gpucc: An Open-Source GPGPU Compiler

Jingyue Wu (jingyue@google.com), Eli Bendersky,
Mark Heffernan, Chris Leary, Jacques Pienaar, Bjarke Roune,
Rob Springer, Xuetian Weng, Artem Belevich, Robert Hundt

One-Slide Overview

- Motivation
  - Lack of a state-of-the-art platform for CUDA compiler and HPC research
  - Binary dependencies, performance tuning, language features, bug turnaround times, etc.
- Solution
  - **gpucc**: the **first** fully-functional, open-source, high performance CUDA compiler
  - based on LLVM and supports C++11 and C++14
  - developed and tuned several general and CUDA-specific optimization passes
- Results highlight (compared with nvcc)
  - up to 51% faster on internal end-to-end benchmarks
  - on par on open-source benchmarks
  - compile time is 8% faster on average and 2.4x faster for pathological compilations

Mixed-Mode CUDA Code

```
template <int N>
__global__ void kernel(
    float *y) {
  ...
}
```

GPU/device

Mixed-Mode CUDA Code

```
template <int N>
void host(float *x) {
  float *y;
  cudaMalloc(&y, 4*N);
  cudaMemcpy(y, x, ...);
  kernel<N><<<16, 128>>>(y);
  ...
}
```

```
template <int N>
__global__ void kernel(
    float *y) {
  ...
}
```

CPU/host

GPU/device

Google

Mixed-Mode CUDA Code

foo.cu

```
template <int N>
void host(float *x) {
  float *y;
  cudaMalloc(&y, 4*N);
  cudaMemcpy(y, x, ...);
  kernel<N><<<16, 128>>>(y);
  ...
}
```

```
template <int N>
__global__ void kernel(
    float *y) {
  ...
}
```

CPU/host

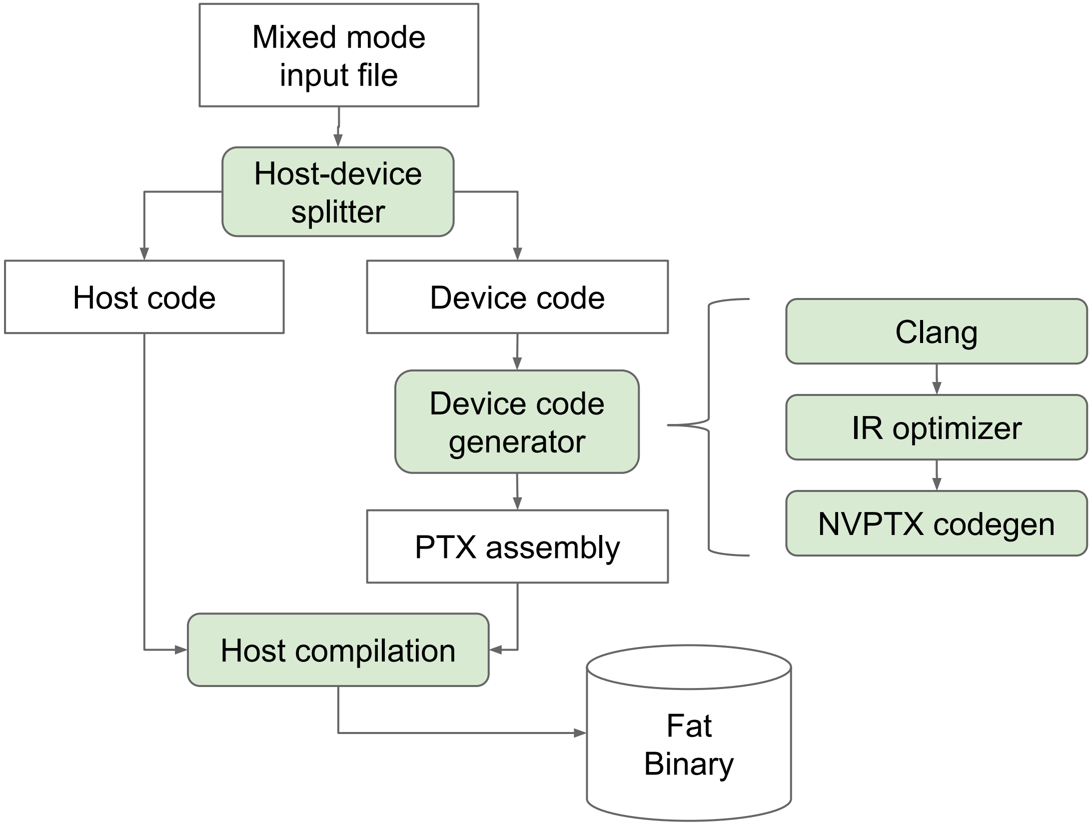
GPU/device

Google

gpucc Architecture (Current and Interim)

