



gpucc: An Open-Source GPGPU Compiler

Jingyue Wu, Artem Belevich, Eli Bendersky, Mark Heffernan, Chris Leary,
Jacques Pienaar, Bjarke Rouné, Rob Springer, Xuétian Weng, 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
 - frontend integrated into Clang so supports **C++11** and **partially C++14**
 - backend integrated into LLVM with **general and CUDA-specific optimizations**
- Results highlight (compared with nvcc 7.0)
 - 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

Compiler Architecture

Mixed-Mode CUDA Code

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



GPU/device

Mixed-Mode CUDA Code

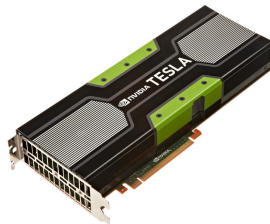
```
int main() {  
    float* arr;  
    cudaMalloc(&arr, 128*sizeof(float));  
    Write42<<<1, 128>>>(arr);  
}
```

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

CPU/host



GPU/device



Mixed-Mode CUDA Code

foo.cu

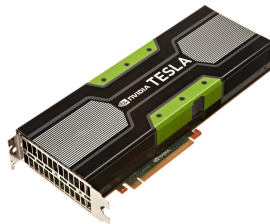
```
int main() {  
    float* arr;  
    cudaMalloc(&arr, 128*sizeof(float));  
    Write42<<<1, 128>>>(arr);  
}
```

