CHAPTER

Numerical considerations

6

CHAPTER OUTLINE

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 extremely slow, 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 and GPUs today. Floating-point representation allows for larger dynamic range of representable data values and more precise representation of tiny data values. These desirable properties make floating-point preferred data representative for modeling the physical and artificial phenomena, such as combustion, aerodynamics, light illumination, and financial risks. Large scale evaluation of these models has been driving the need for parallel computing. As a result, it is important for application programmers to understand the nature of floating-point arithmetic in developing their parallel applications. In particular, we will focus on the accuracy of floating-point arithmetic operations, the precision of floating-point number representation, the stability of numerical algorithms and how they should be taken into consideration in parallel programming.

Programming Massively Parallel Processors. DOI: http://dx.doi.org/10.1016/B978-0-12-811986-0.00006-6

Copyright © 2017 David B. Kirk/NVIDIA Corporation and Wen-mei W. Hwu. Published by Elsevier Inc. All rights reserved

6.1 FLOATING-POINT DATA REPRESENTATION

The IEEE-754 Floating-Point Standard is an effort for the computer manufacturers to conform to a common representation and arithmetic behavior for floating-point data [IEEE 2008]. 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-754 Floating-Point Standard and its more recent IEEE-754 2008 revision [2008]. Therefore, it is important for application developers to understand the concepts 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). With some exceptions that will be detailed later, each (S, E, M) pattern uniquely identifies a numeric value according to the following formula:

$$\text{Value} = (-1)^S * 1.M * \{2^{E-\text{bias}}\} \tag{6.1}$$

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 numeric values, we will sometimes need to express a number either in decimal place value or in binary place value. Numbers expressed in decimal place value will have subscript $_D$ and those in 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 (6.1) requires that all values are derived by treating the mantissa value as $1.M$, which makes the mantissa bit pattern for each floating-point number unique. For example, under this interpretation of the M bits, the only mantissa bit pattern allowed for 0.5_D is the one where all bits that represent M are 0s:

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

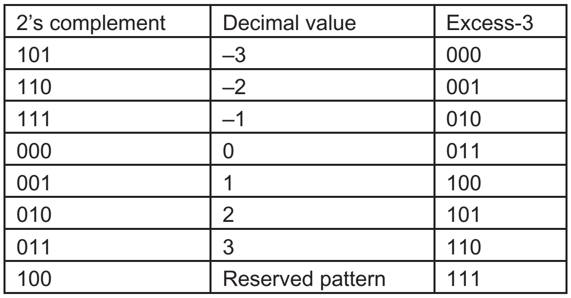
Other potential candidates would be 0.1_B*2^0 and 10.0_B*2^{-2}, but neither fits the form of $1.M$. The numbers that satisfy this restriction will be referred to as normalized numbers. Because all mantissa values that satisfy the restriction are of the form 1.XX, we can omit the "1." part from the representation. 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 m-bit mantissa is effectively an $(m+1)$-bit mantissa.

EXCESS ENCODING OF *E*

The number of bits used to represent E determines the range of numbers that can be represented. Large positive E values result in very large floating-point absolute values. For example, if the value of E is 64, the floating-point number being represented is between 2^{64} ($>10^{18}$) and 2^{65}. You would be extremely happy if this was the balance of your savings account! Large negative E values result in very small floating values. For example, if E value is -64, the number being represented is between 2^{-64} ($<10^{-18}$) and 2^{-63}. This is a very tiny fractional number. The E field allows a floating-point number format to represent a wider range of numbers than integer number formats. We will come back to this point when we look at the representable numbers of a format.

The IEEE standard adopts an excess or biased encoding convention for E. If e bits are used to represent the exponent E, $(2^{e-1}-1)$ 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 adding one to the result. In our 3-bit exponent representation, there are three bits in the exponent ($e = 3$). Therefore, the value $2^{3-1}-1 = 011$ will be added to the 2's complement representation of the exponent value.

The advantage of excess representation is that an unsigned comparator can be used to compare signed numbers. As shown in Fig. 6.1, in our 3-bit exponent representation, the excess-3 bit patterns increase monotonically from –3 to 3 when viewed as unsigned numbers. We will refer to each of these bit patterns as the code for



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

FIGURE 6.1

Excess-3 encoding, sorted by excess-3 ordering.

the corresponding value. For example, 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 –3 to 3, the comparator gives the correct comparison result in terms of which number is larger, smaller, etc. For another 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.

Fig. 6.1 also shows that the pattern of all 1's in the excess representation is a reserved pattern. Note that a 0 value and an equal number of positive and negative values result in an odd number of patterns. Having the pattern 111 as either an even number or odd number would result in an unbalanced number of positive and negative numbers. The IEEE standard uses this special bit pattern in special ways that will be discussed later.

