Implementing a Fast Lucas-Lehmer Test on Programmable Graphics Hardware

Andrew Thall, *Alma College*

***Abstract*— The Lucas-Lehmer test provides a deterministic algorithm for testing whether, for a prime number p, $M_p = 2^p - 1$ is also a prime number. The current work demonstrates that this test can be effectively implemented on a parallel graphics processing unit (GPU). The parallelization was achieved by two main parallel methods: (1) fast multiplication using parallel Fast Fourier transforms in extended precision; (2) fast parallel carry-addition for arbitrary-precision numbers. Extended-precision is necessary in the Fourier transforms to allow single-precision graphics hardware to achieve sufficient precision for tests on non-trivial values of M_p. Methods (1) and (2) allow data to to remain on the graphics card throughout the test and minimize runtime costs of bus traffic between the host and GPU. The algorithm has been implemented in the Cg language and tested on several hardware platforms. The current work demonstrates the viability of current and future GPUs for number theoretic computation.**

[Addenda (2009): While actual implementations of this were not competitive with highly optimized sequential algorithms such as those used by GIMPS, a similar implementation using modern double-precision GPU hardware and CUDA kernels, rather than Cg-shaders, might produce superior runtimes to sequential algorithms.]

***Index Terms*— Parallel Lucas-Lehmer test, Mersenne primes, extended-precision computation, graphics processing units, GPGPU, Fast Fourier Transform, parallel carry-propagation in software**

Manuscript date: August 14, 2007

I. Introduction

MERSENNE primes are prime-valued binary repunit numbers—primes of the form

$$\begin{aligned} M_p &= 2^p - 1 \\ &= \underbrace{1111\ldots1111_2}_{p} \end{aligned}$$

for prime p. While of interest to number theorists, Mersenne primes are also popular targets for computational benchmarking, and are consistently the largest known prime numbers, due to the availability of the efficient Lucas-Lehmer test (Alg. 1). The current largest known prime is $M_{32,582,657}$, discovered by *GIMPS* (*Great Internet Mersenne Prime Search*) participants Cooper and Boone [1]. The Lucas-Lehmer test is simple to implement algorithmically but requires numerically intense computation, needing $\mathcal{O}(p^2 \log p \log\log p)$ sequential operations to test a p-bit number when using FFT-based multiplication.

This paper describes a GPU-based vector-parallel implementation of the Lucas-Lehmer test, depending for speed on computing not only the FFT-squarings but the carry-adds, subtractions, and modular reductions on the GPU. The use of the discrete Fourier transform (DFT) for fast multiplication was described by Schönhage and Strassen (1971) [2], reducing the computational complexity to $\mathcal{O}(p \log p \log\log p)$. It is well-known that fast parallel-reductions can be done for associative operations on vector elements; such parallel-prefix operations can be used in parallel carry-adders for carry propagation (see Ladner and Fischer [3], Balakrishnan and Nandy [4]). This enables $\mathcal{O}(\log p)$ parallel additions including carries on a machine storing one radix-digit at each of p processors. Combining the Schönhage-Strassen method with parallel carry-addition makes possible Alg. 2.

Algorithm 1 The Lucas-Lehmer Test

Require: $p > 2$, a prime number for which $M_p \triangleq 2^p - 1$ is the Mersenne number to be tested

1: **procedure** LUCAS-LEHMER-TEST(p)
2: $s_0 \leftarrow 4$ ▷ Initialize first in sequence
3: **for** $i \leftarrow 1, p-2$ **do**
4: $s_i \leftarrow (s_{i-1}^2 - 2) \mod M_p$
5: **end for**
6: **return** TRUE if $s_{p-2} = 0$, else FALSE
7: **end procedure**

Algorithm 2 A Vector-Parallel Lucas-Lehmer Test

Require: $p > 2$, a prime number for which $M_p \triangleq 2^p - 1$ is the Mersenne number to be tested
Require: n is the smallest power-of-two such that $2p < n$
Require: X and Y are complex arrays of length n

