Interval Arithmetic-based FFT for Large Integer Multiplication

Zibo Gong*, Nathan Zhu*, Matthew Ngaw*, Joao Rivera†,
Larry Tang*, Eric Tang*, Het Mankad*, Franz Franchetti*
*Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA
†Department of Computer Science, ETH Zurich, Switzerland
*{zibog, njzhu, mngaw}@andrew.cmu.edu, †hectorr@inf.ethz.ch, *{lawrenct, erictang, hmankad, franzf}@andrew.cmu.edu

***Abstract*—In this work we propose an interval arithmetic Fast Fourier Transform (FFT) algorithm for large integer multiplication on both CPU and GPU. We utilize techniques of double-double precision, shared memory, and thread parallelization to improve both the efficiency and accuracy of our implementation. Early results show that for CPU, we can achieve correctness on factors of billions of digits in size. On GPU, we see performance speedups compared to existing software libraries, lowering computation cost without adversely impacting accuracy of the result.**

***Index Terms*—Large integer multiplication, interval arithmetic, FFT**

I. Introduction

The multiplication of large integer strings is an important kernel in cryptography systems, number theoretic tasks, computer algebra systems, as well as arithmetic software libraries. The naive method of multiplying two N-digit integers requires $\mathcal{O}(N^2)$ operations which quickly becomes slow and infeasible in an actual application. However, there exists a number of different divide-and-conquer algorithms to reduce the computational complexity of large integer multiplication, including the commonly used Karatsuba algorithm and the Number Theoretic Transform (NTT). Actual implementations of these algorithms are found in arbitrary-precision software libraries such as the GNU Multiple Precision Arithmetic Library (GMP) [1]. Multi-precision library routines performing exact multiplication will typically operate over integers in order to provide provably correct results.

A widely known method for integer multiplication is to use the faster floating-point FFT [2], but this is not implemented in many real systems due to round-off errors associated with floating point arithmetic. One technique to overcome floating-point errors is interval arithmetic [3] which places bounds on rounding errors to provide mathematically correct results. All arithmetic computations then occur on an interval which contains a real number that has some uncertainty. Final multiplication is precise if and only if each digit interval of results bound exact one integer (i.e. $floor(upperbound) == ceil(lowerbound)$).

Contributions. In this paper, we present an interval-arithmetic FFT-based method for large integer multiplication which instead bounds the final integer result within an interval. We show that this technique produces correct results up to billions of bits.

II. Integer Multiplication via Interval Arithmetic FFTs

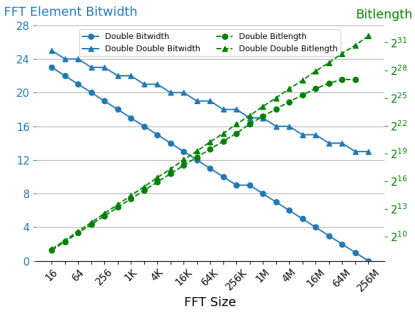
FFT-based Integer Multiplication. The FFT-based multiplication relies on the well known fast convolution via FFT algorithm. One can express the multiplication of integers as a linear convolution between the digits, which is discussed more concretely as follows. Given two integer strings, a and b, bits are first packed into the elements of two input vectors, $x[n]$, $y[n]$. To ensure a linear convolution is performed, each vector is zero padded such that the second half is all zeros. Then, we take the FFT of each to get $X = FFT(x), Y = FFT(y)$. This is followed by an element-wise product, $Z = X \odot Y$ and an inverse FFT to obtain the product vector, $z = FFT^{-1}(Z)$. A final carry over step then gives the final product in the proper form.

Interval-Arithmetic FFT. The shortcomings of floating-point arithmetic are exacerbated by our choice of Cooley-Tukey for our FFT algorithm, who's floating-point error grows on a logarithmic scale in the worst case [4]. Thus, we adopt interval arithmetic, which we utilize by bounding all required constants in the algorithm as well as converting each bit-packed element of the two input vectors into a corresponding interval. Since the algorithm requires certain trigonometric constants, we bound π using machine epsilon constants to ensure results have at most one bit of error in double precision. Additionally, we perform all operations in the multiplication pipeline with its interval complement given by [5].

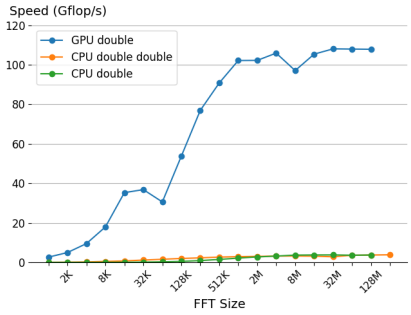
Algorithm Interval-arithmetic FFT-based Integer Multiplication

