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.

Photo Courtesy of Nvidia geforce.com, ORNL, Cray Inc., and Google Inc.
Problem Statement

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

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

v.s.
Different Connotations of Performance/Precision Tradeoffs

- Input: 1024-element 32-bit floating-point array. Each element is $0.1f$.
  - The result computed in infinite precision is $0.1 \times 1024 = 102.4$.

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Machine Result</th>
<th>Difference</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Balance</td>
<td>102.4000015259</td>
<td>1.5259 e-06</td>
<td>3.53 (\mu s)</td>
</tr>
<tr>
<td>Imbalance</td>
<td>102.3990097046</td>
<td>-9.9030 e-04</td>
<td>1.34 (\mu s)</td>
</tr>
</tbody>
</table>

- We show only one concrete input 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:
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:

- **CUDA Program**
- **Performance Profiler**
  - ie. nvprof, Nsight
- **Performance/Precision Tradeoff**
- **CUDA Numerical Precision Profiler**
  - **Sequential Scheduler**
  - **GKLEE**
  - **Code Transformer**
  - **Gappa:**
    - a precision checker/prover for sequential programs

Performance breakdown:

- Race checking

Precision breakdown:

Gappa input language
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

\[ X \in [0, 1] \land Y \in [0, 1] \rightarrow (X + Y) \in [0, 1] \quad \text{Gappa} \rightarrow \text{Not Valid} \]

\[ X \in [0, 1] \land Y \in [0, 1] \rightarrow (X + Y) \in ? \quad \text{Gappa} \rightarrow (X + Y) \in [0, 2] \]

\[ X \in [0, 1] \land Y \in [0, 1] \rightarrow \text{rnd}_{32}(X + Y) - (X + Y) \in ? \quad \text{Gappa} \rightarrow \text{rnd}_{32}(X + Y) - (X + Y) \in [-1 \times 2^{-24}, 1 \times 2^{-24}] \]
Sequential C Program:
Inputs: W,X,Y = [0, 1]
float a, Z;
a = W + X;
Z = a + Y;
check_abs_error(Z);

\[ Z_T = ((W + X) + Y) \land \]
\[ Z_R = \text{rnd}_{32}(\text{rnd}_{32}(W + X) + Y) \land \]
\[ W \text{ in [0, 1]} \land X \text{ in [0, 1]} \land Y \text{ in [0, 1]} \]
\[ \rightarrow (Z_R - Z_T) \text{ in ?} \]

- 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 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.

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Array Size</th>
<th>Abs. Error</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Balance</td>
<td>512</td>
<td>([-0.0215, 0.0215])</td>
<td>3.43 µs</td>
</tr>
<tr>
<td>Imbalance</td>
<td>512</td>
<td>([-0.5767, 0.5767])</td>
<td>1.37 µs</td>
</tr>
<tr>
<td>Balance</td>
<td>1024</td>
<td>([-0.0508, 0.0508])</td>
<td>3.53 µs</td>
</tr>
<tr>
<td>Imbalance</td>
<td>1024</td>
<td>([-2.2994, 2.2994])</td>
<td>1.37 µs</td>
</tr>
<tr>
<td>Balance</td>
<td>2048</td>
<td>([-0.0977, 0.0977])</td>
<td>3.54 µs</td>
</tr>
<tr>
<td>Imbalance</td>
<td>2048</td>
<td>Timeout (10 hours)</td>
<td>1.38 µs</td>
</tr>
</tbody>
</table>
Experimental Results: Reduction

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

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Array Size</th>
<th>Abs. Error</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Balance</td>
<td>512</td>
<td>([-4.0018 \times 10^{-11}, 4.0018 \times 10^{-11}]</td>
<td>3.65 (\mu s)</td>
</tr>
<tr>
<td>Imbalance</td>
<td>512</td>
<td>([-0.0020, 0.0020]]</td>
<td>6.805 (ms)</td>
</tr>
<tr>
<td>Balance</td>
<td>1024</td>
<td>([-9.4587 \times 10^{-11}, 9.4587 \times 10^{-11}]</td>
<td>3.83 (\mu s)</td>
</tr>
<tr>
<td>Imbalance</td>
<td>1024</td>
<td>([-0.0039, 0.0039]]</td>
<td>14.867 (ms)</td>
</tr>
<tr>
<td>Balance</td>
<td>2048</td>
<td>([-1.8190 \times 10^{-10}, 1.8190 \times 10^{-10}]</td>
<td>3.83 (\mu s)</td>
</tr>
<tr>
<td>Imbalance</td>
<td>2048</td>
<td>Timeout (10 hours)</td>
<td>30.5575 (ms)</td>
</tr>
</tbody>
</table>
Experimental Results: Scan

Naive scan

\[
\begin{array}{cccccccc}
X0 & X1 & X2 & X3 & X4 & X5 & X6 & X7 \\
X0 & X0.X1 & X1.X2 & X2.X3 & X3.X4 & X4.X5 & X5.X6 & X6.X7 \\
X0 & X0.X1 & X0.X2 & X0.X3 & X1.X4 & X2.X5 & X3.X6 & X4.X7 \\
X0 & X0.X1 & X0.X2 & X0.X3 & X0.X4 & X0.X5 & X0.X6 & X0.X7 \\
\end{array}
\]

V.S.

