EE 705: Project Description
Integer Factorization using QFS and Bluespec

OV Shashank - 14D070021
Ajinkya Kharalkar - 14D070031
Yogesh Mahajan - 14D070022

May 7, 2018

1 Motivation: RSA Algorithm

In this section we describe the motivation for integer factorization.
RSA (Rivest–Shamir–Adleman) is one of the first public-key cryptosystems and is widely used for secure data transmission. In such a cryptosystem, the encryption key is public and it is different from the decryption key which is kept secret (private). In RSA, this asymmetry is based on the practical difficulty of the factorization of the product of two large prime numbers, the "factoring problem". A user of RSA creates and then publishes a public key based on two large prime numbers, along with an auxiliary value. The prime numbers must be kept secret. Anyone can use the public key to encrypt a message, but with currently published methods, and if the public key is large enough, only someone with knowledge of the prime numbers can decode the message feasibly.

RSA is a relatively slow algorithm, and because of this, it is less commonly used to directly encrypt user data. More often, RSA passes encrypted shared keys for symmetric key cryptography which in turn can perform bulk encryption-decryption operations at much higher speed. This implies that the ability to break the RSA key, may give access to the key of the subsequent encryption and potentially exposing critical information.

Different lengths of keys have been broken by several methods previously and hence it is interesting to see the opportunities in taking the security of the current systems, based on the RSA encryption, to the limit. Moreover the scheme cannot be done without as it is one of those schemes which does not require a secure channel to start of with and hence making this study important.

2 Previous Work (in HPC Lab)

Matrix-Vector product on a binary field using a simple LUT approach and its integration with the Wiedemann Algorithm has been simulated on a Multi-FPGA system using appropriate interfacing protocols. This is in conjunction with the Quadratic Field Sieve method which has not been implemented, but only used to generate the matrices offline.

3 Quadratic Field Sieve

The quadratic sieve algorithm (QS) is an integer factorization algorithm and, in practice, the second fastest method known (after the general number field sieve). It is still the fastest for

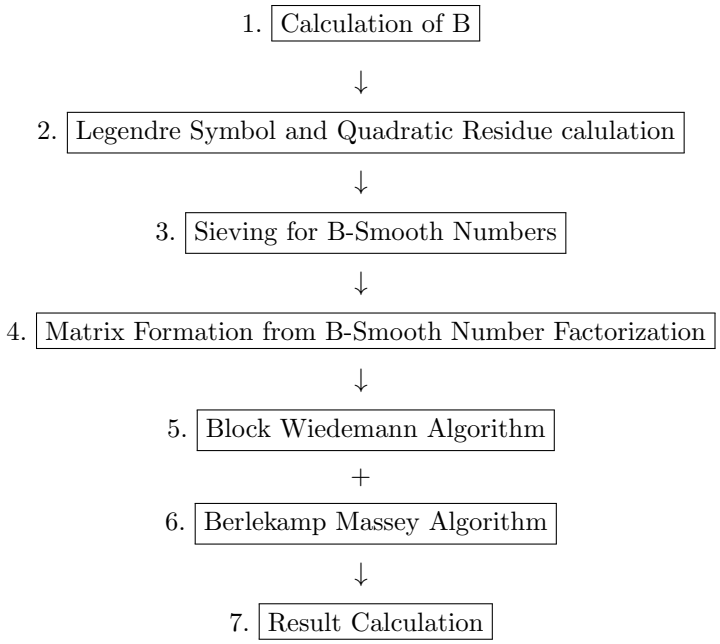
integers under 100 decimal digits or so, and is considerably simpler than the number field sieve. It is a general-purpose factorization algorithm, meaning that its running time depends solely on the size of the integer to be factored, and not on special structure or properties.

The algorithm attempts to set up a congruence of squares modulo n (the integer to be factorized), which often leads to a factorization of n.
The algorithm works in two phases:

- the data collection phase, where it collects information that may lead to a congruence of squares
- the data processing phase, where it puts all the data it has collected into a matrix and solves it to obtain a congruence of squares.

The data collection phase can be easily parallelized to many processors, but the data processing phase requires large amounts of memory, and is difficult to parallelize efficiently over many nodes or if the processing nodes do not each have enough memory to store the whole matrix. The block Wiedemann algorithm can be used in the case of a few systems each capable of holding the matrix which can be implemented on a multi-FPGA system.