1: **procedure** VECTOR-PARALLEL-LUCAS-LEHMER-TEST(p)
2: $X[0] \leftarrow 4$ ▷ Initialize first in sequence
3: **for** $i \leftarrow 1, p-2$ **do**
4: $Y \leftarrow \mathcal{F}(X)$
5: $X \leftarrow Y \cdot Y$, componentwise vector product
6: $Y \leftarrow \mathcal{F}^{-1}(X)$
7: $X \leftarrow Y + (FFFF\ldots FFFE)_n$
8: $Y \leftarrow$ reduce-carrysave-to-carryadd(X)
9: $X \leftarrow$ parallel-prefix-carry(Y)
10: $Y \leftarrow X_{\text{low-order}} + (X_{\text{high-order}} >> p)$
11: $X \leftarrow$ parallel-prefix-carry(Y)
12: $Y \leftarrow X_{\text{low-order}} + (X_{\text{high-order}} >> p)$
13: $X \leftarrow$ parallel-prefix-carry(Y)
14: **end for**
15: **return** TRUE if $X = 0 \mod M_p$, else FALSE
16: **end procedure**

Step 7 deterministically performs the subtraction-by-two using the equivalent operation $Y - 2 = Y + (2^n - 2)$ modulo 2^n. This is $\mathcal{O}(1)$ in the carry-save representation. Step 8 reduces the carry-

save configuration to one with a single possible one-bit carry per byte. Steps 10 and 11, repeated twice, effect the reduction modulo $2^p - 1$ by shifting and adding the high-order p bits to the low-order p bits. Of these operations, the DFT-based multiplies and the carry-adds both have time complexity $\mathcal{O}(\lg p)$, giving this implementation a theoretical time-complexity $\approx \mathcal{O}(p \lg p)$.

This paper will provide the details required to efficiently implement the described algorithm on current GPU hardware.

II. GPU Implementation of the Parallel Lucas-Lehmer Test

The ideal asymptotic complexity of the parallel test is qualified by a number of factors: (a) the operation-count is no longer in terms of CPU clock cycles but in terms of the GPU frame-rate, which is a million times slower; (b) because a GPU does not actually have a processor per pixel,execution-time per operation is not constant but is a step-function of the size of the images used for the convolutions. Since the basic operation—the rendering of a single frame—is GPU-compute-bound, as the convolution-size increases by multiples of two, the compute time increases proportionally. Despite this, current GPU hardware can run this algorithm at comparable speeds to current sequential implementations (see Results below), and the algorithm scales transparently as more powerful GPUs are made available.

To implement the parallel Lucas-Lehmer test on the GPU—specifically, on nVidia G70 and G80-based processors—one must address the following GPU issues:

A) Extended-precision computation;
B) Fast Fourier transformations;
C) Reducing carry-save to carry-add configurations;
D) Fast carry-adds, including subtraction by 2;
E) Reduction of products modulo M_p.

The current implementation takes a naïve approach to the Lucas-Lehmer test, representing the numbers in the spatial domain as arrays of 8-bit binary "digits". More sophisticated approaches, as described in Crandall and Pomerance [5] (Ch. 9) can involve balanced-radix representations, irrational numerical bases, and products $modulo\ 2^n - 1$, affording constant-factor—albeit considerable—speedups over the naïve approach.

A. Extended-Precision Computation

Because a radix-256 number of length N can create frequency components on the order of $(256^2 N)$, we require at least $\log_2 256^2 N = 16 + \log_2 N$ bits of floating-point precision, as well as additional bits to prevent roundoff error (Press et al. [6], p. 918). Current GPUs offer at most a single-precision floating point number with a 24 bit (normalized) mantissa, which is insufficient for the large Mersenne numbers of interest here. Therefore it has been necessary to use extended-precision software to emulate higher-precision float.

Techniques for performing extended-precision arithmetic in software using pairs of machine-precision numbers have a long history: Dekker [7], Wyatt [8], and Brent [9], [10] all did extended precision research prior to the IEEE 754 standard. Priest [11] in 1992 did a full study of extended-precision requirements under the IEEE 754 standard; this provided the theoretical underpinnings for the *doubledoubles* of Briggs [12] and *quad-doubles* of Hida et al [13]. The use and hazards of double- and quad-precision numerical types is discussed by Li et al [14].

