Teaching Computational Thinking for High-performance Programming

Iliya Georgiev, Ivo Georgiev

Abstract. *High performance programming needs skills, generalized as computational thinking, of designing algorithms to meet the major requirements in parallelism, execution efficiency, and memory bandwidth. The paper covers some important topics that are foundation for computational thinking: performance consideration, accuracy/precision and data dependencies.*

Key words: *performance, accuracy and precision, data dependencies, loop programming*

INTRODUCTION

Computational thinking is a key for efficient programming for all palettes from parallel supercomputers to embedded computers. It is considered as an intensive process of formulating problems based on good problem decomposition, selection of algorithms that achieve appropriate accuracy and precision, compromise among parallelism, computational efficiency, performance and information retrieval. The paper emphasizes on the knowledge of computer performance, precision of data processing and related programming techniques that are often needed for creating successful computational solutions to challenging applications.

Important remark: The authors have selected partly the examples in the following manuscript from some available sources that are widely used in both the university and professional societies.

.

PERFORMANCE ANALYSIS

High performance programmers should carefully analyze the performance issues when distributing a program between two independent processing units – central processing unit (CPU) and graphical processing unit (GPU). CPU executes different threads in high concurrent mode that is scheduled and synchronized by the operating system (OS) and run-time environment. GPU is oriented to data-parallel processing, tries to organize the computations in single-instruction-multiple-data mode, but provides simple scheduling and only barrier synchronization. GPU fetches the data from the main memory of the CPU, which is a significant latency in transferring huge information.

Memory hierarchy utilization is an important requirement to achieve high performance (recommended reading is [1]). The primary memory consists of processor registers, cache level one (separate instruction and data cache), cache level two, cache level three and main memory. The exchange of the information between those levels is controlled by a memory management unit and is hidden from the OS and run-time environment. The processor issues main memory addresses, the memory management unit checks whether the data is uploaded in cache one and down through the hierarchy. If the data is in the upper level, it is a *hit*. If it is not, it is called *miss* and the information has to be uploaded from the lower level. The time to upload the missing data block from the lower level is *miss penalty*, when the current thread waits several hundred clock cycles. Average memory access time for only two levels of primary memory (cache and main memory) is given by the formula:

$$\text{Memory access time} = \text{Hit time} + \text{Miss rate} * \text{Miss penalty},$$

where the *hit time* is the time to analyze miss or hit and the time to deliver the data to the upper level, *miss rate* is the percentage of misses related to all memory accesses.

Performance is estimated practically by the CPU time for executing a program. The CPU time formula considering again only processor and main memory is as follows:

$$\text{CPUtime} = IC * \{CPI + \text{Memory accesses/per instruction} * \text{Miss rate}_{cache} * \text{Miss Penalty}) * \text{Clock time},$$

CompSysTech'13 Local Proceedings
ISSN 1314-9687, pp. 36 - 41, 28-29 June 2013, Ruse, Bulgaria
Compright © University of Ruse, 2013

where *IC* is instruction count, *CPI* is clocks/per instruction considering also pipeline stalls because of hardware hazards, *Clock time* is the period of the processor main clock.

For memory hierarchy with 3 caches, *miss penalty* of every level has to be replaced with the memory access to the lower level. The CPU time formula considering the completely primary memory hierarchy now is:

$$\text{CPUtime} = IC * \{CPI_{execution} + \text{Memory accesses/instruction} * \text{Miss rate}_{cache\ 1} * [\text{Hit time}_{cache\ 2} + \text{Miss rate}_{cache\ 2} * (\text{Hit time}_{cache\ 3} + \text{Miss rate}_{cache\ 3} * \text{Miss Penalty}_{main\ memory})]\} * \text{Clock time}$$

Further, the performance analysis becomes more complicated considering the virtual memory organization, which is hardware/software mechanism under operating system control. If the information is not in the main memory, there is a v*irtual memory fault*. The operating system blocks the program under execution and uploads the page or segment from the hard disk to the main memory. After that, the program is unblocked and put in the ready queue of the OS scheduler. The fault could take thousands of clock cycles. In high performance programming, it is recommended general estimation of the code performance especially of cache misses and virtual memory faults in those part of the programs that manipulate huge information.

Programmers also have to understand how the compilers and the run-time environments store two-or-three-dimensional arrays in sequential virtual memory. For example, most C/C++ compilers put two dimensional array row by row (row order), but FORTRAN compilers lay out such arrays column by column (column order).