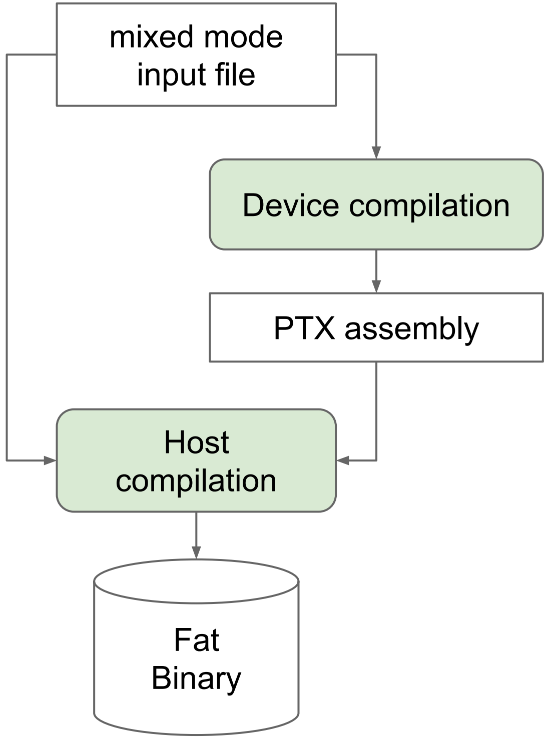
```
template <int N>
__global__ void kernel(
    float *y) {
  ...
}

template <int N>
void host(float *x) {
  float *y;
  cudaMalloc(&y, 4*N);
  cudaMemcpy(y, x, ...);
  kernel<N><<<16, 128>>>(y);
  ...
}
```



Clang Integration (WIP and Long-Term)

- Major issues with the separate compilation
    - Source-to-source translation is complex and fragile
    - Long compilation time
- Clang driver instead of physical code splitting
    - (by Artem Belevich)
    - `$ clang foo.cu ...`
    - `$ clang -x cuda <file> ...`



CPU vs GPU Characteristics

CPU

- Designed for general purposes
- Optimized for latency
- Heavyweight hardware threads
  - Branch prediction
  - Out-of-order execution
  - Superscalar
- Small number of cores per die

GPU

- Designed for rendering
- Optimized for throughput
- Lightweight hardware threads
- Massive parallelism
  - Can trade latency for throughput

Major Optimizations in gpucc

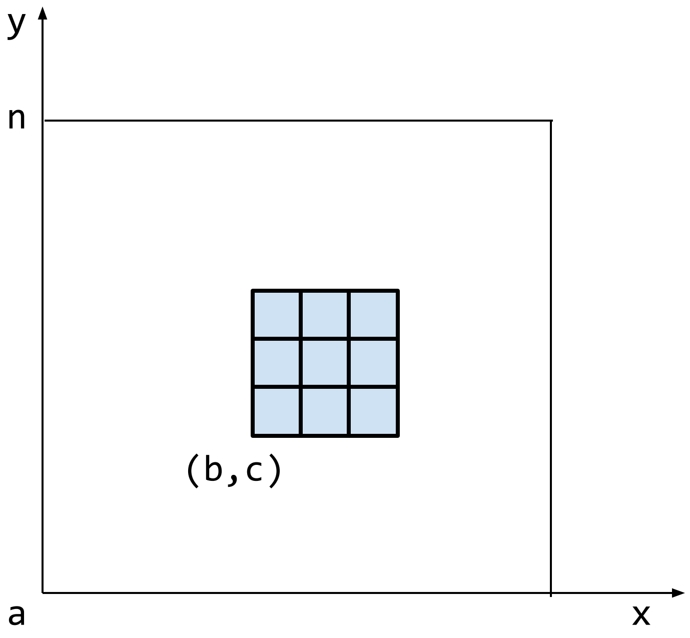
- Straight-line scalar optimizations
- Inferring memory spaces
- Loop unrolling and function inlining
- Memory-space alias analysis
- Speculative execution
- Bypassing 64-bit divisions