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

$$0.5_D = 001000, \quad \text{where } S = 0, E = 010, \text{and } M = (1.)00$$

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

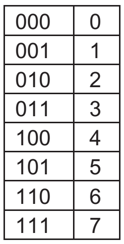
In general, with normalized mantissa and excess-coded exponent, the value of a number with an n-bit exponent is:

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

6.2 REPRESENTABLE NUMBERS

The representable numbers of a representation 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 are shown in Fig. 6.2.

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 Fig. 6.3 where all representable numbers of the 3-bit unsigned integer format are marked with stars.



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

FIGURE 6.2

Representable numbers of a 3-bit unsigned integer format.

The representable numbers of a floating-point format can be visualized in a similar manner. In Fig. 6.4, 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. The reader is encouraged to generate at least part of the no-zero column using the formula given in Section 6.1. Note that with this format, 0 is not one of the representable numbers.

A quick look at how these representable numbers populate the number line, as shown in Fig. 6.5, provides further insights about these representable numbers. In Fig. 6.5, we show only the positive representable numbers. The negative numbers are symmetric to their positive counterparts on the other side of 0.

We can make five observations. First, the exponent bits define the major intervals of representable numbers. In Fig. 6.5, there are three major intervals on each side of 0

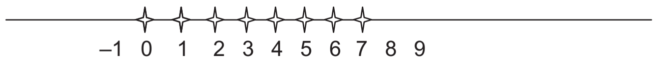


FIGURE 6.3

Representable numbers of a 3-bit unsigned integer format.

		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						

FIGURE 6.4

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

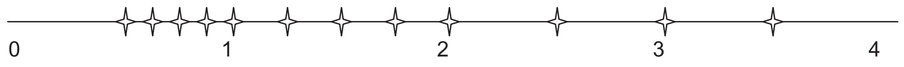


FIGURE 6.5

Representable numbers of the no-zero representation.

because there are two exponent bits. Basically, the major intervals are between powers of 2s. With two bits of exponents and one reserved bit pattern (11), there are three powers of two ($2^{-1} = 0.5_D$, $2^0 = 1.0_D$, $2^1 = 2.0_D$), each starts an interval of representable numbers. Keep in mind that there are also three powers of two ($-2^{-1} = -0.5_D$, $-2^0 = -1.0_D$, $-2^1 = -2.0_D$) on the left of zero, are not shown in Fig. 6.5.

The second observation is that the mantissa bits define the number of representable numbers in each interval. With two mantissa bits, we have four representable numbers in each interval. In general, with N mantissa bits, we have 2^N representable numbers in each interval. If a value to be represented falls within one of the intervals, it will be rounded to one of these representable numbers. Obviously, the larger the number of representable numbers in each interval, the more precisely we can represent a value in the region. Therefore, the number of mantissa bits determines the *precision* of the representation.

The third observation is that 0 is not representable in this format. It is missing from the representable numbers in the no-zero column of Fig. 6.5. Because 0 is one of the most important numbers, not being able to represent 0 in a number representation system is a serious deficiency. We will address this deficiency soon.

The fourth observation is that the representable numbers become closer to each other toward the neighborhood of 0. Each interval is half the size of the previous interval as we move toward zero. In Fig. 6.5, the rightmost interval is of width 2, the next one is of width 1, and the next one is of width 0.5. While not shown in Fig. 6.5, there are three intervals on the left of zero. They contain the representable negative numbers. The leftmost interval is of width 2, the next one is of width 1 and the next one is width 0.5. Since every interval has the same representable numbers, four in Fig. 6.5, the representable numbers become closer to each other as we move toward zero. In other words, the representative numbers become closer as their absolute values become smaller. This is a desirable trend because as the absolute value of these numbers become smaller, it is more important to represent them more precisely. The distance between representable numbers determines the maximal rounding error for a value that falls into the interval. For example, if you have 1 billion dollars in your bank account, you may not even notice that there is a 1 dollar rounding error in calculating your balance. However, if the total balance is 10 dollars, having a 1 dollar rounding error would be much more noticeable!

The fifth observation is that, unfortunately, the trend of increasing density of representable numbers, and thus increasing precision of representing numbers in the intervals as we move toward zero, does not hold for the very vicinity of 0. That is, there is a gap of representable numbers in the immediate vicinity of 0. This is because the range of normalized mantissa precludes 0. This is another serious deficiency. The representation introduces significantly larger (4×) errors when representing numbers between 0 and 0.5 compared to the errors for the larger numbers between 0.5 and 1.0. In general, with m bits in the mantissa, this style of representation would introduce 2^m times more error in the interval closest to zero than the next interval. For numerical methods that rely on accurate detection of convergence conditions based on very small data values, such deficiency can cause instability in execution time and accuracy of results. Furthermore, some algorithms generate small numbers and eventually use them as denominators. The errors in representing these small numbers can be greatly magnified in the division process and cause numerical instability in these algorithms.

