Faster Modular Exponentiation using Double Precision Floating Point Arithmetic on the GPU

Niall Emmart*, Fangyu Zheng†‡, and Charles Weems*

*College of Information and Compute Sciences, University of Massachusetts, Amherst, MA 01003, USA
†State Key Laboratory of Information Security, Institute of Information Engineering,
Chinese Academy of Sciences, Beijing, China
‡Data Assurance and Communication Security Research Center, Chinese Academy of Sciences, Beijing, China

***Abstract*—This paper presents a new approach to integer multiple precision (MP) modular exponentiation, using double-precision floating point (DPF) operations, that is suitable for GPU implementation. We show speedups ranging from 20% to 34% over the best prior GPU times for sizes corresponding to common RSA cryptographic operations (2048 to 4096 bits). Three techniques are described. First, by adding 2^{104} to the high half of the product, and 2^{52} to the low half, we set the implicit leading 1 in the DPF mantissa so that the full 52 explicit bits are available for each half of the 104-bit products of samples. Second, the DPF values are cast bitwise to 64-bit integers for adding the column sums to get the MP result. Normally the cast would require masking off the exponents, but because they are constant, we can include them in the column sums and correct just once for their total. Third, by initializing the column sums with the appropriate negative value to compensate for the exponent sums, no corrective subtraction is needed. Our implementation on an NVIDIA GTX Titan Black GPU achieves between 132.5K and 161.9K modular exponentiations per second of size 1024 bits, with latencies ranging from 21.7 ms to 17.8 ms, making it practical for online RSA applications. Proportional results are shown for 1536 and 2048 bits. The implementation is so efficient that its maximum sustained performance is actually bounded by the thermal limit of the GPU.**

In this paper we examine a new approach to multiple precision (MP) modular exponentiation on the GPU using double precision floating point (DPF) arithmetic. There have been a number of papers that have used floating point arithmetic as the basis for modular multiplication and exponentiation algorithms. The idea is to leverage the high speed floating point units on the GPU to perform integer arithmetic. Moss, Page and Smart [1] use a mixed radix approach with a vector of co-prime moduli, where each modulus is 12-bits in length and compute $a_i \cdot b_i$ fmod m_i using single precision floating point (SPF) arithmetic. Fleissner [2] implements 192-bit modular exponentiation with six 24-bit values. The 24-bit values are further sampled into bytes and the computations on the bytes are done using SPF arithmetic. In [3] Bernstein et al. implement a 280-bit modular multiplication based on a traditional fixed radix number system (FRNS) with 28 limbs of 10-bit samples (each sample is a 10-bit integer stored in a SPF) and a three-full-products approach to Montgomery multiplication [4]. They use a sophisticated scheme to distribute each multiplication across 28 threads in a warp. But in essence, the column sums of a multiple precision product (see Figure 1) can be thought of as dot-products, $\sum_{i,j} a_i \cdot b_j$ where A and B are the MP values to be multiplied and $i + j$ equals the column number. The a_i and b_i samples and all of the computations are done using SPF arithmetic. But we note that $28 \cdot (2^{10} - 1) \cdot (2^{10} - 1)$ can exceed 2^{24} which can result in rounding errors. To work around this problem, Bernstein et al. compute two partial sums of at most 14 terms, which guarantees there won't be any rounding, and then perform the final sum using integer arithmetic. Bernstein et al. have carefully crafted a highly optimized implementation, but unfortunately, 10-bit samples are too small to be efficient. In their subsequent paper [5], Bernstein et al. perform all computations using integer arithmetic, which proved to be much faster. Zheng et al. in [6] (and two follow-on papers [7] and [8]) use floating point arithmetic to implement modular multiplication, modular exponentiation, and RSA. In [6] they use an FRNS with 23 bits per limb and Montgomery's multiple-precision modular reduction, which

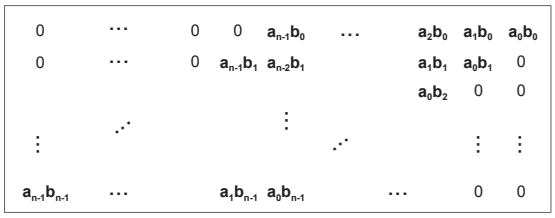

Fig. 1. Product terms to be summed for an n-sample by n-sample multiply.

