Fast Mersenne Prime Testing on the GPU

Andrew Thall
Dept. of Mathematics and Computer Science
Alma College
614 W. Superior St.
Alma, MI 48801 USA
thall@alma.edu

ABSTRACT

The Lucas-Lehmer test for Mersenne primality can be efficiently parallelized for GPU-based computation. The `gpuLucas` project implements an irrational-base discrete weighted transform approach (IBDWT) using balanced-integers, non-power-of-two transforms, and carry-save radix representations. `gpuLucas` uses the CUDA programming language and requires the double-precision floating point capabilities of recent GPUs. Results show up to 7× speedups over benchmark averages for optimized sequential code and factor-of-two speedups over `CUDALucas`, another GPU-based Lucas-Lehmer tester developed independently and with a different optimization strategy. This work demonstrates techniques for implementing GPU-based number theoretic algorithms on very large numbers, including fast multiplication, prefix-sum-based carry-propagation, and the use of carry-save arithmetic with balanced integers. The work presents timing profiles of convolution-based integer multiplication based on the IBDWT, in particular for non-power-of-two transformations, and establishes the usefulness of the software as a GPU benchmarking application and as a platform for large-integer and polynomial experimentation.

Categories and Subject Descriptors

G.4 [**Mathematics of Computing**]: Mathematical software—*Parallel and vector implementations*; I.3.1 [**Computer Graphics**]: Hardware architecture—*Graphics processors*

General Terms

Algorithms, Performance

Keywords

CUDA, discrete weighted transforms, fast carry propagation, GPGPU, IBDWT, Lucas-Lehmer test, Mersenne primes, multiprecision arithmetic, parallel carry-save arithmetic

1. INTRODUCTION AND OVERVIEW

Mersenne prime testing has a long history in numerical computing, from the 1948 work of Newman and Turing on the Manchester Mark I, to the 1952 work by Robinson at UCLA that found the first machine-proven Mersenne prime, $2^{521} - 1$. Mersenne primes—prime numbers of the form $M_p \triangleq 2^p - 1$, for p prime—have interesting mathematical properties but little practical value; nonetheless, they remain popular computational targets on new generation of computer hardware and are the subject of a massive distributed online search, the Great Internet Mersenne Prime Search (GIMPS). One practical use of the Mersenne prime testing software, rather than the numbers themselves, has been for benchmarking and stress-testing new computational hardware, and GIMPS maintains a huge database of profiled machines on their website [21].

Primality testing of Mersenne numbers is made practical by the Lucas-Lehmer test, a simple but computationally intensive algorithm that requires repeated squaring and reduction modulo M_p of p-bit numbers. With search exponents currently greater than 40-million bits, checking a large exponent is a test of speed and accuracy for multiprecision arithmetic.

Modern graphics processing units (GPUs), as massively-parallel, shared-memory architectures, are ideal platforms for the Lucas-Lehmer test. The `gpuLucas` project has parallelized the Lucas-Lehmer test for efficient GPU using an irrational-base discrete weighted transform (*IBDWT*) and balanced-integer approach derived from the work of Crandall and Fagin [1]. The implementation uses the CUDA programming language and requires the double-precision floating point capabilities of current (ca. 2009) GPUs: specifically, the Fermi®-class processors from NVIDIA Corp. The work uses the CUFFT library for non-power-of-two FFTs, and eliminates the need for parallel carry-addition by using carry-save arithmetic on a variable word-size, balanced-integer representation.

Section 1 describes the Lucas-Lehmer test and the IBDWT-method for multiplication modulo M_p. This is followed by a description of a GPU-suitable parallel version, and a discussion of the `gpuLucas` implementation and its optimization issues. Example timings and code-profiling data are presented, comparing `gpuLucas` with the GIMPS `Prime95` sequential code and with `CUDALucas`, another GPU-based implementation developed independently by S. Yamada as a port of existing sequential code. The discussion section compares the optimization strategy of the two GPU-based tests, discusses further improvements to the current project, and concludes with ongoing work and promising directions for further research.

1.1 The Lucas-Lehmer Test

There are only 47 known Mersenne primes at the time of this publication, the largest being $M_{43,112,609}$. Testing the primality of Mersenne numbers is made computationally tractable by the deterministic Lucas-Lehmer test [10], expressed algorithmically in Alg. 1. This algorithm follows

