Parallel, Arbitrary-Precision Arithmetic

Charles Cary

Goal

This project began as an attempt to implement the RSA algorithm on top of Nvidia's CUDA platform. One key aspect of the RSA algorithm is its reliance on large numbers, specifically for the large keys that make the algorithm an effective way to protect information. After working on this problem for just a few days, I realized that the absence of a library for arbitrary-precision arithmetic, the library I would need to manipulate these large numbers, would make my original project especially difficult. Therefore, I set out to develop one of these libraries: a parallel, arbitrary-precision arithmetic library, implemented using CUDA.

Problem

However, creating one of these libraries is a non-trivial problem; the issue with doing parallel arithmetic is that it is hard to isolate the sub-parts of the arithmetic. The 'carrying' and 'borrowing' of digits during addition, multiplication, subtraction, and division creates a sort of necessary serial component to any arithmetic. For example, in addition, one must add the one's place before the tens and tens before hundreds because of the carries that could possibly be created during the addition. Second, even if these seemingly, necessarily-serial components of arithmetic can be made to be done in parallel, the internal dependencies in the arithmetic of numbers means that the various threads of execution working on the problem must be synchronized in such a way that the correct result is obtained despite non-serial computation. This is what makes parallel arithmetic difficult and this is what limits its performance because this synchronization limits the speedup that one can gain from parallelization.

Approach and Methods

As indentified earlier, the first problem of parallel arithmetic is overcoming the inherent-serial nature of arithmetic. Like many computer science problems, the solution to this problem rests in how data is to be structured; the solution created for this paper represents each digit of a number with its own unique binary encoding, specifically its own 32-bit, 2's complement integer, and the entire number is stored as an array of these 32-bit integers, one per decimal digit. Furthermore, the position of the decimal place within the number and the number of digits that compose the number are each stored in their own unsigned, 32-bit integer. The reason for doing this, is that, with slight modification to our typical addition and multiplication algorithms, this representation will allow us to achieve near-independence between the digits when doing arithmetic, which allows for the parallelization of this arithmetic.

With numbers stored in this manner, we will define addition and multiplication in much the same way one would add or multiply two numbers on paper. For addition, we will add two numbers by summing pairs of digits, one from each number, that correspond to the same decimal place. (See fig. 1)

$$a_1a_2a_3...a_n + b_1b_2b_3...b_n = (a_1 + b_1)(a_2 + b_{21})(a_3 + b_3)...(a_n + b_n)$$

Figure 1: addition of numbers stored in this manner and pairing of digits

For multiplication, we will multiply two numbers in the same way most learned to in grade school: by multiply individual digits of the second number against the first to create temporary values that are shifted by placing zeros equal to the position of the digits being multiplied against the first number and finally summing the result. (See fig. 2)

From these definitions, two issues immediately appear: an overflow error could occur within one of the digits during addition, corrupting the addition of the two numbers, and when we wish to view the result of this computation, it is no longer stored in the typical way for decimals where each digit can only contain the values 0 through 9. Instead, individual digits can take on any of 2^{32} values.

$$
\begin{aligned}
&a_1a_2a_3...a_n * b_1b_2b_3...b_n = C \\
&C = \sum_{i=0}^{n-1} c_i \\
&c_0 = (b_n * a_1)(b_n * a_2)...(b_n * a_n) \\
&c_1 = (b_{n-1} * a_1)(b_{n-1} * a_2)...(b_{n-1} * a_n)(0) \\
&c_2 = (b_{n-2} * a_1)(b_{n-2} * a_2)...(b_{n-2} * a_n)(0)(0) \\
&c_x = (b_{n-x} * a_1)(b_{n-x} * a_2)...(b_{n-x} * a_n)[(0)\cdot x]
\end{aligned}
$$

Figure 2: multiplication is similar to grade-school multiplication

The solution to both of these problems rests in normalizing the values stored in the arrays when either of these problems occurs. Here, I define normalization as the process by which a number is returned to a form that resembles the standard way by which human's express numbers; each digit, or now each cell of the array, that represents the digits of a number, is recomputed so that it only represents values zero through nine, yet the value of the number in the array remains the same. The algorithm is quite simple to achieve this; for each cell in the array, mod by ten to determine what needs to be carried over to the cell that represents the next most significant digit in the number. Store this value. Now divide the value in the array by ten and store it in the array. Now, take the carry value we stored and add it to the value stored in the next most significant cell in the array. Repeat this process until the absolute value of no digit is greater than nine. (See fig. 3) Using this algorithm, carries can be done in parallel because each of the digits can be handled simultaneously.

```
Let A = $a_1a_2a_3...a_n$

while(keepcarrying)
{
C = $a_n$ / 10
        $a_n$ %= 10
        $A_{n-1}$ += C
        If any $a_n > 9$
                keepcarrying = True
        Else
                Keepcarrying = False
}
```

Figure 3: pseudocode for normalization

With the process of normalization comes the next question: when does normalization need to be invoked in order to protect against overflow? For addition, we simply check if either of the highest bits of pairs of digits are set to one and normalize both values if either bit is one. For multiplication, the two numbers are to be normalized before multiplication begins and then the same check for addition must be done for each addition during multiplication.