This material is based upon work supported by the National Science Foundation (NSF) under award No. CCF-1525754 and the National Key R&D Program of China under award No. 2017YFB0802103.

is similar to column sums in Figure 1, except each column will have twice as many terms (see Section 2 of [4] for details). The samples and all computations are performed using DPF arithmetic and by using 23 bits per limbs, they can guarantee that the column sum does not exceed 2^{53}, the largest integer that can be stored in a DPF value without rounding. [7] extends the earlier work, using 22 and 23-bit limbs to support RSA sizes that range from 2048 bits to 4096 bits. [7] is the first paper that shows that floating point implementations can outperform the best integer implementations, at least at large sizes. [8] further improves the performance of [6] and [7] with a clever technique using signed floating point values allowing for longer limbs to be used (e.g., 24-bit limbs instead of 23), which means the same MP values can be represented with fewer limbs, and a corresponding improvement in performance. [8] also introduces two new parallel carry resolution algorithms, that are faster than the traditional ripple carry approach and have an added advantage that they run in constant time. [8] is the first paper that addresses all of the requirements to build a GPU based RSA digital signature server: high throughput, reasonable latency and constant time signature generation and verfication.

All of these prior papers have used *narrow samples*, i.e., the number of bits in each sample is always less than or equal to half the number of bits (width) available in the mantissa. In this paper we introduce a new *wide samples* approach, where each sample will use almost the full precision available in the mantissa. Specifically, we use 52 bits per sample stored in DPFs (a similar approach could be used with 23 bits per sample stored in SPFs). The wide samples approach combined with the fast carry resolution algorithms developed in [8] leads to a significant performance gain over the prior work. In particular, comparing this work to [8] (the fastest RSA implementation using floating point arithmetic to date) the proposed approach is 26% faster, 22% faster, and, 36% faster, at 2048-bit, 3072-bit and 4096-bit RSA respectively. In reviewing the literature, we find that Gueron and Krasnov [9] report that Intel plans to extend the AVX-512 vector instruction set to use the DPF units to perform 52-bit integer multiplications within AVX vectors in a future Cannonlake architecture. In contrast, our approach is applicable to any current or future architecture with IEEE-754 DPF support, without requiring any specialized, non-standard hardware.

The rest of this paper is organized as follows, Section I describes the wide samples approach with tricks and optimizations that would allow it to be implemented efficiently on a wide range of processors, including GPUs, DSPs, CPUs, and CPU vector processors (such as AVX). Section II covers our GPU implementation of modular multiplication and modular exponentiation using the wide samples approach. Please note, although RSA is the motivating problem, we do not implement it in this paper. However, it could easily be layered on top of the modular exponentiation routines that we do implement. Section III presents our experiments and results, and we close with conclusions and future work in Section IV.

I. New Approach Using Wide Samples

The idea behind our approach is straightforward. We will use wide samples with a 52-bit positive integer in each sample and will compute the column sums (essentially dot-products) with a combination of double precision and integer arithmetic. Since the samples are 52 bits, the product a pair of a_i and b_i samples will be 104 bits long and won't fit in a double precision value. Instead, for each term in the column sum, i.e., product of a pair of samples, we will compute a 52-bit low product and a 52-bit high product. Computing low and high products of a pair of floating point values is a well known problem dating back to the 1970s, used to compute the error in a floating point multiplication. Dekker [9] solved the problem by splitting a_i and b_i into upper and lower halves and computing the products from the half precision values. Dekker's approach is quite slow, but with the advent of hardware based fused-multiply-and-add (FMA), which is present on the GPU, there is a fast algorithm as follows, where *__fma_rz* is the CUDA built-in function for a double precision FMA with rounding towards zero:

```
full_product(double a_sample, double b_sample)
  p_hi = __fma_rz(a_sample, b_sample, 0.0);
  p_lo = __fma_rz(a_sample, b_sample, -p_hi);
  return (p_hi, p_lo);
}
```

Figure 2. Compute the high and low product of two samples using fused multiply and add

This approach is discussed in [10] and [11]. Internally, the FMA hardware has to compute the product to 106 bits of precision for a DPF multiply. The first multiply in the algorithm returns the most significant 53 bits of the product. The second multiply subtracts the high bits from the product, which become zeros, and are shifted off by the normalizer, returning the least significant 53 bits of the product. Unfortunately, with this algorithm, the low and high products of the terms in the dot-product might not be correctly aligned. To illustrate the problem, consider computing the dot-product of two vectors of samples using the algorithm in Figure 2 and summing the high and low products: let $\vec{A} = \vec{B} = (2^{50}, 1)$ and we wish to compute $\vec{A}$ dot $\vec{B}$ exactly. Computing the full products $p_0 = \text{full\_product}(a_0, b_0) = (2^{100}, 0)$ and $p_1 = \text{full\_product}(a_1, b_1) = (1, 0)$. Adding the high

```
dpf_full_product(double a_sample, double b_sample)
  double c1=2^104, c2=2^104 + 2^52, c3=2^52, sub;

  p_hi = __fma_rz(a_sample, b_sample, c1);
  sub = c2 - p_hi;
  p_lo = __fma_rz(a_sample, b_sample, sub);

  return (p_hi-c1, p_lo-c3);
}
```

