L9: Project Discussion and Floating Point Issues

Outline

- Discussion of semester projects
- Floating point
  - Mostly single precision until recent architectures
  - Accuracy
  - What's fast and what's not
  - Reading:

  Ch 6/7 in Kirk and Hwu, http://courses.ece.illinois.edu/ece498/al/textbook/Chapter6-FloatingPoint.pdf

  NVIDA CUDA Programmer's Guide, Appendix C

Project Proposal (due 3/8)

- Team of 2-3 people
  - Please let me know if you need a partner
- Proposal Logistics:
  - Significant implementation, worth 50% of grade
  - Each person turns in the proposal (should be same as other team members)
- Proposal:
  - 3-4 page document (11pt, single-spaced)
  - Submit with handin program:

    "handin CS6235 prop <pdf-file>"

Project Parts (Total = 50%)

- Proposal (5%)
  - Short written document, next few slides
- Design Review (10%)
  - Oral, in-class presentation 2 weeks before end
- Presentation and Poster (15%)
  - Poster session last week of class, dry run week before
- Final Report (20%)
  - Due during finals – no final for this class

Project Schedule

- Thursday, March 8, Proposals due
- Monday, April 2, Design Reviews
- Wednesday, April 18, Poster Dry Run
- Monday, April 23, In-Class Poster Presentation
- Wednesday, April 25, Guest Speaker

Content of Proposal

I. Team members: Name and a sentence on expertise for each member

II. Problem description
- What is the computation and why is it important?
- Abstraction of computation: equations, graphic or pseudo-code, no more than 1 page

III. Suitability for GPU acceleration
- Amdahl's Law: describe the inherent parallelism. Argue that it is close to 100% of computation. Use measurements from CPU execution of computation if possible.
- Synchronization and Communication: Discuss what data structures may need to be protected by synchronization, or communication through host.
- Copy Overhead: Discuss the data footprint and anticipated cost of copying to/from host memory.

IV. Intellectual Challenges
- Generally, what makes this computation worthy of a project?
- Point to any difficulties you anticipate at present in achieving high speedup

Projects – How to Approach

- Some questions:
  1. Amdahl's Law: target bulk of computation and can profile to obtain key computations…
  2. Strategy for gradually adding GPU execution to CPU code while maintaining correctness
  3. How to partition data & computation to avoid synchronization?
  4. What types of floating point operations and accuracy requirements?
  5. How to manage copy overhead? Can you overlap computation and copying?

Floating Point

- Incompatibility
  - Most scientific apps are double precision codes!
  - Graphics applications do not need double precision (criteria are speed and whether the picture looks ok, not whether it accurately models some scientific phenomena).
  - -> Prior to GTX and Tesla platforms, double precision floating point not supported at all. Some inaccuracies in single-precision operations.
- In general
  - Double precision needed for convergence on fine meshes, or large set of values
  - Single precision ok for coarse meshes

Some key features

- Hardware intrinsics implemented in special functional units faster but less precise than software implementations
- Double precision slower than single precision, but new architectural enhancements have increased its performance
- Measures of accuracy
  - IEEE compliant
  - In terms of "unit in the last place" (ulps): the gap between two floating-point numbers nearest to x, even if x is one of them

What is IEEE floating-point format?

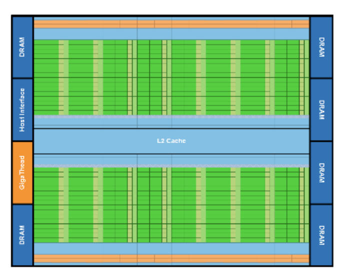
- A floating point binary number consists of three parts:
  - sign (S), exponent (E), and mantissa (M).
  - Each (S, E, M) pattern uniquely identifies a floating point number.
- For each bit pattern, its IEEE floating-point value is derived as:
  - value = $(-1)^S * M * \{2^E\}$, where $1.0 \le M < 10.0_B$
- The interpretation of S is simple: S=0 results in a positive number and S=1 a negative number.

Single Precision vs. Double Precision

- Platforms of compute capability 1.2 and below only support single precision floating point
- Some systems (GTX, 200 series, Tesla) include double precision, but much slower than single precision
  - A single dp arithmetic unit shared by all SPs in an SM
  - Similarly, a single fused multiply-add unit
- Greatly improved in Fermi
  - Up to 16 double precision operations performed per warp (subsequent slides)

Fermi Architecture

- 512 cores
- 32 cores per SM
- 16 SMs
- 6 64-bit memory partitions

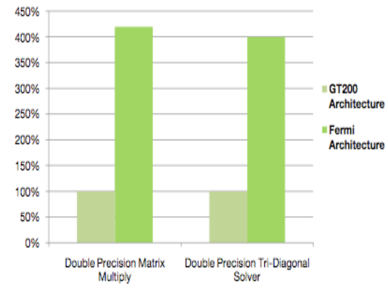


Closer look at Fermi core and SM

- 48 KB L1 cache in lieu of 16 KB shared memory
- 32-bit integer multiplies in single operation
- Fused multiply-add
- IEEE-Compliant for latest standard

