Many-core Processors for Science and Engineering Applications

Lecture 6
Floating-Point Considerations
and
CUDA for Multi-core

© David Kirk/NVIDIA and Wen-mei W. Hwu

Objective

- To understand the fundamentals of floating-point representation
- To know the IEEE-754 Floating Point Standard
- GeForce 8800 CUDA Floating-point speed, accuracy and precision
  - Deviations from IEEE-754
  - Accuracy of device runtime functions
  - -fastmath compiler option
  - Future performance considerations
- To understand CUDA on Multi-cores

© David Kirk/NVIDIA and Wen-mei W. Hwu
Urbana, Illinois, August 10-14, 2009

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

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 $1.0 \leq M < 10.0_{\mathbf{B}}$

- The interpretation of S is simple: S=0 results in a positive number and S=1 a negative number.

Normalized Representation

- Specifying that $1.0_B \leq M < 10.0_B$ makes the mantissa value for each floating point number unique.
  - For example, the only one mantissa value allowed for 0.5_D is M =1.0
    - $0.5_D = 1.0_B * 2^{-1}$
  - Neither $10.0_B * 2^{-2}$ nor $0.1_B * 2^{0}$ qualifies

- Because all mantissa values are of the form 1.XX…, one can omit the "1." part in the representation.
  - The mantissa value of 0.5_D in a 2-bit mantissa is 00, which is derived by omitting "1." from 1.00.

© David Kirk/NVIDIA and Wen-mei W. Hwu

Exponent Representation

- In an n-bits exponent representation, $2^{n-1}-1$ is added to its 2's complement representation to form its excess representation.
  - See Table for a 3-bit exponent representation
- A simple unsigned integer comparator can be used to compare the magnitude of two FP numbers
- Symmetric range for +/- exponents (111 reserved)

2’s complement	Actual decimal	Excess-3
000	0	011
001	1	100
010	2	101
011	3	110
100	**(reserved pattern)**	**111**
101	-3	000
110	-2	001
111	-1	010

© David Kirk/NVIDIA and Wen-mei W. Hwu
Urbana, Illinois, August 10-14, 2009

A simple, hypothetical 5-bit FP format

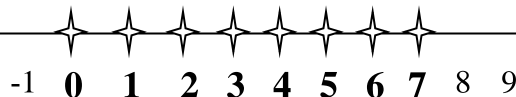
- Assume 1-bit S, 2-bit E, and 2-bit M
  - $0.5_D = 1.00_B * 2^{-1}$
  - 0.5_D = **0 00 00**, where S = 0, E = 00, and M = (1.)00

2's complement	Actual decimal	Excess-1
00	0	01
01	1	10
10	(reserved pattern)	11
11	-1	**00**

© David Kirk/NVIDIA and Wen-mei W. Hwu
Urbana, Illinois, August 10-14, 2009

Representable Numbers

- The representable numbers of a given format is the set of all numbers that can be exactly represented in the format.
- See Table for representable numbers of an unsigned 3-bit integer format



000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

© David Kirk/NVIDIA and Wen-mei W. Hwu

Represent… f a 5-bit Hypot… rmat

Cannot represent Zero!