Major Optimizations in gpucc

- Straight-line scalar optimizations
- Inferring memory spaces
- Loop unrolling and function inlining
- Memory-space alias analysis
- Speculative execution
- Bypassing 64-bit divisions

Straight-Line Scalar Optimizations

```
for (long x = 0; x < 3; ++x) {
  for (long y = 0; y < 3; ++y) {
    float *p = &a[(c+y) + (b+x) * n];
    ... // load from p
  }
}
```



Straight-Line Scalar Optimizations

```
for (long x = 0; x < 3; ++x) {
  for (long y = 0; y < 3; ++y) {
    float *p = &a[(c+y) + (b+x) * n];
    ... // load from p
  }
}
```

loop unroll →

```
p0 = &a[c     + b       * n];
p1 = &a[c + 1 + b       * n];
p2 = &a[c + 2 + b       * n];

p3 = &a[c     + (b + 1) * n];
p4 = &a[c + 1 + (b + 1) * n];
p5 = &a[c + 2 + (b + 1) * n];

p6 = &a[c     + (b + 2) * n];
p7 = &a[c + 1 + (b + 2) * n];
p8 = &a[c + 2 + (b + 2) * n];
```

Straight-Line Scalar Optimizations

```
p0 = &a[c     + b       * n];
p1 = &a[c + 1 + b       * n];
p2 = &a[c + 2 + b       * n];

p3 = &a[c     + (b + 1) * n];
p4 = &a[c + 1 + (b + 1) * n];
p5 = &a[c + 2 + (b + 1) * n];

p6 = &a[c     + (b + 2) * n];
p7 = &a[c + 1 + (b + 2) * n];
```

```
        c + 2
                 b + 2
                (b + 2) * n
        c + 2 + (b + 2) * n
p8 = &a[c + 2 + (b + 2) * n];
```

Straight-Line Scalar Optimizations

```
p0 = &a[c     + b       * n];
p1 = &a[c + 1 + b       * n];
p2 = &a[c + 2 + b       * n];

p3 = &a[c     + (b + 1) * n];
p4 = &a[c + 1 + (b + 1) * n];
p5 = &a[c + 2 + (b + 1) * n];

p6 = &a[c     + (b + 2) * n];
p7 = &a[c + 1 + (b + 2) * n];
```

`c + 2`

`b + 2`

~~(b + 2) * n~~

`c + 2 + (b + 2) * n`

p8 = ~~&a[c + 2 + (b + 2) * n];~~

Addressing mode (base+imm)

`p8 = &a[c + (b + 2) * n] + 2`

- Pointer arithmetic reassociation

Injured redundancy

`(b + 1) * n + n`

- Straight-line strength reduction
- Global reassociation

Pointer Arithmetic Reassociation

```
p0 = &a[c     + b       * n];
p1 = &a[c + 1 + b       * n];
p2 = &a[c + 2 + b       * n];

p3 = &a[c     + (b + 1) * n];
p4 = &a[c + 1 + (b + 1) * n];
p5 = &a[c + 2 + (b + 1) * n];

p6 = &a[c     + (b + 2) * n];
p7 = &a[c + 1 + (b + 2) * n];
p8 = &a[c + 2 + (b + 2) * n];
```

⇒

```
p0 = &a[c + b * n];
p1 = &p0[1];
p2 = &p0[2];

p3 = &a[c + (b + 1) * n];
p4 = &p3[1];
p5 = &p3[2];

p6 = &a[c + (b + 2) * n];
p7 = &p6[1];
p8 = &p6[2];
```

Straight-Line Strength Reduction

```
x = (base+C0)*stride
y = (base+C1)*stride
```

→

```
x = (base+C0)*stride
y = x + (C1-C0)*stride
```

Straight-Line Strength Reduction

```
x = (base+C0)*stride
y = (base+C1)*stride
```

→

```
x = (base+C0)*stride
y = x + (C1-C0)*stride
```

```
x0 = b * n;
p0 = &a[c + x0];
p1 = &p0[1];
p2 = &p0[2];

x1 = (b + 1) * n;
p3 = &a[c + x1];
p4 = &p3[1];
p5 = &p3[2];

x2 = (b + 2) * n;
p6 = &a[c + x2];
p7 = &p6[1];
p8 = &p6[2];
```