Let us consider nested loops that access array data stored in memory in row or column order. Programming the right nesting of the loops can make the loop expressions retrieve the data in the order in which they are stored. In case the array cannot fit in the cache or main memory pages assigned for the program, precise nesting reduces cache misses and/or virtual memory faults using all data in a cache or page before they are replaced.

Assume a program, which declares two-dimensional array

```
int Array2D[][] = new int[128][128]; //the array is laid out row by row
```

The program is activated by OS that implements least recently used replacement policy of a paged virtual memory organization (page size is 1 Ki = 1024 bytes). The OS assigns three pages in the main memory. The program occupies the first page, during execution the array partly will be uploaded by demand in the next two pages (initially they are empty). Let us estimate the number of virtual memory faults for two implementations of the code, which is a nested initialization loop. At the beginning, there is one fault to load two rows (256 words) in the first page and another fault for the next two rows. The first version is:

```
for (int j = 0; j < 128; j++){
    for (int i = 0; i < 128; i++){
        Array2D[i][j] = 0;}}
```

For every two iterations of the internal loop there will be one fault. For the entire execution, the virtual memory faults will be 9192.

In the next version, the internal and the external loops are interchanged. During every 512 executions of the internal loop, there will be two faults.

```
for (int i = 0; i < 128; i++){
    for (int j = 0; j < 128; j++){
        Array2D[i][j] = 0;}}
```

Finally, there are only 64 virtual memory faults.

Loop interchange does not show such performance improvement if both row and column elements are manipulated in the expressions of every iteration. In such cases, the programmer has to decompose the loop writing sub-loops that do not process the entire array but operates on blocks.

FLOATING POINT ACCURACY AND PRECISION

Writing high performance programs that use both central and graphical processors needs strong understanding and analysis of the accuracy and precision of floating-point arithmetic Floating-point formats (single, double and extended) are defined in IEEE 754 standard [3], which

follows the following principles. For normalized numbers the significant (mantissa) is *1.xx..xx*, where integer *1* is called implicit integer bit and is not presented. The exponent is encoded in Excess ($2^{m-1} - 1$) code (*m* is the number of the exponent bits) and all exponents are positive biased numbers. For example, in single floating format the exponent is 8 bits, m=8, $2^{8-1} - 1=127$, 127 is added to the original exponents. The number of the significant bits gives the precision of the format: 23 bits in single format, 52 in double format. Most of the programming languages support only single and double format. For higher precision, some development environments provide special subroutines.

Presentation of zero is by special code (exponent=significant = 0). Additionally special codes are foreseen for infinity and not-a-number (NaN). Floating-point numbers approximate real numbers with limited precision and the arithmetic operations can generate loss of significance. The maximal loss of significance (relative error) after executed operations is the estimation of the accuracy of floating-point arithmetic. Higher accuracy needs such loss to be as smaller as possible. The most common source of significance loss is when the operation generates a result that cannot be represented with exact precision. The programmer has to define some small number ε *(epsilon)*, which is the maximum loss of significance of the calculations (recommended reading is [6]).

When comparing two floating-point values (float or double) for equality it is not recommended to use the equality operator (==). It might consider the numbers to be "relatively equal using **ε**" even if they aren't exactly equal. The following technique with subtraction is preferable to compare floating point numbers *f1* and *f2*:

```
if (Math.abs (f1 - f2) < ε) printf ("Relatively equal!");
```

IEEE 754 requires rounding of the arithmetic operations. Rounding should be done if the significant of the result value needs too many bits to be exact. Rounding is necessary typically when shifting right of the significant in addition and subtraction, when two input operands have different exponents, the significant of the one with the lesser exponent has to be shifted right loosing significant bits. Loss after right shift of the lesser number significant can be several bits.

Because of the limited precision the floating point addition is not associative. Following the recommended reading [2], this can be illustrated with a simple example based on a 5-bit representation that follows the principles of IEEE 754: 1-bit sign, 2-bit exponent, and 2-bit significant (integer bit '1.' not presented). The exponent is Excess ($2^{2-1} - 1 = 1$) *code.* The possible exponents are from $-1_{(10)} = 00_{(2)}$ to $2_{(10)} = 11_{(2)}$ The major intervals are between powers of twos. With two bits of exponent, there are four powers of 2 ($2^{-1} = 0.5$, $2^0 = 1.0$, $2^1 = 2.0$, $2^2 = 4.0$), and each defines a number interval.