0

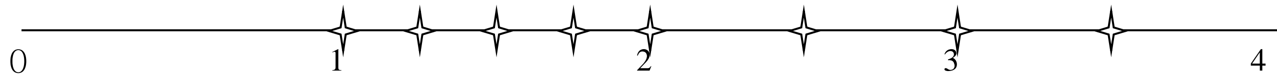
		No-zero		Abrupt underflow		Gradual underflow	
E	M	**S=0**	**S=1**	S=0	S=1	S=0	S=1
00	00	$\mathbf{2^{-1}}$	$\mathbf{-(2^{-1})}$	0	0	0	0
	01	$\mathbf{2^{-1}+1*2^{-3}}$	$\mathbf{-(2^{-1}+1*2^{-3})}$	0	0	$1*2^{-2}$	$-1*2^{-2}$
	10	$\mathbf{2^{-1}+2*2^{-3}}$	$\mathbf{-(2^{-1}+2*2^{-3})}$	0	0	$2*2^{-2}$	$-2*2^{-2}$
	11	$\mathbf{2^{-1}+3*2^{-3}}$	$\mathbf{-(2^{-1}+3*2^{-3})}$	0	0	$3*2^{-2}$	$-3*2^{-2}$
01	00	$\mathbf{2^{0}}$	$\mathbf{-(2^{0})}$	2^{0}	$-(2^{0})$	2^{0}	$-(2^{0})$
	01	$\mathbf{2^{0}+1*2^{-2}}$	$\mathbf{-(2^{0}+1*2^{-2})}$	$2^{0}+1*2^{-2}$	$-(2^{0}+1*2^{-2})$	$2^{0}+1*2^{-2}$	$-(2^{0}+1*2^{-2})$
	10	$\mathbf{2^{0}+2*2^{-2}}$	$\mathbf{-(2^{0}+2*2^{-2})}$	$2^{0}+2*2^{-2}$	$-(2^{0}+2*2^{-2})$	$2^{0}+2*2^{-2}$	$-(2^{0}+2*2^{-2})$
	11	$\mathbf{2^{0}+3*2^{-2}}$	$\mathbf{-(2^{0}+3*2^{-2})}$	$2^{0}+3*2^{-2}$	$-(2^{0}+3*2^{-2})$	$2^{0}+3*2^{-2}$	$-(2^{0}+3*2^{-2})$
10	00	$\mathbf{2^{1}}$	$\mathbf{-(2^{1})}$	2^{1}	$-(2^{1})$	2^{1}	$-(2^{1})$
	01	$\mathbf{2^{1}+1*2^{-1}}$	$\mathbf{-(2^{1}+1*2^{-1})}$	$2^{1}+1*2^{-1}$	$-(2^{1}+1*2^{-1})$	$2^{1}+1*2^{-1}$	$-(2^{1}+1*2^{-1})$
	10	$\mathbf{2^{1}+2*2^{-1}}$	$\mathbf{-(2^{1}+2*2^{-1})}$	$2^{1}+2*2^{-1}$	$-(2^{1}+2*2^{-1})$	$2^{1}+2*2^{-1}$	$-(2^{1}+2*2^{-1})$
	11	$\mathbf{2^{1}+3*2^{-1}}$	$\mathbf{-(2^{1}+3*2^{-1})}$	$2^{1}+3*2^{-1}$	$-(2^{1}+3*2^{-1})$	$2^{1}+3*2^{-1}$	$-(2^{1}+3*2^{-1})$
11	Reserved pattern						

© David Kirk/NVIDIA and Wen-mei W. Hwu
Urbana, Illinois, August 10-14, 2009

Flush to Zero

- Treat all bit patterns with E=0 as 0.0
  - This takes away several representable numbers near zero and lump them all into 0.0
  - For a representation with large M, a large number of representable numbers numbers will be removed.