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 PRIME if $s_{p-2} = 0$, else NOT PRIME
 7: end procedure
```

from Lehmer's proof, extending a result by Lucas, that $2^p - 1$ is prime if and only if it divides the $(p-1)^{th}$ term of the series $\{4, 14, 194, 37634, \ldots, S_k, \ldots\}$, where $S_k = S_{k-1}^2 - 2$. This is made computationally practical by the trivial corollary that the product can be reduced *modulo* $2^p - 1$ following each iteration without altering the divisibility test of the final product.

Practical testing of large values thus requires $p-2$ squarings and modular reductions of a p-bit number. The reduction mod M_p can be done by an $\mathcal{O}(p)$ shift-and-add operation, but the multiplication requires a fast algorithm such as the Schönhage-Strasse method [16], treating multiplication as convolution and using Fast Fourier transforms to accomplish a p-bit multiply in $\mathcal{O}(p \log p \log\log p)$ operations rather than $\mathcal{O}(p^2)$. A Mersenne number M_p can therefore be tested with $\mathcal{O}(p^2 \log p \log\log p)$ bit-operations, or $\mathcal{O}(pN \log N)$ arithmetic operations for an N-word radix representation.

1.2 IBDWT-based Lucas-Lehmer Testing

Transform-based multiplication of multiprecision integers $\mathbf{x}$ and $\mathbf{y}$ typically requires zero-padding arrays of $<= N$, base-W digits to a length $> 2N$ large enough to hold their product, then transforming the arrays, typically with an FFT, doing a component-wise product, and then doing an inverse FFT. This gives a correct polynomial-product; for integers, the base-W radix-representation is restored by a carry operation from low-order digits to highest. Conceptually, each digit in the convolution product holds an entire column-sum for a digit computed using long multiplication, with a final carry to restore the base. Thus, each digit must have enough storage to hold a product term, on the order of NW^2. When using a floating point transform like the FFT, there also has to be enough precision to avoid rounding errors.

Efficient sequential implementations of the Lucas-Lehmer test use a variant of this that avoids zero-padding and does the multiplication modulo M_p directly. This is the irrational-base, discrete weighted transform method of Crandall and Fagin [1]. Its effective use in Lucas-Lehmer testing rests on four concepts: (1) balanced-radix representations, (2) variable-base digit representations, (3) weighted cyclic convolutions and discrete weighted transforms ($DWTs$), and (4) irrational numeric bases.

Balanced radix representations allow signed digit values; thus, for a base-W digit of the number $\mathbf{x}$, we may have digit values x_j in the range $-W/2 \leq x_j < W/2$. This greatly reduces the error bounds on integer multiplication using FFTs; intuitively, this is because each "column-sum" in the convolution product has both positive and negative summands and, while its magnitude is bounded by $N(W/2)^2$, its expected value is much less.

Variable base representations allow each digit x_j to have its own base W_j. This is strange but well-defined; for a balanced, symmetric-even representation, each digit x_j will be in the range $-W_j/2 \leq x_j < W_j/2$.

The use of irrational-base digits and the Weighted Discrete transform is beautiful but lengthy to describe; for a mathematical discussion, see Crandall et al. [2]. In practice, to do a multiplication modulo M_p, we choose an FFT signal-length $N < p$ (see Sec. 3.1.1) and represent our numbers as integers in a variable base representation

$$x = \sum_{j=0}^{N-1} x_j 2^{\lceil pj/N \rceil}, \tag{1.1}$$

with base $W_j = 2^{b_j}$ and the number of bits for digit x_{j-1} being

$$b_j = \lceil pj/N \rceil - \lceil p(j-1)/N \rceil. \tag{1.2}$$

To perform a product $xy \pmod{M_p}$, given $\mathbf{x}$ and $\mathbf{y}$ in this representation, we multiply each component x_j and y_j by the corresponding a_j-component of a weight-signal $\mathbf{a}$ of length N. This is defined by

$$a_j = 2^{\lceil pj/N \rceil - pj/N} \tag{1.3}$$

approximated by floating-point numbers in the interval $[1, 2)$. This permits x and y to be represented by integer components x_j and y_j in base W_j, with the the number of bits in W_j being either $\lceil p/N \rceil$ or $\lfloor p/N \rfloor$, with the floating-point terms a_j giving giving a componentwise correction weights for an irrational-digit, irrational-base representation. The modular product can then be computed using a weighted transform in place of an ordinary Fourier transform, expressable in terms of the latter as $\mathrm{DWT}_{(N,\mathbf{a})}\mathbf{x} = \mathrm{DFT}_{(N)}\mathbf{ax}$. This lets the modular product be computed using ordinary discrete Fourier transforms as

$$xy \pmod{M_p} = \mathbf{a}^{-1}\mathrm{DFT}_{(N)}^{-1}[(\mathrm{DFT}_{(N)}\mathbf{ax})(\mathrm{DFT}_{(N)}\mathbf{ay})].$$

The IBDWT enables Alg. 2 for integer multiplication modulo M_p. The algorithm may use standard or balanced digits and can serve in the inner loop of a Lucas-Lehmer test. The arithmetic complexity of the inner loop remains $\mathcal{O}(N \lg N)$, but the elimination of the $\mathcal{O}(N)$ modular reduction, the reduction in the run length N, and the error reduction from using a balanced representation combine to produce superior runtime constants, making these methods markedly superior to more general large-integer multiplication and modular division algorithms such as those used in the GMP libraries [3].

2. GPU-BASED LUCAS-LEHMER TESTING

Efficient parallelization of an IBDWT-based Lucas-Lehmer test requires two algorithmically fast operations: (1) parallel multiplication of large integers modulo M_p, and (2)

Algorithm 2 Multiplication of x and y modulo M_p. (Crandall)