On the GPU, preliminary exploration of the feasibility and error-characteristics of GPU double-floats has been done using the Cg language by Meredith and Bremer [15] and the Brook language by Da Graça and Defour [16]. Thall [17] implemented full mathematical libraries for double-float and quad-float (henceforth, df_{64} and qf_{128}) computation. This library code was used in the current work to achieve the necessary numerical precision in the frequency components of the Fourier transforms.

As implemented in the Cg language and based on each image pixel storing four single-precision floating point values, a pixel can therefore store two df_{64} numbers. In the current implementation, these are precisely the df_{64} real and imaginary components of the complex array elements. Double-floats offer nearly 48-bits of precision in each "primitive"; this is sufficient for convolution products of 8-bit "digits" but is just shy of the number needed for 16-bit values.

B. A GPU Implementation of the Fast Fourier Transform in df_{64}

A discrete Fourier transform is a linear transformation of a complex vector:

$$\mathcal{F}(\bar{x}) = W\bar{x}$$

where, for complex $\bar{x}$ of length n, $W_{kj} = \omega_n^{kj}$, for

$$\begin{aligned} \omega_n &= \cos(2\pi/n) - i\sin(2\pi/n) \\ &= \exp(-2\pi i/n). \end{aligned}$$

"Fast" versions of the DFT allow the matrix product to be computed in $\mathcal{O}(n \log n)$ operations rather than $\mathcal{O}(n^2)$, and play pivotal roles in signal processing and data analysis.

A general discussion of FFTs on GPU hardware can be found in Moreland et al. [18]. The GAMMA group at UNC-Chapel Hill has released the GPUFFTW library for single-precision FFTs on real and complex input. The libraries for the CUDA language from nVidia also provide implementations of single-precision FFTs on the GPU.

The implementation chosen for this work uses a transposed Stockham autosort framework for the FFT; a discussion such methods is found in Van Loan [19]. The Stockham autosort methods avoid bit-reversal rearrangement of the data at the cost of a temporary storage array (necessary for a GPU-based algorithm in any case) and a varying index scheme for each of the $\ln n$ iterations. The GAMMA group at UNC employ such a framework in their GPUFFTW library.

C. Fast carry-adds and modular reduction on the GPU

Implementation of the Lucas-Lehmer test with parallel Fourier-based multiplication but sequential carry-add and modular reduction operations has shown that a substantial amount of time is spent in the sequential operation, especially in the readout of the product to CPU memory. As an example, for $p = 23,209$, over two-thirds of the computation time was spent in the carry-add and modular-reduction phases; of this fraction, only 15% of the time was spent on the ($\mathcal{O}(n)$) carry-adds and modular reductions, while the other 85% was the cost of the writes and readbacks between CPU and GPU. For a $p = 86,243$ trial with a 256×256 texture, the sequential code was 87% of the (3021 sec) runtime.

If the carry-adds and modular-reductions can be done on the GPU, the transfer times to and from the CPU can be eliminated, and the carry-adds and modular reductions reduced to $\mathcal{O}(\log n)$

operations. Although n is now bound by the frame-rate, rather than the FLOPS/sec of the CPU, the big win is in the elimination of the transfer times.

1) Fast parallel carry-addition and subtraction algorithms and implementations: An $\mathcal{O}(\log n)$ fast carry-add can be performed over n bytes stored 1-per-processor on a vector-based architecture by means of a parallel prefix sum computation (see Ladner [3], also Berman and Paul [20], p. 378). It is straightforward to adapt such a framework to the GPU, which behaves functionally as though it has a processor per pixel.

The convolution product, returned to the spatial domain, is rounded to a radix-256 integer in a *carry-save* configuration, with digits not restricted to be ≤ 255. While in the carry-save state, we can subtract 2 $modulo$ 2^n by simply adding 255 to each digit except the lowest order, to which we add 254. This is an $\mathcal{O}(1)$ operation on the vector-array. It is then necessary to reduce the carry-save configuration to a *carry-add* configuration, summing higher order bits from each digit into higher order bits to leave at most a single bit to be carried from each digit. This reduction depends on the maximum size of the convolution product and can be done in $m/8-1$ parallel-additions, where m is the maximum number of bits in the product.

Once in a carry-add configuration, a prefix-sum algorithm can be used to propagate the carries in $\mathcal{O}(\lg n)$ time.