Figure 3. Normalized high and low products. Note, this algorithm **requires** rounding towards zero. Rounding towards nearest will produce incorrect results.

```
int_full_product(double a_sample, double b_sample)
  double   c1=2^104, c2=2^104 + 2^52, sub;
  uint64_t mask=2^52 - 1;

  p_hi = __fma_rz(a_sample, b_sample, c1);
  sub = c2 - p_hi;
  p_lo = __fma_rz(a_sample, b_sample, sub);

  return (to_u64(p_hi) & mask, to_u64(p_lo) & mask);
}
```

Figure 4. Normalized high and low products. Note, this algorithm **requires** rounding towards zero. Rounding towards nearest will produce incorrect results.

products and low products of $p0$ and $p1$, yields $(2^{100}, 0)$ rather than the desired sum $(2^{100}, 1)$, due to alignment and round-off problems. We can resolve these problems with a small modification, shown in Figure 3. When computing the high product, we add 2^{104} and when computing the low product we subtract off the 2^{104} and add 2^{52}. This forces the alignment of the decimal places in the high and low products, and thus the column sum can be computed by just summing the high terms and (separately) summing the low terms. Returning to the example, computing the dot-product using the algorithm in Figure 3, we will get $p0 = (2^{100}, 0)$ and $p1 = (0, 1)$ and summing highs and lows will result in $(2^{100}, 1)$, as desired.

Since the high and low products have 52-bits of precision, if we compute the sums in the floating point domain, we will need more than the 53-bits of precision provided by a DPF. Instead, we can take advantage of the DPF binary representation and perform the summation in the integer domain, as follows. In the IEEE-754 DPF binary representation, the most significant bit is the sign bit, the next 11 bits are the exponent (with a bias of 1023), and for normal floating point values, the remaining 52 bits form a 53-bit mantissa, where the most significant bit of the mantissa is always implicitly set to one (the implicit bit is not stored in the binary representation which saves a bit). Since the product of two samples is guaranteed to be less than 2^{104}, if we compute $p\_hi = a_i \cdot b_i + 2^{104}$, then 2^{104} will be the most significant bit of the result and will become the implicit bit in the binary representation. The next 52 bits (the explicit bits) of the mantissa will exactly match the 52-bit high product that would have resulted from a product computation in the integer domain. Likewise, computing $p\_lo = a_i \cdot b_i + (2^{104} + 2^{52}) - p\_hi$ ensures that 2^{52} will be the implicit bit of *p_lo* and the next 52 explicit bits will be the low product. Thus we can recover the integer high and low products, by masking the top 12 bits from the DPF binary representation. This is shown in Figure 4, which uses a helper function *to_u64* to convert double precision values to their binary representations. *to_u64* can be implemented with C unions or with inline PTX assembly. On the GPU, floating point values and integer values use the same registers, so the *to_u64* conversion is essentially free.

It turns out that the masking operations end up wasting a significant number of compute cycles, but with one more trick, we can eliminate them, by noticing that the sign bit of *p_hi* is always 0 and the exponent is always 104 plus 1023 (the bias), thus the top 12 bits of *p_hi* is always 0x467, and the top 12 bits of *p_lo* is always 0x433 (52 plus 1023). Instead of the masking operations, we can include the top 12 bits in the column sums and then by tracking how many high and low products have been summed into each column, we can cancel off their total with a single subtraction operation. We can even save the final subtraction by initializing the column sums with the correct negative values. The algorithm

```
#define N 8

void sampled_product(uint64_t *col_sums,
             double *a_samps, *double b_samps) {
  double   c1=2^104, c2=2^104 + 2^52, sub;
  uint64_t mask=2^52 - 1;
  int      i, j;

  for(i=0;i<N;i++) {
    col_sums[i]=make_initial(i, i+1);
    col_sums[2*N-1-i]=make_initial(i+1, i);
  }

  for(i=0;i<N;i++) {
    for(j=0;j<N;j++) {
      p_hi = __fma_rz(a_samps[i], b_samps[j], c1);
      sub = c2 - p_hi;
      p_lo = __fma_rz(a_samps[i], b_samps[j], sub);
      col_sums[i+j+1] += to_u64(p_hi);
      col_sums[i+j] += to_u64(p_lo);
    }
  }
}
```

Figure 5. MP Sampled Product Algorithm

```
uint64_t make_initial(int high_count,
                      int low_count) {
  uint64_t value=high_count*0x467 + low_count*0x433;

  return -((value & 0xFFF)<<52);
}
```

Figure 6. MP Sampled Product Algorithm

in Figure 5 computes an 8-sample (416 bits) by 8-sample multiple precision full product returning the 16 column sums of the result. The algorithm uses a routine called *make_initial* (shown in Figure 6) which takes two arguments, the high product count and low product count, and generates the appropriate initial value for the column sum that will cancel the 0x467s and 0x433s. Since the mantissa part is 52 bits, we can add up to 2^{12} product terms in a column before we have to worry about overflowing the 64-bit integer sums and we can ignore any carry outs generated by the 0x467s and 0x433s.