```
1: Choose run length $N < p$, establishing bit-sizes $b_j$ according to Eq. 1.2, components of the weight signal $\mathbf{a}$ by Eq. 1.3, and representations $\mathbf{x} = \{x_j\}, \mathbf{y} = \{y_j\}$ according to Eq. 1.1, zero-padding to $N$ digits.
2: $\mathcal{X} \leftarrow \mathrm{DWT}_{(N,\mathbf{a})}\mathbf{x}$ and $\mathcal{Y} \leftarrow \mathrm{DWT}_{(N,\mathbf{a})}\mathbf{y}$.
3: $\mathcal{Z} \leftarrow \mathcal{X}\mathcal{Y}$, componentwise.
4: $\mathbf{z} \leftarrow \mathrm{DWT}^{-1}_{(N,\mathbf{a})}\mathcal{Z}$.
5: $\mathbf{z} \leftarrow \mathrm{round}(\mathbf{z})$.
6: Adjust the digits $\{z_n\}$ to the variable digit representation based on $b_j$ bits per digit.
```

parallel radix-restoration of digit values from convolution products. Any parallel system capable of accelerating the FFTs required by the Lucas-Lehmer test iterations may give speedups over sequential platforms. Massively parallel systems such as vector machines have been attractive targets for Lucas-Lehmer implementations, and some of the first results reported by Crandall and Fagin for the IBDWT-method were of timings on Cray supercomputers.

2.1 Prior and Concurrent Work

GPUs were demonstrably capable of accelerating FFTs as early as 2003 (Moreland et al. [11]) , with further work by Govindaraju et al. [4] [5], and with extended-precision *double-float* FFTs by Thall in 2006 [19].

The work by Thall was the first reported parallel Lucas-Lehmer test for the GPU; it used an autosorting, transposed-Stockham transform, implemented using extended precision libraries needed for non-trivial M_p values. It used the Cg-shading language for the GPU computations and extended-precision libraries from prior work [18]. The Cg-based Lucas-Lehmer code served as proof-of-concept on the use of the GPU for large-integer multiplication, implementing a naïve Lucas-Lehmer design with fixed-base, standard representation integers, with modular reduction via parallel-shift-and-add, and parallel carry-propagation for radix restoration via parallel prefix-sum. It failed to run faster than well-written sequential code for a plethora of reasons:

1. the lack of IEEE compliant double precision hardware, instead requiring software-implemented double-floats;
2. the need under Cg and the OpenGL API that each computational step be a separate "rendering pass" by a fragment-shader on data coded into image-texture memory;
3. the limitations on loops and branching under Cg;
4. GPU memory accessible solely through texture lookups and allowing no scattered writes;
5. non-IEEE-754-compliant floats and problematic compilation: e.g., compiler optimizations that change computational results in unpredictable ways.

Modern architectures, and GPGPU languages such as CUDA and OpenCL, have made recent GPUs much more versatile platforms for numerical computing in general, and all of the above problems are non-issues on present systems.

The `CUDALucas` program of Shoichiro Yamada has been developed concurrently and independently of the present `gpuLucas` project. This is a sophisticated implementation of the IBDWT on the GPU, done as a port of the highly optimized sequential program `MacLucasFFTW`. Comparison timings and discussion of implementation differences will be presented below; in general, `CUDALucas` produces Lucas-iteration timings within a factor of two of slower than `gpuLucas`, due to its inability to perform non-power-of-two transforms. There has been no published work on it, but details have been provided directly [22] and in discussions on the mersenneforum.org site.

2.2 A Parallel IBDWT-Based Lucas-Lehmer

Efficient Lucas-Lehmer testing on the GPU requires the elimination of all large data transfers between host and GPU; further, computations should preserve locality for all operations besides the FFTs themselves. Alg. 3 parallelizes the balanced-integer-IBDWT approach with this in mind. At

Algorithm 3 GPU-Parallel-Lucas-Lehmer-Test(M_p)