A final issue with normalization is the memory needed to store the normalized values. As the structure of each of these numbers permits 2^{32} unique values to be stored in each digit, instead of the more familiar ten, once a number is normalized, the amount of memory required to store that number grows. The maximum amount of space needed to store this number is governed by this equation (See fig. 4):

$$x = \frac{\ln(2^{z} - 1)}{\ln(10)} - 1 + n$$

Figure 4: where z is the number of bits used to store each digit (32 for this paper's implementation) and n is the number of digits that the number currently has

In the implementation that accompanies this paper, 32 bits are used to store each value, so for a number of n digits, n + 10 digits should be allocated in order to store a normalized value. The derivation of this equation can be found in fig. 5. Furthermore, it should be noted that through inspection, it was determined that the result of the sum of two numbers never has more digits than the longer of the two numbers plus one. For multiplication, the result is simply never longer than the sum of the number of digits of each number.

Results

The implementation that accompanies this paper features working addition, multiplication, and subsequently normalization. It is able to compute arithmetic correctly for numbers up to 2^{32}-1 digits.

$$\left(2^{32}-1\right)\sum_{i=0}^{n}10^{i} = (10)\sum_{i=0}^{x}10^{i}$$
$$\left(2^{32}-1\right)\frac{10^{n+1}-1}{9} = (10)\frac{10^{x+1}-1}{9}$$
$$\ln((2^{32}-1)(10^{n+1}-1)) = \ln(10^{x+2}-10)$$
$$\ln(10^{x+2}-10) < \ln(10^{x+2})$$
$$\ln(2^{32}-1)+\ln(10^{n+1}-1) = (x+2)\ln(10)$$
$$\ln(2^{32}-1)+\ln(10^{n+1}) = (x+2)\ln(10)$$
$$x = \frac{\ln(2^{32}-1)+\ln(10^{n+1})-2\ln(10)}{\ln(10)}$$
$$x = \frac{\ln(2^{32}-1)}{\ln(10)} + (n-1)$$
$$x \approx 10$$

Figure 5: the derivation of the equation in fig. 4. Here, we are solving for x and z has been set to 32.

However, this is not an ideal solution because of the many limitations imposed on it through the use of CUDA. First, memory cannot be allocated directly on the GPU in CUDA. This means that numbers must be created on the CPU and then pushed over to GPU. This creates a performance issue because only implementations that perform many computations on a few numbers would be efficient; computations that perform a few arithmetic operations on many numbers would face a memory bottleneck in sending all of this data to the GPU. Second, in order to preserve the values of the operands during arithmetic, and to properly perform normalization, during a computation of addition or multiplication, the two operands must be copied into temporary buffers in order to perform computation. This also must be done because a possible normalization means that the memory requirement to store a number may grow during arithmetic. This means that any arithmetic operation is also accompanied by two copy operations in the GPU's global memory. As the global memory of the GPU is relatively slow, compared to shared memory or registers, this limits the performance of this implementation for all sized numbers and especially limits performance on extra large numbers. Finally, as the serial nature of arithmetic has not been fully escaped by the structure of our data, synchronization is needed between the threads of the program in many instances, specifically to prevent threads from wrecking each other's data, which limits the potential speedup of parallelization. Furthermore, CUDA's treatment of synchronization further limits the effectiveness of the implementation; CUDA only allows for the synchronization of threads in a given block, and as synchronization cannot be escaped in arithmetic, this limits the implementation to using just 512 threads in one block. An argument could be made for using more threads and creating a system to lock global memory through some sort of global mutex. However, as global memory is relatively slow and as more and more threads demand more and more locking, this would create a severe performance bottleneck for the implementation. For these three reasons, the implementation that accompanies this paper is limited in its performance.

Conclusions

For these reasons, CUDA is not the ideal platform to implement a parallel, arbitrary-precision arithmetic library. CUDA is good for problems that can be done in parallel that do not require many threads to be synchronized. The heavy inter-dependence of the digits of numbers during computation means that synchronization is necessary in order to calculate correct values. Furthermore, the inability to directly allocate and set memory on the GPU creates a bottleneck for all but the simplest arithmetic.

The ideal platform for parallel, arbitrary-precision arithmetic would counter these two problems. First, it would allow for the effective synchronization of all of the threads during computation. This could be achieved through a small but fast memory, readable and writable by all of the GPU's cores. This memory would work as a space for global control over all of the threads. In order to achieve this speed, it should be on the same die of the cores, not off die like the global memory. In many ways, it should be like the unified L2 cache of some CPU's with multiple cores. Second, the ideal platform would allow for the allocation and setting of memory on the GPU itself. This would eliminate the issue of pushing data over from the main memory in order to do computation and pushing it back in order to view the results. This could be achieved by having a small CPU directly on the GPU that is responsible for memory related functions. There already is talk of doing this.

Next Steps

This implementation is by no means done; many functions essential to an arithmetic library have yet to be completed. In order to perform division, a reciprocal function needs to be developed. One option is to use the Newton-Raphson method built on top of the already working addition and multiplication functions. Second, a function to convert negative numbers to human readable form needs to be created. Third, functions to compare numbers, like greater than, less than, and equal to, need to be created. Finally, a way to monitor the amount of memory numbers are using, and to minimize the amount of unnecessary space numbers occupy, needs to be developed for any practical library.

Sources

"Cuda Developer Documentation". http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/online/group__CUDART__MEMORY_gb17fef862d4d1fefb9dba35bd62a187e.html

"How to get Cuda Device properties". Accessed December 18, 2010. http://codereflect.com/2008/09/22/how-to-get-cuda-device-properties/

Jason Sanders and Edward Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming* (Boston: Addison-Wesley Professional, 2010)

"Nvidia Forums". http://forums.nvidia.com

"Reciprocal & Square Root". Accessed December 19, 2010. http://www.derekroconnor.net/NA/Notes/RecSqRoot.pdf

Wikipedia.org was extensively consulted throughout this project.