rd7, %rd5;
      st.u32  [%rd8], %r8;
      ret;
}
```

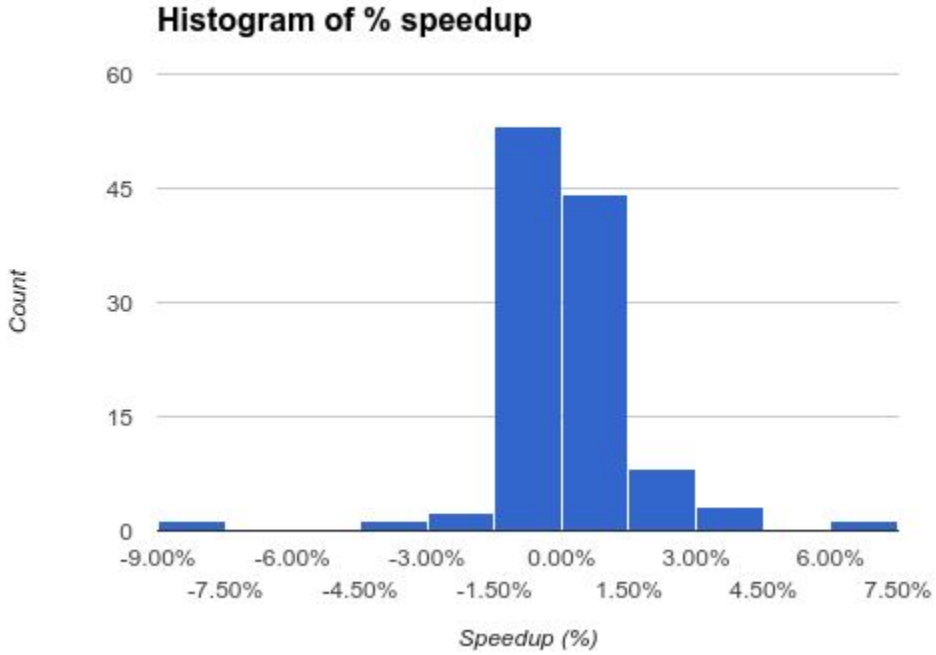
But is it needed?

- Type widening for multhu introduced in http://llvm.org/viewvc/llvm-project?view=revision&revision=121696
  (Mon Dec 13 00:39:01 PST 2010)

Intel Optimization Reference Manual, Assembly/Compiler Coding Rule 21:

Favor generating code using imm8 or imm32 values instead of imm16 values. If imm16 is needed, load equivalent imm32 into a register and use the word value in the register instead.

Does it help? (SHOC)



"Division is really slow! But I know some of my divisors are powers of two ..."

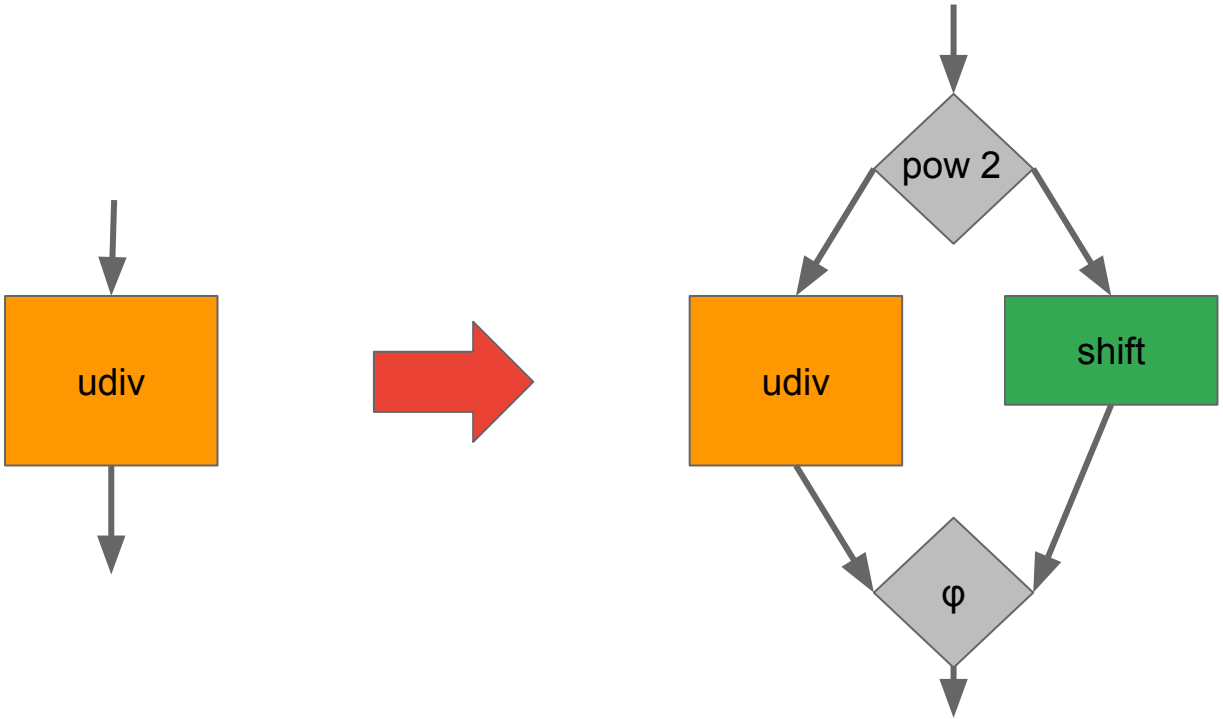
Example input

- Input kernel