```
Require: $p > 2$, a prime for which $M_p \triangleq 2^p - 1$ is to be tested
 1: Choose run length $N < p$, establish bit-sizes $b_j$ according to Eq. 1.2.
 2: Allocate and initialize device arrays for weights $\mathbf{A}, \mathbf{A}_{inv}$ and for $\mathbf{Bits}$, by Eq. 1.3.
 3: Allocate device arrays $\mathbf{S}, \mathbf{S_{llint}}, \mathbf{D}, \mathbf{C}$ of size $N$ storing int32, int64, double, and complex-double values respectively
 4: $\mathbf{S[0]} \leftarrow 4, \mathbf{S[n > 0]} \leftarrow 0$.
 5: for $i \leftarrow 1, p-2$ do
 6:     $\mathbf{D} \leftarrow \mathbf{AS}$                        ▷ componentwise product
 7:     $\mathbf{C} \leftarrow \mathrm{realToComplexFFT}_{(N)}(\mathbf{D})$
 8:     $\mathbf{C} \leftarrow \mathbf{C}^2$                        ▷ componentwise product
 9:     $\mathbf{D} \leftarrow \mathrm{ComplexToRealFFT}_{(N)}(\mathbf{C})$
10:     $\mathbf{S_{llint}} \leftarrow \mathrm{roundToNearest}(\mathbf{A_{inv}D}/N)$      ▷ componentwise rounding
11:     $\mathbf{S} \leftarrow \mathrm{rebalanceToCarrySave}(\mathbf{S_{llint}}, \mathbf{Bits})$   ▷ limited carry propagation
12: end for
13: $\mathbf{S} \leftarrow \mathrm{rebalanceToBalancedInt}(\mathbf{S}, \mathbf{Bits})$      ▷ full carry propagated
14: return PRIME if $\mathbf{S} = \overline{\mathbf{0}}$, else NOT PRIME
```

runtime, the host program allocates storage and initializes variables, then executes the for-loop in Step 5. All operations within this loop, Steps 6–11, are data-parallel operations on the input arrays, and all but the forward and reverse FFTs operate in parallel on a single digit and at most a few of its near neighbors.

2.3 Time and Work Complexity

Regarding only the non-trivial operations within the loop, Steps 6, 8, and 10 are all $\mathcal{O}(1)$ under EREW PRAM assumptions, or $\mathcal{O}(N/P)$ on P processors, consisting of operations on individual words or identically indexed words in multiple arrays.

Steps 7 and 9, the FFTs, can be accomplished in $\mathcal{O}(\lg N)$ time under EREW PRAM assumptions, or $\mathcal{O}(N/P \lg N)$ time on a physical system. Efficient implementation must consider the size-speed tradeoffs of non-power-of-2 FFTs that may be available; architecture and library specific profiling is necessary to determine the most efficient lengths N that give results within acceptable error bounds. (See below, Sec. 2.4.)

In Step 11, carry-addition to restore digit-radices can be parallelized via a parallel prefix-sum (*scan*) operation using a $\oplus_{gpk}$ function over a per-digit *generate-carry/propagate-carry/kill-carry* evaluation. This can be done with $\mathcal{O}(\lg N)$ time complexity and optimal $\mathcal{O}(N)$ work under an EREW PRAM model [8] but is problematic for two reasons: (1) in Lucas-Lehmer testing, parallel carry-adders are "overkill": long carry-propagation-chains are very rare in high-radix arithmetic for typical products; prefix-sums take $\mathcal{O}(\lg N)$ time when only a few $\mathcal{O}(1)$ per digit ripple carries are needed; and (2) $\oplus_{gpk}$ is problematic for balanced integers; it is no longer an associative function and therefore not subject to prefix-scan evaluation. Although the propagation can be divided into separate add-with-carry and subtract-with-borrow scans, this is inelegant and doubly overkill.

A better solution for applications such as Lucas-Lehmer testing is to not resolve the carries at all, leaving the large integer in a *carry-save* representation. For a variable base representation for $\mathbf{x}$, each digit x_j should be in the range $-W_j/2 \le x_j < W_j/2$. However, if any x_j is outside of its range—representing an unpropagated borrow or carry—the digits still represent the same $\mathbf{x}$; further, they still give correct convolution products, so long as the individual product terms are not too large for the precision of the floating-point variables in the FFT. Further, convolution product terms, assuming random inputs, are still distributed about a mean of zero for each digit, and there is no observable increase in the error bounds. Shift-based modular reduction becomes problematic, but an IBDWT-approach eliminates the need for this in any case.

The balanced, carry-save representation proved a very effective technique in parallel Lucas-Lehmer implementation. While a sequential implementation requires an $\mathcal{O}(N)$ scan in any case for digit-adjustment, the use of carry-save arithmetic eliminates the need for any non-local computation other than the Fourier transforms. The computational cost over $\mathcal{O}(p)$ iterations is therefore $\mathcal{O}(p(\log N + \mathcal{O}(1)))$, or $\mathcal{O}(pN/P \log N + pN/P)$ for P processors.

2.4 Error Bounds on DWT-Based Convolution

Average and worst-case error-bounds for FFTs, DWTs, and multiplication modulo Mersenne numbers were calculated by Percival [14]; the findings showed that requiring FFT multiplication to be provably immune to fatal rounding increases runtime by under a factor two versus runlengths N tuned to the average case. Percival gave experimental results showing that the range of maximum errors for multiplications of random inputs is small and predictable, although atypical inputs may produce much larger errors.

Because of the predictability of the observed average error, it is possible to make the selection of runlength N based on short test-runs across a range of possible lengths; for a computation that may take hours or days, this adds a negligible amount to the runtime, and acceptable values may vary based on unpredictable factors, including new releases of FFT libraries. This may be done on a case-by-case basis, but test-runs are short enough to allow ranges of Mersenne numbers testable by transforms of a given length to be tabulated.

Runtime error testing is also used, most commonly by checking the maximum per-digit rounding difference required to restore the unweighted floating point product-digit to an integer value. Errors becoming more likely to have occurred when absolute distance of a digit from its nearest integer approaches 0.5. A heuristic threshold is set, and results are accepted only as probabilistic until confirmed by tests with other runlengths.