If the loops of the algorithm in Figure 5 are completely unrolled and we make some reasonable assumptions about the code generated by the CUDA tool chain, then an N-sample by N-sample product will require exactly $3N^2$ double precision floating point operations and $4N^2$ 32-bit additions (two 32-bit addition instructions per 64-bit addition in the algorithm). This can be extremely fast on GPU cards with high double precision throughput.

II. Implementation of Modular Exponentiation using Wide Samples

In this section, we discuss the implementation of modular multiplication and exponentiation. Although we do not implement RSA, it could easily be layered on top of the modular exponentiation routines. Since the motivating problem is typically RSA, our modular exponentiation routine needs to meet a few key requirements: 1) it must achieve high throughput at moderate batch sizes; 2) the latency must be reasonable (<100 ms); 3) the running time must be independent of the secret key so that we don't leak information; 4) we must support 1024-bit, 1536-bit and 2048-bit modular exponentiations, which are suitable for RSA-2048 through RSA-4096.

There are a number of algorithms to compute a modular exponentiation, $A^K \mod P$. The most well known is exponentiation by squaring, but it's quite slow. Two faster techniques are fixed window exponentiation (also known as k-ary exponentiation) and sliding window exponentiation. The sliding window method is the fastest, but has the potential to leak information about the secret key, so we use a fixed window approach. To ensure constant time, we always do a multiplication step for each group of w bits, even in the case where the window bits are zero. For more on exponentiation algorithms, please see [12]. The modular exponentiation algorithms all require a modular multiplication subroutine to do the actual computations.

For modular multiplication there are several choices to compute $A \cdot B \mod P$. Several papers have explored mixed radix number systems (see for example [1], [13]). Our wide samples approach could be used to support a mixed radix approach, but it's our belief, based on our prior evaluations, that while these parallelize well, they are less performant than a traditional FRNS. In the FRNS space, there are two common approaches that avoid long division, which is very slow on the GPU. These are the Barrett reduction [14] and the Montgomery product [4]. At relatively small sizes using the $O(N^2)$ algorithms, the Montgomery product is more efficient and regular, thus we use Montgomery for our implementation.

Further if we relax the requirement that the Montgomery product returns a value between 0 and $P-1$, and instead allow the return of values between 0 and $2P-1$, then the correction step can be eliminated entirely, as per Orup [15] and later Walter [16]. Eliminating the correction step improves performance and it removes another potential source of timing attacks.

For the required modular multiplication, which range in size from 1024 bits to 2048 bits, the GPU does not have enough register resources to run a multiplication per thread, and keep everything on chip. So instead we use a warp parallel approach pioneered by Jang et al. [17] where the multiple precision values, A, B, P, and S are partitioned across a group of threads as shown in Figure 7. In this example, a 1024-bit modular multiplication

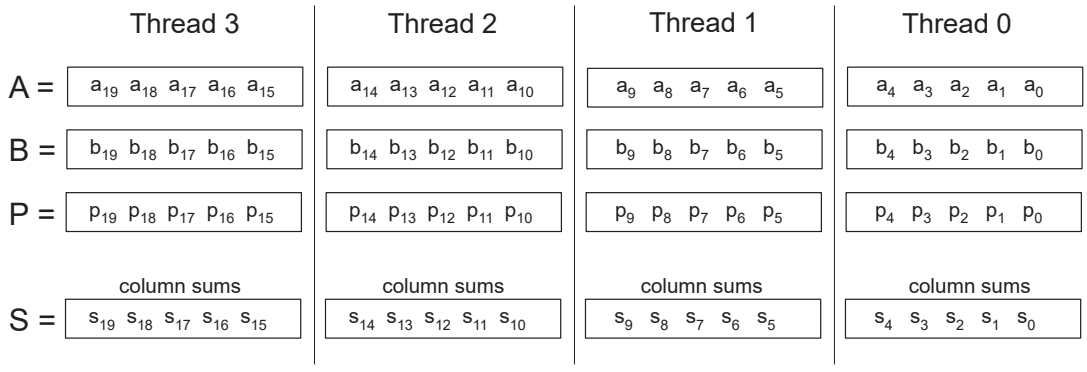

Fig. 7. 1040-bit (20 samples at 52-bits/sample) modular multiplication is partitioned acress 4 threads

is computed using 20 samples of 52 bits per sample. The samples are partitioned across 4 threads. Since 20×52 is 1040, the top 16 bits are initially zero padded, but it gives us the extra bit of space needed for Orup & Walter's method. At a high level, the Montgomery product is computed as follows:

1: $S \leftarrow 0$
2: **for** i=0 to 19 **do**
3: $\quad S \leftarrow S + A \cdot b_i$
4: $\quad S \leftarrow S + P \cdot q_i$
5: $\quad S \leftarrow S/2^{52}$
6: **end for**
7: **return** S

where q_i is the smallest positive value less than 2^{52} such that $S + P \cdot q_i$ is evenly divisible by 2^{52} and $S/2^{52}$ is implemented by shifting all of the column sums one position to the right. This organization of the computation is typically called Coarsely Integrated Operand Scanning (CIOS), see [18] for details.

