# 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"

Google

## 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;
```

Google

* 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
[ ... ]
```

Google

## 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
```

Google

## 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
```

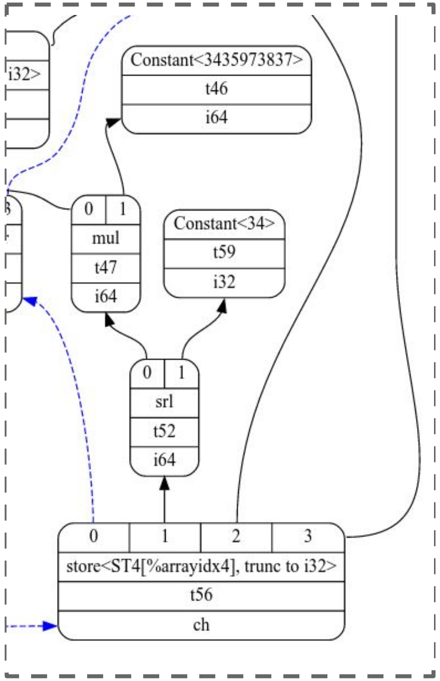
Google

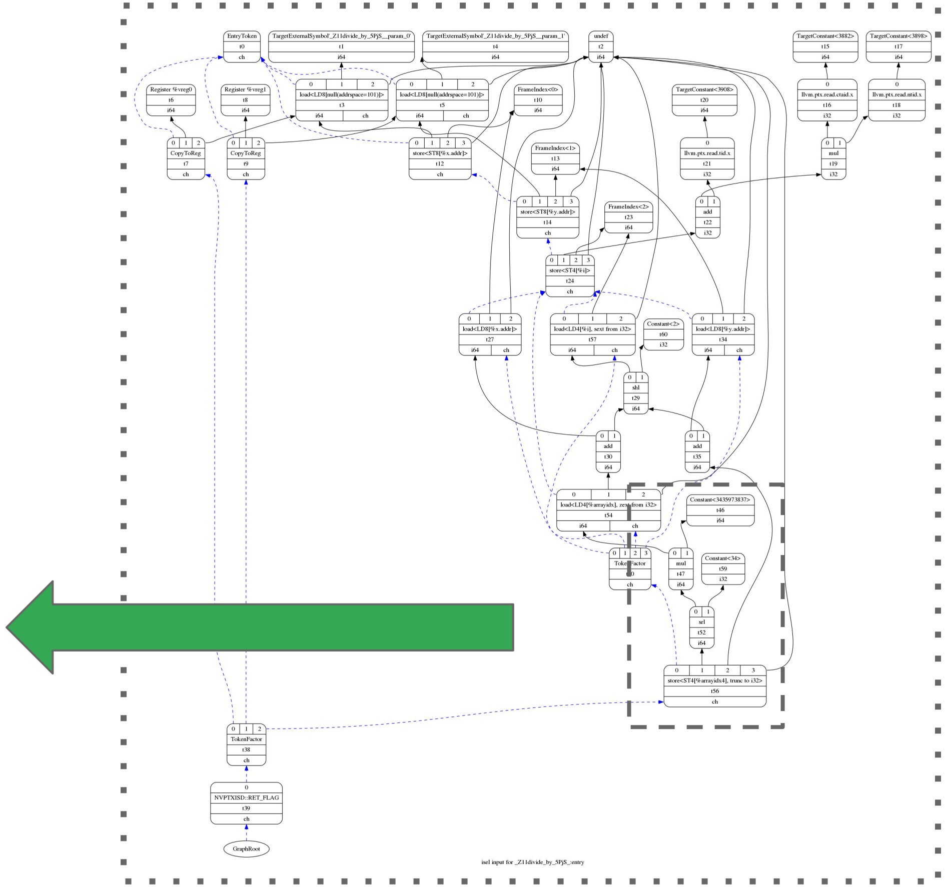
## Print ISEL input

```
__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





EntryToken
GraphRoot
NVPTXISD::RET_FLAG
TokenFactor
store<ST4[%arrayidx4], trunc to i32>
isel input for _Z11divide_by_5PjS_:entry

Google

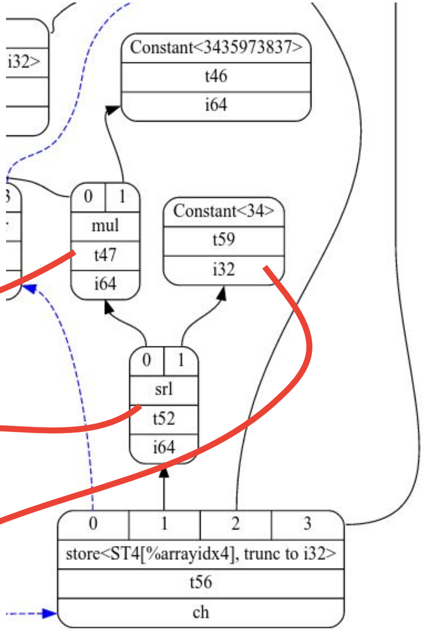
## NVPTX backend

- 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)>;
```