0



© David Kirk/NVIDIA and Wen-mei W. Hwu
Urbana, Illinois, August 10-14, 2009

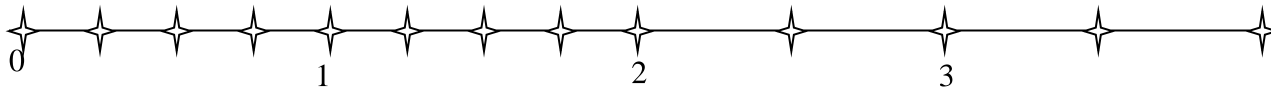
Flush to Zero

		No-zero		**Flush to Zero**		Denormalized	
E	M	S=0	S=1	**S=0**	**S=1**	S=0	S=1
00	00	2^{-1}	$-(2^{-1})$	**0**	**0**	0	0
	01	$2^{-1}+1*2^{-3}$	$-(2^{-1}+1*2^{-3})$	**0**	**0**	$1*2^{-2}$	$-1*2^{-2}$
	10	$2^{-1}+2*2^{-3}$	$-(2^{-1}+2*2^{-3})$	**0**	**0**	$2*2^{-2}$	$-2*2^{-2}$
	11	$2^{-1}+3*2^{-3}$	$-(2^{-1}+3*2^{-3})$	**0**	**0**	$3*2^{-2}$	$-3*2^{-2}$
01	00	2^{0}	$-(2^{0})$	$\mathbf{2^{0}}$	$\mathbf{-(2^{0})}$	2^{0}	$-(2^{0})$
	01	$2^{0}+1*2^{-2}$	$-(2^{0}+1*2^{-2})$	$\mathbf{2^{0}+1*2^{-2}}$	$\mathbf{-(2^{0}+1*2^{-2})}$	$2^{0}+1*2^{-2}$	$-(2^{0}+1*2^{-2})$
	10	$2^{0}+2*2^{-2}$	$-(2^{0}+2*2^{-2})$	$\mathbf{2^{0}+2*2^{-2}}$	$\mathbf{-(2^{0}+2*2^{-2})}$	$2^{0}+2*2^{-2}$	$-(2^{0}+2*2^{-2})$
	11	$2^{0}+3*2^{-2}$	$-(2^{0}+3*2^{-2})$	$\mathbf{2^{0}+3*2^{-2}}$	$\mathbf{-(2^{0}+3*2^{-2})}$	$2^{0}+3*2^{-2}$	$-(2^{0}+3*2^{-2})$
10	00	2^{1}	$-(2^{1})$	$\mathbf{2^{1}}$	$\mathbf{-(2^{1})}$	2^{1}	$-(2^{1})$
	01	$2^{1}+1*2^{-1}$	$-(2^{1}+1*2^{-1})$	$\mathbf{2^{1}+1*2^{-1}}$	$\mathbf{-(2^{1}+1*2^{-1})}$	$2^{1}+1*2^{-1}$	$-(2^{1}+1*2^{-1})$
	10	$2^{1}+2*2^{-1}$	$-(2^{1}+2*2^{-1})$	$\mathbf{2^{1}+2*2^{-1}}$	$\mathbf{-(2^{1}+2*2^{-1})}$	$2^{1}+2*2^{-1}$	$-(2^{1}+2*2^{-1})$
	11	$2^{1}+3*2^{-1}$	$-(2^{1}+3*2^{-1})$	$\mathbf{2^{1}+3*2^{-1}}$	$\mathbf{-(2^{1}+3*2^{-1})}$	$2^{1}+3*2^{-1}$	$-(2^{1}+3*2^{-1})$
11				Reserved pattern			

0

Denormalized Numbers

- The actual method adopted by the IEEE standard is called denromalized numbers or gradual underflow.
  - The method relaxes the normalization requirement for numbers very close to 0.
  - whenever E=0, the mantissa is no longer assumed to be of the form 1.XX. Rather, it is assumed to be 0.XX. In general, if the n-bit exponent is 0, the value is
    - $0.M * 2^{-2^{(n-1)}+2}$



© David Kirk/NVIDIA and Wen-mei W. Hwu
Urbana, Illinois, August 10-14, 2009

Denormalization

		No-zero		Flush to Zero		**Denormalized**	
E	M	S=0	S=1	S=0	S=1	**S=0**	**S=1**
00	00	2^{-1}	$-(2^{-1})$	0	0	**0**	**0**
	01	$2^{-1}+1*2^{-3}$	$-(2^{-1}+1*2^{-3})$	0	0	**$1*2^{-2}$**	**$-1*2^{-2}$**
	10	$2^{-1}+2*2^{-3}$	$-(2^{-1}+2*2^{-3})$	0	0	**$2*2^{-2}$**	**$-2*2^{-2}$**
	11	$2^{-1}+3*2^{-3}$	$-(2^{-1}+3*2^{-3})$	0	0	**$3*2^{-2}$**	**$-3*2^{-2}$**
01	00	2^{0}	$-(2^{0})$	2^{0}	$-(2^{0})$	2^{0}	$-(2^{0})$
	01	$2^{0}+1*2^{-2}$	$-(2^{0}+1*2^{-2})$	$2^{0}+1*2^{-2}$	$-(2^{0}+1*2^{-2})$	$2^{0}+1*2^{-2}$	$-(2^{0}+1*2^{-2})$
	10	$2^{0}+2*2^{-2}$	$-(2^{0}+2*2^{-2})$	$2^{0}+2*2^{-2}$	$-(2^{0}+2*2^{-2})$	$2^{0}+2*2^{-2}$	$-(2^{0}+2*2^{-2})$
	11	$2^{0}+3*2^{-2}$	$-(2^{0}+3*2^{-2})$	$2^{0}+3*2^{-2}$	$-(2^{0}+3*2^{-2})$	$2^{0}+3*2^{-2}$	$-(2^{0}+3*2^{-2})$
10	00	2^{1}	$-(2^{1})$	2^{1}	$-(2^{1})$	2^{1}	$-(2^{1})$
	01	$2^{1}+1*2^{-1}$	$-(2^{1}+1*2^{-1})$	$2^{1}+1*2^{-1}$	$-(2^{1}+1*2^{-1})$	$2^{1}+1*2^{-1}$	$-(2^{1}+1*2^{-1})$
	10	$2^{1}+2*2^{-1}$	$-(2^{1}+2*2^{-1})$	$2^{1}+2*2^{-1}$	$-(2^{1}+2*2^{-1})$	$2^{1}+2*2^{-1}$	$-(2^{1}+2*2^{-1})$
	11	$2^{1}+3*2^{-1}$	$-(2^{1}+3*2^{-1})$	$2^{1}+3*2^{-1}$	$-(2^{1}+3*2^{-1})$	$2^{1}+3*2^{-1}$	$-(2^{1}+3*2^{-1})$
11		Reserved pattern					