```
cudaMemcpy(A, host_A, sizeof(double)*SIZE,
      cudaMemcpyHostToDevice);
cudaMemcpy(Ainv, host_Ainv, sizeof(double)*SIZE,
      cudaMemcpyHostToDevice);
cudaMemcpy(bpw, host_bpw, sizeof(unsigned char)*SIZE,
      cudaMemcpyHostToDevice);
loadValue4ToFFTarray<<<GRID, BDIM>>>(signal_d,
      SIZE);
for (int iter = 2; iter < TESTP; iter++) {
   cufftExecD2Z(signal_d, signal_z);
   complexPointwiseSqr<<<FFTGRID,
         BDIM>>>(signal_z, SIZE/2 + 1);
   cufftExecZ2D(signal_z, signal_d);
   invDWTprodMinus2<<<GRID, BDIM>>>(signal_ll,
         signal_d, Ainv, SIZE);
   llint2BALIrrInt<<<GRID, BDIM>>>(iArr, hiArr,
         signal_ll, bpw, SIZE);
   if (iter < TESTP - 1)
      loadIntToDoubleIBDWT<<<GRID,
            BDIM>>>(signal_d, iArr, hiArr, A, SIZE);
}
// on loop exit, rebalance iArr by full add-with-carry of
      hiArr. This is the final Lucas-Lehmer product.
```

Figure 1: CUDA code-outline of main Lucas-Lehmer loop of `gpuLucas`. The code for convolution-error testing has been omitted.

3. THE GPULUCAS IMPLEMENTATION

The `gpuLucas` software is written in the CUDA programming language from NVIDIA Corporation, and currently runs only on Fermi® class NVIDIA hardware, with testing on GTX 480 and Tesla 2050 GPUs. The following discussion assumes basic knowledge of CUDA programming and the associated Fermi GPU memory structure (global, shared, constant memory, warps, and bank conflicts, etc.); on-line reference materials are plentiful; recommended are the official documentation [13], CUDA 3.2 SDK example code, the GPU Gems series [12], as well as recent texts by Sanders and Kandrot [15] and by Kirk and Hwu [9].

A code outline of the main test-loop is shown in Fig. 1. It is a straightforward implementation of Alg. 3, using library calls to CUFFT for double-to-complex and complex-to-double FFTs and custom kernels for the rest. The overall optimization strategy was fourfold:

1. eliminate any array copies or host operations on GPU data

2. keep data operations as local as possible to within each data array, using shared memory within thread-blocks for inter-thread communication

3. use block-level synchronization (via `__syncthreads()` and global synchronization (via separate kernel calls) as seldom as possible.

4. use integer arithmetic and bitwise operations for exact arithmetic.

In terms of speed, modern GPUs remain double-precision-challenged; while the recent Fermi-architecture GPUs are much better in this regard, 32-bit int and float operations are still substantially faster than 64-bit double performance. Thus, it is still faster to convert to integer for most operations, using double-precision only for the weighted multiplies and the FFTs. Single-precision floating point is faster than integer for most operations, but bitwise Boolean operations allow exact arithmetic for all but the DWT and the rounding of the resulting product terms.

3.1 Implementation and Optimization

The project made use of open-source libraries for several routines. The CUDPP library provides efficient CUDA parallel-prefix scan operations as described by Harris, Sengupta, and Owens [6]; the parallel-prefix code was modified for parallel carry-adds and borrows, as described below in Sec. 3.1.6, but in final implementation was used only for max operations over convolution rounding errors. The computation of weights for the IBDWT was done on the host in extended precision using the `qd` library of Hida et al. [7].

3.1.1 *Selection of runlength N and word-sizes*

The current code, in common with GIMPS `Prime95`, uses non-power-of-two transforms. This eliminates sharp jumps in runtime when changing from transforms of length 2^m to 2^{m+1}. Thus, to test $M_{43,112,609}$, which is too large for a 2^{21} transform, one can use $N = 2^{21} + 2^{19} = 2^{19} * 5$ instead of $N = 2^{22}$, giving a nearly 2× speedup. Modern libraries such as FFTW and CUFFT employ mixed-radix methods and can handle powers-of-small prime transformations very efficiently. CUFFT allows runlengths $N = 2^a * 3^b * 5^c * 7^d$; as part of this project, runlengths have been profiled and tabulated for different products of powers, and maximum acceptable word-sizes have been computed for each. The current `gpuLucas` now chooses a runlength for acceptable word-sizes based on known FFT timings for possible choices of N.

3.1.2 *Computation of digit bit-lengths*

An array **b** of bit-lengths for each radix position is precomputed on the host processor according to Eq. 1.2. An array of bit-vectors **bpw** is then created for each digit x_j,

$$\mathbf{bpw}[j][i] = \begin{cases} 1 & \text{if } b_{j-i} = \lceil p/N \rceil \\ 0 & \text{if } b_{j-i} = \lfloor p/N \rfloor \end{cases} \qquad (3.1)$$

This sets bit-i if digit x_{j-i} has the higher bit-length, wrapping as necessary to $x_{N-(j-i)}$ when $j - i < 0$. At most six lower-order values may generate carries to any given x_j in the radix-rebalancing stage, given convolution products stored as IEEE-754 double precision values and given minimum bits per digit $(8, 9)$; i.e., no product digit can contribute bits directly to more than six higher order terms. By selecting N to generate the largest word-sizes that give convolution products storable in a double, prior terms needed for a radix-restoration may be reduced to only 2 or 3.

3.1.3 *Initialization of* $\mathbf{a}$ *and* $\mathbf{a}^{-1}$

Because the weight-computation $a_j = 2^{\lceil jp/N \rceil - jp/N}$ can produce catastrophic cancelation when transform-size N is not a power of two, weights are computed on the CPU in double-double precision and then loaded to global memory.

3.1.4 *Setting up the DWT and entering the main loop*

Initially, the value 4.0 is loaded to `signal_d[0]`, with the rest of the N components set to zero, since 4 is balanced and $a_0 = 1.0$. On subsequent iterations, the FFT array is initialized by carry-save values from the previous iteration, which are multiplied componentwise by the weights a_j. The forward and inverse transforms are done by the CUFFT library calls, and the complex squaring by a kernel call to compute the product on the $2N + 1$ complex values.

Following the convolution,`invDWTprodMinus2()` multiplies product terms componentwise by the `Ainv` weights, subtracts 2 from the lowest, then rounds to the nearest `long long int` and writes this to global memory. CUDA provides library functions for `double2ll()` in device code that takes a rounding-mode parameter as an argument. A modified `invDWTprodMinus2()` is used when error terms need to be computed, with additional code to do a parallel-prefix scan of error terms to find the maximum.

3.1.5 *Distributing convolution product digits to base-W_j carry-save configuration*

Base-reestablishment is done by `llint2BALIrrInt()`. This is the most costly operation besides the FFTs—see profiling data in the results section. There are separate routines for different word-size pairs depending on how many previous terms can carry to the current one; at runtime, a pointer to the correct function is assigned and used for the entire test. Some details from the $(16, 17)$ case are shown below. Product terms are loaded to shared memory, including three extra terms that may carry in from the previous block and with wrap-around from the final block to block 0. Once loaded to shared memory, the correct bits from the current and prior 3 terms are added, using the correct multipliers and signs.

```
// get bits in current digits and preceding 3
unsigned char bperW = bpw[tid];
int isHi = bperW & 1;
const int BITS = LO_BITS + isHi;
const int BASE = BASE_LO << isHi;
const int MASK = BASE − 1;

// mask off and add contribution of this and preceding digits'
// values to current digit's rebalanced value
int shiftBits = 0;
int sum = signs[tba + 3]∗(digits[tba + 3] & MASK);
for (int i = 1; i < 3; i++) {
   bperW >>= 1;
   isHi = bperW & 1;
   shiftBits += LO_BITS + isHi;
   sum += signs[tba + 3 − i]∗((digits[tba + 3 − 1] >>
          shiftBits) & MASK);
}
```

Finally, a partial balanced-integer rebalancing is done, computing carry or borrow out from current digit digit and writing the current digit to `iArr` and its carry to `hiArr`.

