Analysis and Parallelization of RSA based Algorithms

DEEPAK SRIVATSAV, IIIT Delhi
SURYATEJ REDDY, IIIT Delhi

ACM Reference Format:
Deepak Srivatsav and Suryatej Reddy. 2019. Analysis and Parallelization of RSA based Algorithms. 1, 1 (April 2019), 6 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 PROJECT OVERVIEW

Through this project, we aim to analyse the algorithm and performance of RSA Encryption and Decryption, and also a number of attacks on weak RSA.

2 LITERATURE REVIEW

First we started off by reading the paper titled " A method for obtaining digital signatures and public-key cryptosystems" [1] which was the first paper to propose the RSA encryption technique. After understanding key concepts involved in RSA, we started searching for parallel implementations when we found the paper titled "Analysis Of RSA Algorithm Using GPU Programming". In this paper, the authors address the issue of large messages and propose the division of a message into blocks for encryption and decryption. Although this does utilize the GPU efficiently to encryption the message in parallel, the major issue with RSA is the time taken in the modular exponentiation process. We aim to make the modular exponentiation process efficient and achieve a speedup over the above mentioned paper.

3 ANALYSIS OF THE ALGORITHM

3.1 RSA Algorithm

RSA is an asymmetric cryptographic algorithm that is widely used in public key cryptography. It involves a public key and a private key. After generating keys, there is a modular exponentiation required.

$$c = m^e \pmod{n}$$

Here m is the message to be encrypted , (e, n) is the public key, $n = p * q$, where p and q are primes, and c is the encrypted message. The paper we used as reference solves the problems of large messages but we noticed that the modular exponentiation implemented in the paper is an efficient serial execution and involved no parallel component. Every block of the message was calculating the modular exponent value serially. The modular exponentiation implemented in the paper uses a repeated square-and-multiply method. The method makes use of the fact that if the e value is even, then the modular exponential is calculated as

$$m^e \pmod{n} = (m^{\frac{e}{2}} \pmod{n} * m^{\frac{e}{2}} \pmod{n}) \pmod{n}$$

Authors' addresses: Deepak Srivatsav, 2016030, IIIT Delhi; Suryatej Reddy, 2016102, IIIT Delhi.

XXXX-XXXX/2019/4-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

If e is odd, then we simply multiply the above equation with $g^e \pmod n$ This can be viewed as a collection of independent tasks that can be parallelized on the GPU. The dependency graph for this is shown in figure.