One method that can accommodate 0 into a normalized floating-point number system is the *abrupt underflow* convention, which is illustrated in the second column of Fig. 6.4. 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 (between -1.0 and $+1.0$) and makes them all 0. Due to its simplicity, some mini-computers in the 1980s used abrupt underflow. Even to this day, some arithmetic units that need to operate in high-speed still use abrupt underflow convention. Although this method makes 0 a representable number, it creates an even larger gap between representable numbers in 0's vicinity, as shown in Fig. 6.6. It is obvious, when compared with Fig. 6.5, that the gap of representable numbers has been enlarged significantly (by 2×) from 0.5 to 1.0. As we explained before, this is very problematic for many numerical algorithms whose correctness relies on accurate representation of small numbers near zero.

The actual method adopted by the IEEE standard is called denormalization. The method relaxes the normalization requirement for numbers very close to 0. As shown in Fig. 6.8, whenever $E = 0$, the mantissa is no longer assumed to be of the form 1.XX. Rather, it is assumed to be 0.XX. The value of the exponent is assumed to be the same as the previous interval. For example, in Fig. 6.4, the denormalized representation 00001 has exponent value 00 and mantissa value 01. The mantissa is assumed to be 0.01 and the exponent value is assumed to be the same as that of the previous interval: 0 rather than -1. That is, the value that 00001 represents is now $0.01*2^0 = 2^{-2}$. Fig. 6.7 shows the representable numbers for the denormalized format. The representation now has uniformly spaced representable numbers in the close vicinity of 0. Intuitively, the denormalized convention takes the four numbers in the last interval of representable numbers of a no-zero representation and spreads them out to cover the gap area. This eliminates the undesirable gap in the previous two methods.

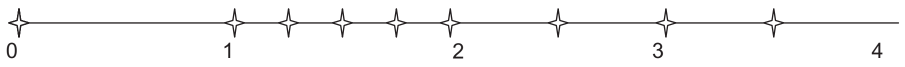


FIGURE 6.6

Representable numbers of the abrupt underflow format.

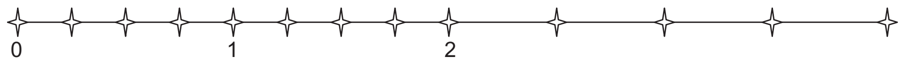


FIGURE 6.7

Representable numbers of a denormalization format.

Note that the distances between representable numbers in the last two intervals are actually identical. In general, if the *n*-bit exponent is 0, the value is:

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

As we can see, the denormalization formula is quite complex. The hardware also needs to be able to detect whether a number falls into the denormalized interval and choose the appropriate representation for that number. The amount of hardware required to implement denormalization in high speed is quite significant. Implementations that use a moderate amount of hardware often introduce thousands of clock cycles of delay whenever a denormalized number needs to be generated or used. This was the reason why early generations of CUDA devices did not support denormalization. However, virtually all recent generations of CUDA devices, thanks to the increasing number of available transistors of more recent fabrication processes, support denormalization. More specifically, all CUDA devices of compute capability 1.3 and up support denormalized double-precision operands, and all devices of compute capability 2.0 and up support denormalized single-precision operands.

In summary, 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. The precision of a floating-point representation can be improved by adding more bits to mantissa. Adding one bit to the representation of the mantissa improves the precision by reducing the maximal error by half. Thus, a number system has higher precision when it uses more bits for mantissa. This is reflected in double precision versus single precision numbers in the IEEE standard.

6.3 SPECIAL BIT PATTERNS AND PRECISION IN IEEE FORMAT

We now turn to more specific details of the actual IEEE format. When all exponent bits are 1s, the number represented is an infinity value if the mantissa is 0. It is a Not a Number (NaN) if the mantissa is not 0. All special bit patterns of the IEEE floating-point format are described in Fig. 6.8.

Exponent	Mantissa	Meaning
11…1	$\neq 0$	NaN
11…1	$= 0$	$(-1)^S*\infty$
00…0	$\neq 0$	denormalized
00…0	$= 0$	0

FIGURE 6.8

Special bit patterns in the IEEE standard format.

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! With the additional three bits of exponent, the double precision format also extends the number of intervals of representable numbers. This extends the range of representable numbers to very large as well as very small values.

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

NaN is generated by operations whose input values do not make sense, for example, 0/0, $0*\infty$, ∞/∞, $\infty-\infty$. They are also used for data that have not been properly initialized in a program. There are two types of NaN's in the IEEE standard: signaling and quiet. Signaling NaN's (SNaNs) should be represented with the most significant mantissa bit cleared, whereas Quiet NaN's are represented with most significant mantissa bit set.

Signaling NaN causes an exception when used as input to arithmetic operations. For example, the operation (1.0+ signaling NaN) raises an exception signal to the operating system. 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. The current generation of GPU hardware does not support signaling NaN. This is due to the difficulty of supporting accurate signaling during massively parallel execution.

