gpucc: An Open-Source GPGPU Compiler

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

Presenters: Bryce Messmann, Rohit Kandula

Date: 11 December 2019

Background

- The two dominant software platforms for GPUs are CUDA (by NVIDIA) and OpenCL.
- CUDA is widely-used, but it is proprietary!
- Open-source compilers exist for CUDA, but they are not as powerful as NVIDIA's proprietary compiler (which is itself limited).

There is almost no research on CUDA-based compiler optimizations.

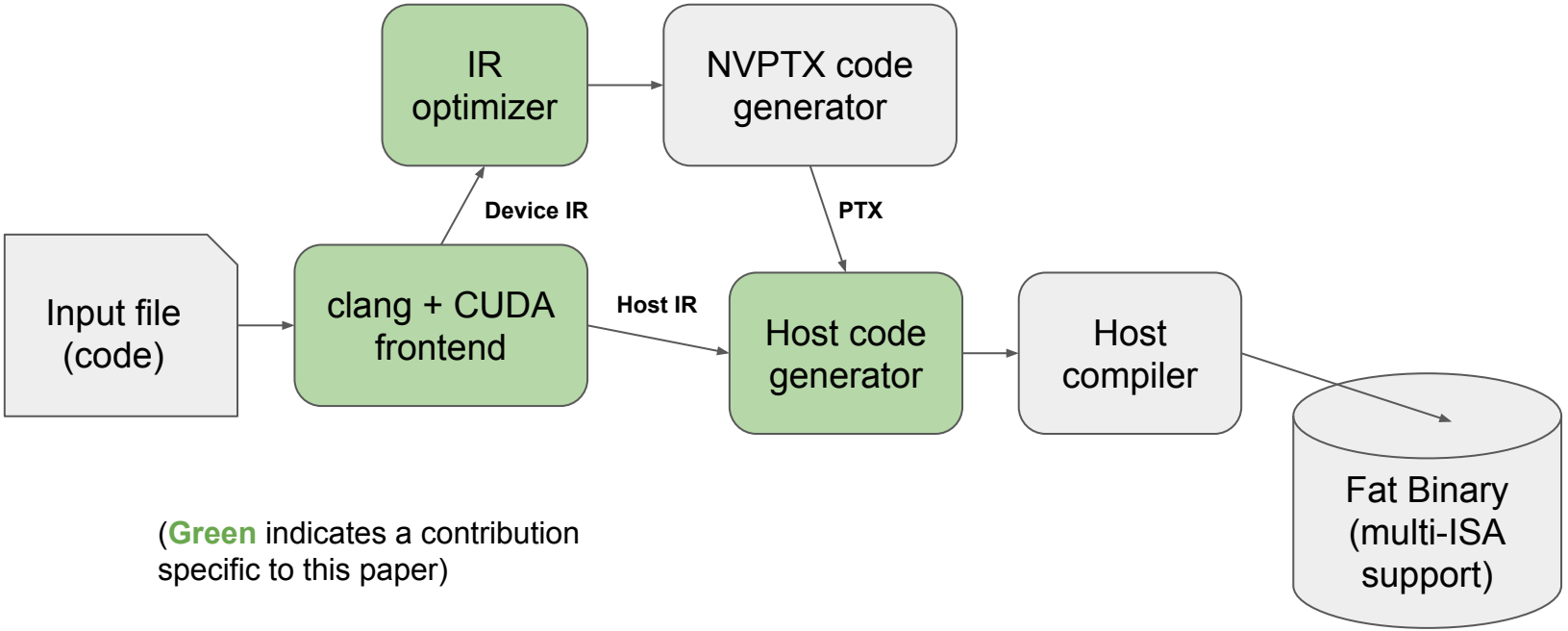
This is a large bottleneck in GPU compiler research.

gpucc

- gpucc is “**a fully functional, open-source, high performance, CUDA-compatible toolchain**, based on LLVM and Clang”

- Targets CUDA, and includes many general and CUDA-specific optimizations

- Compared to NVIDIA’s proprietary nvcc compiler:
  - Significantly faster compile time
  - On-par runtime performance



(**Green** indicates a contribution specific to this paper)

IR-Level Optimizations

- `gpucc` includes many optimizations for runtime performance
  - Standard optimizations (e.g. `-O3` in LLVM) are intended for CPUs
  - The optimizations here target GPUs, and the CUDA platform specifically