The Algorithm flow is as follows:



4 Implementation

From the above algorithm flow the following were implemented by us:

- Step 1, 2, 3 and 4: In C++ based on the algorithm details mentioned ahead
- Step 5: Step 5 involves repeated Boolean Matrix Vector Multiplication. It was implemented in Bluespec using the Sparse Matrix approach.
- Step 5 and 6: Using only basic algorithmic techniques and not much optimizations, these two steps were also implemented in Bluespec

5 Algorithm and Optimizations

5.1 Basic Idea of Factorization using QFS

Given a large number N such that

$$N = A \times B \text{ where A and B are prime numbers of approximately the same order}$$

Consider two integers x and y such that

$$x^2 = y^2 \ (mod\ N) \text{ and } x \neq \pm y \ (mod\ N)$$

Basically: $x^2 - y^2$ divisible by N and $(x \pm y)$ is not divisible by N

$$\implies A = gcd(x - y, N);\ B = gcd(x + y, N)$$

In Quadratic Field Sieve we wish to find congruences such that

$$x_i^2 = a_i \ (mod\ N) \text{ where } a = \prod a_i \text{ is a square say } y^2$$

$$\implies If x = \prod x_i, \{then\} x^2 = y^2 \ (mod\ N)$$

and we hope that the condition on $x \pm y$ gets satisfied. Otherwise we go for a different set of x_i and the corresponding a_i.

5.2 B Smooth Numbers

Optimization: Note that if some x^2 mod N has a large prime factor to the first power, then if we are to involve this particular residue in our subset with square product, there will have to be another x'^2 mod N that has the same large prime factor. So, what if we do this systematically and throw away any x^2 mod n that has a prime factor exceeding B, say? That is, suppose we keep only the B-smooth numbers.

Based on statistics and the requirement to find a square within a certain limit of number of x_is,

$$B \approx \sqrt{e^{\sqrt{\ln n . \ln \ln n}}}$$

The exact choice may be different and is empirically decided. For example in out code B was chosen to be higher by 800 to satisfactorily obtain the desired results within bounded time because we were dealing with significantly small numbers

5.3 Matrix Generation

Let us associate an "exponent vector" to a B-Smooth number $m = \prod p_i^{e_i}$, where $p_1, p_2, ..., p_{\pi(B)}$ ate the primes upto B and each exponent $e_i \geq 0$. The exponent vector is

$$v(m) = (e_1, e_2, ...e_{\pi(B)})$$

If $m_1, m_2, ...m_k$ are all B-Smooth, then $\prod_{i=1}^{k} m_i$ is a square iff $\sum_{i=1}^{k} v(m_i)$ has all even coordinates.

Optimization: We reduce the exponent vectors modulo 2 and $\pi(B)$ think of them in the vector space $F_2^{\pi(B)}$. The field of scalars of this vector space is $F_2^{\pi(B)}$ which has only the two elements 0,1. Thus a linear combination of different vectors in this vector space is precisely the same thing as a subset sum; the subset corresponds to those vectors in the linear combination that have the coefficient 1. So the search for a nonempty subset of integers with product being a square is reduced to a search for a linear dependency in a set of vectors.

5.4 Obtaining B-Smooth Numbers and Exponent Vectors

NOTE: This section has been completely implemented in C++

Optimization: Choose primes to be tested based on the property $\left(\frac{n}{p}\right) = 1$ where $\left(\frac{n}{p}\right)$ is the Legendre symbol. This ensures that there exist x such that

$$x^2 = n \ (mod \ p_i)$$

which implies that if we have x going from $\sqrt{n}$ to n, $a_i = x^2 \ mod \ n = x^2 - n$ and hence $x^2 - n$ has p_i as a factor. All the primes which do not pass the Legendre Symbol are ignored because anyways they will not satisfy the above property which we desired.

Also obtaining the lowest a_i such that $a_i^2 = n \ (mod \ p_i)$ proves as good starting point for an approximate sieving.

- Basic Sieving: Sieve the sequence $(x^2 - n), x = \sqrt{n}, \sqrt{n} + 1, ...$ for B-Smooth values using primes generated as above until K such pairs of $(x, x^2 - n)$ are obtained where K is the number of primes generated. This gives a square matrix of future steps.
- Logarithmic Approximation: Instead of sieving using multiplications, using approximate logarithms of the primes and use addition instead. This introduces error but can be compensated by having an error margin and also skipping some B-Smooth numbers does not hurt.
- Prime Powers: Not to sieve by higher power of primes because it is more common for numbers to have only unity power of higher primes as the numbers tend to higher values.
- Using the Quadratic Residue: Add a $\log p_i$ increment (rounded to the nearest integer) starting at offsets x_i, x'^i, the least integers $\geq \sqrt{N}$ that are congruent $(mod \ p_i)$ to $a_i, -a_i$, respectively, and at every spacing p_i from there forward in the array.

5.4.1 High Precision Class

For the purpose of static implementation of high-precision numbers, a complete high-precision class was also implemented handling large number using the existent 64-Bit word.