2) Fast parallel modular reduction on the GPU: Given fast parallel addition-with-carry, modular reduction $\mod 2^p-1$ can be implemented by left-shifting the high-order p bits of a $2p$-bit product by p and then adding the low-order p bits. If there is a carry-out of the p^{th} bit of this sum, the value added to the lowest order term and any carry propagated once again.

III. Results and Discussion

A. Performance

[Results omitted. See implementation issues below.]

B. Implementation Issues

The code currently does fast-carry-adds on the GPU; because this is a linear sequential operation, it is architecture-dependent whether, for problem-sizes n of interest, the cost the frame-rate-based $\mathcal{O}(\lg n)$ operations on the GPU will be less than the CPU-clock-rate-based $\mathcal{O}(n)$ operations on the CPU. The cost of transferring the texture-data to and from the graphics card for each frame must be factored into the process. Thus, bus speeds and memory transfer rates make the problem highly dependent on details of both GPU and CPU system configurations.

Many obvious optimizations of the parallel routines were not done. (1) A factor of two reduction in the texture-size is attainable using standard methods for doing Fourier transforms of real-valued data by rearrangement of the data into a complex-valued array of half-the size. (2) The current implementation uses only even powers-of-two for the convolution arrays, using only square textures for storage. A trivial change to non-square textures would improve runtimes by a factor of two for values of M_p only requiring twice the space of a previous M_{p-1} but currently using four times as much. (3) The subtraction of two from the low-order byte seldom involves a "borrow" from a higher-order byte; the parallel carry-add of 2^p-2 might be done only rarely, with a single read-back from and write to the low-order byte being used more often. (4) Similarly, the second iteration of the shift-and-carry-add for the modular remainder operation is only necessary if there is a carry out from the p^{th} bit of the array after the first shift-and-carry-add; this could be checked with the read-back of a single byte and the costly second iteration done only as necessary.

These and similar optimizations would be desirable before using such code for serious Mersenne prime hunting.

C. Future Evolution of the GPU as a Platform for High-Precision Numerical Computation

The next generation of GPUs (see NVidia announcements) will have support for d_{64} double-precision variables. This will eliminate the need for df_{64} numbers for intermediate range values; for larger arbitrary precision numbers, it is likely that double-double and quad-double primitives will be immediately usable based on the current df_{64} and qf_{128} implementations. With higher precision primitives and larger texture memories, GPUs will prove to be very powerful platforms for high-precision computations. The availability of double-precision without emulation will also allow the use of tools such as Nvidia's *CUDA* language to greatly simplify the programming of such applications.

The use of CUDA would also avoid the OpenGL graphics-oriented API; studies of parallel-prefix operations by Harris et al [21] show an up-to-7-fold speedup of CUDA-based parallel-prefix operations over OpenGL-based implementations on the same hardware. Factors contributing to this are given as on-chip shared memory, thread synchronization functionality, and the ability to do scatter-writes to memory. All of these factors should translate directly to similar speedups in the parallel Lucas-Lehmer test.

[Note: as of 2009, IEEE-compliant double-precision arithmetic is available on commercial GPUs, accessible (on NVidia cards) using CUDA kernels but not exposed in the graphics pipeline in such systems as Cg. Use of double-precision would allow much greater efficiency than the double-float emulation used in this study; use of CUDA kernels in place of Cg shaders would likewise boost efficiency, perhaps to the point where GPU methods could equal or surpass the performance of the highly optimized sequential algorithms used by GIMPS.]

Acknowledgment

The author would like to thank Dr. Dinesh Manocha and his colleagues in UNC's GAMMA group for their GPUFFTW work. Thanks to Bob Silverman for correspondence and for fast sequential code for the Lucas-Lehmer test. Thanks also to Dr. Jan Prins at UNC-Chapel Hill for a critique of early ideas regarding parallel Lucas-Lehmer testing in 1995.

REFERENCES

[1] Website, "GIMPS: The Great Mersenne Prime Search," http://www.mersenne.org/prime.htm, 2007.

[2] A. Schönhage and V. Strassen, "Schnelle Multiplikation grosser Zahlen," *Computing*, vol. 7, pp. 281–292, 1971.

[3] R. E. Ladner and M. J. Fischer, "Parallel prefix computation," *J. ACM*, vol. 27, no. 4, pp. 831–838, 1980.