- All optimizations are done within LLVM

Optimization 1/6: Loop Unrolling / Function Inlining

- Jumps are expensive on GPUs
  - Multiple “small” simultaneous threads (SIMD execution)
  - Little or no out-of-order execution
  - Pass-in-memory calling adds delay

- gpucc performs more aggressive unrolling and inlining
  - Reduces number of jumps, improving performance
  - Also promotes stack variables to registers (due to constant propagation / SROA)

- Manual options: `#pragma unroll` and `__forceinline__`

Optimization 2/6: Inferring Memory Spaces

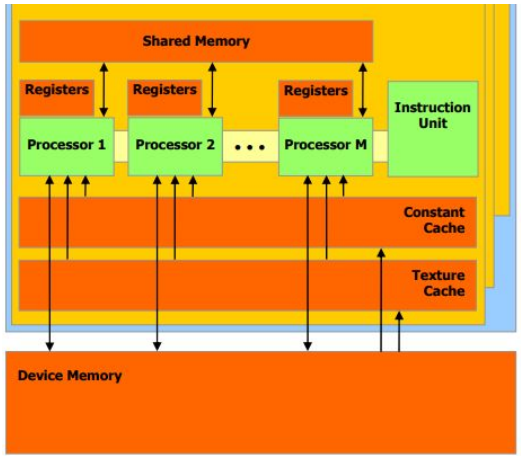
- gpucc can “propagate” memory spaces from initial definition to users

- This knowledge allows for performance optimizations (e.g. `ld.shared` instruction instead of generic `ld` instruction)

Points to shared memory

Also must point to shared memory (for every iteration)!

```
1  __shared__ float a[1024];
2  float *p = a;
3  float *end = a + 1024;
4  while (p != end) {
5    float v = *p;
6    ... // use "v"
7    ++p;
8  }
```



Optimization 3/6: Memory-Space Alias Analysis

- Two pointers to different memory spaces won’t access the same memory
- gpucc can detect this! (using memory-space inference)

- This improves dead-store elimination

Optimization 4/6: Bypassing 64-Bit Division

- NVIDIA GPUs don’t have a “divide” unit
- They perform division using a long sequence of simpler instructions
  - ~20 instructions for 32-bit division
  - ~70 instructions for 64-bit division

- Most division can be done in 32 bits (divisor and dividend are small)
- 64-bit division is automatically converted to 32-bit division when possible

Optimization 5/6: Straight-Line Scalar Optimizations

- Eliminate Partial Redundancy:
  - `(b+1)*n -> b*n + n`
- Useful for Array accesses with unrolled loops
  - Matrix Multiplication
  - Dot product
  - Back Propagation
- Adopted by other backends in LLVM:
  - AMDGPU
  - PowerPC
  - ARM64

Optimization 5/6: Straight-Line Scalar Optimizations

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

Loads a 3x3 submatrix at indices (b,c) in the array a

```
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];
```

after unrolling

Optimization 5/6: Straight-Line Scalar Optimizations

- Inefficiencies:
  - Partial Redundancy:
    - `(b)*n, (b+1)*n, (b+2)*n`
    - `=> (b)*n, (b)*n+n, (b)*n+n+n`
  - Doesn't use `var+immOff` addressing:
    - `var` -> stored in register
    - `immOff` -> 32-bit immediate
    - `c+2+(b+2)*n => (c+(b+2)*n)+2`

- 3 optimizations under this class
  - Pointer Arithmetic Reassociation: to map loads to var+immOff
  - Straight-Line Strength Reduction & Global Reassociation: Eliminate partial redudancy