```
__global__ void divide_by_a(int a, unsigned int* x, unsigned int* y) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    y[i] = x[i] / a;
}
```

- 'a' is not constant so regular power of 2 transformation won't apply
- BUT division is so much slower than compare ...

What should the pass do?



Code for pass: https://gist.github.com/jpienaar/e4857c8460db3f2d4805

Fast path for avoiding divides at runtime

- Common request == utility exists!
- Existing transform utility to bypass slow division:

BypassSlowDivision.cpp
// This file contains an optimization for div and rem on architectures that
// execute short instructions significantly faster than longer instructions.
// For example, on Intel Atom 32-bit divides are slow enough that during
// runtime it is profitable to check the value of the operands, and if they are
// positive and less than 256 use an unsigned 8-bit divide.

- For GPUs it might pay to be even more aggressive!
  - Or not - profiling your app will indicate!
  - Is pass needed?

Add a new pass

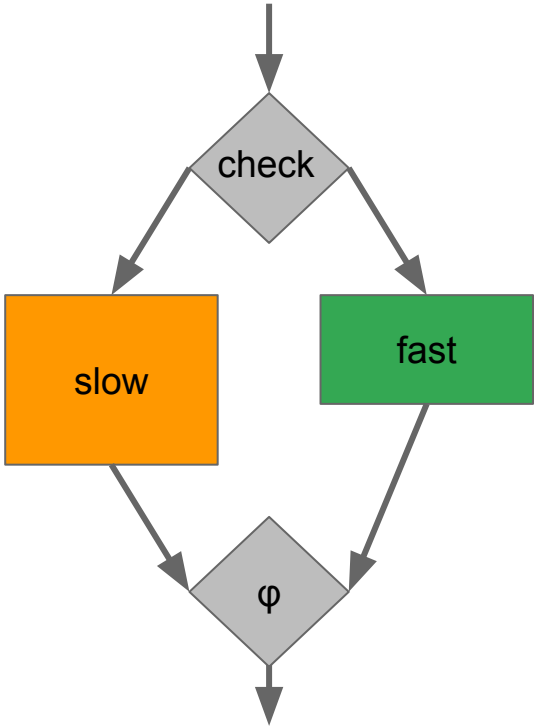
```
struct NVPTXBypassDivPowerTwo : public FunctionPass {
  static char ID;