Google

## 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();
}
```

Google

## 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;
}
```

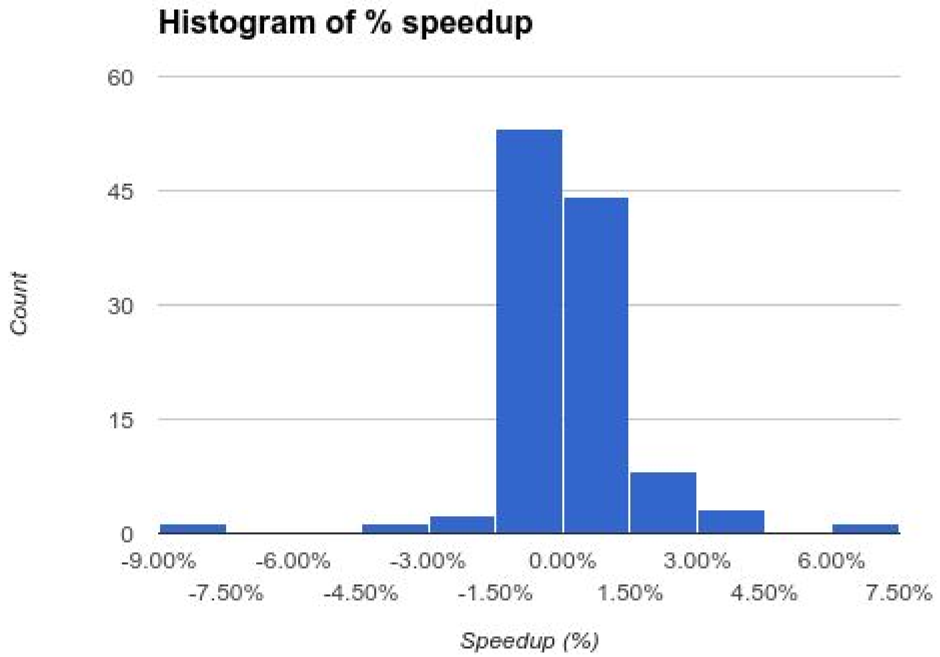
## But is it needed?

- 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 ..."