```
  // do a partial rebalancing, computing carry or borrow out
        from current digit
  // and storing in hival
  int HBASE = BASE >> 1;
  int hival = 0;
  if (sum < −HBASE)
       hival = −((−sum + HBASE) >> BITS);
  else if (sum >= HBASE)
       hival = (sum + HBASE) >> BITS;

  // writes to global memory
  iArr[tid] = sum − (hival << BITS);
  hiArr[tid] = hival;
}
```

No further rebalancing is done following this stage; the `hiArr` carry or borrow is simply added to the following `iArr` term when the double array is loaded for the next iteration, leaving the representation in a carry-save configuration.

3.1.6 Full Rebalancing by GPK-carry-addition

The current implementation uses no GPK-carry resolution in the main loop; they are used in the final rebalancing Step 13, though mainly as an act of bravado; a single $\mathcal{O}(n)$ linear scan on the CPU would work just as well. For applications requiring full carry-adds on very large integers, however, they are deterministic and faster than a CPU-based sequential carry; the current code provides them for both balanced and unbalanced representations, and also includes fast-carry-based routines for conversion between the two.

The implementation of the GPK algorithm used modified routines from the open-source CUDPP library [6]. The `CUDPP_GPK` function was added to the already defined `ADD`, `MULTIPLY`, `MIN`, and `MAX`. The code modification was nontrivial; the existing CUDPP codebase assumes a commutative scan-function and required careful modification of the highly optimized code to ensure that the non-commutative $\oplus_{gpk}$ was correctly composited and applied to correctly ordered operands throughout the parallel scan. The `CUDPP_GPK` function can be used equivalently for either carry-addition or subtract-with-borrow, by modifying the test for setting the $\{G, P, K\}$ status prior to the scan.

4. RESULTS AND COMPARISONS

4.1 Target Platforms

Target platforms for the implementation were an NVIDIA GTX 480 graphics card and a Tesla 2050 GPGPU device, running on an Intel Core i7 CPU 940 host (2930 MHz) under Windows 7. Both GPUs are Fermi architecture, giving CUDA capability 2.0: fairly IEEE-compliant double-precision, 32-bit integers (and fast 64 bit extensions) with bit operations, and atomic memory operations. The Tesla has a slower CUDA clock and fewer processors, but has superior double-precision performance due to full enabling of the processor; the GTX 480, designed as a high-end commodity graphics card, opts for more CUDA cores and a higher clock speed. In Lucas-Lehmer experiments, the two cards are nearly even...the superior double precision throughput sped the FFTs and high-precision multiplies and rounding, while the faster clock and greater number of CUDA cores decreased communication costs for the FFT and sped single precision and integer computation.

The most recent NVIDIA drivers allow overclocking of GPUs at the OS level; increasing the clock rate of the Tesla 2050 to equal that of the GTX produced results superior to the less expensive card. (We also record GPU temperatures in excess of $87°C$ for the overclocked Tesla during a long run; how healthy this is for the processor remains to be seen.)

All Tesla results below are processor overclocked to 1404 MHz, with ECC memory mode disabled.

4.2 CUDA Profiling

Table 1 shows kernel timing profile of representative runlengths using the NVIDIA Compute Visual Profiler 3.2.0. The FFT library calls (divided into multiple kernel calls, internally) were most costly operations. The `llint2BALIrrInt()` kernel, took the next longest, and the other expensive operation being the `loadInt2DoubleIBDWT` with its double-precision dependencies. As expected, with identical processor clock speeds (1402 GHz), the greater number of CUDA cores on the GTX 480 led to better timings on the $\mathcal{O}(N/P)$ operations, even the double-precision `complexPointwiseSqr`, but for the FFTs, the Tesla made up for fewer processors by the superior double-precision performance. The similar balance of operation timings between the $N = 256K$ and $N = 8192K$ runlengths seems odd, but were confirmed by independent timings of the CUFFT real-to-complex and complex-to-real transforms, which scale linearly with N for signal-lengths between 2^{18} and 2^{24}.

CUDA kernel or library calls	% of time $N = 256K$	% of time $N = 8192K$
`cufftExecD2Z()` and `cufftExecZ2D()`	64.7 (61.3)	68.0 (62.9)
`complexPointwiseSqr()`	5.2 (5.8)	4.6 (5.6)
`invDWTprodMinus2()`	8.0 (8.8)	7.4 (8.8)
`llint2BALIrrInt()`	13.7 (14.7)	12.1 (13.9)
`loadInt2DoubleIBDWT()`	8.4 (9.2)	7.7 (9.2)

Table 1: Profiling `gpuLucas` for small and large convolution lengths. Timings are on a GTX 480 and (in parentheses) overclocked Tesla 2050.

p for M_p	time (sec)	N	bases (lo, hi)	max ε
44497	4.2	4 K	(10, 11)	1.6e-7
86243	8.1	8 K	(10, 11)	1.5e-7
86243	8.1	4 K	(21, 22)	2.5e-1
216091	20.3	16K	(13, 14)	9.5e-6
216091	25.7	12K	(17, 18)	3.6e-3
859433	136.4	64K	(13, 14)	2.2e-5
859433	111.6	48K	(17, 18)	7.8e-3
1257787	197.4	64K	(19, 20)	8.6e-2
3021377	1237 (0.34 hr)	160K	(18, 19)	5.4e-2
6972593	6410 (1.78 hr)	384K	(17, 18)	3.7e-2
13466917	23069 (6.41 hr)	768K	(17, 18)	2.5e-2
32582657	141998 (39.4 hr)	2048K	(15, 16)	5.1e-3
32582657	125338 (34.8 hr)	1792K	(17, 18)	2.3e-1
42643801	208300 (57.9 hr)	2304K	(18, 19)	1.8e-1

Table 2: Sample execution timing for `gpuLucas`. Timings are for the overclocked Tesla 2050. Timing are for the main loop of the Lucas-Lehmer test, and ignore a small constant setup time.