Assume that we need to add $0.5_{(10)}=1.0_{(2)} * 2^{-1}$ (the 5-bit code is 0 - sign, 00 – exponent, 00 – significant), with $4_{(10)}=1.0_{(2)} * 2^2$ (the 5-bit code is 0 11 00).

Due to the difference in exponent values, the significant value of the first number must be right shifted to make the exponents equal, after that to add the significants and at the end to perform normalization (if needed) of the result. The addition becomes $(0.00_{(2)} + 1.0_{(2)}) * 2^2$ (the 5-bit code is 0 - sign, 11 – exponent, 00 –significant), but has to be $(0.001_{(2)} + 1.0_{(2)}) * 2^2$. Here there is loss of significance.

Using rounding, the loss should be half the value of the least significant bit (LSB) that is often called machine epsilon. If the computer is designed to perform arithmetic and rounding operations perfectly, the greatest error of arithmetic operation should be no more than half of the LSB. In practice, some of the more complex arithmetic-logical operations, such as division and transcendental functions, are typically implemented with iterative approximation algorithms. If the computer does not perform a sufficient number of rounding, the result may have an error larger than half of the LSB.

High performance algorithms often must calculate a large number of values. For example, the dot product in matrix multiplication needs to add pair-wise products of the elements of both matrices. Ideally, the order of adding these elements should not affect the final result if addition is an associative operation. The order of summation because of the finite precision affects the accuracy of the result and in general addition is not associative.

Assume sequential addition of the following numbers:

$$
\begin{aligned}
&3.0_{(10)} + 1.0_{(10)} + 0.5_{(10)} + 0.75_{(10)} = \\
&= 1.0_{(2)}*2^{1}\ [5\text{-bit code } 0\ 10\ 00] + 1.0_{(2)}*2^{0}\ [5\text{-bit code } 0\ 01\ 00] + \\
&+ 1.0_{(2)}*2^{-1}[5\text{-bit code } 0\ 00\ 00] + 1.1_{(2)}*2^{-1}[5\text{-bit code } 0\ 00\ 10] = \\
&= 1.0_{(2)}*2^{2}\ [0\ 11\ 00] + 1.1_{(2)}*2^{-1}[0\ 00\ 10] = \\
&= 1.0_{(2)}*2^{2}\ [0\ 11\ 00]
\end{aligned}
$$

After the addition of first two numbers the result is $1.0_{(2)}*2^2$ [5-bit code 0 11 00]. Summation with the third one gives $1.0_{(2)}*2^2$, but the exact result should be $1.001*2^2$, which is a loss of precision. Continuing addition, there is again loss of precision and we receive final sum $1.0_{(2)}*2^2$ [5-bit code 0 11 00].

Further, let us consider parallel execution, where 0.5 + 0.75 are added in one processor thread and 3.0 + 1.0 in another. The algorithm then adds the pair sums:

$$
\begin{aligned}
&[0.5_{(10)} + 0.75_{(10)}] + [3.0_{(10)} + 1.0_{(10)}]= \\
&= 1.01_{(2)}*2^{0}\ [5\text{-bit code } 0\ 01\ 01] + 1.0_{(2)}*2^{2}\ [5\text{-bit code } 0\ 11\ 00] = \\
&= 1.01_{(2)}*2^{2}\ [5\text{-bit code } 0\ 11\ 01]
\end{aligned}
$$

The result is $1.01*2^2$ [5-bit code 0 11 01], which is more precise than the sequential execution. This is because the sum of the third and fourth numbers is large enough that it can now affect the addition result. Depending on the estimated precision of the application the experienced programmers can accept such variations in the result or reorganize the program to ensure that the data are sorted or grouped in such a way that provides the most accurate results.

A common technique to maximize floating-point arithmetic accuracy is to presort data in ascending numerical order before computation, if it is possible. Dividing the numbers into groups guarantees that numbers with numerical values close to each other are in the same group and the results are more accurate. Having the numbers in ascending number provides greater accuracy. Sorting is frequently used in massively data parallel numerical algorithms. However, if there are dependencies between the numbers (weak data parallelism) simple sorting is not efficient and more complicated analyses has to be used.

In addition, the previous example illustrates the reason for usage of Excess code for floating point exponent: the smallest numbers have the smallest exponent, which is not possible if 2's complement code is used, where negative numbers are encoded by greater binary codes.

