Floating-point Precision vs Performance Trade-offs

Wei-Fan Chiang

School of Computing, University of Utah

Problem Statement

- Accelerating computations using graphical processing units has made significant advances, finding applications in systems ranging from large-scale HPC systems to mobile platforms.

[0] Photo Courtesy of Nvidia geforce.com, ORNL, Cray Inc., and Google Inc.

Problem Statement

- **Performance increases must not come at the expense of precision!**

[0] Photo Courtesy of KSL.com and Desert News

Problem Statement

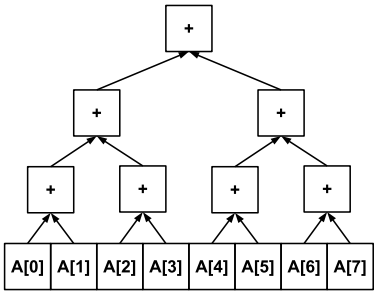
- We have evidence that this is happening!
  - We will present details pertaining to many important GPU primitives where this tradeoffs appears not to have been precisely characterized.
- State of the art: Tool support to make these tradeoffs to assist programmers writing specific applications lacking!

Different Connotations of Performance/Precision Tradeoffs

- Performance : Average elapsed time for typical inputs
- Precision : Numerical uncertainty of outputs (e.g. interval of uncertainty) relative to input uncertainty
- Approaches to achieve this tradeoffs:
  - Algorithms that terminate with coarse approximations (e.g., of iterations).
  - **Our main interest so far:** Algorithms that fundamentally rearrange the computations
    - e.g., reduce the number of operations
    - minimize synchronization.
    - change the divide-and-conquer schemes (e.g. like in CILK)

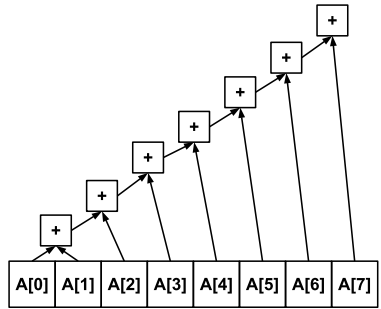
Different Connotations of Performance/Precision Tradeoffs

Balance



V.S.

Imbalance



Different Connotations of Performance/Precision Tradeoffs

- Input 1: 1024-element 32-bit floating-point array. Each element is $0.1f$.
  - The result computed in infinite precision is $0.1 \times 1024 = 102.4$.
- Input 2: 1024-element 32-bit floating-point array. 512 element are $100.3f$, and 512 element are $0.3f$.
  - Elements are randomly permuted.
  - The result computed in infinite precision is $(100.3 + 0.3) \times 512 = 51507.2$.

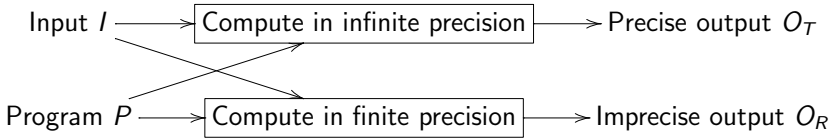
Kernel	Input	Machine Result	Difference	Performance
Balance	input 1	102.4000015259	1.5259 e-06	**3.53** μs
Imbalance	input 1	102.3990097046	**-9.9030 e-04**	1.34 μs
Balance	input 2	51507.203125	0.003125	**3.54** μs
Imbalance	input 2	51507.67578125	**0.47578125**	1.30 μs

- We only show two concrete inputs here. How about other inputs?

Goal of Our Work

- Short term goal: Provide tools to analyze precision/performance tradeoffs
- Longer term goal:
  - Evolve guidelines that help achieve these tradeoffs by construction.
  - Integrate tools into usable CUDA programming / verification flows.
  - Demonstrate on a wide variety of practical examples.

Basics of Floating-point Precision Analysis



- Informal analysis does not suffice. Absolute and relative errors are used to indicate the degree of imprecision.
- Absolute (abs.) error of the program output is:

$$O_R - O_T$$

- Relative (rel.) error of the program output is:

$$(O_R - O_T)/O_T$$

Related Work: Precision Profiling for Sequential Programs

- Dynamic analysis v.s. Interval/affine arithmetic based analysis
- Dynamic analysis [Hollingsworth, 2011] [Benz and Hack, 2012]:
  - Using very wide floating-point numbers to approximate real (infinite precision) numbers.
  - Pros: Fast computation of abs. or rel. errors.
  - Pros: Detects catastrophic cancellation and points out the root-cause instruction(s).
  - Cons: Requires concrete inputs. Not a generic analysis.