  NVPTXBypassDivPowerTwo() : FunctionPass(ID) {}

  bool runOnFunction(Function &F) override;

  const char *getPassName() const override {
    return "Bypass divide power two";
  }
};
```

```
void NVPTXPassConfig::addCodeGenPrepare() {
  addPass(createBypassDivPowerTwo());
  TargetPassConfig::addCodeGenPrepare();
}
```

```
namespace llvm {
void initializeNVPTXBypassDivPowerTwoPass
(PassRegistry&);
}

INITIALIZE_PASS(NVPTXBypassDivPowerTwo,
    "Nvptx-bypass-div-power-two",
    "Insert runtime check to use shift instead of div",
    false, false)

FunctionPass *llvm::createBypassDivPowerTwo() {
  return new NVPTXBypassDivPowerTwo();
}
```

Working of the pass

- runOnFunction(Function &F)
  - For every basic block
    - For every instruction in the basic block
      - Is this a binary operation that performs division?
        a. Split original block in 2 (before divide, after divide)
        b. Add 2 new basic blocks (slow path, fast path)
        c. Insert check for power of 2 (use population counting)
        d. Fast path: shift input trailing-zeros (divisor) left
        e. Slow path: divide input by divisor
        f. After divide: join via phi-node



Difference in resultant output

old.s

```
        .param .u64 _Z11divide_by_aiPjS__param_2) {
+--
        ld.u32  %r71, [%rd6];
        ld.u32  %r82, [%SP+0];









        div.u32         %r94, %r71, %r82;









        ld.u64  %rd8, [%SP+16];

        add.s64         %rd10, %rd8, %rd5;
        st.u32  [%rd10], %r94;
        ret;
}
```

new.s