4.3 Timings and Comparisons

Table 2 gives sample execution times for the testing of representative Mersenne numbers. Given the ability to use non-power-of-two runlengths and to fine-tune for higher base-values, a few examples of the same M_p with different runlengths are shown. For small values of p, non-power-of-two runlengths may make little difference or even be detrimental, but appreciable speedups are seen for larger p when decreasing the runlength.

Table 3 gives timings for a single Lucas-Lehmer iteration for different runlengths N, based on runs of `gpuLucas` and `CUDALucas` on the test GPUs and `GLucas` and `Prime95` on the test host. `GLucas` shows its age as older sequential code, cross-platform but used mainly for hardware stress-testing.

runlength ($K = 2^{10}$)	~max. p for M_p	gpuLucas (msec)	CUDALucas	Glucas	Prime95 1 T, 1 core	Prime95 8 T, 4 cores
1024 K	18M	2.20 (2.04)	2.74 (2.45)	51	14.4	5.6
2048 K	36M	4.46 (4.36)	4.43 (4.10)	104	29.2	10.3
2304 K	40M	5.05 (4.89)		124		
2560 K	46M	5.81 (5.46)		143	37.3	14.5
3072 K	52M	7.13 (6.61)		169	46.0	17.5
3584 K	60M	8.18 (7.62)		211	54.3	19.4
4096 K	69M	9.11 (8.80)	9.04 (8.12)	225	61.4	21.8
4608 K	81M	10.41 (9.70)		268		
5120 K	90M	11.85 (10.97)		303	85.8	28.9
6144 K	108M	14.57 (14.41)		359	99.1	33.8
7168 K	123M	16.73 (15.23)		445	118.1	42.8
8192 K	145M	18.62 (17.45)	18.2 (15.6)	479	133.3	46.4

Table 3: Comparison timings for a single Lucas-Lehmer squaring and rebalancing: `gpuLucas` vs. `CUDALucas`, `Glucas` and `Prime95` benchmarks. GPU timings are on an NVIDIA GTX 480 and (in parentheses) overclocked Tesla 2050. `Glucas` and `Prime95` are on an Intel Core i7 CPU 940 processor at 2930 Mhz.

`Prime95`, the current GIMPS tester, tracks a steady 6.5–7x slower for 1 thread on one core, but gets within 2.3–2.5x when using 8 threads on all 4 cores of the host. In discussions on mersenneforum.org, `Prime95` experts note that its FFTs are well-tuned only for single-core, "with multi-threading wedged in as an afterthought," and that it will always achieve better throughput by assigning one exponent per core. The other GPU implementation, `CUDALucas`, matches `gpuLucas` closely on the GTX 480 and beats it on the Tesla 2050; however, given its current inability to use non-power-of-two transforms, it runs 1-2x slower in practice, since Mersenne numbers too large for a given length 2^m must use the next larger 2^{m+1}. This will be discussed in more detail below.

5. DISCUSSION

5.1 Comparison of `gpuLucas` and `CUDALucas`

`CUDALucas` has been developed by Yamada as a direct port of `MacLucasFFTW`, itself a port to FFTW of a program tracing its roots back to original code by Crandall et al. While `gpuLucas` was designed ground-up as a data-parallel application, taking a 1-digit-per-thread approach and using data-parallel algorithms whenever possible, `CUDALucas` retains much of its original character as highly optimized sequential code. It does all computation in floating point, and loops over multiple digits in each thread for base-restoration, using similar optimizations for carries and exact rounding as were used sequentially. It uses the CUFFT complex-to-complex transforms, but sets up its real data using real-to-complex FFT code from Takuya Ooura's FFT package [20], which is ostensibly faster than the CUFFT real-to-complex setup. It is this that prevents it from using non-power-of-two transforms, however, since the front-end code is strictly power-of-two.

`CUDALucas` is much more complex than `gpuLucas`, but with some nice touches, and it matches `gpuLucas` for speed at its power-of-two transform sizes. A function timing-profile comparison with `gpuLucas` remains to be done; it will be noteworthy whether the expected slow-down of keeping all computations in double-precision was offset by the rewritten real-to-complex front-end for the CUFFT routines.

5.2 Issues with current implementation

`gpuLucas` uses the CUDA CUFFT library code; this has optimizations for real-to-complex transforms, but is not perfectly suited for DFT-based multiplication, which does not require sorted data in the frequency domain. Other approaches may give constant factor speedups over the general-purpose CUDA libraries. A canonical reference for real-valued FFT algorithms is Sorensen et al. [17].

Older versions of the CUDA library had a limit of 2^{23} real or complex elements in a transform; this proved not to be a problem given the reduced runlengths produced by the IBDWT method and through the use of non-power-of-two FFTs. Older drivers did, however, have problems with accuracy for non-power-of-two transforms; there were also bugs in the cuFFT library that prevented more than 2^{24} FFT transformation from being done sequentially by an application. All of these problems have been corrected as of the Nov. 2010 CUDA 3.2 release.

Runtime convolution error checking remains ad hoc. The current implementation checks for maximum rounding errors every $N/50$ iterations by an $\mathcal{O}(\lg n)$ reduction over per-word roundoff distances, using a CUDPP prefix-scan with a `CUDPP_MAX` function. This adds no significant amount to the runtime, but if a per-iteration error test is desired, it can be accomplished by an `atomic_or()` to a global variable on the very improbable event of a $|x_j - \text{round(x\_j)}|$ value exceeding the designated threshold < 0.5.