Work-efficient Scan

\[
\begin{array}{cccccccc}
X0 & X0.X1 & X2 & X0.X3 & X4 & X4.X5 & X6 & X0.X7 \\
X0 & X0.X1 & X2 & X0.X3 & X4 & X4.X5 & X6 & X4.X7 \\
X0 & X0.X1 & X2 & X0.X3 & X4 & X4.X5 & X6 & X6.X7 \\
X0 & X1 & X2 & X3 & X4 & X5 & X6 & X7 \\
\end{array}
\]

Origin from GPU Gem3. [Nguyen, 2007]

Three scan implementations: naive, work-efficient, and work-efficient with zero bank conflict.
Each element is assigned as [-100.0, 100.0] when measuring precision.

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Array Size</th>
<th>32-bit Precision</th>
<th>64-bit Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naive</td>
<td>512</td>
<td>[-0.0195, 0.0195]</td>
<td>[-3.6366 e-11, 3.6366e-11]</td>
</tr>
<tr>
<td>Work-efficient</td>
<td>512</td>
<td>[-0.0322, 0.0322]</td>
<td>[-0.0019, 0.0019]</td>
</tr>
<tr>
<td>Work-efficient (bf-free)</td>
<td>512</td>
<td>[-0.0322, 0.0322]</td>
<td>[-0.0019, 0.0019]</td>
</tr>
<tr>
<td>Naive</td>
<td>1024</td>
<td>[-0.0449, 0.0449]</td>
<td>[-8.3652 e-11, 8.3652 e-11]</td>
</tr>
<tr>
<td>Work-efficient</td>
<td>1024</td>
<td>[-0.0703, 0.0703]</td>
<td>[-0.0039, 0.0039]</td>
</tr>
<tr>
<td>Work-efficient (bc-free)</td>
<td>1024</td>
<td>[-0.0703, 0.0703]</td>
<td>[-0.0039, 0.0039]</td>
</tr>
</tbody>
</table>
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:
    - 1 bit sign
    - 8 bit exponent
    - 23 bit mantissa
    - $\text{if } sign = 0 : \text{mantissa } \times 2^{\text{exponent} - 127}$
    - $\text{if } sign = 1 : -1 \times \text{mantissa } \times 2^{\text{exponent} - 127}$

- 64-bit floating-point number:
  - 1 bit sign
  - 11 bit exponent
  - 52 bit mantissa
  - $\text{if } sign = 0 : \text{mantissa } \times 2^{\text{exponent} - 1023}$
  - $\text{if } sign = 1 : -1 \times \text{mantissa } \times 2^{\text{exponent} - 1023}$
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.:
      \[ \frac{X}{Y}, \quad -1 \leq Y \leq 1 \]
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 a polynomial.
  - Pros: Relatively less pessimistic.
  - Cons: More involved.
    - $X = X_0 + X_1 \times \epsilon_1$. $X - X = 0$. 

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:
    \[
    V_R = V_T \times (1 + \epsilon_M \times \alpha). \quad -1 \leq \alpha \leq 1
    \]
    
    \( \epsilon_M \), machine epsilon, is a constant which only depends on the size of mantissa.
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 elements are $100.3f$, and 512 elements are $0.3f$.
  - Elements are randomly permuted.
  - The result computed in infinite precision is $(100.3 + 0.3) \times 512 = 51507.2$.

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Input</th>
<th>Machine Result</th>
<th>Difference</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Balance</td>
<td>input 1</td>
<td>102.4000015259</td>
<td>1.5259 e-06</td>
<td>3.53 $\mu s$</td>
</tr>
<tr>
<td>Imbalance</td>
<td>input 1</td>
<td>102.3990097046</td>
<td>-9.9030 e-04</td>
<td>1.34 $\mu s$</td>
</tr>
<tr>
<td>Balance</td>
<td>input 2</td>
<td>51507.203125</td>
<td>0.003125</td>
<td>3.54 $\mu s$</td>
</tr>
<tr>
<td>Imbalance</td>
<td>input 2</td>
<td>51507.67578125</td>
<td>0.47578125</td>
<td>1.30 $\mu s$</td>
</tr>
</tbody>
</table>

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