Related Work: Precision Profiling for Sequential Programs

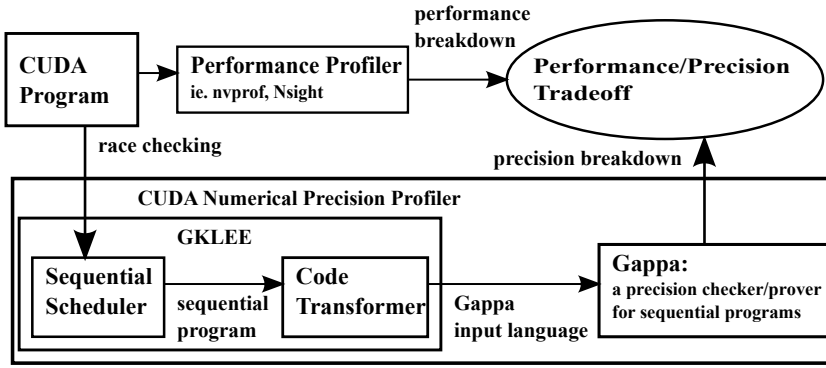
- Interval/affine arithmetic based analysis:
  - Supports assigning intervals (value ranges) as program inputs.
  - Executes each operator in interval/affine arithmetic.
  - Analyzes the worst-case abs. or rel. errors for outputs.
  - Pros: Generic analysis.
  - Cons: Over-approximation.
- Interval/affine arithmetic based analyzer:
  - Gappa [Melquiond, 2006], Gappa++ [Linderman, 2011], FLUCTUAT [Putot and Goubault, 2009], and SmartFloat [Kuncak, 2011].

Our work: Precision Profiling for CUDA Programs

- We currently focus on CUDA programs.
- We only focus on race-free programs.
- We do interval/affine arithmetic based analysis.
- We use Gappa as our back-end analyzer.
- We sequentialize CUDA programs.
- Race checking and program sequentialization is done by GKLEE [Li, 2012].

Our work: Precision Profiling for CUDA Programs

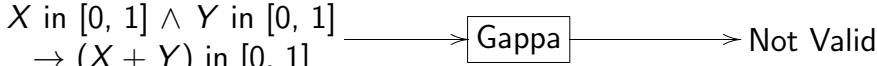
- Tool Flow:

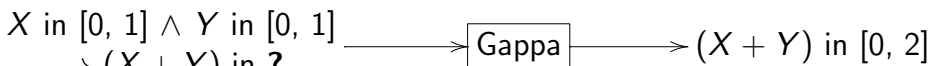


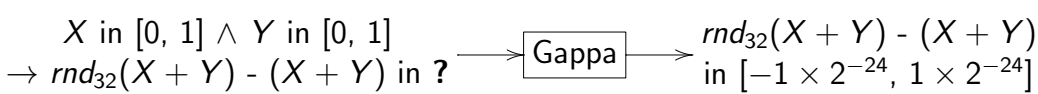
Technical Details: Gappa

- Gappa is a saturation and interval arithmetic based numerical constraint prover.
- Gappa takes logic formulas as input.
- Gappa checks validity or solves input logic formulas.
- Gappa provides built-in IEEE-754 rounding functions for estimating rounding uncertainty.

Technical Details: Gappa







Technical Details: Sequential C to Gappa Input Language

Sequential C Program:
Inputs: W,X,Y = [0, 1]

```
float a, Z;
a = W + X;
Z = a + Y;
check_abs_error(Z);
```