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

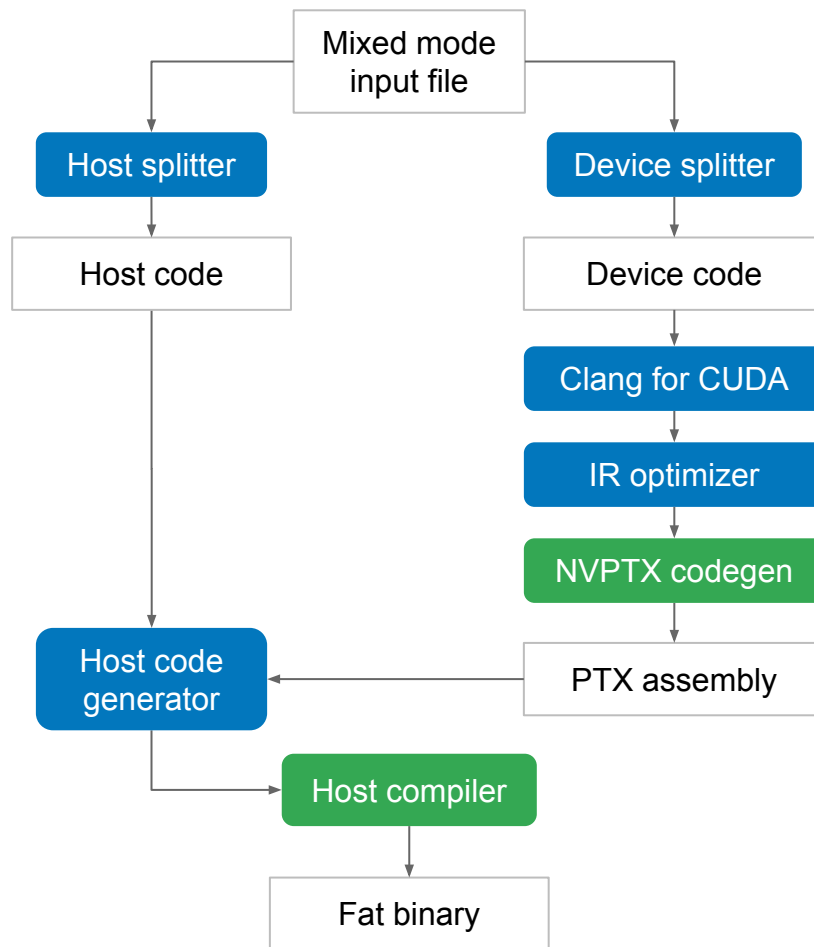
CPU/host



GPU/device



Separate Compilation



Separate Compilation

Disadvantages

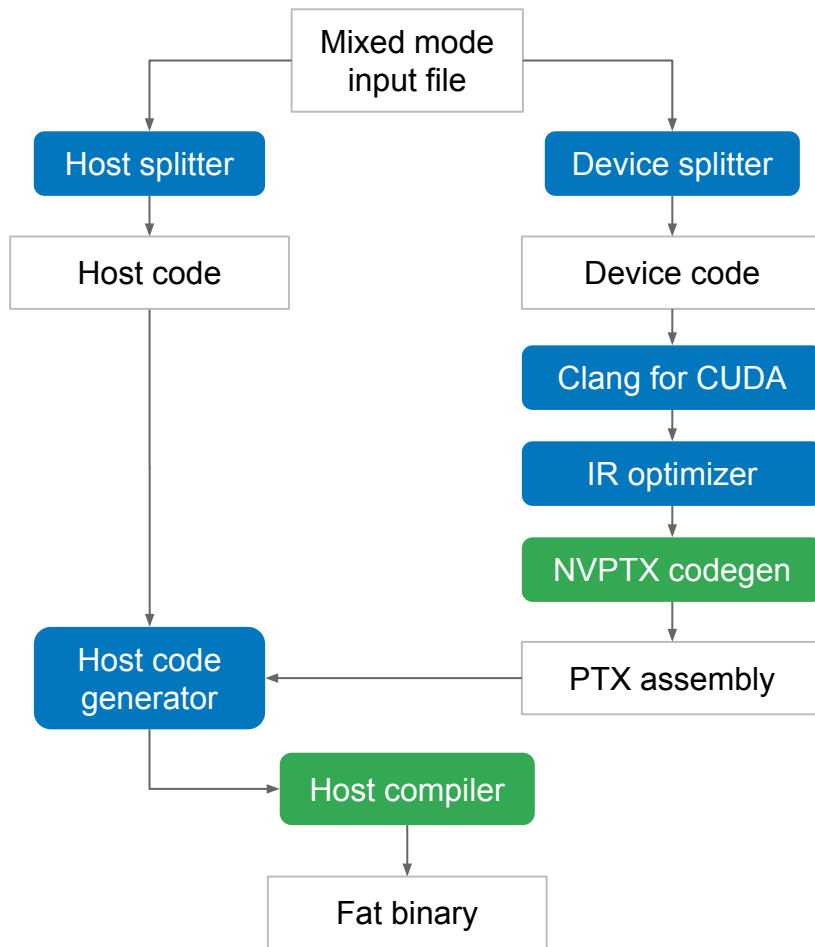
- Source-to-source translation is fragile

```
template <int batchSize>
__global__ void kernel(float* input,
                      int len) {
```