```
        .param .u64 _Z11divide_by_aiPjS__param_2) {
+--
        ld.u32  %r71, [%rd6];
        ld.u32  %r82, [%SP+0];
        popc.b32        %r12, %r82;
        setp.ne.s32             %p1, %r12, 1;
        @%p1 bra        LBB2_2;
        bra.uni         LBB2_1;
LBB2_1:
        clz.b32         %r13, %r2;
        mov.u32         %r14, 31;
        sub.s32         %r15, %r14, %r13;
        shr.u32         %r3, %r71, %r15;
        mov.u32         %r16, %r3;
        bra.uni         LBB2_3;
LBB2_2:
        div.u32         %r94, %r71, %r82;
        mov.u32         %r16, %r94;
        bra.uni         LBB2_3;
LBB2_3:
        mov.u32         %r5, %r16;
        ld.s32  %rd7, [%SP+24];
        ld.u64  %rd8, [%SP+16];
        shl.b64         %rd9, %rd7, 2;
        add.s64         %rd10, %rd8, %rd9;
        st.u32  [%rd10], %r5;
        ret;
}
```

Google

Google

Contributing to gpucc

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

"We have a performant open-source GPU compiler. Now what?"

Compiler development

- Compilers are tuned to provide good performance on important* applications and benchmarks
- Where important can be 1) benchmarks, 2) widely used kernels, or 3) applications that the compiler developer cares about
- gpucc performs better than nvcc on our internal applications as that was our first target
    - It performs well on open-source benchmarks too!

But it should be the best for applications you care about too!

How to contribute to LLVM

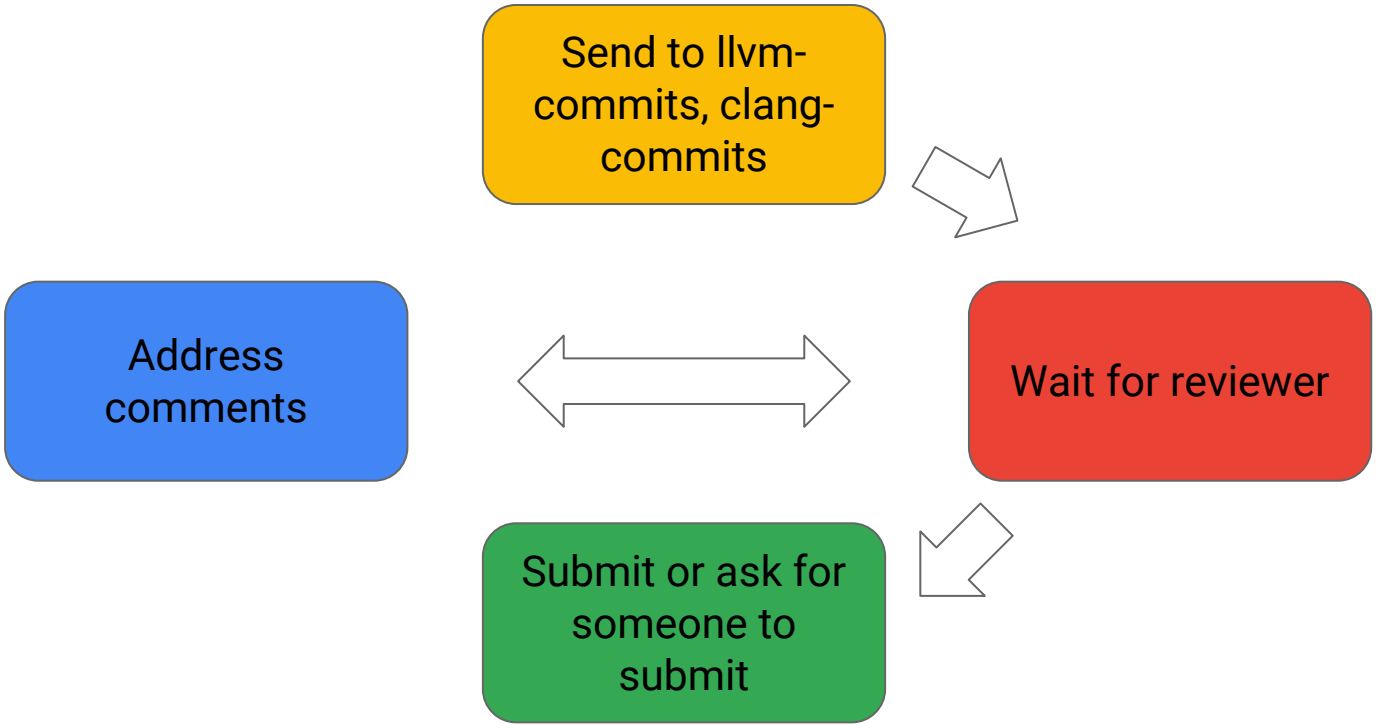
- License
  - Published under the "University of Illinois/NCSA Open Source License" BSD-style
  - Some of them are also published under MIT licence (dual-license)
  - No copyright assignment
- Community
  - Friendly
  - Mainly professional (Apple, Google, ARM, Linaro, Intel, etc)
  - But many individual and academic contributors
  - Usually fast to answer to comments/questions
- Resources
  - LLVM Developer Policy
  - "How to contribute to LLVM, Clang, Etc" talk by Sylvestre Ledru @ FOSDEM 2014

The LLVM/Clang community

	LLVM	Clang
Project Activity	Very High	Very High
Homepage	http://llvm.org/	http://clang.llvm.org/
Project License	NCSA	NCSA
Contributors (All Time)	613 developers	459 developers
Commits (All Time)	134,904 commits	62,633 commits
Initial Commit	over 14 years ago	over 8 years ago
Contributors (Past 12 Months)	336 developers	265 developers
Commits (Past 12 Months)	15,426 commits	5,983 commits

Source: OpenHub

Contributing a patch



Getting help

- Mailing lists:
  - LLVM
    - LLVM-dev
    - LLVM-commits
  - Clang
    - cfe-users
    - cfe-dev
    - cfe-commits
- IRC: irc.oftc.net #llvm

Keep patches small and focussed

Discuss the design on the mailing list, before starting the development

Review patches

Organization of LLVM source code

- $llvm/lib/Target/NVPTX
  NVPTX backend & NVPTX specific optimizations
- $llvm/lib/Transforms/Utils
  General transformation utils
- $llvm/CodeGen/ and $llvm/CodeGen/SelectionDAG
  General code generations and transformations on instruction selection DAG
- $llvm/tools/clang/lib/Sema/SemaCUDA.cpp
  Semantic analysis for CUDA constructs
- $llvm/tools/clang/include/clang/Basic/BuiltinsNVPTX.def,
  $llvm/tools/clang/lib/CodeGen/CGCUDABuiltin.cpp
  CUDA specific builtins

Using out-of-tree

- Use gpucc/Clang/LLVM to create tools & plugins
- Useful resources:
  - "Writing an LLVM Pass"
    http://llvm.org/docs/WritingAnLLVMPass.html
  - "Tutorial: Building, Testing and Debugging a Simple out-of-tree LLVM Pass"
    http://www.llvm.org/devmtg/2015-10/#tutorial1
  - "Samples for using LLVM and Clang as a library"

    http://eli.thegreenplace.net/2014/samples-for-using-llvm-and-clang-as-a-library/

    https://github.com/eliben/llvm-clang-samples/tree/master/src_clang
    http://eli.thegreenplace.net/tag/llvm-clang
  - "LLVM for Grad Students"
    http://adriansampson.net/blog/llvm.html

Action Items

Concrete bugs and feature requests

- Frontend support
  - Texture memory
  - C++14
- NVPTX Backend
  - More intrinsics
- CUDA runtime support
  - Dynamic allocation and deallocation
- Micro-optimizations / peephole

Loops with strided accesses