Quiet NaN generates another quiet NaN without causing an exception when used as input to arithmetic operations. For example, the operation (1.0+ 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

Now that we have a good understanding of the IEEE floating-point format, we are ready to discuss the concept of arithmetic accuracy. While the precision is determined by the number of mantissa bits used in a floating-point number format, the accuracy is determined by the operations performed on a floating number. 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. For example, a multiplication generates a product value that consists of twice the number of bits than either of the input values. For another example, adding two floating-point numbers can be done by adding their mantissa values together if the two floating-point values have identical exponents. When two input operands to a floating-point addition have different exponents, the mantissa of the one with the smaller exponent is repeatedly divided by 2 or right-shifted (i.e., all the mantissa bits are shifted to the right by one bit position) until the exponents are equal. As a result, the final result can have more bits than the format can accommodate.

Alignment shifting of operands can be illustrated with a simple example based on the 5-bit representation in Fig. 6.4. Assume that we need to add 1.00_B*2^{-2}(0, 00, 01) to $1.00*2^1{}_D$ (0, 10, 00), i.e., we need to perform $1.00_B*2^1 + 1.00_B*2^{-2}$. Due to the difference in exponent values, the mantissa value of the second number needs to be right-shifted by 3-bit positions before it is added to the first mantissa value. That is, the addition becomes $1.00_B*2^1 + 0.001_B*2^1$. The addition can now be performed by adding the mantissa values together. The ideal result would be 1.001_B*2^1. However, we can see that this ideal result is not a representable number in a 5-bit representation. It would have required three bits of mantissa and there are only two mantissa bits in the format. Thus, the best one can do is to generate one of the closest representable numbers, which is either 1.01_B*2^1 or 1.00_B*2^1. By doing so, we introduce an error, 0.001_B*2^1, which is half the place value of the least significant place. We refer to this as 0.5_D 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_D ULP. To our knowledge, this is the accuracy achieved by the addition and subtraction operations in all CUDA devices today.

In practice, some of the more complex arithmetic hardware units, such as division and transcendental functions, are typically implemented with polynomial approximation algorithms. If the hardware does not use a sufficient number of terms in the approximation, the result may have an error larger than 0.5_D ULP. For example, if the ideal result of an inversion operation is 1.00_B*2^1 but the hardware generates a 1.10_B*2^1 due to the use of an approximation algorithm, the error is 2_D ULP since the error ($1.10_B - 1.00_B = 0.10_B$) is two times bigger than the units in the last place (0.01_B). In practice, the hardware inversion operations in some early devices introduce an error that is twice the place value of the least place of the mantissa, or 2 ULP. Thanks to the more abundant transistors in more recent generations of CUDA devices, their hardware arithmetic operations are much more accurate.

6.5 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 perform a sum reduction on four numbers in our 5-bit representation: $1.00_B{*}2^0{+}1.00_B{*}2^0{+}1.00_B{*}2^{-2}{+}1.00_B{*}2^{-2}$.

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

$$1.00_B{*}2^0 + 1.00_B{*}2^0 + 1.00_B{*}2^{-2} + 1.00_B{*}2^{-2} = 1.00_B{*}2^1 + 1.00_B{*}2^{-2} + 1.00_B{*}2^{-2} = 1.00_B{*}2^1 + 1.00_B{*}2^{-2} = 1.00_B{*}2^1$$

Note that in the second step and third step, the smaller operand simply disappears because it is too small compared to the larger operand.

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

$$(1.00_B{*}2^0 + 1.00_B{*}2^0) + (1.00_B{*}2^{-2} + 1.00_B{*}2^{-2}) = 1.00_B{*}2^1 + 1.00_B{*}2^{-1} = 1.01_B{*}2^1$$

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 now affects the addition result. 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 ensure that the data is sorted or grouped in a way that the parallel algorithm results in the most accurate results.

A common technique to maximize floating-point arithmetic accuracy is to presort data before a reduction computation. In our sum reduction example, if we presort the data according to ascending numerical order, we will have the following:

$$1.00_B{*}2^{-2} + 1.00_B{*}2^{-2} + 1.00_B{*}2^0 + 1.00_B{*}2^0$$

When we divide up the numbers into groups in a parallel algorithm, say the first pair in one group and the second pair in another group, numbers with numerical values close to each other are in the same group. Obviously, the sign of the numbers needs to be taken into account during the presorting process. Therefore, when we perform addition in these groups, we will likely have accurate results. Furthermore, some parallel algorithms use each thread to sequentially reduce values within each group. Having the numbers sorted in ascending order allows a sequential addition to get higher accuracy. This is a reason why sorting is frequently used in massively parallel numerical algorithms. Interested readers should study more advanced techniques such as compensated summation algorithm, also known as Kahan's Summation Algorithm, for getting even a more robust approach to accurate summation of floating-point values [Kahan 1965].

6.6 LINEAR SOLVERS AND NUMERICAL STABILITY

While the order of operations may cause variation in the numerical outcome of reduction operations, it may have even more serious implications on some types of computation such as solvers for linear systems of equations. In these solvers, different numerical values of input may require different ordering of operations in order to find a solution. If an algorithm fails to follow a desired order of operations for an input, it may fail to find a solution even though the solution exists. Algorithms that can always find an appropriate operation order, thus finding a solution to the problem as long as it exists for any given input values, are called *numerically stable*. Algorithms that fall short are referred to as *numerically unstable*.