→

```
x0 = b * n;
p0 = &a[c + x0];
p1 = &p0[1];
p2 = &p0[2];

x1 = x0 + n;
p3 = &a[c + x1];
p4 = &p3[1];
p5 = &p3[2];

x2 = x1 + n;
p6 = &a[c + x2];
p7 = &p6[1];
p8 = &p6[2];
```

Global Reassociation

```
x0 = b * n;
p0 = &a[c + x0];
p1 = &p0[1];
p2 = &p0[2];

x1 = x0 + n;
p3 = &a[c + x1];
p4 = &p3[1];
p5 = &p3[2];

x2 = x1 + n;
p6 = &a[c + x2];
p7 = &p6[1];
p8 = &p6[2];
```

Google

Global Reassociation

```
x0 = b * n;          x0 = b * n;
p0 = &a[c + x0];     i0 = c + x0;
p1 = &p0[1];
p2 = &p0[2];

x1 = x0 + n;         x1 = x0 + n;
p3 = &a[c + x1];     i1 = c + x1;
p4 = &p3[1];
p5 = &p3[2];

x2 = x1 + n;
p6 = &a[c + x2];
p7 = &p6[1];
p8 = &p6[2];
```

Global Reassociation

```
x0 = b * n;
p0 = &a[c + x0];
p1 = &p0[1];
p2 = &p0[2];

x1 = x0 + n;
p3 = &a[c + x1];
p4 = &p3[1];
p5 = &p3[2];

x2 = x1 + n;
p6 = &a[c + x2];
p7 = &p6[1];
p8 = &p6[2];
```

```
x0 = b * n;
i0 = c + x0;

x1 = x0 + n;
i1 = c + x1; // = c+(x0+n) = (c+x0)+n
```

Google

Global Reassociation

```
x0 = b * n;
p0 = &a[c + x0];
p1 = &p0[1];
p2 = &p0[2];

x1 = x0 + n;
p3 = &a[c + x1];
p4 = &p3[1];
p5 = &p3[2];

x2 = x1 + n;
p6 = &a[c + x2];
p7 = &p6[1];
p8 = &p6[2];
```

```
x0 = b * n;
i0 = c + x0;

x1 = x0 + n;
i1 = c + x1;
```

```
i1 = i0 + n;
```

Global Reassociation

```
x0 = b * n;
p0 = &a[c + x0];
p1 = &p0[1];
p2 = &p0[2];

x1 = x0 + n;
p3 = &a[c + x1];
p4 = &p3[1];
p5 = &p3[2];

x2 = x1 + n;
p6 = &a[c + x2];
p7 = &p6[1];
p8 = &p6[2];
```

```
x0 = b * n;
i0 = c + x0;
p0 = &a[i0];

x1 = x0 + n;
i1 = c + x1;
p3 = &a[i1];
```

→

```
i1 = i0 + n;
p3 = &a[i1];
```

→

```
p3 = &p0[n];
```

Global Reassociation

```
x0 = b * n;
p0 = &a[c + x0];
p1 = &p0[1];
p2 = &p0[2];

x1 = x0 + n;
p3 = &a[c + x1];
p4 = &p3[1];
p5 = &p3[2];

x2 = x1 + n;
p6 = &a[c + x2];
p7 = &p6[1];
p8 = &p6[2];
```

```
x0 = b * n;
i0 = c + x0;
p0 = &a[i0];

x1 = x0 + n;
i1 = c + x1;
p3 = &a[i1];
```

```
i1 = i0 + n;
p3 = &a[i1];
```

```
p3 = &p0[n];
```

```
x0 = b * n;
p0 = &a[c + x0];
p1 = &p0[1];
p2 = &p0[2];

p3 = &p0[n];
p4 = &p3[1];
p5 = &p3[2];

p6 = &p3[n];
p7 = &p6[1];
p8 = &p6[2];
```

Summary of Straight-Line Scalar Optimizations

```
p0 = &a[c     + b       * n];
p1 = &a[c + 1 + b       * n];
p2 = &a[c + 2 + b       * n];

p3 = &a[c     + (b + 1) * n];
p4 = &a[c + 1 + (b + 1) * n];
p5 = &a[c + 2 + (b + 1) * n];

p6 = &a[c     + (b + 2) * n];
p7 = &a[c + 1 + (b + 2) * n];
p8 = &a[c + 2 + (b + 2) * n];
```

