

© David Kirk/NVIDIA and Wen-mei Hwu, 2006-2008

This is a draft chapter from an upcoming CUDA textbook by David Kirk from NVIDIA and Prof. Wen-mei Hwu from UIUC.

Please send any comment to dkirk@nvidia.com and w-hwu@uiuc.edu

This material is also part of the IAP09 CUDA@MIT (6.963) course. You may direct your questions about the class to Nicolas Pinto (pinto@mit.edu).

Chapter 6

Floating Point Considerations

In the early days of computing, floating point arithmetic capability was found only in mainframes and supercomputers. Although many microprocessors designed in the 1980's started to have floating point coprocessors, their floating-point arithmetic speed was about three orders of magnitude slower than that of mainframes and supercomputers. With advances in microprocessor technology, many microprocessors designed in the 1990's, such as Intel Pentium III and AMD Athlon, started to have high performance floating point capabilities that rival supercomputers. High speed floating point arithmetic has become a standard feature for microprocessors today. As a result, it has also become important for application programmers to understand and take advantage of floating point arithmetic in developing their applications. In particular, we will focus on the accuracy of floating point arithmetic, the precision of floating point number representation, and how they should be taken into consideration in parallel computation.

6.1. Floating Point Format

The IEEE Floating Point Standard is an effort for the computer manufacturers to conform to a common representation and arithmetic convention for floating point data. Most, if not all, of the computer manufacturers in the world have accepted this standard. In particular, virtually all microprocessors designed in the future will either fully conform to or almost fully conform to the IEEE Floating Point Standard. Therefore, it is important for application developers to understand the concept and practical considerations of this standard.

A floating point number system starts with the representation of a numerical value as bit patterns. In the IEEE Floating Point Standard, a numerical value is represented in three groups of bits: sign (S), exponent (E), and mantissa (M). Each (S, E, M) pattern uniquely identifies a numeric value according to the following formula:

$$\text{value} = (-1)^S * M * \{2^E\}, \text{ where } 1.0 \leq M < 2.0 \quad (1)$$

The interpretation of S is simple: S=0 means a positive number and S=1 a negative number. Mathematically, any number, including -1, when raised to the power of 0, results

in 1. Thus the value is positive. On the other hand, when -1 is raised to the power of 1, it is -1 itself. With a multiplication by -1, the value becomes negative. The interpretation of M and E bits are, however, much more complex. We will use the following example to help explain the interpretation of M and E bits.

Assume for the sake of simplicity that each floating point number consists of a 1-bit sign, 3-bit exponent, and 2-bit mantissa. We will use this hypothetical 6-bit format to illustrate the challenges involved in encoding E and M. As we discuss these values, we will sometime need to express number as either in decimal place value or in binary place value. Numbers expressed in decimal place value will have subscript D and those as binary place value will have subscript B. For example, 0.5_D ($5 * 10^{-1}$ since the place to the right of the decimal point carries a weight of 10^{-1}) is the same as 0.1_B ($1 * 2^{-1}$ since the place to the right of the decimal point carries a weight of 2^{-1}).

Normalized representation of M

Formula (1) requires that $1.0_B \leq M < 10.0_B$, which makes the mantissa value for each floating point number unique. For example, the only one mantissa value allowed for 0.5_D is $M = 1.0$:

$$0.5_D = 1.0_B * 2^{-1}$$

Another potential candidate would be $0.1_B * 2^0$, but the value of mantissa would be too small according to the rule. Similarly, $10.0_B * 2^{-2}$ is not legal because the value of the mantissa is too large. In general, with the restriction that $1.0_B \leq M < 10.0_B$, every floating point number has exactly one legal mantissa value. The numbers that satisfy this restriction will be referred to as normalized numbers. Because all mantissa values that satisfy the restriction are of the form 1.XX, we can omit the “1.” part from the representation. Therefore, the mantissa value of 0.5 in a 2-bit mantissa representation is 00, which is derived by omitting “1.” from 1.00. This makes a 2-bit mantissa effectively a 3-bit mantissa. In general, with IEEE format, an n-bit mantissa is effectively an (n+1)-bit mantissa.