Optimization 5/6: 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];
```

after unrolling

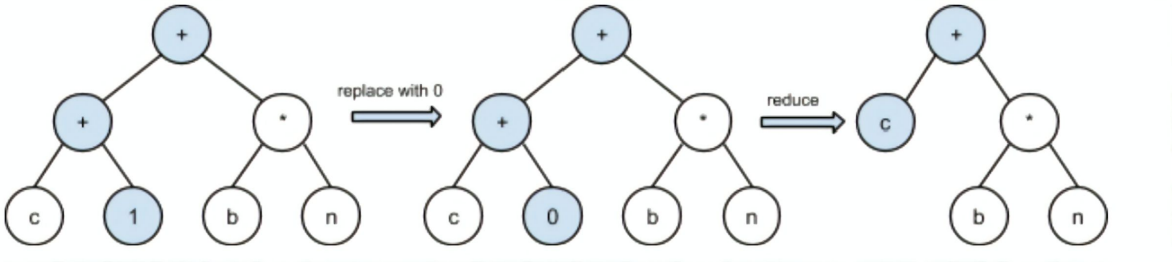
```
p0 = &a[c+b*n];
p1 = &p0[1];
p2 = &p0[2];


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


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

p1..p8 take 1 cycle

p1..p8 can be folded to reg+immOff mode and free up registers

Optimization 5.1/6: Pointer Arithmetic Reassociation

- PAR tries to extract extract additive integer constant from pointer address expression.
  - *variable part + constant offset*
- NVPTX codegen folds to reg+immOff using simple pattern matching
- PAR promotes better CSE
  - &a[c+1+b*n] -> &a[c+b*n] + 1 -> &p0[1] -> ld.f32 [%rd1+4]



Optimization 5.1/6: 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];
```

after unrolling

```
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];
```

after PAR+CSE

Optimization 5.2/6: Straight-Line Strength Reduction

`x = (b+C0)*s;`	`x = b+C0*s;`	`x = &b[C0*s];`
`y = (b+C1)*s;`	`y = b+C1*s;`	`y = &b[C1*s];`
`y = x+(C1-C0)*s`		

Strength reduction forms and replacements.

- Works on dominator paths instead of loops.
- `b,s` integer variables. `C0,C1` are integer constants.
- SLSR identifies candidates in the same form and replaces them

Optimization 5.2/6: Straight-Line Strength Reduction

`x = (b+C0)*s;`	`x = b+C0*s;`	`x = &b[C0*s];`
`y = (b+C1)*s;`	`y = b+C1*s;`	`y = &b[C1*s];`
`y = x+(C1-C0)*s`		

Strength reduction forms and replacements.

- `C1-C0` is often -1 or 1 or a fixed stride
- Increases dependencies and could hurt ILP
  - NVIDIA K40 doesn't use out-of-order execution and has 2 integer units
  - Not so much of a problem as scheduling window is small

Optimization 5.1/6: Straight-Line Strength Reduction

```
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];
```

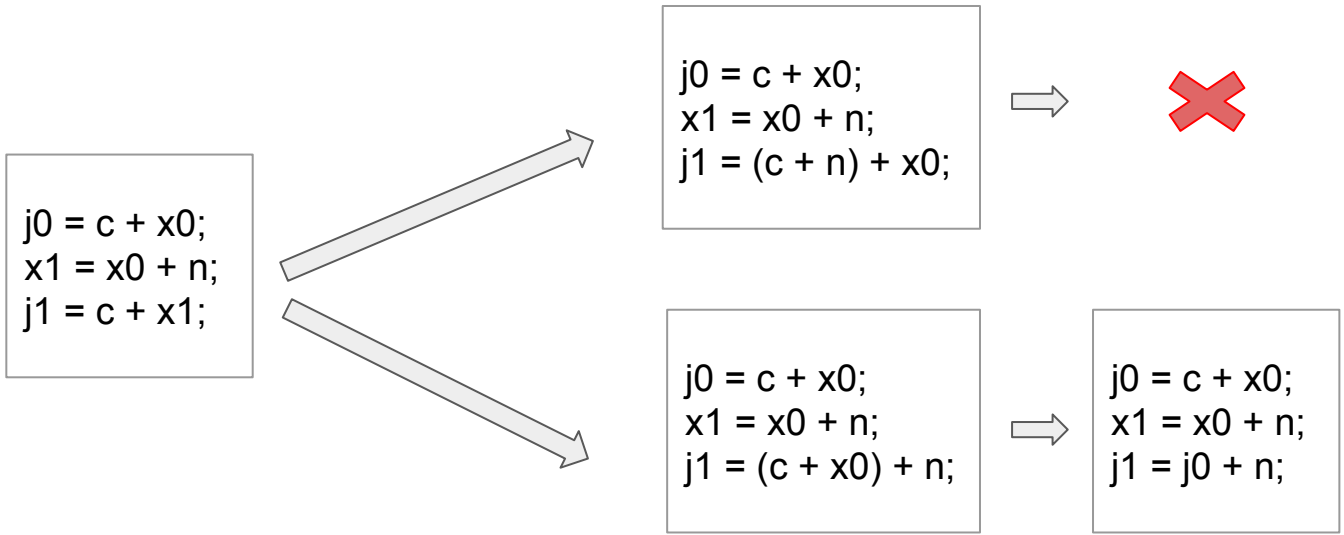
after PAR+CSE