→

```
x0 = b * n;
p0 = &a[c + x0];
p1 = &p0[1];
p2 = &p0[2];

p3 = &p0[n];
p4 = &p3[1];
p5 = &p3[2];

p6 = &p3[n];
p7 = &p6[1];
p8 = &p6[2];
```

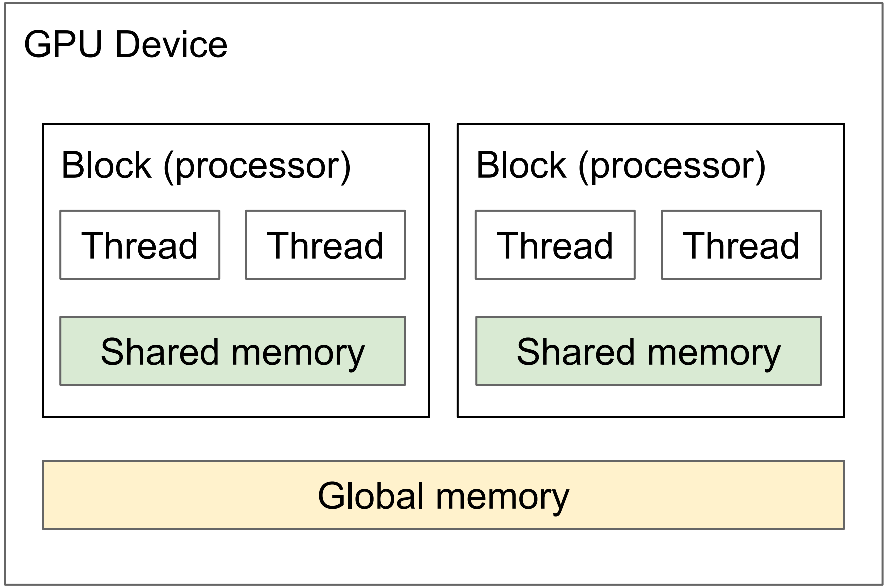
Design doc: https://goo.gl/4Rb9As

Google

Optimizations

- Straight-line scalar optimizations
- Inferring memory spaces
- Loop unrolling and function inlining
- Memory-space alias analysis
- Speculative execution
- Bypassing 64-bit divisions

Inferring Memory Spaces



Load/store PTX assembly instructions

- Special
  - `ld.shared/st.shared`
  - `ld.global/st.global`
  - `...`
- Generic
  - `ld/st`
  - Overhead in checking (e.g. ~10% slower than `ld.shared`)
  - Alias analysis suffers

Inferring Memory Spaces

Memory space qualifiers

```
__global__ void foo(int i) {
  __shared__ float a[1024];
  float* p = a;
  while (p != a + 1024) {
    float v = *p; // expect ld.shared.f32
    ...
    ++p;
  }
}
```

Fixed-point data-flow analysis

- First assumes all derived pointers are *special*
- Then iteratively reverts the assumption if an incoming value is generic
- Design doc: https://goo.gl/5wH2Ct