The modular multiplier consists of three subroutines. Figure 8 computes a row product, i.e., $S \leftarrow S + X \cdot t$, where X is a multiple precision value and t is a 52-bit sample, using our wide samples approach. The template parameter *limbs* is the number of limbs in each thread. Figure 9 computes the full Montgomery product of A, B, and P, and returns the resulting column sums. It uses three template arguments where *n* specifies the total number of limbs in an instance, *threads* is the number of threads assigned to each instance and *limbs* is the number of 52-bit limbs in each thread. Note, in some cases, *n* might be less than $threads \cdot limbs$. The algorithm first initializes the column sums. Then it iterates through the limbs of B, accumulating $A \cdot b_i$ followed by $P \cdot q_i$ into the column sums. Since the column sums are 64 bits per limb, rather than the 52 bits of the sample, the column sums form a redundant representation where the upper 12 bits of each column overlap with the next (more significant) column sum. At the end of each iteration, we must shift the column sums one sample, or 52 bits to the right. To implement this, each thread splits the least significant column sum into an upper 12 bits and a lower 52 bits. The upper 12 bits gets added to the next column to the left. The lower 52 bits gets sent, using a *__shfl*, to the thread to its right for inclusion in the most significant column. Once we have iterated through all n of the b_i terms, the column sums will represent Montgomery product in redundant (overlapped) form.

```
template<int limbs>
void rowmul(uint64_t sums[limbs+1],
            double term, double v[limbs]) {
  double hi, temp, lo, c1=2^104, c2=2^104+2^52;

  for(int index=0;index<limbs;index++) {
    hi=__fma_rz(term, v[index], c1);
    temp=hi-c2;
    lo=__fma_rz(term, v[index], temp);
    sums[index+1] += to_u64(hi);
    sums[index] += to_u64(lo);
  }
}
```

Figure 8. 52-bit sampled row multiplier

```
template<int n, int threads, int limbs>
void modmul(uint64_t sums[limbs+1], double a[limbs],
     double b[limbs], double p[limbs], double np0) {
 uint32_t groupBase=threadIdx.x & ~(threads-1);
 uint64_t send64, mask52=0xFFFFFFFFFFFFFull;
 double   term, temp, c1=2^104;

 // initialize the low limbs
 for(int word=0;word<limbs;word++)
   sums[word]=make_initial(2*word, 2*word+2);

 for(int i=0;i<n;i++) {
   // initialize the high limb
   if(i<n-1-limbs)
     sums[limbs]=make_initial(2*limbs, 2*limbs);
   else if(i<n-1)
     sums[limbs]=make_initial(2*(n-1-i), 2*(n-2-i));

   // accumulate b_i * A, followed by q_i * P
   term=__shfl(b[i % limbs], i/limbs+groupBase);
   rowmul<limbs>(sums, term, a);

   temp=(double)(sums[0] & mask52);
   term=__fma_rz(temp, np0, c1);       // high prod
   term=__fma_rz(temp, np0, c1-term); // low prod
   term=__shfl(term, groupBase);
   rowmul<limbs>(sums, term, p);

   // because of the Montgomery property, the least
   // significant 52 bits of the least significant
   // thread will always be zero.  So we can use
   // wrap around without any additional zeroing.
   send64=__shfl(sums[0] & mask52, tid.x+1, threads);
   sums[0]=sums[1] + (sums[0])>>52);

   // all other limbs shift right by one column
   for(int word=1;word<limbs-1;word++)
     sums[word]=sums[word+1];
   sums[limbs-1]=sums[limbs] + send64;
 }
 // the high limb is empty, zero it
 sums[limbs]=0;
}
```

Figure 9. 52-bit sampled modular multiplication based on a CIOS Montgomery product

Before the next multiply step can be computed, we have to return the result to a non-redundant 52-bit sampled form, which is equivalent to resolving the carries between the columns. In a sequential processing situation, this would be a trivial problem, just start at the right most column and sweep the carries across to the left. Jang et al. [17] used the following parallel approach:

1: *resolve the carries within each thread*
2: **while** *any thread has a carry out* **do**
3: $\quad$ *move the carry to the next higher thread*
4: $\quad$ *resolve carries within each thread*
5: **end while**