Excess encoding of E

If n bits are used to represent the exponent E, the value $2^{n-1}-1$ is added to the two's complement representation for the exponent to form its excess representation. A two's complement representation is a system where the negative value of a number can be derived by first complementing every bit of the value and add one to the result. In our 3-bit exponent representation, there are three bits in the exponent. Therefore, the value $2^{3-1}-1 = 011$ will be added to the 2's complement representation of the exponent value. The following table shows the 2's complement representation and the excess presentation of each decimal exponent value. In our example, the exponent for 0.5_D is -1. The two's

complement representation of -1 can be derived by first complementing 001, the representation of 1, into 110 and then adding 001 to get 111. The excess representation adds another 011 to the 2's complement representation, as shown in the table, which results in 010.

2's complement	Decimal value	Excess-representation
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

Figure 1 Excess-3 encoding, sorted by 2's complement ordering

The advantage of excess representation is that an unsigned comparator can be used to compare signed numbers. As shown in Figure 2, the excess-3 code increases monotonically from -3 to 3. The code for -3 is 000 and that for 3 is 110. Thus, if one uses an unsigned number comparator to compare excess-3 code for any number from -1 to 3, the comparator gives the correct comparison result in terms of which number is larger, smaller, etc. For example, if one compares excess-3 codes 001 and 100 with an unsigned comparator, 001 is smaller than 100. This is the right conclusion since the values that they represent, -2 and 1, have exactly the same relation. This is a desirable property for hardware implementation since unsigned comparators are smaller and faster than signed comparators.

2's complement	Decimal value	Excess-3
100	(reserved pattern)	111
101	-3	000
110	-2	001
111	-1	010
000	0	011
001	1	100
010	2	101
011	3	110

Figure 2 Excess-3 encoding, sorted by excess-3 ordering

Now we are ready to represent 0.5_D with our 6-bit format:

$$0.5_D = 0\ 010\ 00, \text{ where } S = 0, E = 010, \text{ and } M = (4-)00$$

That is, the 6-bit representation for 0.5_D is 001000.

With normalized mantissa and excess-coded exponent, the value of a number with an n -bit exponent is

$$(-1)^S * 1.M * 2^{(E - (2^{n-1}) + 1)}$$

6.2. Representable Numbers

The representable numbers of a number format are the numbers that can be exactly represented in the format. For example, if one uses a 3-bit unsigned integer format, the representable numbers would be:

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

Figure 3 Representable numbers of a 3-bit unsigned format

Neither -1 nor 9 can be represented in the format given above. We can draw a number line to identify all the representable numbers, as shown in Figure 4 where all representable numbers of the 3-bit unsigned integer format are marked with stars.

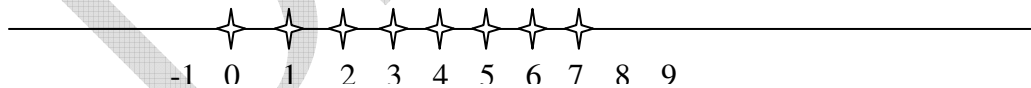


Figure 4 representable numbers of a 3-bit unsigned integer format

The representable numbers of a floating point format can be visualize in a similar manner. In Table 1, we show all the representable numbers of what we have so far and two variations. We use a 5-bit format to keep the size of the table manageable. The format consists of 1-bit S , 2-bit E (excess-1 coded), and 2-bit M (with “1.” part omitted). The no-zero column gives the representable numbers of the format we discussed thus far. Note that with this format, 0 is not one of the representatble numbers.

		No-zero		Abrupt underflow		Denorm	
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						

Table 1 Representable numbers of no-zero, abrupt underflow, and denorm formats

A look at how these representable numbers populate the number line, as shown in Figure 5, provides further insights about these representable numbers. In Figure 5, we show only the positive representable numbers. The negative numbers are symmetric to their positive counterparts on the other side of 0.



Figure 5. Representable numbers of the no-zero representation

Three things stand out. First, 0 is not representable in this format. Because 0 is one of the most important numbers, not being able to represent 0 in a number representation system is a serious deficiency. Second, the representable numbers become closer to each other towards the neighborhood of 0. This is a desirable behavior because as the absolute value of these numbers become smaller, it is more important to represent them more accurately. Having the representable numbers closer to each other makes it possible to represent numbers more accurately. Unfortunately this trend does not hold for the very vicinity of 0, which leads to the third point: there is a gap of representable numbers in the vicinity of 0. This is because the range of normalized mantissa precludes 0. This is another serious deficiency. The representation introduces significantly larger errors when representing numbers between 0 and 0.5 compared to the errors for the larger numbers between 0.5 and 1.0.