In some cases, numerical stability considerations can make it more difficult to find efficient parallel algorithms for a computational problem. We can illustrate this phenomenon with a solver that is based on Gaussian Elimination. Consider the following system of linear equations:

$$3X + 5Y + 2Z = 19 \qquad \text{(Equation 1)}$$

$$2X + 3Y + Z = 11 \qquad \text{(Equation 2)}$$

$$X + 2Y + 2Z = 11 \qquad \text{(Equation 3)}$$

As long as the three planes represented by these equations have an intersection point, we can use Gaussian elimination to derive the solution that gives the coordinate of the intersection point. We show the process of applying Gaussian elimination to this system in Fig. 6.9, where variables are systematically eliminated from lower positioned equations.

In the first step, all equations are divided by their coefficient for the X variable: 3 for Equation 1, 2 for Equation 2, and 1 for Equation 3. This makes the coefficients for X in all equations the same. In step two, Equation 1 is subtracted from Equation 2 and Equation 3. These subtractions eliminate variable X from Equation 2 and Equation 3, as shown in Fig. 6.9.

We can now treat Equation 2 and Equation 3 as a smaller system of equations with one fewer variable than the original equation. Since they do not have variable X, they can be solved independently from Equation 1. We can make more progress by eliminating variable Y from Equation 3. This is done in step 3 by dividing Equation 2 and Equation 3 by the coefficients for their Y variables: $-1/6$ for Equation 2 and 1/3 for Equation 3. This makes the coefficients for Y in both Equation 2 and Equation 3

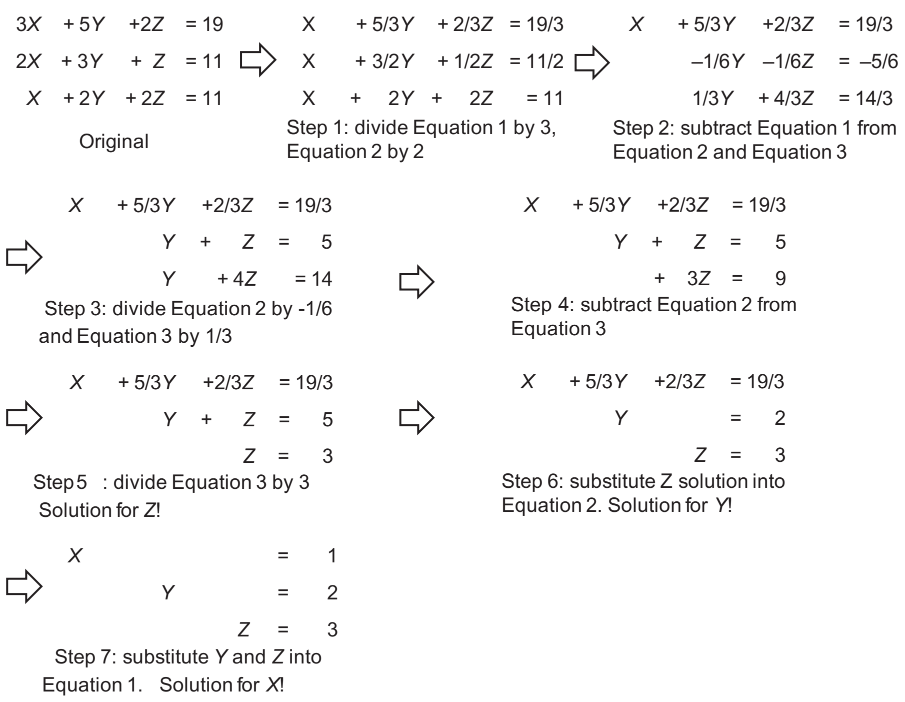

3X + 5Y +2Z = 19
2X + 3Y + Z = 11
X + 2Y + 2Z = 11
Original
X + 5/3Y + 2/3Z = 19/3
X + 3/2Y + 1/2Z = 11/2
X + 2Y + 2Z = 11
Step 1: divide Equation 1 by 3, Equation 2 by 2
X + 5/3Y +2/3Z = 19/3
–1/6Y –1/6Z = –5/6
1/3Y + 4/3Z = 14/3
Step 2: subtract Equation 1 from Equation 2 and Equation 3
X + 5/3Y +2/3Z = 19/3
Y + Z = 5
Y + 4Z = 14
Step 3: divide Equation 2 by -1/6 and Equation 3 by 1/3
X + 5/3Y +2/3Z = 19/3
Y + Z = 5
+ 3Z = 9
Step 4: subtract Equation 2 from Equation 3
X + 5/3Y +2/3Z = 19/3
Y + Z = 5
Z = 3
Step5 : divide Equation 3 by 3 Solution for Z!
X + 5/3Y +2/3Z = 19/3
Y = 2
Z = 3
Step 6: substitute Z solution into Equation 2. Solution for Y!
X = 1
Y = 2
Z = 3
Step 7: substitute Y and Z into Equation 1. Solution for X!