0

Arithmetic Instruction Throughput

- 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

© David Kirk/NVIDIA and Wen-mei W. Hwu
Urbana, Illinois, August 10-14, 2009

Arithmetic Instruction Throughput

- 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

© David Kirk/NVIDIA and Wen-mei W. Hwu

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()

Make your program float-safe!

- Future hardware will have double precision support
  - G80 is single-precision only
  - Double precision will have additional performance cost
  - Careless use of double or undeclared types may run more slowly on G80+
- Important to be float-safe (be explicit whenever you want single precision) to avoid using double precision where it is not needed
  - Add ‘f’ specifier on float literals:
    - `foo = bar * 0.123;    // double assumed`
    - `foo = bar * 0.123f;   // float explicit`
  - Use float version of standard library functions
    - `foo = sin(bar);    // double assumed`
    - `foo = sinf(bar);   // single precision explicit`

© David Kirk/NVIDIA and Wen-mei W. Hwu

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

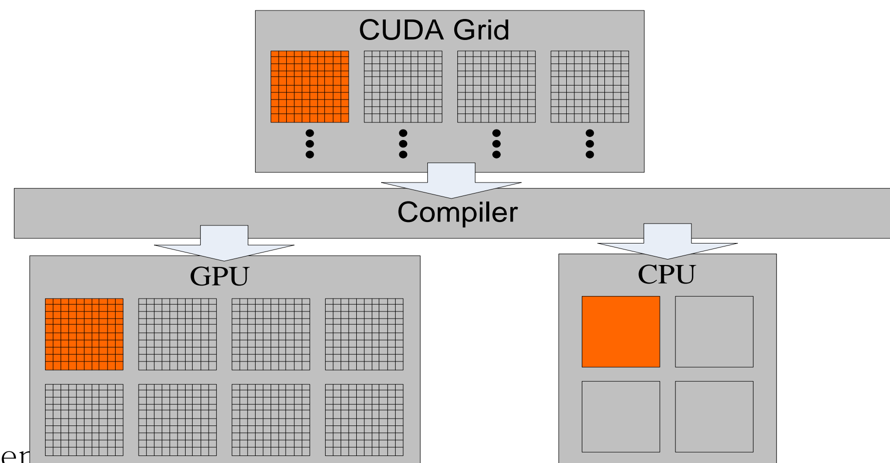
- Division is non-compliant (2 ulp)
- Not all rounding modes are supported
- Denormalized numbers are supported in G280 and beyond
- No mechanism to detect floating-point exceptions

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

CUDA Kernels on Multi-core CPU

- Most application developers do not want to write special code for each architecture – CUDA kernels are currently special code for GPUs
  - MCUDA allows efficient execution of CUDA kernels on multi-core CPUs
- A single GPU thread is too small for a CPU Thread
  - CUDA emulation does this and performs poorly
- CPU cores designed for ILP, SIMD
  - Optimizing compilers work well with iterative loops
- Turn GPU thread blocks from CUDA into iterative CPU loops