The multiplication code takes 60–70% of the inner loop computation. If this were better optimized, the rest of the code might be improved as well by making better use of the GPU cores, using "warp-aware" computation similar to the parallel-scan optimizations employed by Harris et al. [6]; these involve detailed knowledge of the SIMD processors in the cores used by *warps* of 16–32 threads within each block, allowing synchronized updates to shared memory without explicit synchronization mechanisms.

5.3 Ongoing and Future Work

In the immediate future, the `gpuLucas` code will be released to the public, along with the FFT non-power-of-two profiling database and the code that generated it. This work may be developed further into a standard suite of benchmarking tools to profile FFT runtime and error-bounds with respect to runlength and wordsize for large integer and polynomial convolution and DWTs, establishing values for guaranteed accuracy using Percival's criteria, along with the current values, for applications where randomness assumptions on operands allow for more latitude.

The prefix-scan fast-carry modifications to CUDPP will also be released, either by integration of the GPK-function and necessary code fixes into the main project, or independently under the public license.

There has been no attempt to integrate this code with the formats and protocols for its use by the GIMPS project. A number of participants have expressed interest in seeing this happen. There is also interest in adapting the current software to do more general Lucas-Lehmer-Reisel primality testing, for numbers of the form $k2^n - 1$, with $2^n > k$. There are a number of other number theoretic applications for IBDWT methods, and countless applications for large integer and polynomial manipulation. The current codebase provides a simple and clean testbed for experiments, profiling, and large-integer library development.

Acknowledgments

The author would like to thank Shoichiro Yamada, creator of `CUDALucas`, for explanation of design decisions in his implementation. Thanks to Thomas Duell for providing more timings of `CUDALucas`, and to Dr. Greg Childers the many others on mersenneforum.org who have provided information on current software and offers of help with the testing and distribution of `gpuLucas`. Thanks to Harris, Sengupta and Owens for their CUDPP software that I hacked so mercilessly, and to Dr. Mark Harris personally for advice and encouragement.

6. REFERENCES

[1] Crandall, R., and Fagin, B. Discrete weighted transforms and large-integer arithmetic. *Mathematics of Computation 62*, 205 (1994), 305–324.

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

[3] Gaudry, P., Kruppa, A., and Zimmermann, P. A GMP-based implementation of Schönhage-Strassen's large integer multiplication algorithm. In *ISSAC '07: Proceedings of the 2007 International Symposium on Symbolic and Algebraic Computation* (New York, NY, USA, 2007), ACM, pp. 167–174.

[4] Govindaraju, N. K., Larson, S., Grey, J., and Manocha, D. A memory model for scientific algorithms on graphics processors. Tech. Rep. Microsoft Technical Report MSR TR 2006 108, The University of North Carolina at Chapel Hill and Microsoft Corp., 2006.

[5] Govindaraju, N. K., and Manocha, D. A memory model for scientific algorithms on graphics processors. Tech. rep., The University of North Carolina at Chapel Hill, 2007.

[6] Harris, M., Sengupta, S., and Owens, J. D. *Parallel Prefix Sum (Scan) with CUDA*. Addison-Wesley, 2007, ch. 39, pp. 851–876.

[7] Hida, Y., Li, X. S., and Bailey, D. H. Algorithms for quad-double precision floating point arithmetic. In *Proceedings of the 15th Symposium on Computer Arithmetic (ARITH '01)* (Washington, DC, 2001), N. Burgess and L. Ciminiera, Eds., IEEE Computer Society, pp. 155–162.

[8] Hillis, W. D., and Steele, Jr., G. L. Data parallel algorithms. *Commun. ACM 29*, 12 (1986), 1170–1183.

[9] Kirk, D. B., and Hwu, W. W. *Programming Massively Parallel Processors*. Elsevier, 2010.

[10] Lehmer, D. H. An extended theory of Lucas' functions. *Ann. Math. 31*, 3 (Jul 1930), 419–448. The original paper on the Lucas-Lehmer test.

[11] Moreland, K., and Angel, E. The FFT on a GPU. In *Graphics Hardware 2003* (2003), M. Doggett, W. Heidrich, W. Mark, and A. Schilling, Eds., Eurographics.

[12] Nguyen, H., Ed. *GPU Gems 3*. Addison-Wesley, 2007.

[13] Nvidia Corporation. *NVIDIA CUDA Reference Manual, Version 3.2*, 3.2 ed. Santa Clara, CA, 2010. `http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_Toolkit_Reference_Manual.pdf`.

[14] Percival, C. Rapid multiplication modulo the sum and difference of highly composite numbers. *Mathematics of Computation 72*, 41 (2002), 387–395.

[15] Sanders, J., and Kandrot, E. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley, 2011.

[16] Schönhage, A., and Strassen, V. Schnelle Multiplikation grosser Zahlen. *Computing 7* (1971), 281–292.

[17] Sorensen, H. V., Jones, D. L., Heideman, M. T., and Burrus, C. S. Real-valued fast Fourier transform algorithms. *IEEE Trans. on Acoustics, Speech, and Signal Processing ASSP-35*, 6 (June 1987), 849–863.

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

[19] Thall, A. Implementing a fast Lucas-Lehmer test on programmable graphics hardware. Tech. rep., Technical Report 007-2, Alma College, 614 W. Superior St., Alma, MI, 48801-1599, June 2007.

[20] Website. Ooura's Mathematical Software Packages. `http://www.kurims.kyoto-u.ac.jp/~ooura/`, 2006.

[21] Website. PrimeNet CPU Benchmarks (GIMPS). `http://www.mersenne.org/report_benchmarks/`, 2010.

[22] Yamada, S. Discussion of `CUDALucas` implementation. Personal correspondence, 2011.