FIGURE 6.9

Gaussian elimination and backward substitution for solving systems of linear equations.

the same. In step four, Equation 2 is subtracted from Equation 3, which eliminates variable Y from Equation 3.

For systems with larger number of equations, the process would be repeated more. However, since we have only three variables in this example, the third equation has only the Z variable. We simply need to divide Equation 3 by the coefficient for variable Z. This conveniently gives us the solution $Z = 3$.

With the solution for Z variable in hand, we can substitute the Z value into Equation 2 to get the solution $Y = 2$. We can then substitute both $Z = 3$ and $Y = 2$ into Equation 1 to get the solution $X = 1$. We now have the complete solution for the original system. It should be obvious why step 6 and step 7 form the second phase of the method called backward substitution. We go backwards from the last equation to the first equation to get solutions for more and more variables.

In general, the equations are stored in matrix forms in computers. Since all calculations only involve the coefficients and the right-hand-side values, we can just store these coefficients and right-hand-side values in a matrix. Fig. 6.10 shows the matrix view of the Gaussian elimination and back substitution process. Each row of the matrix corresponds to an original equation. Operations on equations become operations on matrix rows.

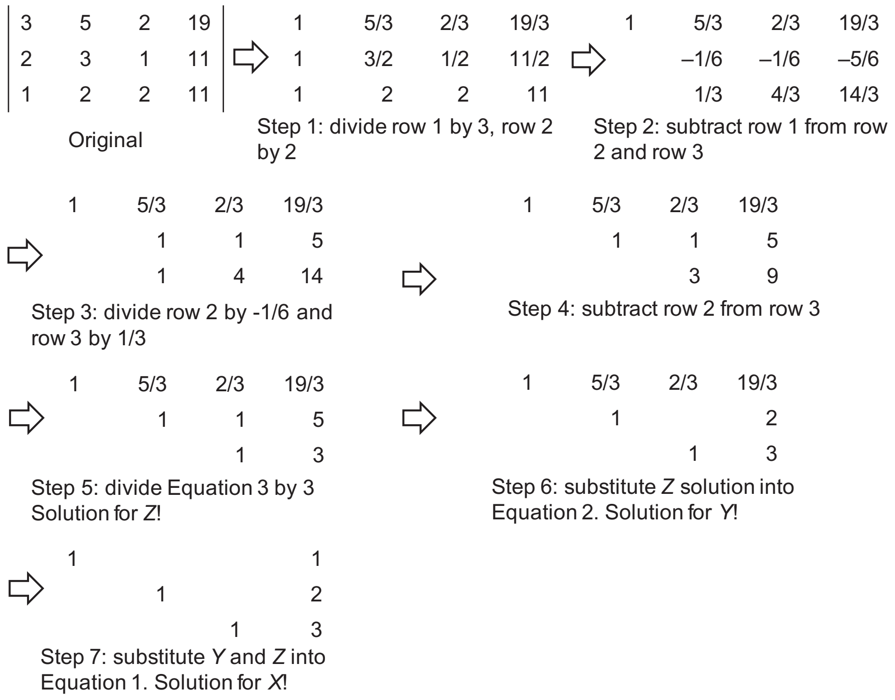

3 5 2 19
2 3 1 11
1 2 2 11
Original
1 5/3 2/3 19/3
1 3/2 1/2 11/2
1 2 2 11
Step 1: divide row 1 by 3, row 2 by 2
1 5/3 2/3 19/3
–1/6 –1/6 –5/6
1/3 4/3 14/3
Step 2: subtract row 1 from row 2 and row 3
1 5/3 2/3 19/3
1 1 5
1 4 14
Step 3: divide row 2 by -1/6 and row 3 by 1/3
1 5/3 2/3 19/3
1 1 5
3 9
Step 4: subtract row 2 from row 3
1 5/3 2/3 19/3
1 1 5
1 3
Step 5: divide Equation 3 by 3 Solution for Z!
1 5/3 2/3 19/3
1 2
1 3
Step 6: substitute Z solution into Equation 2. Solution for Y!
1 1
1 2
1 3
Step 7: substitute Y and Z into Equation 1. Solution for X!

FIGURE 6.10

Gaussian elimination and backward substitution in matrix view.

After Gaussian elimination, the matrix becomes a triangular matrix. This is a very popular type of matrix for various physics and mathematics reasons. We see that the end goal is to make the coefficient part of the matrix into a diagonal form, where each row has only a value 1 on the diagonal line. This is called an identity matrix because the result of multiplying any matrix multiplied by an identity matrix is itself. This is also the reason why performing Gaussian elimination on a matrix is equivalent to multiplying the matrix by its inverse matrix.