```
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];
```

after SLSR

Optimization 5.3/6: Global Reassociation

- Reorders commutative operations for better redundancy elimination
- Similar to
  - Enhanced Scalar Replacement but linear time complexity
  - Reassociative pass but “global”



Optimization 5.3/6: 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];
```

after SLSR

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

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

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

global reassociation

Optimization 5.3/6: Global Reassociation

Stack of dominating instructions that compute *E*

Pre-order traversal guarantees all dominators of *I* than computer *E* are in the stack

Keeps popping until top of the stack dominates *I*

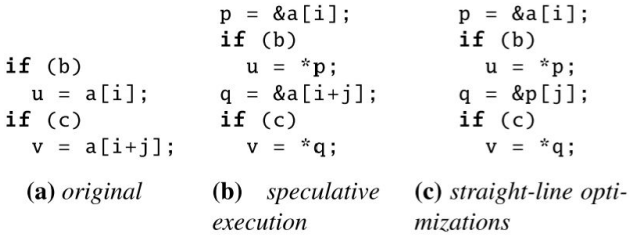
Algorithm : *Global reassociation.*

Data: dominators(E) maintains a list of observed instructions that compute E
Input : the original program P and a schedule S
Output: the specialized program
GlobalReassociation(F)
  foreach instruction I in pre-order of $domtree(F)$ **do**
    // + is used to represent any commutative operator.
    if $I = a + b$ **then**
      $E \leftarrow \text{expr}(I)$
      dominators$[E] \leftarrow$ dominators$[E] + I$
    if $I = (a + b) + c$ and $a + b$ is used only once **then**
      $E_1 \leftarrow \text{expr}(a + c)$
      $D \leftarrow \text{ClosestMatchingDom}(E_1, I)$
      if $D \neq$ nil **then**
        Rewrite I to $D + b$
      else
        $E_2 \leftarrow \text{expr}(b + c)$
        $D \leftarrow \text{ClosestMatchingDom}(E_2, I)$
        if $D \neq$ nil **then**
          Rewrite I to $D + a$
ClosestMatchingDom(E, I)
  $D \leftarrow$ dominators$[E]$
  while $D \neq \emptyset$ and $\neg$dominate(D, I) **do**
    popback(D)
  if $D = \emptyset$ **then**
    return nil
  return back(D)

- Pruning guarantees linear time
- Safe due to pre-order traversal

Optimization 6/6: Speculative Execution

- Straight-Line Scalar Optimizations do not work on non dominating instructions
- Solution: Hoist side effect free instructions from conditional basic blocks
- Increases dominance and likelihood of SLSO
- Also promotes predication
  - Reduces conditional basic blocks small which in turn triggers predicated execution



```
if (b)
  u = a[i];
if (c)
  v = a[i+j];
```

(a) *original*

```
p = &a[i];
if (b)
  u = *p;
q = &a[i+j];
if (c)
  v = *q;
```

(b) *speculative execution*

```
p = &a[i];
if (b)
  u = *p;
q = &p[j];
if (c)
  v = *q;
```