The special codes for infinity and NaN can be used to organize checkpoints for loss of precision during the debugging of high performance programs. All representable numbers are between negative infinity and positive infinity. Exponent overflow/underflow or division by 0 can create infinity. Any numbers divided by infinity result in 0.

A NaN is generated by operations whose input values do not make sense—for example, 0/0, 0*1,1/1,1–1. In some languages, for example JavaScript, NaN are also used for declared but not initialized data. There are two types of NaNs in the IEEE standard: signaling and quiet. Signaling NaNs (SNaNs) has 0 in the first bit of the significant, whereas quiet NaNs has 1. A signaling NaN causes an exception when used as an operand. Signaling NaNs are used in situations where the programmer would like to make sure that the program execution is interrupted whenever any NaN values are used in floating-point computations. Usual practice is to store all of the uninitialized data as signaling NaN and to discover wrong addressed operands. Quiet NaN generates a quiet NaN when used as input to arithmetic operations; for example, the operation (1.0*quiet NaN) generates a quiet NaN. Quiet NaNs are typically used in applications where the programmer can detect data corruption.

Some programming teams have standard very well proved subroutines that use infinity or NaN and can be incorporated in the programs to analyze possible accumulated loss of precision.

DATA DEPENDANCIES AND LOOP PROGRAMMING

Distributing high performance program between central processor and graphical processor needs both analysis of the data and skills in writing subroutines (especially loops with reduced data dependence) that will be given to graphical processor for execution.. The graphical processor is especially efficient for data-parallel computations - the same program is executed on many data elements in parallel with high arithmetic intensity (the ratio of arithmetic operations to memory accesses). Data-parallel processing maps data elements to parallel processing threads. Many applications that process large data sets can use a data-parallel programming model to speed up

the computations. For example, in 3D image rendering, large sets of pixels and vertices are mapped to parallel threads.

Here is one example of a subroutine written in CUDA C language. CUDA C development environment extends C language by allowing the programmer to define C functions, called *kernels.* The kernel subroutines are executed *n* times in parallel by *n* different *CUDA threads*, as opposed to only once execution like regular C functions [4].

A kernel is defined using the *__global__* declaration specifier. The number of CUDA threads which execute that kernel for a given kernel call is specified using a new <<<…>>>*execution configuration* syntax (see CUDA C Language Extensions [5]). Each thread that executes the kernel is given a unique *thread* identifier that is accessible within the kernel through the built-in *threadIdx* variable.

For multithreading processing *threadIdx* is a 3-component vector, so that threads can be identified using one-dimensional, two-dimensional, or three-dimensional *thread index*, forming an one-dimensional, two-dimensional, or three-dimensional thread block. This provides a natural way to invoke computation across the elements in array data structures with different dimensions.

The index of a thread and its thread identifier relate to each other in a straightforward way. For one-dimensional block, they are the same; for a two-dimensional block of size (D_x, D_y), the thread identifier of a thread of index (x, y) is $(x + y\,D_x)$; for a three-dimensional block of size (D_x, D_y, D_z), the thread identifier of a thread of index (x, y, z) is $(x + y\,D_x + z\,D_x\,D_y)$.

As an example, the following code adds two matrices *A* and *B* of size $n \times n$ and stores the result into matrix *C*:

```
// Kernel function for matrix multiplication
__global__ void MultMatrix(float X[n][n], float Y[n][n],
                           float Z[n][n]) {
    int i = threadIdx.x;
    int j = threadIdx.y;
    Z[i][j] = X[i][j] * Y[i][j];
}
int main(){
     ...
    // The kernel function is called with one GPU block of n * n * 1 threads
    int numOfBlocks = 1;
    dim3 threadsInBlock(n, n);
    MultMatrix<<<numOfBlocks, threadsInBlock>>>(X, Y, Z);
      ... }
```

The previous example is fully data independent and convenient for parallel processing. The expression `Z[i+1][j+1] = X[i+1][j+1] * Y[i+1][j+1] does` not depend on the result of the previous expression `Z[i][j] = X[i][j] * Y[i][j].`

There are many applications with data dependencies in loop expressions or array references, which make parallel execution not efficient in graphical processor. Dependencies also generate hazards and pipelining stalls in both CPU and GPU. Understanding data dependencies is fundamental in implementing efficient subroutines in multi core and multi threaded computers. The CPU time depends on the longest sequence of dependent expression, since calculations that are related to the previous calculations in the expression sequence must be executed in order. However, most algorithms do not consist of just a long chain of dependent calculations; there are usually opportunities to execute independent calculations in parallel.