In general, it is straightforward to design a parallel algorithm for the Gaussian elimination procedure that we described in Fig. 6.10. For example, we can write a CUDA kernel and designate each thread to perform all calculations to be done on a row of the matrix. For systems that can fit into shared memory, we can use a thread block to perform Gaussian elimination. All threads iterate through the steps. After each division step, all threads participate in barrier synchronization. They then all perform a subtraction step, after which one thread will stop its participation since its designated row has no more work to do until the backward substitution phase. After the subtraction step, all threads need to perform barrier synchronization again to ensure that the next step will be done with the updated information. With systems of equations with many variables, we can expect reasonable amount of speedup from the parallel execution.

Unfortunately, the simple Gaussian elimination algorithm we have been using can suffer from numerical instability. This can be illustrated with the example:

$$5Y + 2Z = 16 \qquad \text{(Equation 1)}$$

$$2X + 3Y + Z = 11 \qquad \text{(Equation 2)}$$

$$X + 2Y + 2Z = 11 \qquad \text{(Equation 3)}$$

We will encounter a problem when we perform step 1 of the algorithm. The coefficient for the X variable in Equation 1 is zero. We will not be able to divide Equation 1 by the coefficient for variable X and eliminate the X variable from Equation 2 and Equation 3 by subtracting Equation 1 from Equation 2 and Equation 3. The reader should verify that this system of equation is solvable and has the same solution $X = 1$, $Y = 2$, and $Z = 3$. Therefore, the algorithm is numerically unstable. It can fail to generate a solution for certain input values even though the solution exists.

This is a well-known problem with Gaussian elimination algorithms and can be addressed with a method commonly referred to as *pivoting*. The idea is to find one of the remaining equations whose coefficient for the lead variable is not zero. By swapping the current top equation with the identified equation, the algorithm can successfully eliminate the lead variable from the rest of the equations. If we apply pivoting to the three equations, we end up with the following set:

$$2X + 3Y + Z = 11 \qquad \text{(Equation 1', original Equation 2)}$$

$$5Y + 2Z = 16 \qquad \text{(Equation 2', original Equation 1)}$$

$$X + 2Y + 2Z = 11 \qquad \text{(Equation 3', original Equation 3)}$$

Note that the coefficient for X in Equation 1' is no longer zero. We can proceed with Gaussian elimination, as illustrated in Fig. 6.11.

The reader should follow the steps in Fig. 6.11. The most important additional insight is that some equations may not have the variable that the algorithm is eliminating at the current step (see row 2 of Step 1 in Fig. 6.11). The designated thread does not need to do the division on the equation.

In general, the pivoting step should choose the equation with the largest absolute coefficient value among all the lead variables and swap its equation (row) with the current top equation as well as swap the variable (column) with the current variable. While pivoting is conceptually simple, it can incur significant implementation complexity and performance overhead. In the case of our simple CUDA kernel implementation, recall that each thread is assigned a row. Pivoting requires an inspection and perhaps swapping of coefficient data spread across these threads. This is not a big problem if all coefficients are in the shared memory. We can run a parallel max reduction using threads in the block as long as we control the level of control flow divergence within warps.

However, if the system of linear equations is being solved by multiple thread blocks or even multiple nodes of a compute cluster, the idea of inspecting data spread