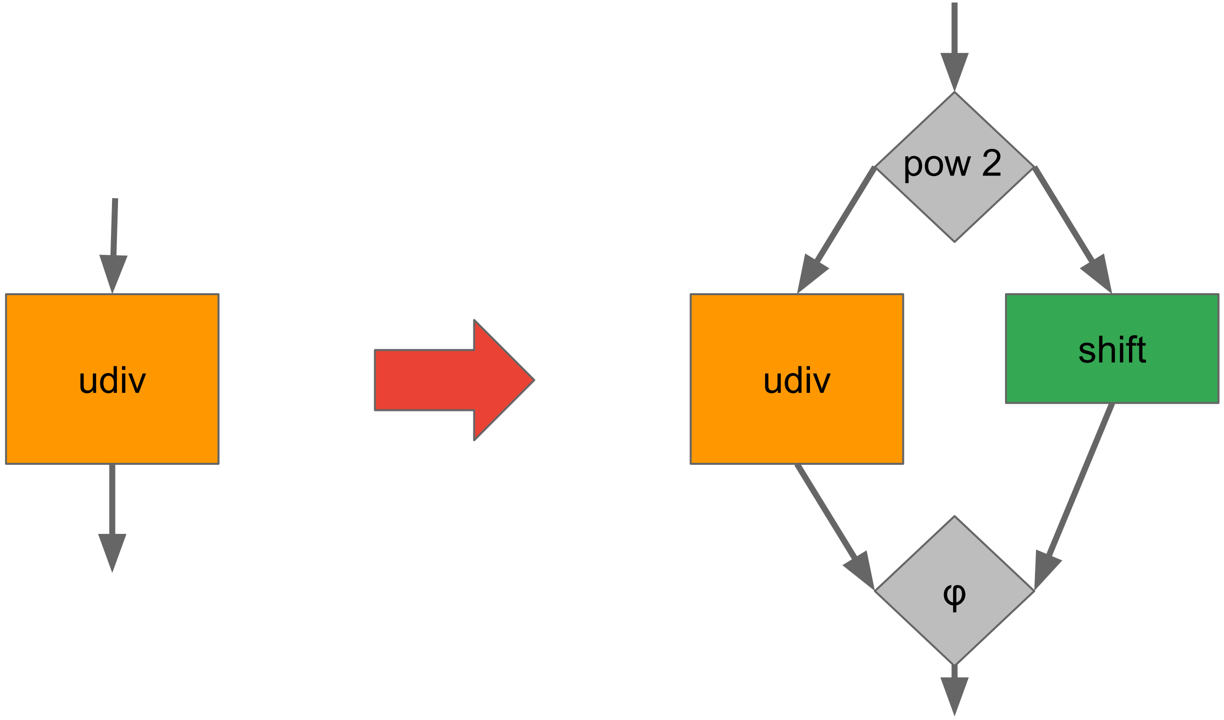
## Example input

- 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?

Google

## 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";
  }
};
```

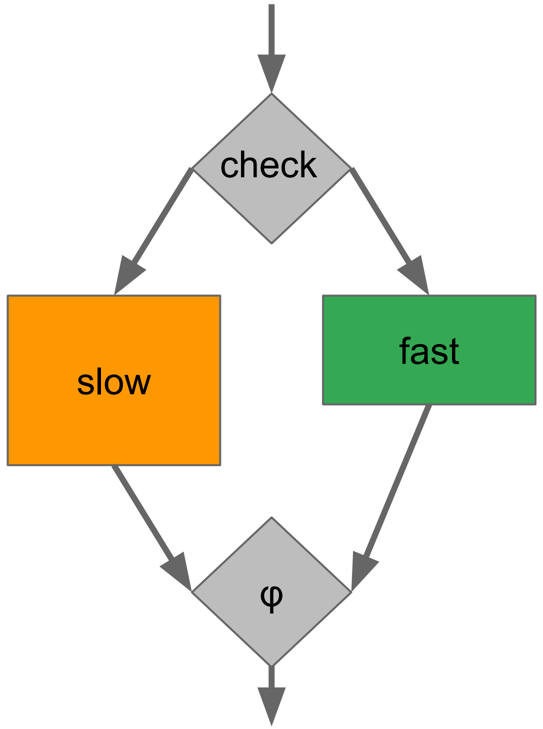
```
void NVPTXPassConfig::addCodeGenPrepare() {
  addPass(createBypassDivPowerTwo());
  TargetPassConfig::addCodeGenPrepare();
}
```

```
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();
}
```

Google

## 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



Google

# 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;
}
```