In the following example, we will use the hardware terminology for dependencies: read-after-write that are also called true dependencies (expression *j* depends on the result of the previous expression *i*) , write-after-read (expression *j* overwrites some variable before the previous expression *i* reads it as a result) and write-after-write (expression *j* overwrites some variable before the previous expression *i* writes it as a result):

```
for (i=2; i<10000; i++) {
    X[i] = Y[i] + X[i];     // E1 – expression 1
    Z[i-1] = X[i] - W[i];   // E2 – expression 2
    X[i-1] = Y[i] * a;      // E3 – expression 3
    Y[i+1] = Y[i]- b;   }   // E4 – expression 4
```

The dependencies are:

a. Read-after-write from E1 to E2 on X[i]: E2 reads X[i] after E1 writes it.

b. Read-after-write from E4 to E1 on Y[i+1]: it is a loop-carried dependence because E1 in the next iteration reads Y[i+1] after E4 in the current iteration writes it.

c. Read-after-write from E4 to E3 on Y[i+1]: it is a loop-carried dependence because E3 in the next iteration reads Y[i+1] after E4 in the current iteration writes it.

d. Read-after-write from E4 to E4 on Y[i]: it is loop-carried dependence because E4 in the next iteration reads Y[i+1] after E4 in the current iteration writes it.

e. Write-after-write from E1 to E3 on X[i]: it is loop-carried dependence because E1 in the next iteration writes X[i+1] after E3 in the current iteration writes it.

f. Write-after-read from E1 to E3 on X[i]: it is loop-carried dependence because E1 in the next iteration reads X[i] after E3 writes it in the current iteration of the loop body.

g. Write-after-read from E2 to E3 on X[i]: it is loop-carried dependence because E2 in the next iteration reads X[i] after E3 writes it in current iteration of the loop body.

For efficient parallel-multithreaded execution, loop iterations must be independent of all others. The presented loop is purely written and cannot be distributed in different cores or given to GPU for execution. The true dependences *b, c* and *d* are loop-carried, so the loop iterations are dependent (such loop is declared not parallel). The code has to be rewritten to some code that is functionally equivalent of the first version and does not have loop-carried dependencies.

CONCLUSIONS

A designer with strong computational thinking has to be able to create programs by analyzing the whole functionality both of the computer and the running program. The paper summarizes some of the knowledge that can build the needed computational thinking for high performance programming:

a. Analyzing the migrating of the data between the levels of the memory hierarchy (caching and virtual memory), memory bandwidth and estimation in advance the maximum and minimum performance, array data layout and loop transformations.
b. Achieving floating-point precision and accuracy using sorting and other techniques.
c. Understanding parallel execution models, data dependencies for efficient decomposition of the program in parallel and multithreaded central and graphical processors.

The ground rules, presented in the paper, allow the programming teams to be much more creative in applying efficient algorithm techniques.

REFERENCES

[1] Hennessy J., Patterson D., Computer Architecture: A Quantitative Approach, Morgan Kaufman, 2012.

[2] Kirk D., Hwu W., Programming Massively Parallel Processor, Morgan Kaufman, 2010.

[3] IEEE 754 Standard for Floating Point Arithmetic, /ieeexplore.ieee.org/, 2008.

[4] NVIDIA CUDA C language Programming Guide, http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, 2011.

[5] NVIDIA CUDA C language extension, http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#c-language-extensions, 2012.

[6] David Monniaux, The pitfalls of verifying floating-point computations". *ACM Transactions on Programming Languages and Systems* 30 (3): article #12. doi:10.1145/1353445.1353446. ISSN 0164-0925, 2008.

ABOUT THE AUTHORS

Iliya Georgiev, Prof. Dr., Department of Mathematical and Computer Sciences, Metro State University of Denver, Campus box 38, P.O. Box 173362, Denver 80217, USA, E-mail: gueorgil@mscd.edu, ilgeorg@yahoo.com; URL: http://rowdy.mscd.edu/~gueorgil/.

Ivo Georgiev, Ph.D., TeraFold Biologics, Lafayette, CO 80026, E-mail: ivogeorg@gmail.com