One method that has been used to accommodate 0 into a normalized number system is the *abrupt underflow* convention, which is illustrated by the second column of Table 1. Whenever E is 0, the number is interpreted as 0. In our 5-bit format, this method takes away eight representable numbers (four positive and four negative) in the vicinity of 0 and makes them all 0. Although this method makes 0 representable, it does create an even larger gap between representable numbers in 0's vicinity, as shown in Figure 6. It is obvious, when compared with Figure 5, the gap of representable numbers has been enlarged significantly with the vicinity of 0. This is very problematic since many numeric algorithms rely on the fact that the accuracy of number representation is higher for the very small numbers near zero. These algorithms generate small numbers and eventually use them as denominators. The errors for these small numbers can be greatly magnified in the division process.

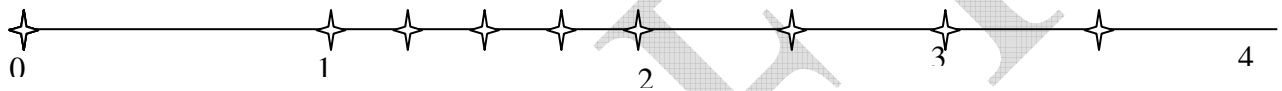


Figure 6 Representable numbers of the abrupt underflow format

The actual method adopted by the IEEE standard is called denormalization. The method relaxes the normalization requirement for numbers very close to 0. That is, 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

$$0.M * 2^{-2^{(n-1)} + 2}$$

For example, in Table 1, the denormalized representation 00001 has exponent value 00 and mantissa value 01. Using the denormalized formula, the value it represents is $0.01 * 2^0 = 2^{-2}$. Figure 7 shows the representable numbers for the denormalization format. The representation now has a uniformly spaced representable numbers in the close vicinity of 0. This eliminates the undesirable gap in the previous two methods.

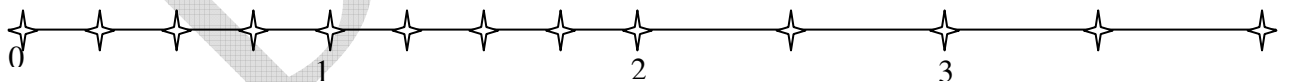


Figure 7. Representable numbers of a denormalization format.

We are now ready to discuss the concept of precision. The precision of a floating point representation is measured by the maximal error that we can introduce to a floating point number by representing that number as one of the representable numbers. The smaller the error is, the higher the precision. In Figure 7, the error introduced by a number representation is always smaller than 0.25_D , which occurs if we choose the largest representative number that is smaller than the number we would like to represent. Obviously, the precision of a floating point representation can be improved by adding more

bits to mantissa. Adding one bit to the representation in Figure 7 would improve the precision by reducing the maximal error by half. Thus, we say that a number system has higher precision when it uses more bits for mantissa.

6.3. Special Bit Patterns and Precision

The actual IEEE format has one more special bit pattern. When all exponent bits are 1s, the number represented is an infinity value if the mantissa is 0 or a Not a Number (NaN) if the mantissa is not 0. All special bit patterns of the IEEE floating point format are shown in the following table.

exponent	mantissa	meaning
11...1	$\neq 0$	NaN
11...1	$=0$	$(-1)^S * \infty$
00...0	$\neq 0$	denormalized
00...0	$=0$	0

All other numbers are normalized floating-point numbers. Single precision numbers have 1-bit S, 8-bit E, and 23-bit M. Double precision numbers have 1-bit S, 11-bit E, and 52-bit M. Since a double precision number has 29 more bits for mantissa, the largest error for representing a number is reduced to $1/2^{29}$ of that of the single precision format!

All representable numbers fall between $-\infty$ (negative infinity) and $+\infty$ (positive infinite). An ∞ can be created by overflow, e.g., divided by zero. Any representable number divided by $+\infty$ or $-\infty$ results in 0.

NaN (Not a Number) is generated by operations whose input values do not make sense, for example, $0/0$, $0*\infty$, ∞/∞ , $\infty - \infty$. They are also used to for data that have not be properly initialized in a program. There are two types on NaN's in the IEEE standard: signaling and quiet.

Signaling NaN causes an exception when used as input to arithmetic operations. For example the operation $1.0 + \text{signaling NaN}$ raises an exception. Signaling NaN's are used in situations where the programmer would like to make sure that the program execution be interrupted whenever any NaN values are used in floating point computations. These situations usually mean that there is something wrong with the execution of the program. In mission critical applications, the execution cannot continue until the validity of the execution can be verified with a separate means. For example, software engineers often mark all the uninitialized data as signaling NaN. This practice ensures the detection of using uninitialized data during program execution.