On average, this approach is quite fast. However, it is possible that the while loop will run multiple times

as a carry ripples from the least significant thread all the way through to the most significant thread. Since there is a loop structure, it will not run in constant time, and therefore runs the risk of leaking information about the secret key. Dong et al. came up with two clever solutions to this problem. For the first they essentially built a hierarchial carry look-ahead adder in software using *__ballot* instructions (see [8] for details). For the second, they realized that their multiplier allowed for a small amount of redundancy. So rather than having to ripple a carry all the way to the left, they only need to push it as far as the next thread. With their multiplier design, the next thread can always absorb a single carry in, and when it does so, the representation becomes mildly redundant. It turns out the same trick works here in our wide samples approach. Instead of requiring samples to range from 0 to $2^{52}-1$ (inclusive), we will allow them to range from 0 to 2^{52}. As discussed in Section 1, if the product of the a and b sample is less than 2^{104} then everything just works. Thus if a and/or b are less than 2^{52}, it is fine. The only questionable case is when both a and b are exactly 2^{52}. However, when we run this case through the high-low multiplier and subtract off the required 0x467 and 0x433 from the top 12 bits, we find the high word is 2^{52} and the low word is 0, exactly as required. Thus, we can allow the wider range in the samples which allows us to use a very efficient carry resolution routine, shown in Figure 10.

```
template<int threads, int size>
void resolve(uint64_t *sums) {
  uint64_t mask52=0xFFFFFFFFFFFFFull, send64;

  #pragma unroll
  for(int index=1;index<size;index++) {
    sums[index]+=sums[index-1]>>52;
    sums[index-1]=sums[index-1] & mask52;
  }

  // 12-bit carry to next thread
  send64=__shfl(sums[size-1], tid.x-1, threads);
  sums[size-1]=sums[size-1] & mask52;

  // must propagate through two limbs to ensure a
  // single bit carry which can be absorbed
  sums[0]+=send64>>52;
  sums[1]+=sums[0]>>52;
  sums[0]=sums[0] & mask52;
}
```

Figure 10. 52-bit sampled carry resolution routine

These three routines serve as the implementation of the modular multiplier. The other routines required to implement fixed window exponentiation are straightforward, requiring only data loading, data storing, and some sampling routines that convert a standard 32-bit FRNS to and from a 52-bit sampled representation.

III. Experimental Setup and Results

To evaluate our wide samples approach to modular exponentiation, we ran experiments on a GeForce GTX Titan Black card, which uses a Kepler micro-architecture with 15 SMs with a nominal clock rate of 889 MHz and a maximum clock rate of 1280 MHz. The Titan Black card supports high throughput double precision floating point operations. The card is hosted by a 64-bit Ubuntu Server (version 16.04 LTS) machine, with an MSI Z270M motherboard and an Intel Core i5-7400 running at 3.0 GHz with 16 GB of main memory. We have installed CUDA version 8.0 and NVIDIA driver version 375.26 and use GMP version 6.1.1 to verify the modular exponentiation results generated on the GPU.

To test the performance for each size k (1024, 1536, or, 2048 bits), we generate the number of random instances in accordance with Table I. Each random instance consists of three randomly generated values A, K, and P, each of which is k bits in length. There are no restrictions on A or K, but P must be odd as required for use in Montgomery reductions. The test procedure is straightforward. Generate the random instances on the CPU and for each instance pre-compute two terms that are dependent on P: $R \mod P$ and $R^2 \mod P$, where R is $2^{52\cdot\lceil k/52\rceil}$. Then copy the five k-bit values for each instance to the GPU. On the GPU we first launch warm up kernels, followed by the timing runs. Once all the timing runs are complete, we copy the results back to the CPU where they are checked for correctness using GMP. For the timing runs, we launch the kernel and measure the execution time. The kernel loads the instance data from GPU global memory, computes $A^K \mod P$, and writes the result back to global memory. The timing runs do not include any of the CPU processing time such as generating the random instance data, the pre-computations, copying the instances to and from the GPU or verifying the result.

The modular exponentiation kernel can be compiled with a number of different parameters, such as threads per instance, samples per thread, threads per block, maximum number of registers to use per thread, etc. For each size k, we test a few of the possibilities and pick the set of parameters that produces the best overall throughput. The parameter values are summarized in Table I. One result that is surprising is that we achieve the best performance with a single large block per SM rather than multiple smaller blocks. With other kernels it's common to see multiple smaller blocks outperform a single large block.