5.5 BMVM using Sparse Matrix Structures

The matrix generated by the sieving step being a boolean matrix is quite sparse and amenable to sparse storage based implementation to reduce the storage size. Some of the common Sparse Matrix formats are

- COO: Coordinate format where the a tuple of the Non-Zero element and its Row and Column Index are stored.
- CSR: Compressed Sparse Row format stores the column indices of all the Non-Zero indices separately and the row information separately in another matrix. This is indeed better than the COO format provided the number of Non-Zero Element (NNZ) is greater than the dimension of the matrix which is usually the case.
- ELL: This format performs compression only in the columns and the row structure is maintained. Basically the number of columns is reduced to the highest number of Non-Zero Elements per row and the column indices are only stored.

On analyzing the sparse structure of the matrix generated by the above processes, the following observations were made:

- The Non-Zero density is higher on the lower primes side and hence is a blocked approach of splitting the large matrix into blocks and using the Perfect Difference Network style communication protocol, leads to a significant imbalance among the processors. This can be corrected by permuting the columns because the order of columns has no particular significance, and the permutation can be restored later.
- The matrix on row-wise analysis shows that is particularly suitable for the ELL Format as the NNZ per row do not vary too much and the mean is pretty close to the maximum value.

Two Implementations of the BMVM were tried

5.5.1 CSR Format

Using this format the BMVM can be done element wise. Some analysis parameters for this format are

- Size Compression: The size occupied by this format can be calculated as

  $$N \log_2 D + D \log_2 N$$

  which looks particularly symmetric. This method gives the best compression ratio.

  $$\text{Compression Ratio} = (N \log_2 D + D \log_2 N)/D^2$$

  The resulting second term is significantly smaller than 1 and the ratio majorly depends directly on the sparsity of the matrix.
- Time Taken: The time taken by this method is approximately the NNZ in the matrix. If NNZ is less than the dimension of the matrix, this could be faster than the row based approach
- Logic Hardware: Apart from the Size Compression the hardware utilized in this method is also absolutely minimal consisting of only one XOR gate and a multiplexer appearing in the arithmetic critical path.

5.5.2 ELL Format

Using this format the BMVM can be done row at a time. Some analysis parameters for this format are

- Size Compression: The size occupied by this format can be calculated as

  $$D(\log_2 D) \max(\mathrm{N_{row}})$$

  This method gives the best compression ratio if the NNZ per row is constant.

  $$\text{Compression Ratio} = (\log_2 D) \max(\mathrm{N_{row}})/D \approx N \log_2 D/D^2$$

- Time Taken: The time taken by this method is approximately the Dimension of the matrix. Hence this approach in general is faster than the CSR format.
- Logic Hardware: Because Row operations are performed at once, as many XOR Gates and multiplexers as the $\max N_{\text{row}}$ are required.

From the above discussion and the earlier observation the ELLPACK format seems to be the choice to take.

5.6 Wiedemann Algorithm

Let M be an $n \times n$ square matrix over some finite field F, let x_{base} be a random vector of length n, and let $x = Mx_{base}$. Consider the sequence of vectors $S = \left[x, Mx, M^2x...\right]$ obtained by repeatedly multiplying the vector by the matrix M; let y be any other vector of length n, and consider the sequence of finite-field elements $S_y = \left[y \cdot x, y \cdot Mx, y \cdot M^2x \dots\right]$

We know that the matrix M has a minimal polynomial; by the Cayley–Hamilton theorem we know that this polynomial is of degree (which we will call n_0 no more than n. Say $\sum_{r=0}^{n_0} p_r M^r = 0$. Then $\sum_{r=0}^{n_0} y \cdot (p_r(M^r x)) = 0$; so the minimal polynomial of the matrix annihilates the sequence S and hence S_y.

But the Berlekamp–Massey algorithm allows us to calculate relatively efficiently some sequence $q_0 \dots q_L$ with $\sum_{i=0}^{L} q_i S_y[i+r] = 0 \ \forall \ r$. Our hope is that this sequence, which by construction annihilates $y \cdot S$, actually annihilates S; so we have $\sum_{i=0}^{L} q_i M^i x = 0$. We then take advantage of the initial definition of x to say $\sum_{i=0}^{L} q_i M^i x_{\text{base}} = 0$ and so $\sum_{i=0}^{L} q_i M^i x_{\text{base}}$ is a hopefully non-zero kernel vector of M.