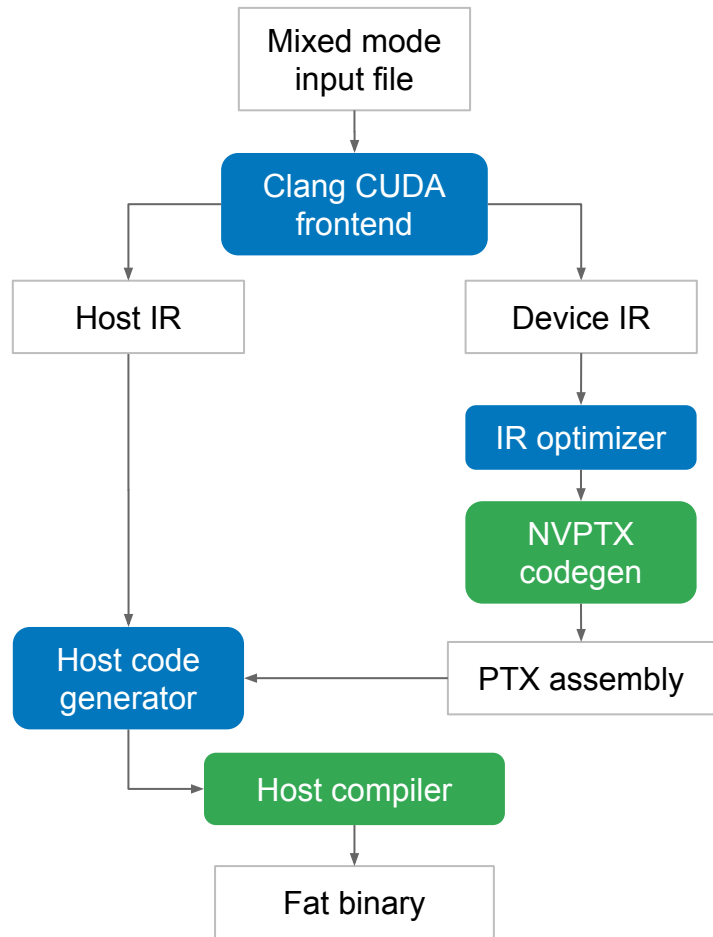
```
    ...
}
```

```
void host(float* input, int len) {
    if (len % 16 == 0) {
        kernel<16><<<1, len/16>>>
            (input, len);
    }
    ...
}
```