Key Issue: Synchronization

- Suspend and Wakeup
  - Move on to other threads
  - Begin again after all hit barrier

```
Matrixmul(A[ ], B[ ], C[ ])
{
    __shared__ Asub[ ][ ], Bsub[ ][ ];
    int a,b,c;
    float Csub;
    int k;
    …
    for(…)
     {

          Asub[tx][ty] = A[a];
           Bsub[tx][ty] = B[b];

          __syncthreads();


          for( k = 0; k < blockDim.x; k++ )
              Csub += Asub[ty][k] + Bsub[k][tx];

          __syncthreads();

     }
    …
}
```

© David Kirk/NVIDIA and Wen-mei W. Hwu
Urbana, Illinois, August 10-14, 2009

Synchronization solution

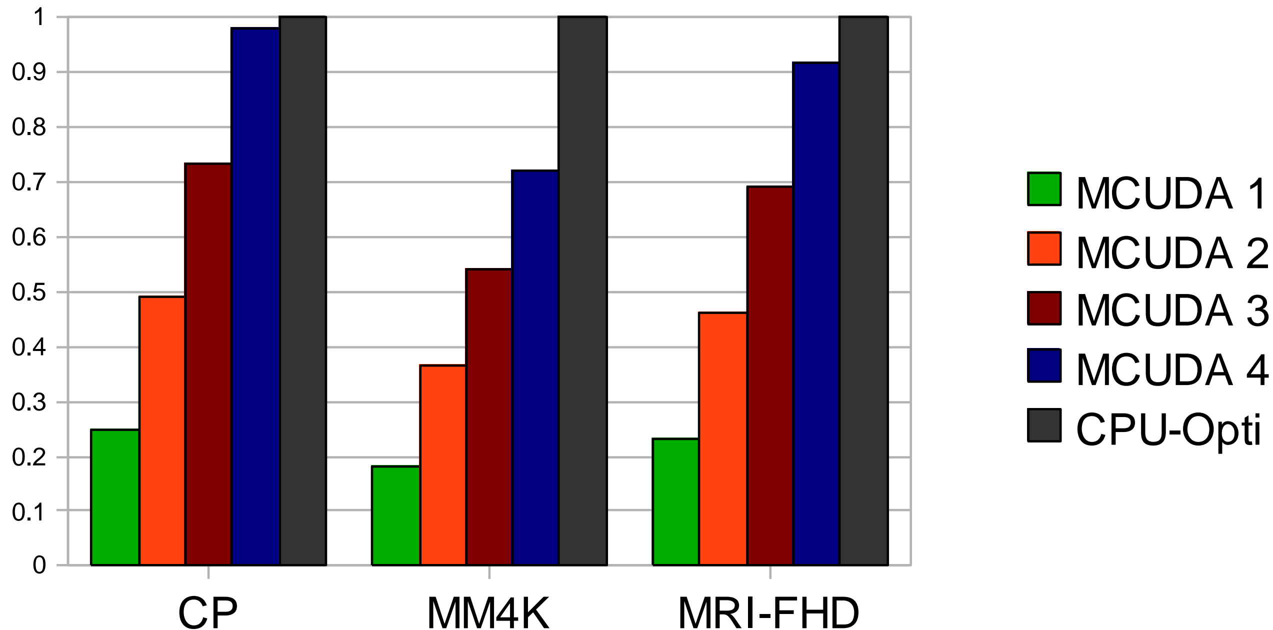
- Loop fission
  - Break the loop into multiple smaller loops according to syncthreads()
  - Not always possible with __synchthreads() in control flow

```
Matrixmul(A[ ], B[ ], C[ ])
{
    __shared__ Asub[ ][ ], Bsub[ ][ ];
    int a,b,c;
    float Csub;
    int k;
    …
    for(…)
    {
        for(ty=0; ty < blockDim.y; ty++)
          for(tx=0; tx < blockDim.x; tx++)
        {
          Asub[tx][ty] = A[a];
          Bsub[tx][ty] = B[b];
        }

        for(ty=0; ty < blockDim.y; ty++)
          for(tx=0; tx < blockDim.x; tx++)
        {
          for( k = 0; k < blockDim.x; k++ )
              Csub += Asub[ty][k] + Bsub[k][tx];
        }

    }
    …
}
```

© David Kirk/NVIDIA and Wen-mei W. Hwu
Urbana, Illinois, August 10-14, 2009

Bigger Picture Performance Results

- Consistent speed-up over hand-tuned single-thread code
- Best optimizations for GPU and CPU not always the same



*Over hand-optimized CPU

**Intel MKL, multi-core execution