Double Precision Arithmetic

- Up to 16 DP fused multiply-adds can be performed per clock per SM



Hardware Comparison

GPU	G80	GT200	Fermi
Transistors	681 million	1.4 billion	3.0 billion
CUDA Cores	128	240	512
Double Precision Floating Point Capability	None	30 FMA ops / clock	256 FMA ops /clock
Single Precision Floating Point Capability	128 MAD ops/clock	240 MAD ops / clock	512 FMA ops /clock
Warp schedulers (per SM)	1	1	2
Special Function Units (SFUs) / SM	2	2	4
Shared Memory (per SM)	16 KB	16 KB	Configurable 48 KB or 16 KB
L1 Cache (per SM)	None	None	Configurable 16 KB or 48 KB
L2 Cache (per SM)	None	None	768 KB
ECC Memory Support	No	No	Yes
Concurrent Kernels	No	No	Up to 16
Load/Store Address Width	32-bit	32-bit	64-bit

GPU Floating Point Features

	G80	SSE	IBM Altivec	Cell SPE
Precision	IEEE 754	IEEE 754	IEEE 754	IEEE 754
Rounding modes for FADD and FMUL	Round to nearest and round to zero	All 4 IEEE, round to nearest, zero, inf, -inf	Round to nearest only	Round to zero/ truncate only
Denormal handling	Flush to zero	Supported, 1000's of cycles	Supported, 1000's of cycles	Flush to zero
NaN support	Yes	Yes	Yes	No
Overflow and Infinity support	Yes, only clamps to max norm	Yes	Yes	No, infinity
Flags	No	Yes	Yes	Some
Square root	Software only	Hardware	Software only	Software only
Division	Software only	Hardware	Software only	Software only
Reciprocal estimate accuracy	24 bit	12 bit	12 bit	12 bit
Reciprocal sqrt estimate accuracy	23 bit	12 bit	12 bit	12 bit
log2(x) and 2^x estimates accuracy	23 bit	No	12 bit	No

Summary: Accuracy vs. Performance

- A few operators are IEEE 754-compliant
  - Addition and Multiplication
- … but some give up precision, presumably in favor of speed or hardware simplicity
  - Particularly, division
- Many built in intrinsics perform common complex operations very fast
- Some intrinsics have multiple implementations, to trade off speed and accuracy
  - e.g., intrinsic __sin() (fast but imprecise) versus sin() (much slower)

Deviations from IEEE-754

- Addition and Multiplication are IEEE 754 compliant
  - Maximum 0.5 ulp (units in the least place) error
- However, often combined into multiply-add (FMAD)
  - Intermediate result is truncated
- Division is non-compliant (2 ulp)
- Not all rounding modes are supported in G80, but supported now
- Denormalized numbers are not supported in G80, but supported later
- No mechanism to detect floating-point exceptions (seems to be still true)

Arithmetic Instruction Throughput (G80)

- int and float add, shift, min, max and float mul, mad: 4 cycles per warp
  - int multiply (*) is by default 32-bit
    - requires multiple cycles / warp
  - Use __mul24() / __umul24() intrinsics for 4-cycle 24-bit int multiply
- Integer divide and modulo are expensive
  - Compiler will convert literal power-of-2 divides to shifts
  - Be explicit in cases where compiler can't tell that divisor is a power of 2!
  - Useful trick: foo % n == foo & (n-1) if n is a power of 2

Arithmetic Instruction Throughput (G80)

- Reciprocal, reciprocal square root, sin/cos, log, exp: 16 cycles per warp
  - These are the versions prefixed with "__"
  - Examples:__rcp(), __sin(), __exp()
- Other functions are combinations of the above
  - y / x == rcp(x) * y == 20 cycles per warp
  - sqrt(x) == rcp(rsqrt(x)) == 32 cycles per warp

Runtime Math Library

- There are two types of runtime math operations
  - __func(): direct mapping to hardware ISA
    - Fast but low accuracy (see prog. guide for details)
    - Examples: __sin(x), __exp(x), __pow(x,y)
  - func() : compile to multiple instructions
    - Slower but higher accuracy (5 ulp, units in the least place, or less)
    - Examples: sin(x), exp(x), pow(x,y)
- The -use_fast_math compiler option forces every func() to compile to __func()

Next Class

- Next class
  - Discuss CUBLAS 2/3 implementation of matrix multiply and sample projects
- Remainder of the semester:
  - Focus on applications and parallel programming patterns