In Table II we give the throughput (modular exponentiations per second) and latency of the wide samples approach on a Titan Black card. As can be seen from the table, a single warm up run followed by small number of timing runs gives the best performance. As we increase the number of timing runs, the performance drops, but then levels off. In the last two rows for each size, we do

TABLE I
PARAMETER THAT DELIVER THE BEST THROUGHPUT ON 1024, 1536 AND 2048 BIT MODULAR EXPONENTIATION

Bits	Threads / Instance	Samples / Thread	Threads / Block	Max Registers	Blocks	Instances
1024	4	5	768	80	15	2880
1536	8	4	768	80	15	1440
2048	8	5	768	80	15	1440

TABLE II
PERFORMANCE RESULTS FOR THREE SIZES AND DIFFERENT WARM UP COUNTS AND TIMING RUN COUNTS

Bits	Warm Up Runs	Timing Runs	Average Throughput	Average Latency
1024	1	4	161.9K	17.8 ms
	1	50	138.2K	20.8 ms
	1	100	135.7K	21.2 ms
	100	200	133.0K	21.7 ms
	100	500	132.5K	21.7 ms
1536	1	4	45.2K	31.9 ms
	1	50	38.1K	37.8 ms
	1	100	37.7K	38.2 ms
	100	200	37.1K	38.8 ms
	100	500	37.1K	38.8 ms
2048	1	4	19.4K	74.4 ms
	1	50	17.8K	81.0 ms
	1	100	17.7K	81.3 ms
	100	200	17.5K	82.3 ms
	100	500	17.5K	82.3 ms

100 warm up runs and then either 200 or 500 timing runs. The throughput and latency of these last two rows are similar and represents the steady state for the size. The *nvidia-smi* utility can be used to monitor the current state of the GPU, including temperature, power consumption, clock rates, etc. What we see is that at the start of a set of runs, the GPU core clock rate is 1280 MHz and the power consumption of the GPU hits the maximum allowed, which is 250 watts. The GPU responds by ratcheting down the core clock rate. At the end of the first few runs it has dropped down to 1030 MHz, and by about the 100th run, the clock rate settles between 862 MHz and 875 MHz and the power consumption hovers at around 245 watts. We can conclude that the kernel is not compute bound or memory bound, it's actually limited by power consumption.

In Table III we compare our results against some recent paper that have implemented 1024-bit (and larger) modular exponentation or 2048-bit RSA. For each paper, we list the architecture, # of SMs, clock rate, integer multiplies per clock cycle, a scaling factor, scaled performance results, and whether the algorithms would be expected to run in constant time and with a low enough latency to be suitable for a GPU based RSA solution. We compute a scaling factor so we can compare results across GPU cards, which have differnces in the # of SMs, clock rates, integer multiplies per cycle and whether the paper measures modular exponentiations or RSA decryptions. At 1024 bits, we find that the proposed wide samples approach significantly outperforms the prior results with the exception of [19], where it is marginally faster. However, it is worth noting that at 1024-bits [19] is using an instance per thread model, and has a latency well over 100 ms. At 1536 and 2048 bits the wide samples approach is 20% and 34% faster than the next fastest result [8], respectively. Further, we are comparing our steady state performance to what likely a peak performance rate in the other papers.

TABLE III
PERFORMANCE COMPARISON

	Jang et al. [17]	Emmart and Weems [19]	Yang et al. [20]	Dong et al. [7]	Dong et al. [8]	Proposed
CUDA platform	GTX 580	GTX780Ti	GT 750m	GTX Titan	GTX Titan Black	GTX Titan Black
Architecture	Fermi	Kepler	Kepler	Kepler	Kepler	Kepler
# of SMs	16	15	2	14	15	15
Int Mul/SM (/Clock)	16	32	32	32	32	32
Shader clock (MHz)	1544	876	967	837	889	889
RSA / Mod Exp	0.49	-	0.49	0.49	0.49	-
Scaling Factor	2.203	1.015	14.072	2.322	2.04	1
Scaled Performance						
1024-bit Mod Exp (ops/s)	26,500	129,700	73,800	98,000	107,600	132,500
1536-bit Mod Exp (ops/s)	-	-	-	28,200	31,000	37,100
2048-bit Mod Exp (ops/s)	3,700	10,900	-	13,400	13,100	17,500
Constant Time (all sizes)	No	No	No	No	Yes	Yes
Latency Under 100ms	Yes	No	No	Yes	Yes	Yes

Based on the throughput results, the good latency characteristics, and favorable comparisons to other papers, we believe a wide samples approach would be a good choice for a GPU based RSA digital signature server.

IV. Conclusions and Future Work

The important contribution of this paper is the combination of three tricks that allow efficient use of wide samples. The first trick is the injection of 2^{104} and 2^{52} into the floating point computations to solve the normalization/alignment problems. The second is taking advantage of the IEEE 754 representation, whereby the least significant 52 bits of the double precision value exactly match the integer high and low products, and performing the column sums with unsigned 64-bit integers. The third trick is to take advantage of the fact that the upper 12 bits of a high product are always 0x467 and for a low product they are always 0x433. Rather than repeatedly masking off the top bits, we can include them in the column sums and cancel them off with a single subtraction or alternatively by initializing the column sum with the right initial value. Combined with the carry resolution routines of [8], the wide samples approach is the best performing CUDA implementation of modular exponentiation to date, at the three tested sizes.