```
__global__ void kernel(float* output, int numIterations) {
  for (int i = 0; i < numIterations; i += 2) {
    output[i] = output[i+1];
  }
}
```

- Good: loop gets unrolled a few times
- Bad:
  - gpucc fails to fold the offsets from the induction variable into the constant addressing mode
  - Uses 64-bit indexing even where 32-bit would suffice

Automatically generated memset sub-optimal

```
__global__ void kernel(float* output, int N) {
  for (int i = 0; i < N; ++i) {
    output[i] = 0;
  }
}
```

- The PTX output by gcudacc changes that to be 1 byte at a time
  - Idiom recognize as memset
  - Generated memset bad

Unnecessary copies

- Why copy into r16 before copying into r5?

- Caveat: PTXAS is an optimizing assembler. Performs more optimizations than one might expect.

new.s

```
      .param .u64 _Z11divide_by_aiPjS__param_2) {
+-- not important lines
      ld.u32  %r71, [%rd6];
      ld.u32  %r82, [%SP+0];
      popc.b32        %r12, %r82;
      setp.ne.s32           %p1, %r12, 1;
      @%p1 bra        LBB2_2;
      bra.uni         LBB2_1;
LBB2_1:
      clz.b32 %r13, %r2;
      mov.u32               %r14, 31;
      sub.s32     %r15, %r14, %r13;
      shr.u32     %r3, %r71, %r15;
      mov.u32     %r16, %r3;
      bra.uni         LBB2_3;
LBB2_2:
      div.u32       %r94, %r71, %r82;
      mov.u32     %r16, %r94;
      bra.uni         LBB2_3;
LBB2_3:
      mov.u32     %r5, %r16;
      ld.s32  %rd7, [%SP+24];
      ld.u64  %rd8, [%SP+16];
      shl.b64       %rd9, %rd7, 2;
      add.s64       %rd10, %rd8, %rd9;
      st.u32  [%rd10], %r5;
      ret;
}
```

Research opportunities

Past

- nvcc: CUDA → *blackbox* → PTX
- libnvvm: closed-source and works on outdated IR
- NVPTX codegen (IR→PTX) with little optimization

Present: complete open-source pipeline from CUDA to PTX

- Frontend: more language extensions? Other languages?
- Backend: more architectures (e.g., CUDA on AMDGPU)? Target SASS?
- Debugging: better profiling? Static analysis?
- **Optimizer: more optimizations?**

Discover optimization opportunities

What optimizations do you currently have to perform in source code?

CUDA C Best Practices (docs.nvidia.com/cuda/cuda-c-best-practices-guide/)

- Global memory coalescing
- Tune occupancy
- Avoid divergence
- Optimizations across host-device boundary
- Fast math
- ...

Global memory coalescing