Flow of Wiedemann algorithm:

1. Two random vectors x and y of length n are generated.
2. Krylov subspace k is generated, which is defined as

$$K = [x, Ax, A^2x, ..., A^{k-1}x]$$
$$K_y = y \cdot K = [y \cdot x, y \cdot Ax, y \cdot A^2x, ..., y \cdot A^{k-1}x]$$

3. Considering the sequence $s_i = K_y[i]$, a sequence C with minimal length l is generated using the Berlekamp-Massey algorithm so that

$$\sum_{i=0}^{l-1} c_i s_{l-1-i} = 0$$

   where $C = [c_{l-1}, c_{l-2}, ..., c_0]$.
   Another sequence M is generated so that, $m_i = c_{l-1-i}$

4. Let $u = M$. If u is non-trivial and $Au = 0$, u lies in the null space of A. Otherwise we start from step 1 again till we get a non-trivial u in the null space of A.

5.6.1 Berlekamp-Massey algorithm

Berlekamp-Massey algorithm is an algorithm which gives the shortest LFSR for a given binary sequence. For an input sequence of $[s_{n-1}, s_{n-2}, ..., s_0]$ of length n, this algorithm produces a ouput sequence $[c_{l-1}, c_{l-2}, ..., c_0]$ of length l such that,

$$\sum_{i=0}^{l-1} c_i s_{l-1-i} = 0$$

The algorithm proceeds through the following steps:

1. Let $s_{n-1}, s_{n-2}, ..., s_0$ be the input bit stream with length n.
2. Initialize two zero arrays b and c of length n except b_0 =1 and c_0 =1.

3. Assign $l = 0$ and $m = -1$

4. For $N = 0$ step 1 while $N < n$:

   - Let d be the discrepancy define as $s_N + c_1 s_{N-1} + c_2 s_{N-2} + ... + c_L s_{N-L}$
   - If $d = 0$, then c is already the polynomial which takes care of the portion from $N - L$ to N.
   - Else, c is copied in t.Set $c_{N-m} = c_{N-m} + b_0$ $c_{N-m-1} = c_{N-m-1} + b_1$,...., $c_{n-1} = c_{n-1} + b_{n-N+m-1}$. If $L \leq N/2$, set $L = N + 1 - L$, $m = N$ and $b = t$. Else leave L, m and b alone, where + is the exclusive OR operator.

6 Bluespec System Verilog

Bluespec SystemVerilog) is a high-level language used in the design of electronic systems (ASICs, FPGAs and systems). BSV is used across the spectrum of applications—processors, memory subsystems, interconnects, multimedia and communication I/O devices and processors, high-performance computing accelerators, etc. It is a very high-level language as well as fully synthesizable to hardware. This combination of high level and full synthesizability enables many of these activities that were previously only done in software simulation now to be moved easily to FPGA-based execution. This can speed up execution by three to six orders of magnitude (1000x to 1,000,000x).

In BSV, the model of atomic rules is fundamentally parallel.The atomicity of rules is a powerful tool in thinking about correct behaviour in the presence of complex parallelism. Atomicity of rules allows reasoning about invariants one rule at-at-time. Unlike C++ and SystemC TLM, BSV is architecturally transparent.BSV's powerful types and static elaboration mechanisms provide great power to express architectural structures very succinctly and programmatically Also BSV has a superior behavioural semantics—Atomic Rules and Interfaces—which is a higher-level abstraction for concurrency and is much better suited to the task of describing the fine-grain, multi-rate, heterogeneous parallelism found in hardware systems. BSV has much stronger parametrization, which directly affects everything: code size, code structure, code reuse, and code correctness.

7 Running the Codes

- The folder Sparse contains all the codes used to perform the Sparse part. All the C++ codes can be compiled as `g++ -O3 -std=c++11 <name>.cpp -o <name>`

  The sub-folder ELL_BMVM contains Bluespec code for the ELLPACK format along with the Makefile

  Similarly the sub-folder CSR_BMVM contains Bluespec code for the ELLPACK format along with the Makefile

- The folder Sieve++ contains the Sieve code and the preceding codes on Legendre Symbol and other supplementary codes. The Sieve code can again be compiled as `g++ -O3 -std=c++11 sieve.cpp -o sieve` The other codes can be run using the supplied Makefile

- The codes in the HP folder, uint192.h and uint192.cpp correspond to the High-Precision Class. It has been integrated with the rest of the codes and the Makefile is included

- The folder Wiedemann includes the bluespec codes for Berlekamp-Massey algorithm and the Wiedemann algorithm. This folder contains sub-folders for individual modules and one sub-folder named "Wiedemann" which has the integrated codes. These codes can be run using a Makefile present in the folder. This folder also contains a README text file which dictates the procedure of running the codes.

8 Bibliography

1. Crandall, R., Pomerance, C. (eds.): Prime Numbers - A Computational Perspective. Springer, New York (2005). https://doi.org/10.1007/0-387-28979-8
2. Wikipedia
3. Learning Bluespec

9 Updates

- BMVM Codes have been made modular using typedefs
- Bug: BMVM codes produce correct result only when compiled using `{make vsim}`
- Wiedemann Algorithm and other codes have been added