[4] S. Balakrishnan and S. K. Nandy, "Arbitrary precision arithmetic—SIMD style," in *VLSID '98: Proceedings of the Eleventh International Conference on VLSI Design: VLSI for Signal Processing*. Washington, DC, USA: IEEE Computer Society, 1998, p. 128.

[5] R. Crandall and C. Pomerance, *Prime Numbers: A Computational Perspective*, 2nd ed. Springer, 2005.

[6] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C: The Art of Scientific Computing*, 2nd ed. Cambridge University Press, 1992.

[7] T. J. Dekker, "A floating point technique for extending the available precision," *Numerische Mathematik*, vol. 18, pp. 224–242, 1971.

[8] W. T. Wyatt, Jr., D. W. Lozier, and D. J. Orser, "A portable extended precision arithmetic package and library with fortran precompiler," *ACM Trans. Math. Softw.*, vol. 2, no. 3, pp. 209–231, 1976.

[9] R. P. Brent, "A FORTRAN multiple-precision arithmetic package," *ACM Trans. on Math. Software*, vol. 4, no. 1, pp. 57–70, March 1978.

[10] R. P. Brent and H. T. Kung, "A regular layout for parallel adders," *IEEE Transactions on Computers*, vol. 31, no. 3, pp. 260–264, 1982. [Online]. Available: citeseer.ist.psu.edu/brent82regular.html

[11] D. M. Priest, "On properties of floating point arithmetics: Numerical stability and the cost of accurate computations," Ph.D. dissertation, University of California, Berkeley, 1992.

[12] K. Briggs, "Doubledouble floating point arithmetic," http://keithbriggs.info/doubledouble.html, Tech. Rep., 1998.

[13] Y. Hida, X. S. Li, and D. H. Bailey, "Algorithms for quad-double precision floating point arithmetic," in *Proceedings of the 15th Symposium on Computer Arithmetic (ARITH '01)*, N. Burgess and L. Ciminiera, Eds. Washington, DC: IEEE Computer Society, 2001, pp. 155–162. [Online]. Available: http://citeseer.ist.psu.edu/hida01algorithms.html

[14] X. S. Li, J. W. Demmel, D. H. Bailey, G. Henry, Y. Hida, J. Iskandar, W. Kahan, S. Y. Kang, A. Kapur, M. C. Martin, B. J. Thompson, T. Tung, and D. J. Yoo, "Design, implementation and testing of extended and mixed precision BLAS," *ACM Trans. Math. Softw.*, vol. 28, no. 2, pp. 152–205, 2002.

[15] J. Meredith, D. Bremer, L. Flath, J. Johnson, H. Jones, S. Vaidya, and R. Frank, "The GAIA Project: evaluation of GPU-based programming environments for knowledge discovery," Presentation, HPEC '04, Boston, MA, 2004.

[16] G. Da Graça and D. Defour, "Implementation of float-float operators on graphics hardware," *preprint*, may 2006.

[17] A. L. Thall, "Extended-precision floating-point numbers for GPU computation," Poster Session, ACM SIGGRAPH '06 Annual Conference, Boston, MA, August 2006.

[18] K. Moreland and E. Angel, "The FFT on a GPU," in *Graphics Hardware 2003*, M. Doggett, W. Heidrich, W. Mark, and A. Schilling, Eds. Eurographics, 2003.

[19] C. van Loan, *Computational Frameworks for the Fast Fourier Transform*, ser. Frontiers in Applied Mathematics. SIAM, 1992, Another wonderful book from Charlie van Loan. Let's send this guy lots of money.

[20] K. A. Berman and J. L. Paul, *Fundamentals of Sequential and Parallel Algorithms*. PWS Publishing Company, 1996.

[21] M. Harris, S. Sengupta, and J. D. Owens, "Parallel prefix sum (scan) with CUDA," in *GPU Gems 3*, H. Nguyen, Ed. Addison-Wesley, 2007, pp. 851–876.

Andrew Thall is Assistant Professor of Mathematics and Computer Science at Alma College. He received his Ph.D. (2004) in Computer Science from the University of North Carolina at Chapel Hill, where his main areas of research were in medical-image analysis and medial-based geometric surface modeling. His current research interests include numerical computation, GPGPU, and computer science education.