There is still a lot of work to be done in the future. First, we would like to integrate these routines into an RSA digital signature server and validate that the RSA routines are indeed secure, with no correlation between the distribution of running times and the secret key (as was done in [8]). Second, these same techniques could be used to implement elliptic curve cryptography (ECC) primitives. It would be interesting to compare the performance of ECC based on wide samples to other ECC implementations in the literature. Finally, we believe that it might be possible to improve the performance of libraries such as QD and CAMPARY using the same techniques we have developed here.

References

[1] A. Moss, D. Page, and N. P. Smart, "Toward acceleration of RSA using 3D graphics hardware," in *IMA International Conference on Cryptography and Coding*, ser. LNCS, vol. 4887. Cirencester, UK: Springer-Verlog, December 2007, pp. 364–383.

[2] S. Fleissner, "GPU-accelerated Montgomery exponentiation," in *International Conference on Computational Science (ICCS)*, ser. LNCS, vol. 4487. Beijing, China: Springer, May 2007, pp. 213–220.

[3] D. J. Bernstein, T. R. Chen, C. M. heng, T. Lange, and B. Y. Yang, "Ecm on graphics cards," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2009, pp. 483–501.

[4] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of Computation*, vol. 44, no. 170, pp. 519–521, 1985.

[5] D. J. Bernstein, H. C. Chen, M. S. Chen, C. M. Cheng, C. H. Hsiao, T. Lange, Z. C. Lin, and B. Y. Yang, "The billion-mulmodper-second PC," in *Workshop Record of Special-purpose Hardware for Attacking Cryptographic Systems (SHARCS)*, vol. 9, Lausanne, Switzerland, September 2009, pp. 131–144.

[6] F. Zheng, W. Pan, J. Lin, J. Jing, and Y. Zhao, "Exploiting the floating-point computing power of GPUs for RSA," in *International Conference on Information Security*, ser. LNCS, vol. 8783. Hong Kong, China: Springer, October 2014, pp. 198–215.

[7] J. Dong, F. Zheng, W. Pan, J. Lin, J. Jing, and Y. Zhao, "Utilizing the double-precision floating-point computing power of GPUs for RSA acceleration," *Security and Communication Networks*, September 2017.

[8] J. Dong, F. Zheng, N. Emmart, J. Lin, and C. Weems, "sDPF-RSA: Utilizing floating-point computing power of GPUs for massive digitial signature computations," *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, to appear.

[9] T. J. Dekker, "A floating-point technique for extending the available precision," *Numerische Mathematik*, vol. 18, no. 3, pp. 224–242, 1971.

[10] A. H. Karp and P. Markstein, "High-precision division and square root," *ACM Transactions on Mathematical Software (TOMS)*, vol. 23, no. 4, pp. 561–589, 1997.

[11] Y. Hida, X. S. Li, and D. H. Bailey, "Algorithms for quad-double precision floating point arithmetic," in *Computer Arithmetic, 2001. Proceedings. 15th IEEE Symposium on*. IEEE, 2001, pp. 155–162.

[12] R. P. Brent and P. Zimmermann, *Modern Computer Arithmetic*. Cambridge University Press, 2010, vol. 18.

[13] N. Cruz-Cortés, E. Ochoa-Jiménez, L. Rivera-Zamarripa, and F. Rodríguez-Henríquez, "A GPU parallel implementation of the RSA private operation," in *Latin American High Performance Computing Conference*. Springer, 2016, pp. 188–203.

[14] P. Barrett, "Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor," in *Conference on the Theory and Application of Cryptographic Techniques*. Springer, 1986, pp. 311–323.

[15] H. Orup, "Simplifying quotient determination in high-radix modular multiplication," in *Computer Arithmetic, 1995., Proceedings of the 12th Symposium on*. IEEE, 1995, pp. 193–199.

[16] C. D. Walter, "Montgomery exponentiation needs no final subtractions," *Electronics Letters*, vol. 35, no. 21, pp. 1831–1832, 1999.

[17] K. Jang, S. Han, S. Han, S. B. Moon, and K. Park, "SSLShader: Cheap SSL acceleration with commodity processors." in *8th USENIX Conference on Networked Systems Design and Implementation (NSDI), Proceedings of the*. Boston, USA: USENIX Association, March 2011.

[18] Ç. K. Koç, T. Acar, and B. S. Kaliski, "Analyzing and comparing Montgomery multiplication algorithms," *IEEE Micro*, vol. 16, no. 3, pp. 26–33, 1996.

[19] N. Emmart and C. Weems, "Pushing the performance envelope of modular exponentiation across multiple generations of GPUs," in *2015 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. Hyderabad, India: IEEE, May 2015, pp. 166–176.

[20] Y. Yang, Z. Guan, H. Sun, and Z. Chen, "Accelerating RSA with fine-grained parallelism using GPU," in *In: Lopez J., Wu Y. (eds) Information Security Practice and Experience. Lecture Notes in Computer Science, vol 9065. Springer, Cham*, May 2015, pp. 454–468.