- Waste compilation time



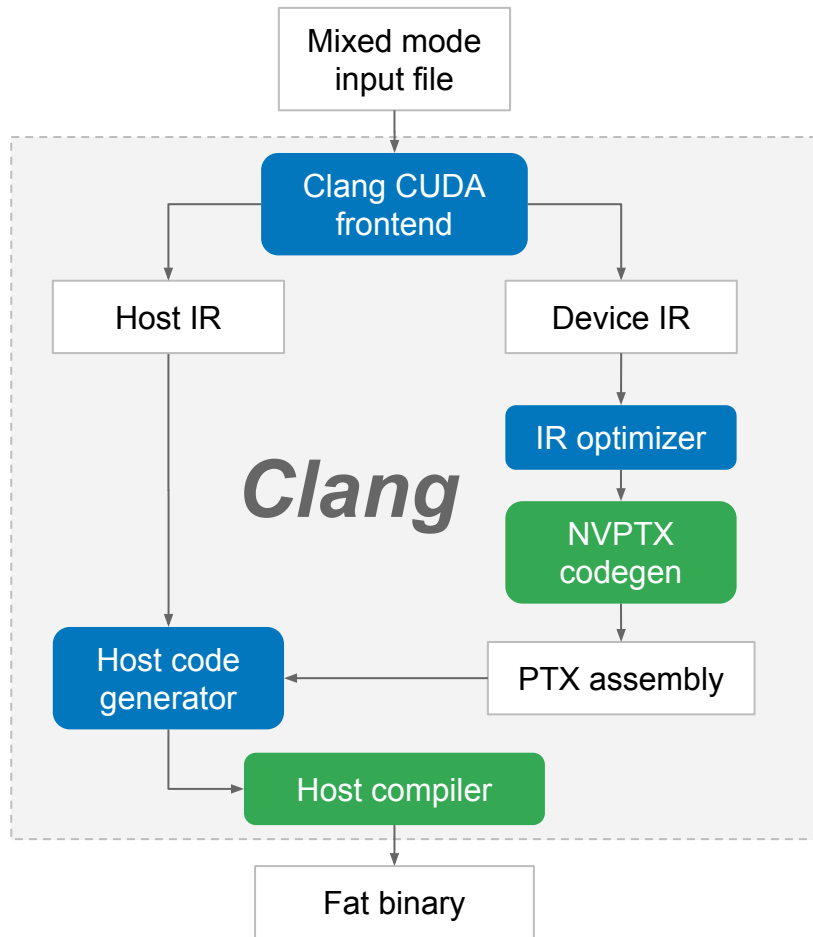
Dual-Mode Compilation



Clang Integration

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

More user guide at bit.ly/gpucc-tutorial



Optimizations

CPU vs GPU Characteristics

CPU

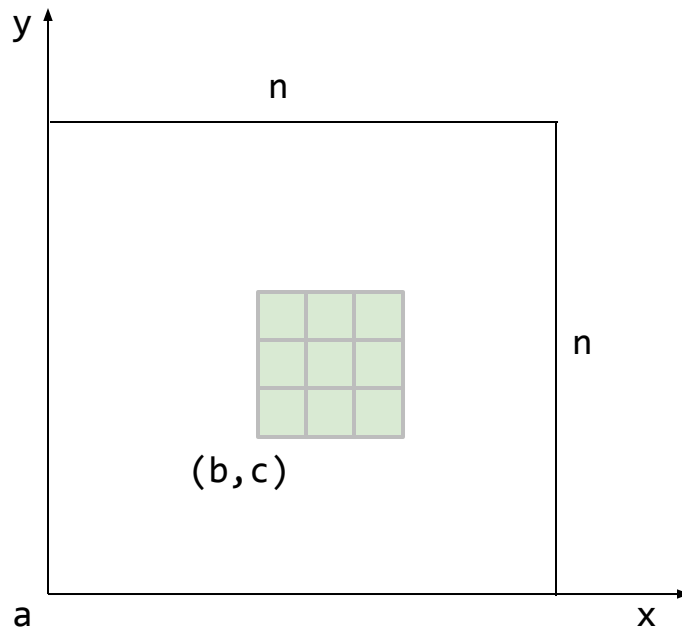
- Designed for general purposes
- Optimized for latency
- Heavyweight hardware threads
 - Branch prediction
 - Out-of-order execution
 - Superscalar
- Small number of cores per die

GPU

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

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];
```

Addressing mode (base+imm)

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

- Pointer arithmetic reassociation

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

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

$x = (\text{base} + C_0) * \text{stride}$
 $y = (\text{base} + C_1) * \text{stride}$  $x = (\text{base} + C_0) * \text{stride}$
 $y = x + (C_1 - C_0) * \text{stride}$

$x_0 = b * n;$
 $p_0 = \&a[c + x_0];$
 $p_1 = \&p_0[1];$
 $p_2 = \&p_0[2];$

$x_1 = (b + 1) * n;$
 $p_3 = \&a[c + x_1];$
 $p_4 = \&p_3[1];$
 $p_5 = \&p_3[2];$

$x_2 = (b + 2) * n;$
 $p_6 = \&a[c + x_2];$
 $p_7 = \&p_6[1];$
 $p_8 = \&p_6[2];$

$x_0 = b * n;$
 $p_0 = \&a[c + x_0];$
 $p_1 = \&p_0[1];$
 $p_2 = \&p_0[2];$

$x_1 = x_0 + n;$
 $p_3 = \&a[c + x_1];$
 $p_4 = \&p_3[1];$
 $p_5 = \&p_3[2];$

$x_2 = x_1 + n;$
 $p_6 = \&a[c + x_2];$
 $p_7 = \&p_6[1];$
 $p_8 = \&p_6[2];$

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];
```

Global Reassociation

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

```
x1 = x0 + n;  
p3 = &a[c + x1];    i1 = c + x1 = c + (x0 + n)  
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];    i0 = c + x0;  
p1 = &p0[1];  
p2 = &p0[2];
```

```
x1 = x0 + n;  
p3 = &a[c + x1];    i1 = c + x1 = c + (x0 + n)  
p4 = &p3[1];        = (c + x0) + n = i0 + n  
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];    i0 = c + x0;  
p1 = &p0[1];  
p2 = &p0[2];
```

```
x1 = x0 + n;  
p3 = &a[c + x1];    i1 = c + x1;   i1 = i0 + n;  
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];    i0 = c + x0;  
p1 = &p0[1];        p0 = &a[i0];  
p2 = &p0[2];
```

```
x1 = x0 + n;  
p3 = &a[c + x1];    i1 = c + x1;    i1 = i0 + n;  
p4 = &p3[1];        p3 = &a[i1] = &a[i0 + n]  
p5 = &p3[2];        = &a[i0] + n = &p0[n]
```