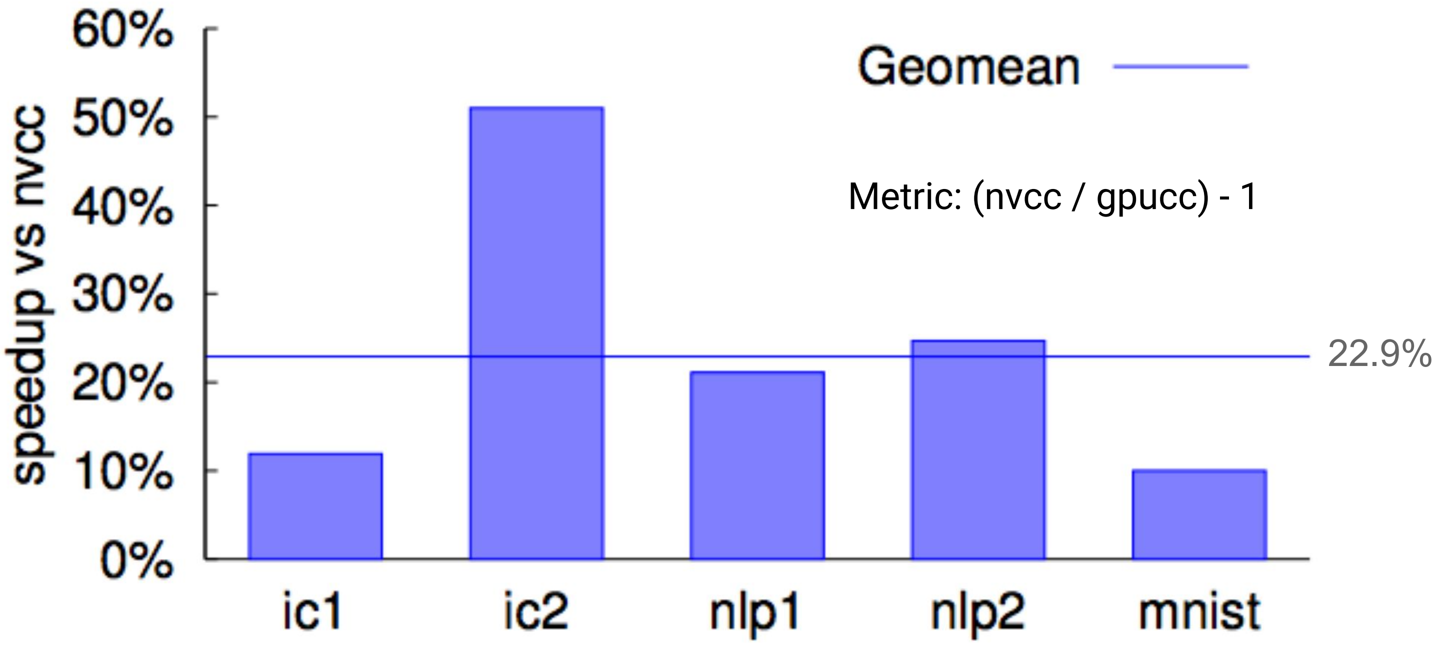
Other Optimizations

- Loop unrolling and function inlining
  - Higher threshold
  - `#pragma unroll` (by Mark Heffernan)
  - `__forceinline__`
- Memory-space alias analysis (by Xuetian Weng)
  - Different special memory spaces do not alias.
- Speculative execution (by Bjarke Roune)
  - Hoists instructions from conditional basic blocks.
  - Promotes straight-line scalar optimizations
- Bypassing 64-bit divides (by Mark Heffernan)
  - 64-bit divides (~70 machine instructions) are much slower than 32-bit divides (~20).
  - If the runtime values are 32-bit, perform a 32-bit divide instead.