Input: vector x and y of size N
Output: vector z of size 2N

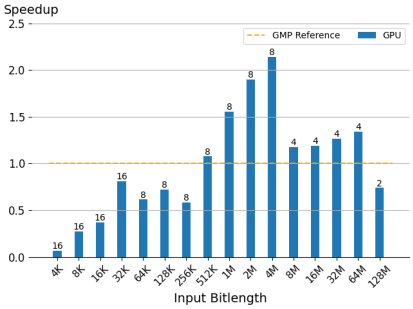
$i1 \leftarrow$ **ZeroPad**(x)
$i2 \leftarrow$ **ZeroPad**(y) ▷ zero padding to size of 2N
$f1 \leftarrow$ **IntervalFFT**$(i1)$
$f2 \leftarrow$ **IntervalFFT**$(i2)$
$prod \leftarrow$ **Mul**$(f1, f2)$ ▷ point-wise multiplication
$raw\_retv \leftarrow$ **IntervalIFFT**$(prod)$
$z \leftarrow$ **Carry**(raw_retv) ▷ Propagate carries



(a) Maximum bitwidth per input vector element and corresponding bitlength



(b) Gflop/s of algorithm on GPU and CPU



(c) Comparison with GMP

Fig. 1: (a) The maximum bitlength using double is 128M while for double-double implementation we can reach billions of bits. (b) On GPU, performance peaks between 16M and 128M at around 108 Gflop/s. On CPU, performance peaks at 4 Gflop/s for both double and double-double implementations at 32M and 256M, respectively. (c) The number of bits packed in each FFT element is written above the bar. The smallest possible FFT that can pack enough bits into each element to represent the operands was used to generate this plot.

III. Results

Experimental Setup. We tested our algorithm on two platforms: an Intel Xeon E7-4850 v3 with 3 TB of RAM and an NVIDIA Tesla V100-32GB. A radix-2 iterative Cooley-Tukey FFT is implemented on both platforms. We avoid expensive bit reversal operations by implementing a decimation in frequency (DIF) forward FFT and decimation in time (DIT) inverse FFT.

Max bitwidth and bitlength. Using more bits to represent a vector element can provide more performance speedup due to smaller FFT sizes but will lead to a higher likelihood of losing multiplication precision. A precision error takes the form of intervals bounding more than one integer before the "carry" portion of the pipeline. In the CPU implementation, we used double-double (128 bits floating point data type) interval arithmetic [3] to alleviate the round-off errors. In the GPU implementation, we used double interval arithmetic since double-double is not yet supported.

We obtained the maximum input length that can be supported by first obtaining the maximum bitwidth for an input vector element for a given FFT size and then multiplying the two numbers. The aggregated results are shown in the Figure 1a. The maximum bitwidth follows an approximately linear decrease w.r.t. FFT sizes. Using double-double we can reach billions of bits. However it greatly undermines the multiplication performance due to more flop involved.

Performance. The performance results on both platforms are demonstrated in Figure 1b. We calculate the Gflop/s by counting all the operations included in the multiplication pipeline and dividing by runtime. For our CPU implementation, we used OpenMP to help speed up our algorithm through parallelization. On the GPU, we used shared memory to reduce expensive global memory access for small-size inputs.

Comparison to GMP. Using the optimal bitwidth that can be packed in an element, we tested our GPU implementation against GMP. GMP is used to compare computation time and check for correctness as well. For GMP we used `mpz_init`, `mpz_urandomm`, `mpz_mul` to initialize and multiply random inputs of given sizes. Speedup is shown in Figure 1c. The labels above the bars represent the optimal bitwidth without precision error. We see a gradually increasing speedup when bitwidth is fixed but when it comes to an edge point where current bitwidth cannot be supported due to the round-off errors the speedup drops a bit and then increases again with increasing FFT size if bitwidth keeps identical. We see a comparable performance to the GMP libraries and higher than 2x speedup in the case where input size is 4M.

IV. Conclusion and Future Work

The proposed multiplication pipeline utilizes both FFT convolution to reduce computational complexity as well as interval arithmetic for greater numerical precision. We believe the demonstrated speedups and precision improvements are a strong indicator that going beyond proof-of-concept could have significant results. Specifically, there is notable room for optimization, such as taking advantage of real-FFT symmetries, using pruned FFTs, and combining our algorithm with Karatsuba to achieve optimal performance.

References

[1] G. (2016), "Gnu multiple precision arithmetic library," "https://gmplib.org/", 2020.

[2] A. P. Dieguez, M. Amor, R. Doallo, A. Nukada, and S. Matsuoka, "Efficient high-precision integer multiplication on the gpu," *The International Journal of High Performance Computing Applications*, vol. 36, no. 3, pp. 356–369, 2022.

[3] T. Hickey, Q. Ju, and M. H. Van Emden, "Interval Arithmetic: From Principles to Implementation," *J. ACM*, vol. 48, no. 5, p. 1038–1068, sep 2001.

[4] W. M. Gentleman and G. Sande, "Fast fourier transforms: For fun and profit," ser. AFIPS '66 (Fall). New York, NY, USA: Association for Computing Machinery, 1966, p. 563–578.

[5] J. Rivera, F. Franchetti, and M. Püschel, "An Interval Compiler for Sound Floating-Point Computations," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2021, pp. 52–64.