```
void copy(float* input, float* output, int n) {
  for (int i = 0; i < n; ++i)
    output[i] = input[i];
}
```

Uncoalesced access

```
__global__ void copy(float* input, float* output, int n) {
  for (int i = threadIdx.x * (n / 32);
       i < (threadIdx.x + 1) * (n / 32); ++i) {
    output[i] = input[i];
  }
}
```

Uncoalesced access

```
__global__ void copy(float* input, float* output, int n) {
  for (int i = threadIdx.x * (n / 32);
       i < (threadIdx.x + 1) * (n / 32); ++i) {
    output[i] = input[i];
  }
}
```

Uncoalesced access

```
__global__ void copy(float* input, float* output, int n) {
  for (int i = threadIdx.x * (n / 32);
       i < (threadIdx.x + 1) * (n / 32); ++i) {
    output[i] = input[i];
  }
}
```

Coalesced access


```
__global__ void copy(float* input, float* output, int n) {
  for (int block = 0; block < n / 32; ++block) {
    int i = block * 32 + threadIdx.x;
    output[i] = input[i];
  }
}
```

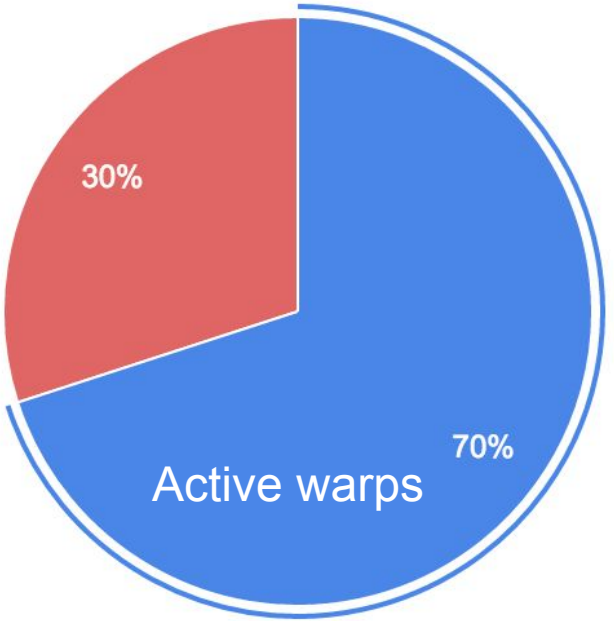
Occupancy

Factors contributing to occupancy (bit.ly/cuda-occupancy-calculator)

- Number of registers each block uses
- Shared memory size
- Block size

Trade off between occupancy and latency

Feedback directed optimizations (FDO)?



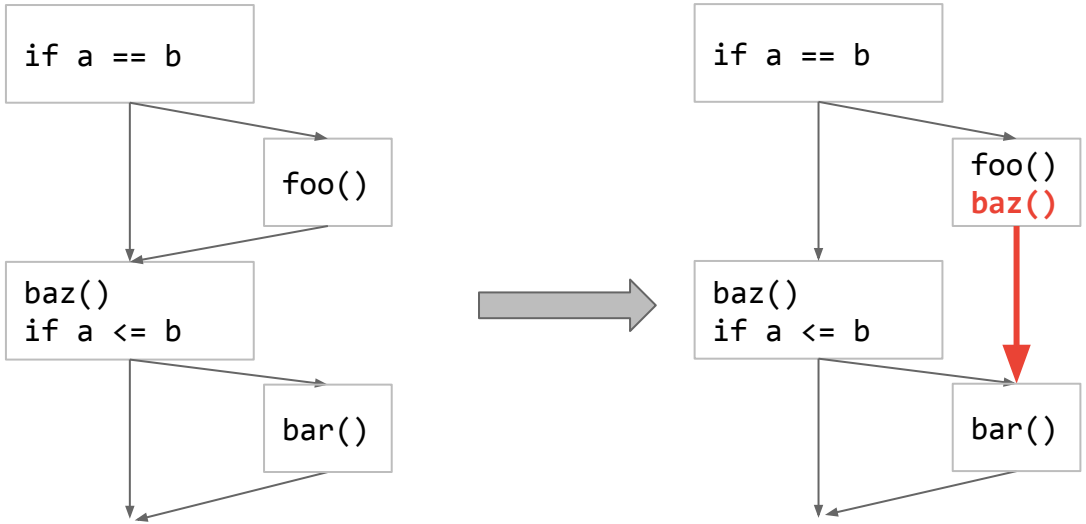
Divergence