# 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?"

Google

## 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

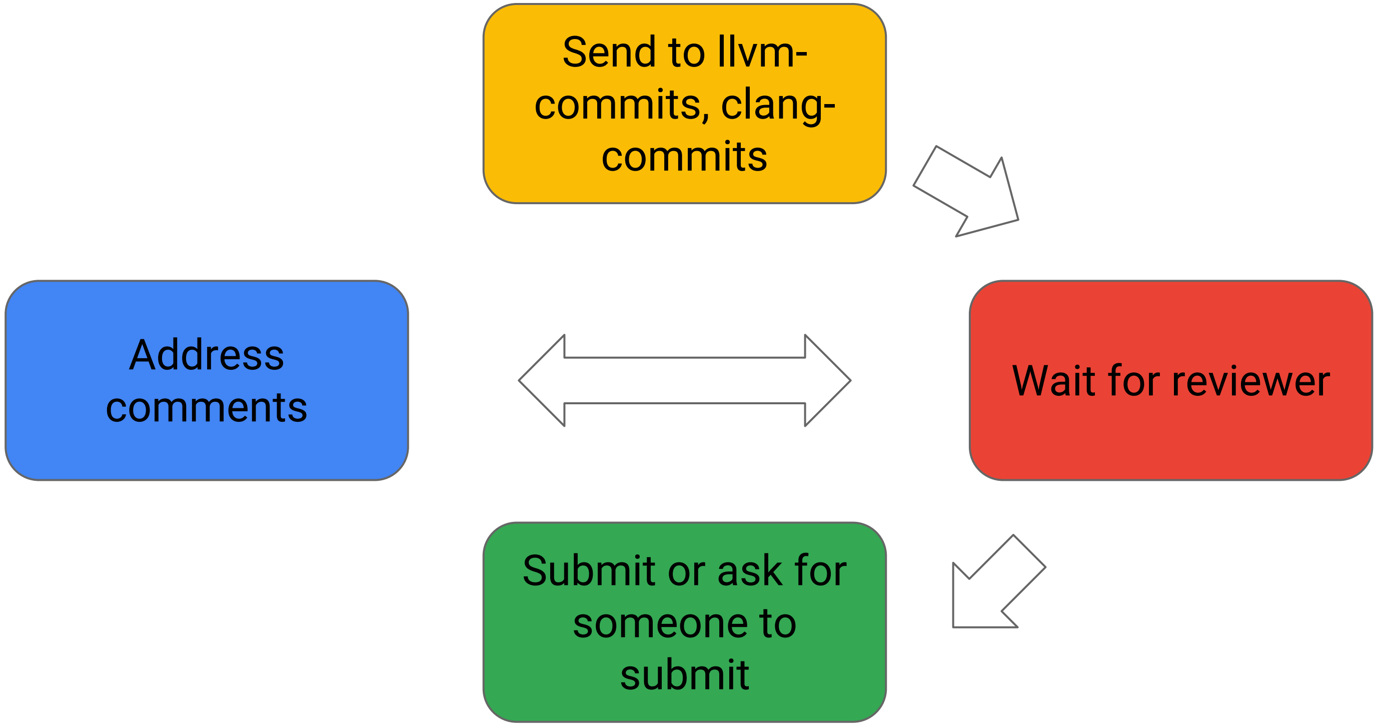
- License
  - Published under the "University of Illinois/NCSA Open Source License" BSD-style
  - Some of them are also published under MIT licence (dual-license)
  - No copyright assignment
- Community
  - Friendly
  - Mainly professional (Apple, Google, ARM, Linaro, Intel, etc)
  - But many individual 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://llvm.org/) | [http://clang.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


Send to llvm-commits, clang-commits
Address comments
Wait for reviewer
Submit or ask for someone to submit

## 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

Google

## 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 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?
- **Optimizer: more optimizations?**

## Discover optimization opportunities

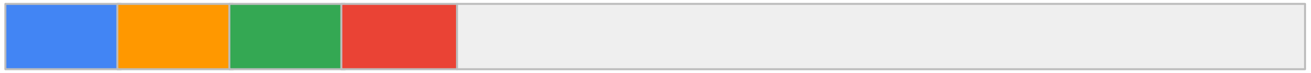
What optimizations do you currently have to perform in source code?

- Global memory coalescing
- Tune occupancy
- Avoid divergence
- Optimizations across host-device boundary
- Fast math
- More in *CUDA C Best practices* (http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/)

## Global memory coalescing

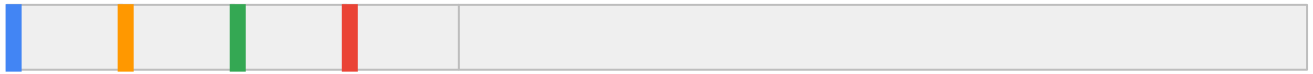
```
void copy(float* input, float* output, int n) {
  for (int i = 0; i < n; ++i)
    output[i] = input[i];
}
```

## Global memory coalescing



```
__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];
  }
}
```

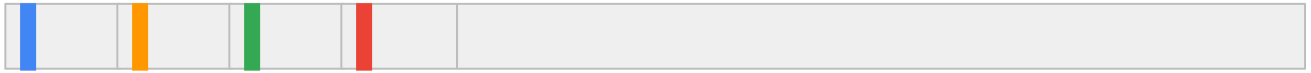
## Uncoalesced access



```
__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];
  }
}
```