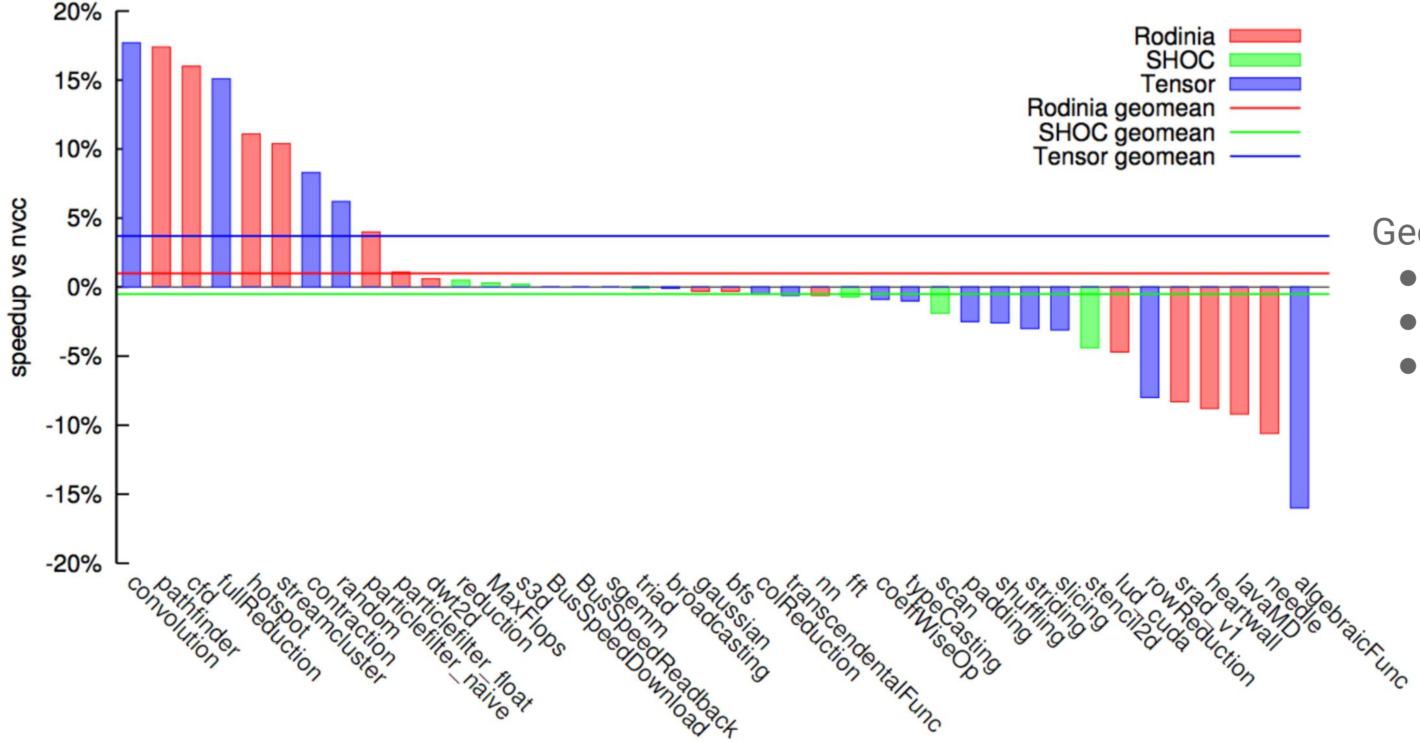
Evaluation

- Benchmarks
  - End-to-end internal benchmarks
    - ic1, ic2: image classification
    - nlp1, nlp2: natural language processing
    - mnist: handwritten digit recognition
  - Open-source benchmark suites
    - Rodinia: reduced from real-world applications
    - SHOC: scientific computing
    - Tensor: heavily templated CUDA C++ library for linear algebra
- Machine setup
  - GPU: NVIDIA Tesla K40c
- Baseline: nvcc 7.0 (latest at the time of the evaluation)

Performance on End-to-End Benchmarks



Performance on Open-Source Benchmarks

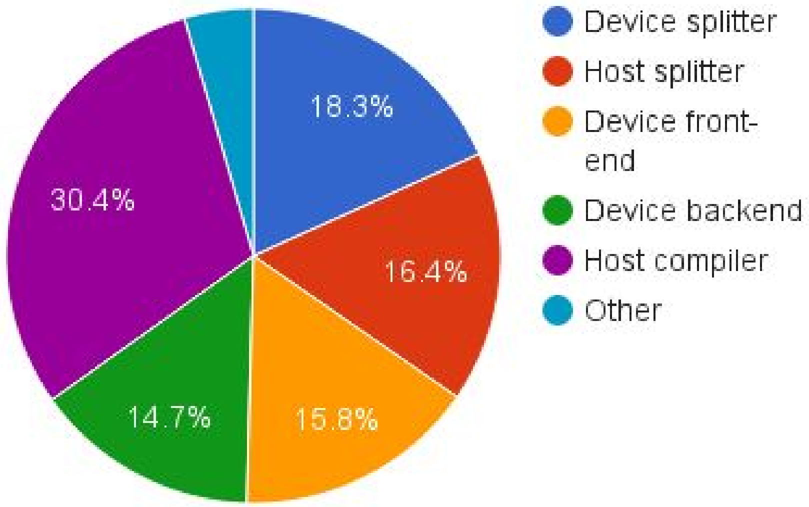


Geomean speedup

- Tensor: 3.7%
- Rodinia: 0.8%
- SHOC: -0.5%

Compilation Time

- 8% faster than nvcc on average (per translation-unit)
- 263.1s (nvcc) vs 109.8s (gpucc) — 2.4x speedup
- Should be even faster with Clang integration



Conclusions and Future Work

- gpucc: fully-functional, open-source, high performance CUDA compiler
    - Enable reproducible GPU compiler and architecture research
    - Ease deployment of GPUs in restricted and controlled environments
- Concepts and insights are general and applicable to other GPU platforms
- Future work on open-sourcing (Q1 2016)
    - Clang integration is in progress
    - Target-specific optimization pipeline
    - Driver-only runtime support
    - Lots of documentation and packaging