```
if (threadIdx.x % 2 == 0) {
  foo();
} else {
  bar();
}
```

CPU: max(foo, bar)

GPU: foo + bar

How divergence affects jump threading



foo + bar + baz

foo + bar + **baz * 2**

Divergence Analysis:
http://bit.ly/divergence-analysis

Optimizations across cross and device boundaries

```
__global__ void kernel(float* input,
                       float* output) {
  ... blockDim.x ...
}

float* input = cudaMemAlloc(...);
float* output = cudaMemAlloc(...);
kernel<<<grid_dim, 128>>>(input, output);
```

Optimizations across cross and device boundaries

```
__global__ void kernel(float* input,
                       float* output) {
  ... blockDim.x ...
}

float* input = cudaMemAlloc(...);
float* output = cudaMemAlloc(...);
kernel<<<grid_dim, 128>>>(input, output);
```

Constant propagation

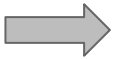
Optimizations across cross and device boundaries

```
__global__ void kernel(float* input,
                       float* output) {
  ... blockDim.x ...
}

float* input = cudaMemAlloc(...);
float* output = cudaMemAlloc(...);
kernel<<<grid_dim, 128>>>(input, output);
```

Constant propagation
Alias analysis

Current workaround: templates and annotations

```
__global__ void kernel(float* input,
                       float* output) {
  ... blockDim.x ...
}

float* input = cudaMemAlloc(...);
float* output = cudaMemAlloc(...);
kernel<<<grid_dim, 128>>>(input, output);
```



```
template <int kBlockDim>
__global__ void kernel(
    float* __restricted__ input,
    float* __restricted__ output) {
  ... kBlockDim ...
}

float* input = cudaMemAlloc(...);
float* output = cudaMemAlloc(...);
kernel<128><<<grid_dim, 128>>(input, output);
```

Constant propagation
Alias analysis

Fast math: div → mul

```
// n = 1024
__global__ void foo(float *input, float *output, int n) {
  for (int i = 0; i < n; ++i)
    output[i] = input[i] / 255.0f;
}
```

	w/o fast math	w/ fast math
PTX	`div.rn.f32`	`div.approx.ftz.f32` **`input[i] * (1.0f / 255.0f)`**
run time	990.72 us (2.6x)	373.92 us

Takeaways from this tutorial

- The missions of gpucc
  - enable compiler research
  - enable industry breakthroughs
  - enable community contributions
- In a good shape but can be improved a lot
- **Use and contribute to gpucc!** (http://bit.ly/llvm-cuda)