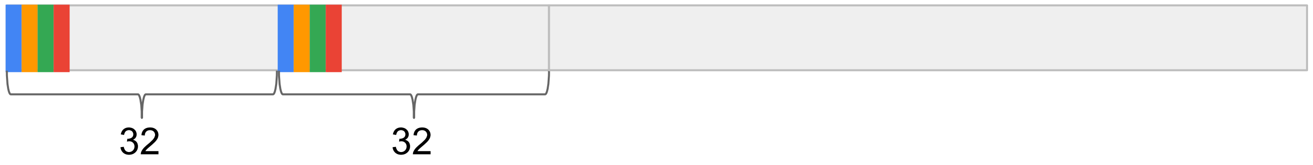
Google

## 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];
  }
}
```

Google

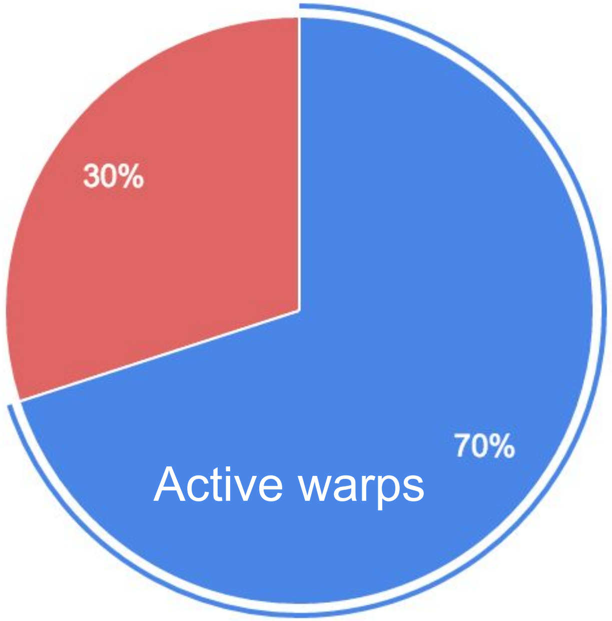
## Occupancy

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

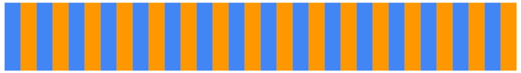
- Number of registers each block uses
- Shared memory size
- Block size

Trade off between occupancy and latency

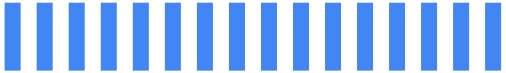
Feedback directed optimizations (FDO)?

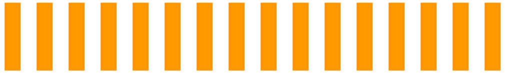


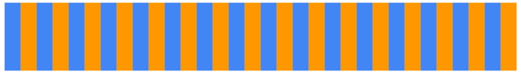
## Divergence



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



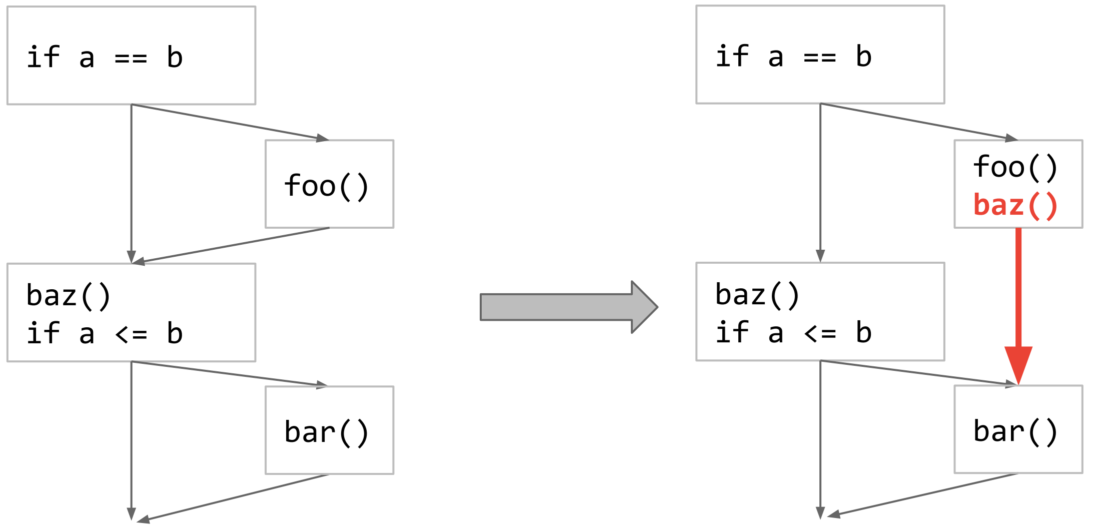




CPU: max(foo, bar)

GPU: foo + bar

# How divergence affects jump threading



foo + bar + baz

foo + bar + **baz * 2**

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

Google

## 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`<br>**`input[i] * (1.0f / 255.0f)`** |
| run time | 990.72 us (2.6x) | 373.92 us |

Google

## Takeaways from this tutorial

- The missions of gpucc
  - state-of-the-art platform for GPU compiler research
  - enables more industrial deployment
- In a good shape but can be improved a lot
- **Use and contribute to gpucc!** (http://bit.ly/llvm-cuda)