Backup

Use gpucc in Standalone Mode

Device code: axpy.cu

```
#include "cuda_builtin_vars.h"

#define __global__ __attribute__((global))

__global__ void axpy(float a, float* x, float* y) {
  y[threadIdx.x] =  a * x[threadIdx.x];
}

$ clang -cc1 -triple nvptx64-nvidia-cuda -target-cpu sm_35 -include <path to cuda_builtin_vars.h> -emit-llvm
-fcuda-is-device -O3 axpy.cu -o axpy.ll
$ llc axpy.ll -o axpy.ptx -mcpu=sm_35
```

Use gpucc in Standalone Mode

Host code: axpy.cc

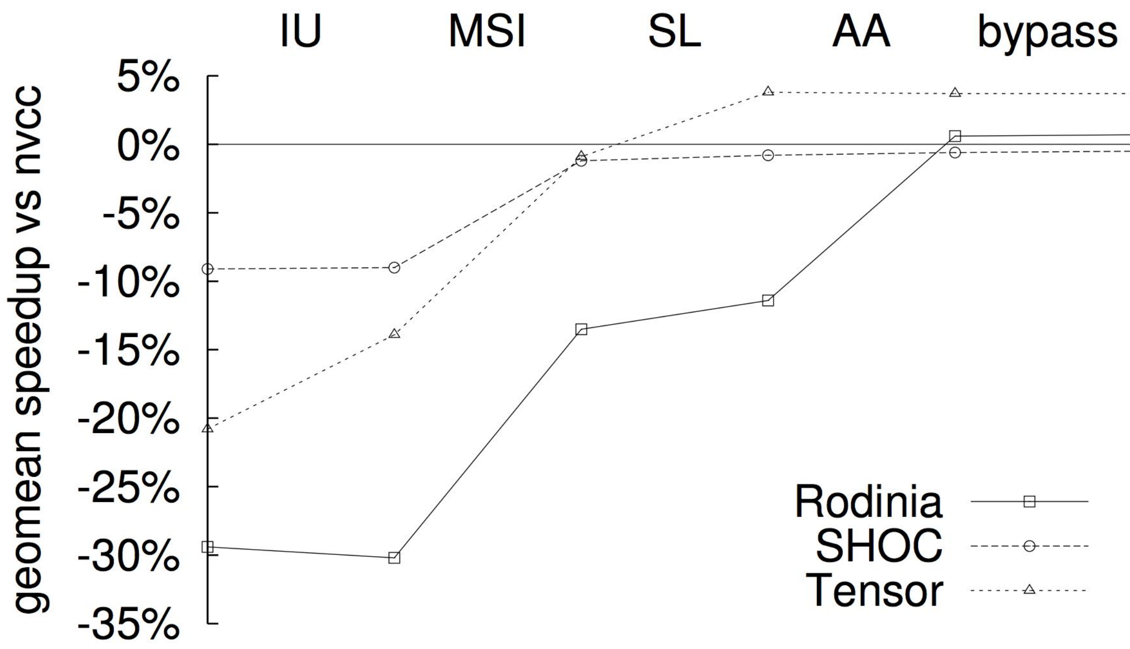
```
CUmodule module;
ExitOnError(cuModuleLoad(&module, "axpy.ptx"));
CUfunction kernel;
ExitOnError(cuModuleGetFunction(&kernel, module, "_Z4axpyfPfS_"));
void* args[3] = {(void *)&a, (void *)&device_x, (void *)&device_y};
ExitOnError(cuLaunchKernel(kernel,
                           /*grid*/1, 1, 1,
                           /*block*/kDataLen, 1, 1,
                           /*sharedMemBytes*/0,
                           /*stream*/0,
                           args,
                           /*extra*/0));
```

```
$ clang++ axpy.cc -I<parent directory of cuda.h> -L<parent directory of libcuda.so> -lcuda
```

Function Overloading

- Function overloading
    - `__host__   void abs() {...}`
    - `__device__ void abs() {...}`

Effects of Optimizations



- **IU**: inline and unroll
- **MSI**: memory space inference
- **SL**: straight-line scalar optimizations
- **AA**: memory-space alias analysis
- **bypass**: bypassing 64-bit divides

Opt	Benchmark	Speedup
IU	ic1	~10x
MSI	ic1	~3x
bypass	ic2	50%
SL	ic1	28.1%

Straight-Line Strength Reduction

```
x = (base+C0)*stride
y = (base+C1)*stride
```

```
x = base + C0*stride
y = base + C1*stride
```

```
x = &base[C0*stride]
y = &base[C1*stride]
```

```
x = (base+C0)*stride
y = x + (C1-C0)*stride
```