(c) *straight-line optimizations*

Evaluation

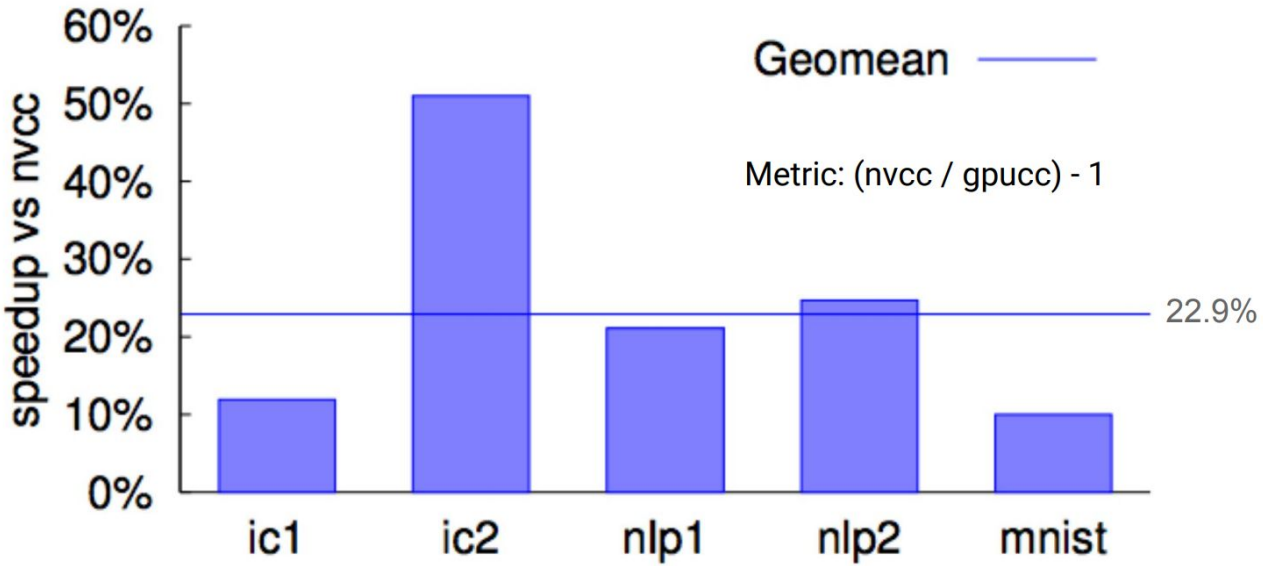
Benchmarks:

- **Rodinia**: based on various “real-world” applications (data mining, medical imaging, etc.)
- **SHOC**: scientific computing benchmarks
- **Tensor**: benchmarks using the Tensor module in Eigen 3.0 (C++ linear algebra library)

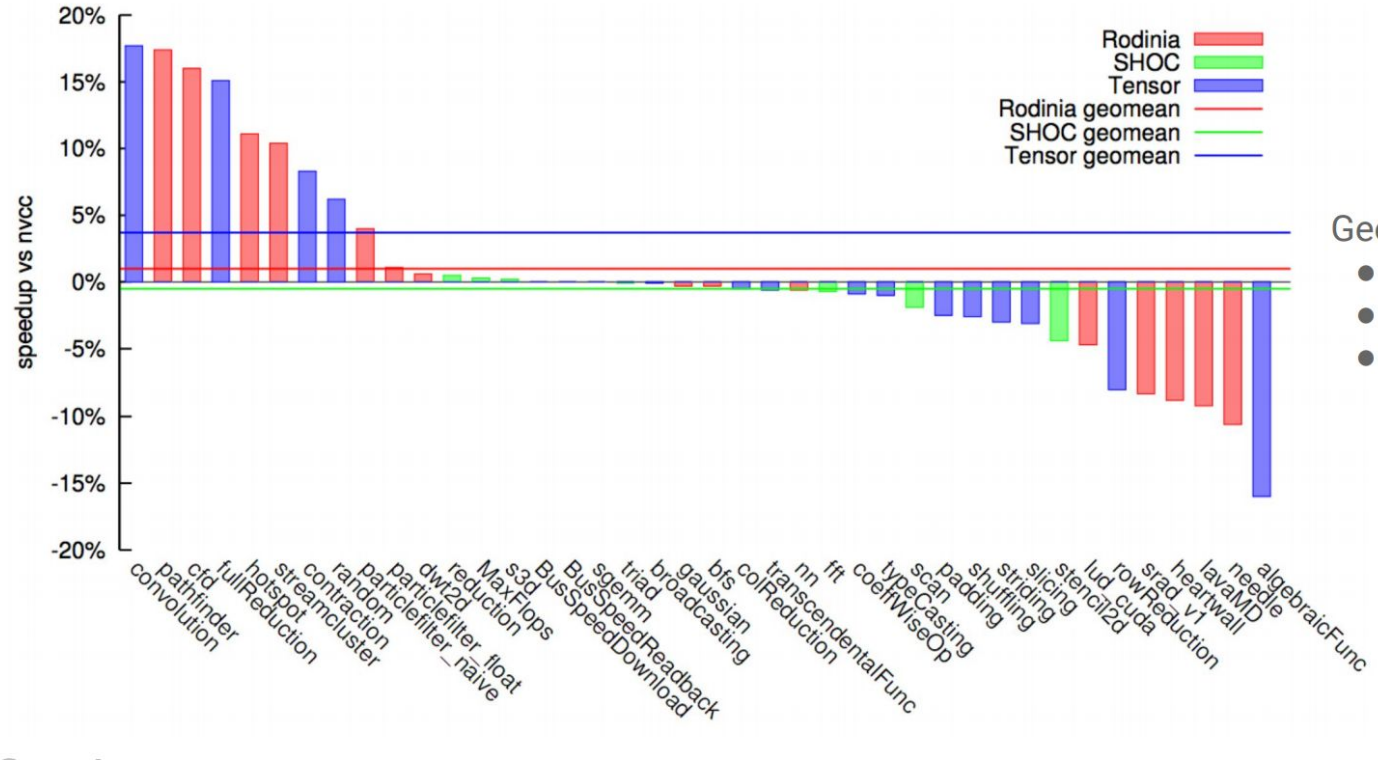
Metrics:

- Runtime performance comparison
- Compilation time
- Effects of optimizations

Performance on End-to-End Benchmarks



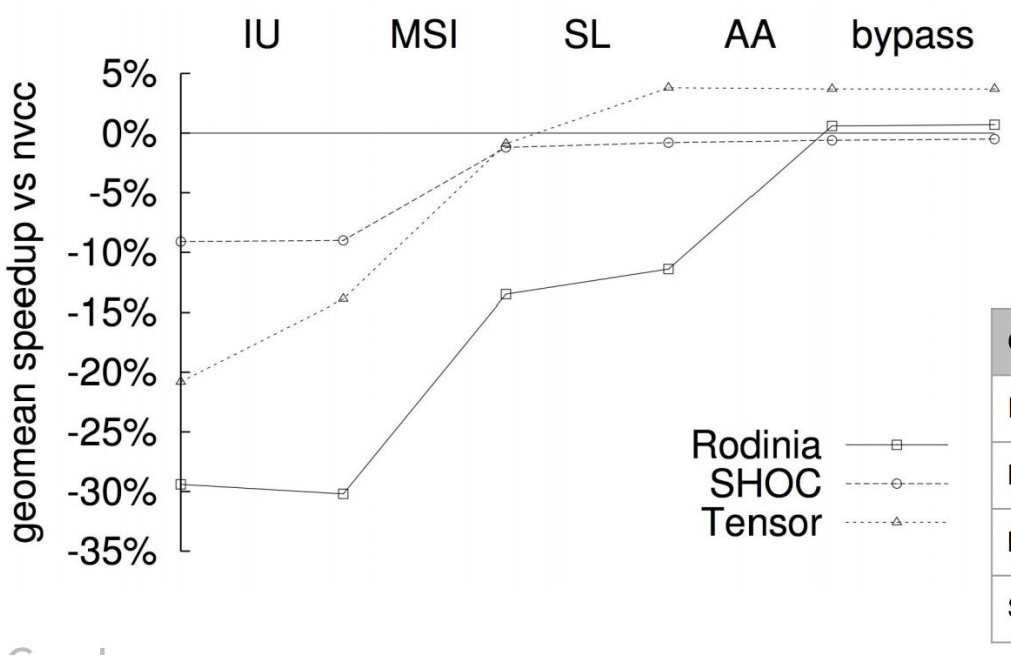
Performance on Open-Source Benchmarks



Geomean speedup

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

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%

Conclusion

- gpucc
  - Open Source
  - High performance
  - CUDA compiler
- Faster (or on-par) Code w.r.t nvcc
- Faster or comparable compilation times
- Now part of clang/LLVM
- Research Opportunities:
  - Frontend: more language extensions? Other languages?
  - Backend: more architectures (e.g., CUDA on AMDGPU)? Target SASS?
  - Debugging: better profiling? Static analysis?
  - Optimizer: more optimizations?

Questions?