$\Rightarrow$

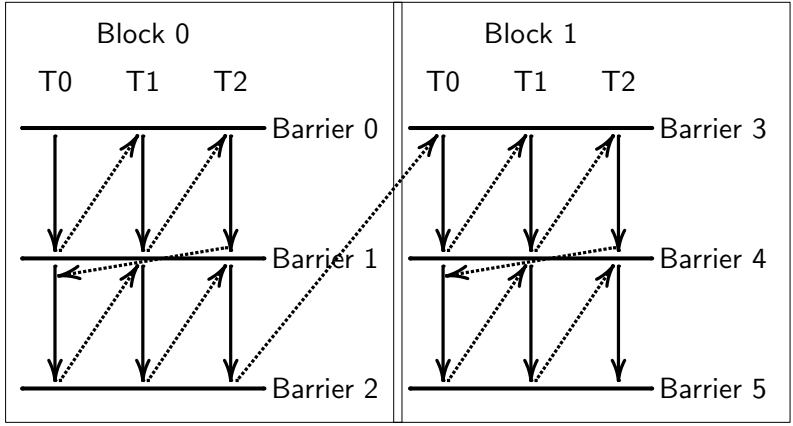
Logic Formula

$$Z_T = ((W + X) + Y) \wedge$$
$$Z_R = rnd_{32}(rnd_{32}(W + X) + Y) \wedge$$
$$W \text{ in } [0, 1] \wedge X \text{ in } [0, 1] \wedge Y \text{ in } [0, 1]$$
$$\rightarrow (Z_R - Z_T) \text{ in } \mathbf{?}$$

- Each variable (Z) for precision checking is represented by an expression tree.
- Expression tree is composed by inputs as leaves and operators as internal nodes.
- Each operator is treated as an uninterpreted function.

Technical Details: Sequentializing CUDA Programs

- GKLEE explores one canonical schedule to check data race for a CUDA program.
- We sequentialize the CUDA program by that canonical schedule.

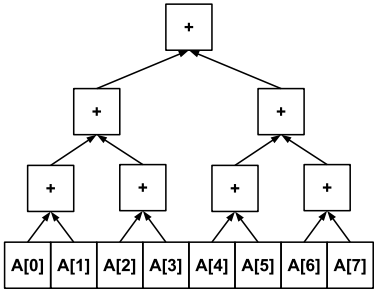


Experimental Results

- We collect sets of GPU programs. Each set contains different implementations of the same computation.
  - Two implementations of reduction.
  - Two implementations of scan (prefix-sum).
- We assume that, by giving the same input to implementations in the same set, they are designed to generate the same output.
- We measure absolute error to on output to indicate floating-point imprecision.
- GPU: Nvidia GeForce GTX 480.
- We assign both 32-bit and 64-bit inputs in all our experiments.

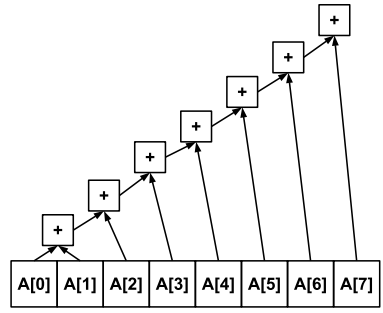
Experimental Results: Reduction

Balance



V.S.

Imbalance



- Balance reduction:
  - The *reduce3* kernel in CUDA SDK 5.0.
- Imbalance reduction:
  - Implemented by *atomicAdd* operation. The *atomicAdd* for 64-bit floating-point number is implemented by *atomicCAS*.

Experimental Results: Reduction

- 32-bit Floating-point Number:
- Each element is assigned as [-100.0f, 100.0f] when measuring precision.

Kernel	Array Size	Abs. Error	Performance
Balance	512	[-0.0215, 0.0215]	3.43 μs
Imbalance	512	[-0.5767, 0.5767]	1.37 μs
Balance	1024	[-0.0508, 0.0508]	3.53 μs
Imbalance	1024	[-2.2994, 2.2994]	1.37 μs
Balance	2048	[-0.0977, 0.0977]	3.54 μs
Imbalance	2048	Timeout (10 hours)	1.38 μs

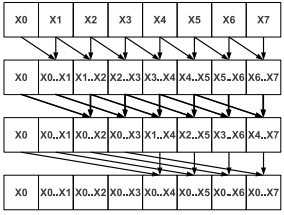
Experimental Results: Reduction

- 64-bit Floating-point Number:
- Each element is assigned as [-100.0, 100.0] when measuring precision.

Kernel	Array Size	Abs. Error	Performance
Balance	512	[-4.0018 e-11, 4.0018 e-11]	3.65 μs
Imbalance	512	[-0.0020, 0.0020]	6.805 *ms*
Balance	1024	[-9.4587 e-11, 9.4587 e-11]	3.83 μs
Imbalance	1024	[-0.0039, 0.0039]	14.867 *ms*
Balance	2048	[-1.8190 e-10, 1.8190 e-10]	3.83 μs
Imbalance	2048	Timeout (10 hours)	30.5575 *ms*

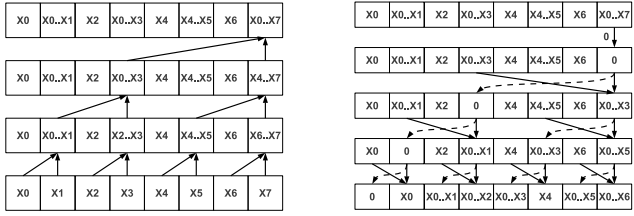
Experimental Results: Scan

Naive scan



V.S.