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


Global Reassociation

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

```
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];
```



```
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: bit.ly/straight-line-optimizations

Other Major Optimizations

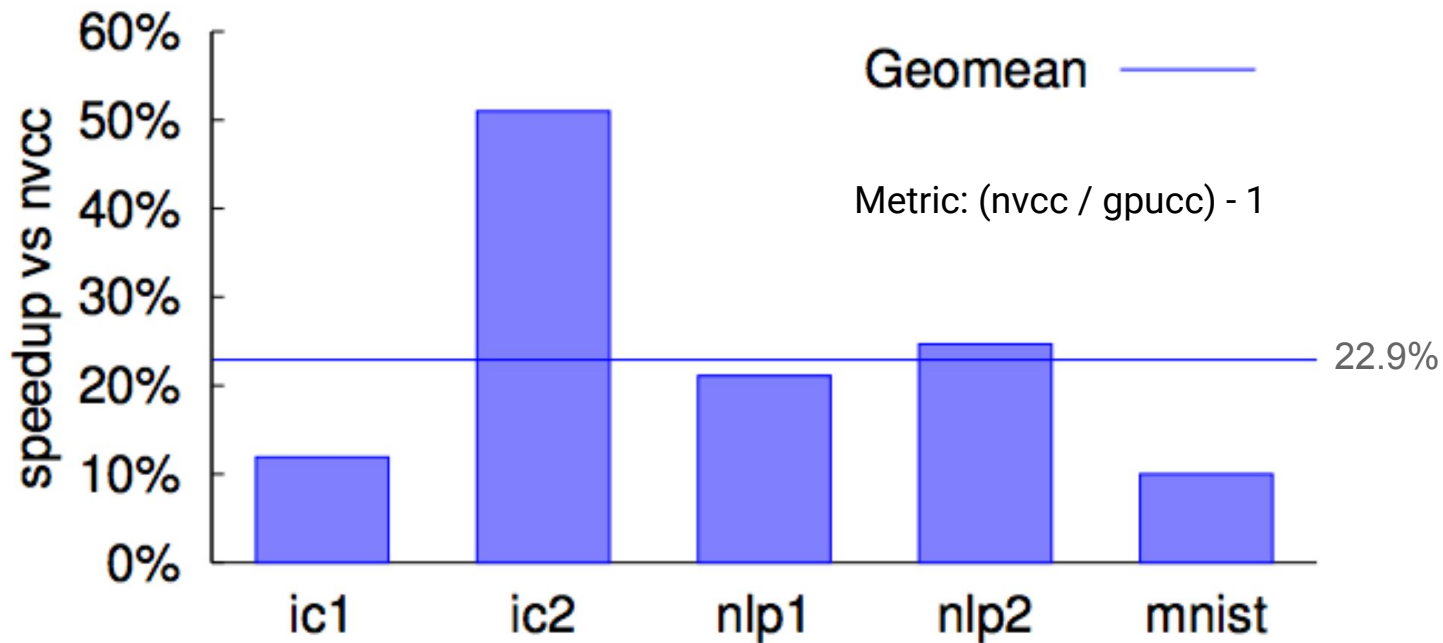
- Loop unrolling and function inlining
 - Higher threshold
 - [`#pragma unroll`](#)
 - `__forceinline__`
- [Memory space inference](#): emit specific memory accesses
- [Memory space alias analysis](#): different specific memory spaces do not alias
- [Speculative execution](#)