Quiet NaN generates a quiet NaN when used as input to arithmetic operations. For example, the operation $(1.0 + \text{quiet NaN})$ generates a quiet NaN. Quiet NaN's are typically used in applications where the user can review the output and decide if the application should be re-run with a different input for more valid results. When the results are printed, Quiet NaN's are printed as "NaN" so that the user can spot them in the output file easily.

6.4. Arithmetic Accuracy and Rounding of Mantissa

Now that we have a good understanding of the IEEE floating point format, we now move to the concept of arithmetic accuracy. The accuracy of a floating point arithmetic operation is measured by the maximal error introduced by the operation. The smaller the error is, the higher the accuracy. The most common source of error in floating point arithmetic is when the operation generates a result that cannot be exactly represented and thus requires rounding. Rounding occurs if the mantissa of the result value needs too many bits to be represented exactly. The cause of rounding is typically pre-shifting in floating point arithmetic. When two input operands to a floating-point addition or subtraction have different exponents, the mantissa of the one with the smaller exponent is typically right-shifted until the exponents are equal. As a result, the final result can have more bits than the format can accommodate.

This can be illustrated with a simple example based on the 5-bit representation in Table 1. Assume that we need to add $1.00 * 2^{-2}$ (0, 00, 01) to $1.00 * 2^1$ (0, 10, 00), that is, we need to perform $1.00 * 2^1 + 1.00 * 2^{-2}$. The ideal result would be $1.001 * 2^1$. However, we can easily see that this ideal result is not a representable number in a 5-bit representation. Thus, the best one can do is to generate the closest representable number, which is $1.01 * 2^1$. By doing so, we introduce an error, $0.001 * 2^1$, which is half the place value of the least significant place. We refer to this as 0.5 ULP (Units in the Last Place). If the hardware is designed to perform arithmetic and rounding operations perfectly, the most error that one should introduce should be no more than 0.5 ULP. This is the accuracy achieved by the addition and subtraction operations in G80/280.

In practice, some of the arithmetic hardware units, such as division and transcendental functions, are typically implemented with iterative approximation algorithms. If the hardware does not perform sufficient number of iterations, the result may have an error larger than 0.5 ULP. For example, the division in G80/280 can introduce an error that is twice the place value of the least place of the mantissa.

6.4. Algorithm Considerations

Numerical algorithms often need to sum up a large number of values. For example, the dot product in matrix multiplication needs to sum up pair-wise products of input matrix elements. Ideally, the order of summing these values should not affect the final total since addition is an associative operation. However, with finite precision, the order of summing these values can affect the accuracy of the final result. For example, if we need to add four numbers in our 5-bit representation: $1.00 * 2^0 + 1.00 * 2^0 + 1.00 * 2^{-2} + 1.00 * 2^{-2}$.

If we add up the numbers in strict sequential order, we have the following sequence of operations:

$$\begin{aligned} 1.00*2^0 + 1.00*2^0 + 1.00*2^{-2} + 1.00*2^{-2} &= 1.00*2^1 + 1.00*2^{-2} + 1.00*2^{-2} \\ &= 1.00*2^1 + 1.00*2^{-2} = 1.00*2^1 \end{aligned}$$

Note that in the second step and third step, the smaller operand simply disappears because they are too small compared to the larger operand. The pre-shifting of mantissa produces a zero value for the smaller operand.

Now, let's consider a parallel algorithm where the first two values are added and the second two operands are added in parallel. The algorithm then add up the pair-wise sum:

$$\begin{aligned} (1.00*2^0 + 1.00*2^0) + (1.00*2^{-2} + 1.00*2^{-2}) &= 1.00*2^1 + 1.00*2^{-1} \\ &= 1.01*2^1 \end{aligned}$$

Note that the results are different from the sequential result! This is because the sum of the third and fourth values is large enough that it remains non-zero after pre-shifting. This discrepancy between sequential algorithms and parallel algorithms often surprises application developers who are not familiar with floating point precision and accuracy considerations. Although we showed a scenario where a parallel algorithm produced a more accurate result than a sequential algorithm, the reader should be able to come up with a slightly different scenario where the parallel algorithm produces a less accurate result than a sequential algorithm. Experienced application developers either make sure that the variation in the final result can be tolerated or to ensure that the data is grouped in a way that the parallel algorithm results in the most accurate results.