Work-efficient Scan



- Origin from GPU Gem3. [Nguyen, 2007]
- Three scan implementations: naive, work-efficient, and work-efficient with zero bank conflict.

Experimental Results: Scan

- Each element is assigned as [-100.0, 100.0] when measuring precision.

Kernel	Array Size	32-bit Precision	64-bit Precision
Naive	512	[-0.0195, 0.0195]	[-3.6366 e-11, 3.6366e-11]
Work-efficient	512	[-0.0322, 0.0322]	[-0.0019, 0.0019]
Work-efficient (bf-free)	512	[-0.0322, 0.0322]	[-0.0019, 0.0019]
Naive	1024	[-0.0449, 0.0449]	[-8.3652 e-11, 8.3652 e-11]
Work-efficient	1024	[-0.0703, 0.0703]	[-0.0039, 0.0039]
Work-efficient (bc-free)	1024	[-0.0703, 0.0703]	[-0.0039, 0.0039]

Limitations

- Scalability.
- Identify the core reason of imprecision.

Conclusions

- **Performance increases must not come at the expense of precision!**
- Our precision profiling tool helps identify the performance/precision tradeoffs for GPU programming.

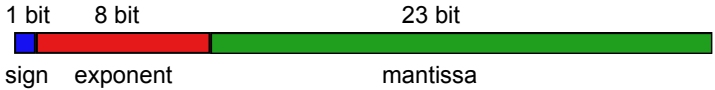
Future Work

- Evolve guidelines that help achieve performance/precision tradeoffs by our profiling tool.
- Improve scalability of our profiling tool.
- Automatically identify the core reason of floating-point imprecision.
- Apply our precision profiler to practically CUDA library.

Thank you.
Questions?

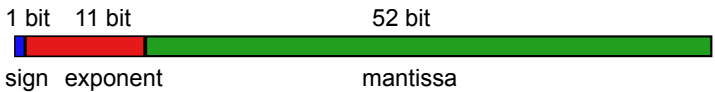
Basics of Floating-point Precision Analysis

- IEEE-754 standard:
  - 32-bit floating-point number:



$$\text{if } sign = 0 \quad : \quad mantissa \times 2^{exponent-127}$$
$$\text{if } sign = 1 \quad : \quad -1 \times mantissa \times 2^{exponent-127}$$

  - 64-bit floating-point number:



$$\text{if } sign = 0 \quad : \quad mantissa \times 2^{exponent-1023}$$
$$\text{if } sign = 1 \quad : \quad -1 \times mantissa \times 2^{exponent-1023}$$

Basics of Floating-point Precision Analysis

- Using abs. or rel. for measuring floating-point imprecision has some pros. and cons.
- Using absolute error:
  - Pros: Easy to understand/compute.
  - Cons: Sensitive to magnitude of operands.
- Using relative error:
  - Pros: Comparatively insensitive to magnitude of operands.
  - Cons: Non-intuitive.
  - Cons:
    - Potential division by zero *IF using interval/affine arithmetic, e.g.:*

$$X/Y. \quad -1 \leq Y \leq 1$$

Basics of Floating-point Precision Analysis: Methods for Precision Estimation

- Interval arithmetic
  - Represents each value as a range and operates on ranges. Each range (interval) is represented by an upper and a lower bound.
  - Pros: Relatively simpler.
  - Cons: Often overly pessimistic.
    - X in [-1, 1]. X - X = [-2, 2].
- Affine arithmetic
  - Similar to interval arithmetic. But represents each range by an polynomial.
  - Pros: Relatively less pessimistic.
  - Cons: More involved.
    - $X = X_0 + X_1 * \epsilon_1$. X - X = 0.

Basics of Floating-point Precision Analysis

- IEEE-754/854 standard:
  - Each operator computes the result exactly (in infinite precision) then rounds.
  - For a binary operator $\oplus$ and its two operands, op_1 and op_2, $\oplus$ will compute the exact result, V_T, then round to the rounded result V_R.
  - The relationship between V_T and V_R is:

$$V_R = V_T \times (1 + \epsilon_M \times \alpha). \quad -1 \leq \alpha \leq 1$$

ϵ_M, machine epsilon, is a constant which only depends on the size of mantissa.