 - Hoists instructions from conditional basic blocks.
 - Promotes straight-line scalar optimizations
- [Bypassing 64-bit divides](#)
 - 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

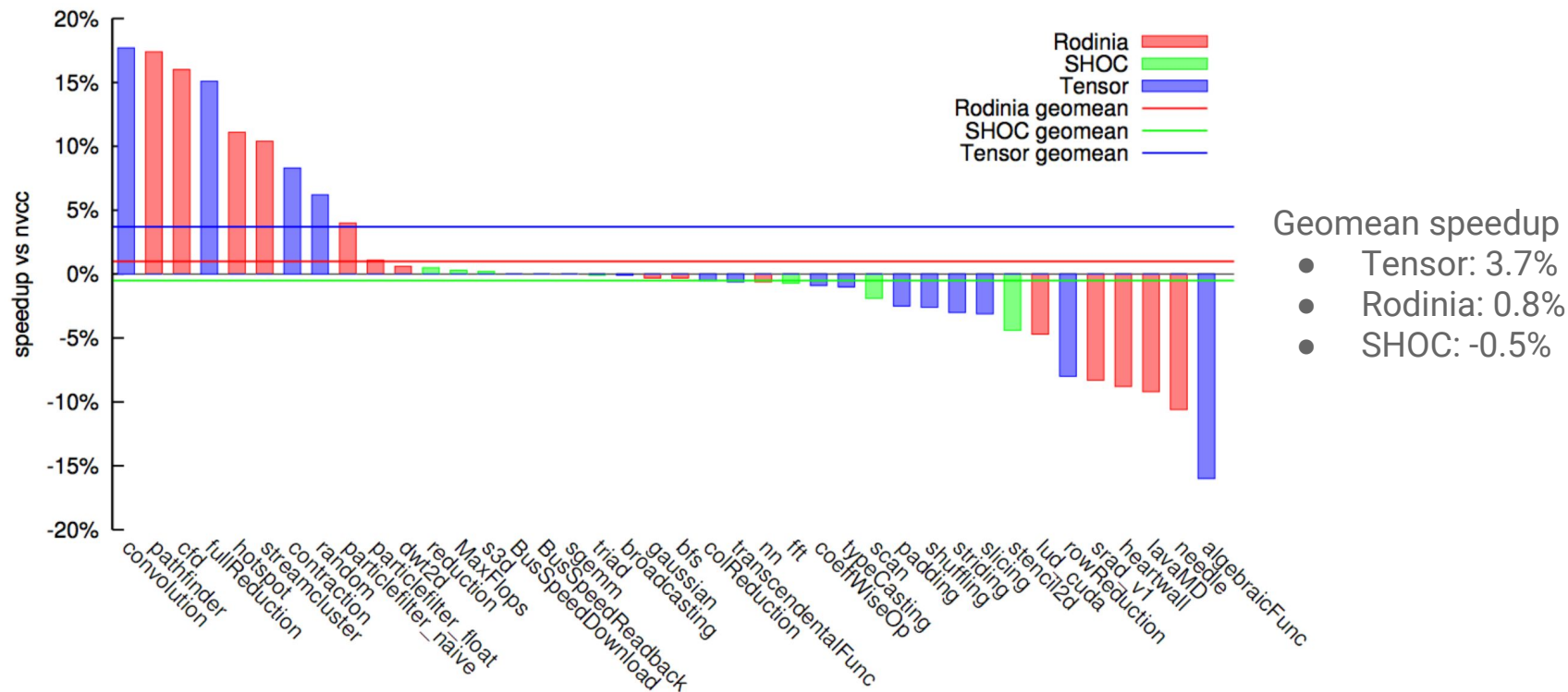
Evaluation

- 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



Conclusions and Future Work

- The missions of gpucc
 - enable compiler research
 - enable industry breakthroughs
- Concepts and insights are applicable to other GPU platforms
- Future work
 - functionality: texture, C++14, more intrinsics, dynamic allocation, ...
 - performance: more optimizations
- Community contributions (bit.ly/gpucc-tutorial)