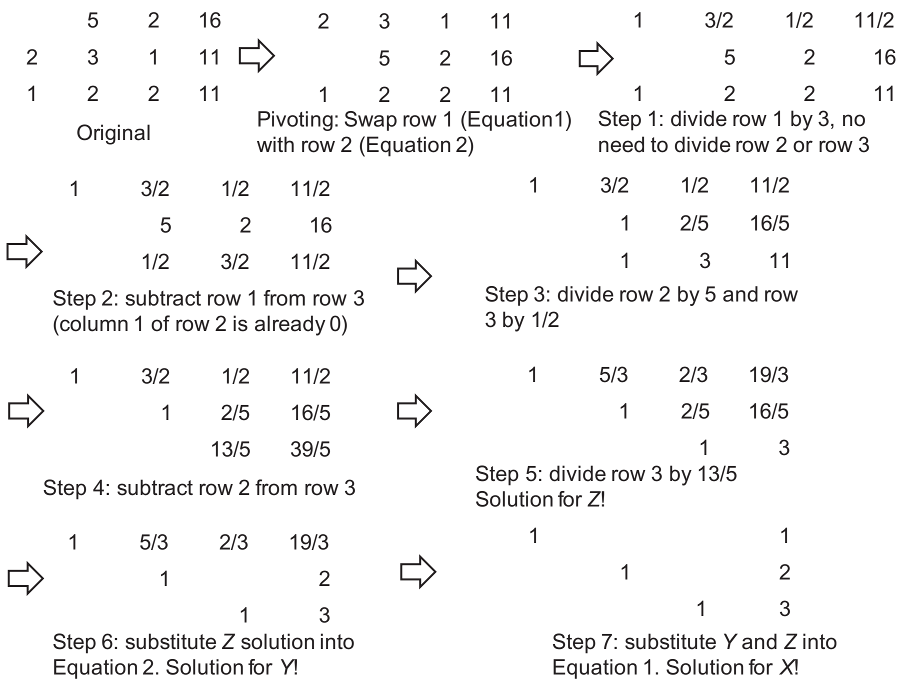

5 2 16
2 3 1 11
1 2 2 11
Original
2 3 1 11
5 2 16
1 2 2 11
Pivoting: Swap row 1 (Equation1) with row 2 (Equation 2)
1 3/2 1/2 11/2
5 2 16
1 2 2 11
Step 1: divide row 1 by 3, no need to divide row 2 or row 3
1 3/2 1/2 11/2
5 2 16
1/2 3/2 11/2
Step 2: subtract row 1 from row 3 (column 1 of row 2 is already 0)
1 3/2 1/2 11/2
1 2/5 16/5
1 3 11
Step 3: divide row 2 by 5 and row 3 by 1/2
1 3/2 1/2 11/2
1 2/5 16/5
13/5 39/5
Step 4: subtract row 2 from row 3
1 5/3 2/3 19/3
1 2/5 16/5
1 3
Step 5: divide row 3 by 13/5 Solution for Z!
1 5/3 2/3 19/3
1 2
1 3
Step 6: substitute Z solution into Equation 2. Solution for Y!
1 1
1 2
1 3
Step 7: substitute Y and Z into Equation 1. Solution for X!

FIGURE 6.11

Gaussian elimination with pivoting.

across multiple thread blocks or multiple compute cluster nodes can be an extremely expensive proposition. This is the main motivation for *communication avoiding algorithms* that avoids a global inspection of data such as pivoting [Ballard 2011]. In general, there are two approaches to this problem. Partial pivoting restricts the candidates of the swap operation to come from a localized set of equations so that the cost of global inspection is limited. This can, however, slightly reduce the numerical accuracy of the solution. Researchers have also demonstrated that randomization tends to maintain a high level of numerical accuracy for the solution.

6.7 SUMMARY

This chapter introduces the concepts of floating-point format and representable numbers that are foundational to the understanding of precision. Based on these concepts, we also explain the denormalized numbers and why they are important in many numerical applications. In early CUDA devices, denormalized numbers were not supported. However, later hardware generations support denormalized numbers. We have also explained the concept of arithmetic accuracy of floating-point operations. This is important for CUDA programmers to understand the potential lower accuracy of fast arithmetic operations implemented in the special function units. More importantly, the readers should now have a good understanding of why parallel algorithms often can affect the accuracy of calculation results and how one can potentially use sorting and other techniques to improve the accuracy of their computation.

6.8 EXERCISES

1. Draw the equivalent of Fig. 6.5 for a 6-bit format (1-bit sign, 3-bit mantissa, 2-bit exponent). Use your result to explain what each additional mantissa bit does to the set of representable numbers on the number line.

2. Draw the equivalent of Fig. 6.5 for another 6-bit format (1-bit sign, 2-bit mantissa, 3-bit exponent). Use your result to explain what each additional exponent bit does to the set of representable numbers on the number line.

3. Assume that in a new processor design, due to technical difficulty, the floating-point arithmetic unit that performs addition can only do "round to zero" (rounding by truncating the value toward 0). The hardware maintains sufficient number of bits that the only error introduced is due to rounding. What is the maximal ulp error value for add operations on this machine?

4. A graduate student wrote a CUDA kernel to reduce a large floating-point array to the sum of all its elements. The array will always be sorted with the smallest values to the largest values. To avoid branch divergence, he decided to implement the algorithm of Fig. 6.4. Explain why this can reduce the accuracy of his results.

5. Assume that in an arithmetic unit design, the hardware implements an iterative approximation algorithm that generates two additional accurate mantissa bits of the result for the sin() function in each clock cycle. The architect decided to allow the arithmetic function to iterate 9 clock cycles. Assume that the hardware fills in all remaining mantissa bits as 0's. What would be the maximal ulp error of the hardware implementation of the sin() function in this design for the IEEE single-precision numbers? Assume that the omitted "1." mantissa bit must also be generated by the arithmetic unit.

REFERENCES

Ballard, G., Demmel, J., Holtz, O., & Schwartz, O. (2011). Minimizing communication in numerical linear algebra. *SIAM Journal of Matrix Analysis Applications*, *32*(3), 866–901.

<http://developer.nvidia.com/content/precision-performance-floating-point-and-ieee-754-compliance-nvidia-gpus>.

IEEE Microprocessor Standards Committee, Draft Standard for Floating-Point Arithmetic P754, Most recent revision January 2008.

Kahan, W. (January 1965). Further remarks on reducing truncation errors. *Communications of the ACM*, *8*(1), 40. http://dx.doi.org/10.1145/363707.363723.

This page intentionally left blank