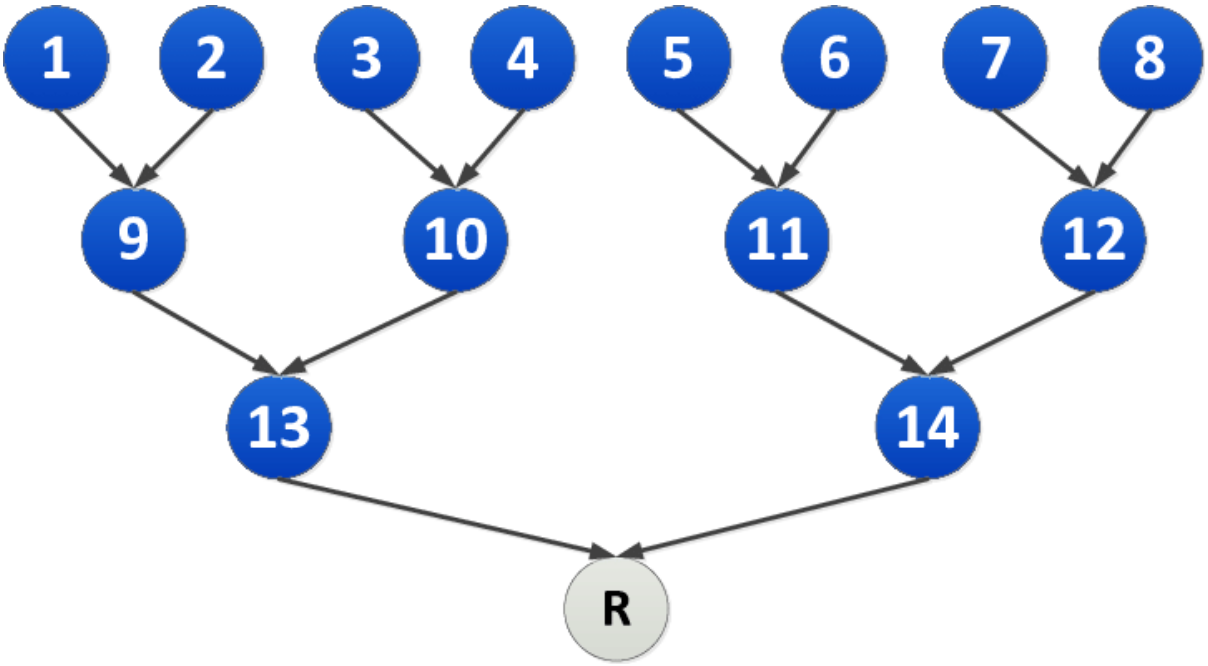

Fig. 1. Task Dependency Graph For Modular Exponentiation Algorithm Mentioned Above. The tasks one through eight have the value m^2 stored with them and they multiply it further to form multiples of 2 as we go lower in the graph. Finally, R contains the value m^e

3.2 Fermat Factorization Attack on RSA

As mentioned above, RSA requires the usage of two large prime numbers p and q. However, when p and q are close in value, i.e for example, if p and q are 512 bit numbers, and the first 256 significant bits are the same, then it is possible to find p and q from the modulus n. This can be done by approximating the value of all $a^2 - n$, $\forall a \in [\sqrt{n}, n]$. If any such value of a is a perfect square, then for this value of a, $b = \sqrt{a^2 - n}$, and our two prime numbers would be a-b and a+b.

3.3 Pollard P-1 Factorization Attack on RSA

This factorization works only for specific sets of numbers. It is an example of algebraic-group factorization. Given some modulus n, we try to find some smoothness bound B such that we can define some M, where q's are primes,

$$M = \Pi_{q \leq B} q^{log_q B}$$

Since n is typically odd, we choose some a coprime to n (we consider a=2), and compute $g = gcd(a^M - 1, n)$. If we find g to be such that $1 < g < n$, then g is one of the factors of n. If g=1, then increase B. If g=n, then decrease B and repeat.

4 PARALLELIZATION STRATEGY

4.1 Encryption - Modular Exponentiation

After visualizing the task dependency graph, we observed that the best way to parallelize this algorithm would be a modified reduction operation. For the mid review, our implementation of reduction algorithm applies modular multiplication to an array of integers filled with the value of m^2. The size of the array depends on value of the exponent. This modification has brought down the time complexity of calculating exponents from $O(n)$ to $O(\log n)$ (if we take multiplication operations to be one operation). This is a considerable improvement over the baseline paper we have selected. After mid evaluation, to optimize further we implemented our algorithm to work for larger values of exponent (10x the midsem value). For this we had to implement a batch exponentiation technique to caluculate exponents in batches and then merge them together.

4.2 Fermat Attack

This attack shows prominence over its serial version when p and q are primes with a large size, such as 512 bits. To this end we have implemented an exploratory analysis parallel version of the code.

4.3 Pollard P-1 Factorization

Once again, we faced constraints with respect to multi-precision integers with cuda. However, we manage to use exploratory analysis and dynamic parallelism in the parallel and optimized version of this code. It also included a parallel implementation of Sieve of Erathosthenes, and an optimized GCD implementation that works faster than the typical Euclidean algorithm.

5 PERFORMANCE EVALUATIONS

5.1 Encryption - Modular Exponentiation

One of our key deliverable for the mid term evaluation was implementing our reference paper. We have not only implemented the paper but also improved on their performance using the modifications mentioned above. We have evaluated our algorithm by calculating the speedup with reference to the paper.

$$speedup = \frac{T_{paper}}{T_{ourcode}}$$

We plotted a speedup graph by varying values of the exponent e.

After mid review, we could run our code on larger values of the exponent and thereby achieve higher speedup.

Table 1. Paper Time vs Improved Time for various exponent values

Exponent Value - e	**Paper Time (GPU) (ms)**	**Improved time (ms)**	**Speedup**
20	0.014112	0.022848	0.617647
2000	0.153632	0.026528	5.791315
4000	0.29408	0.040448	7.27057
12000	0.856736	0.037952	22.5742
20000	1.420992	0.045696	31.096638

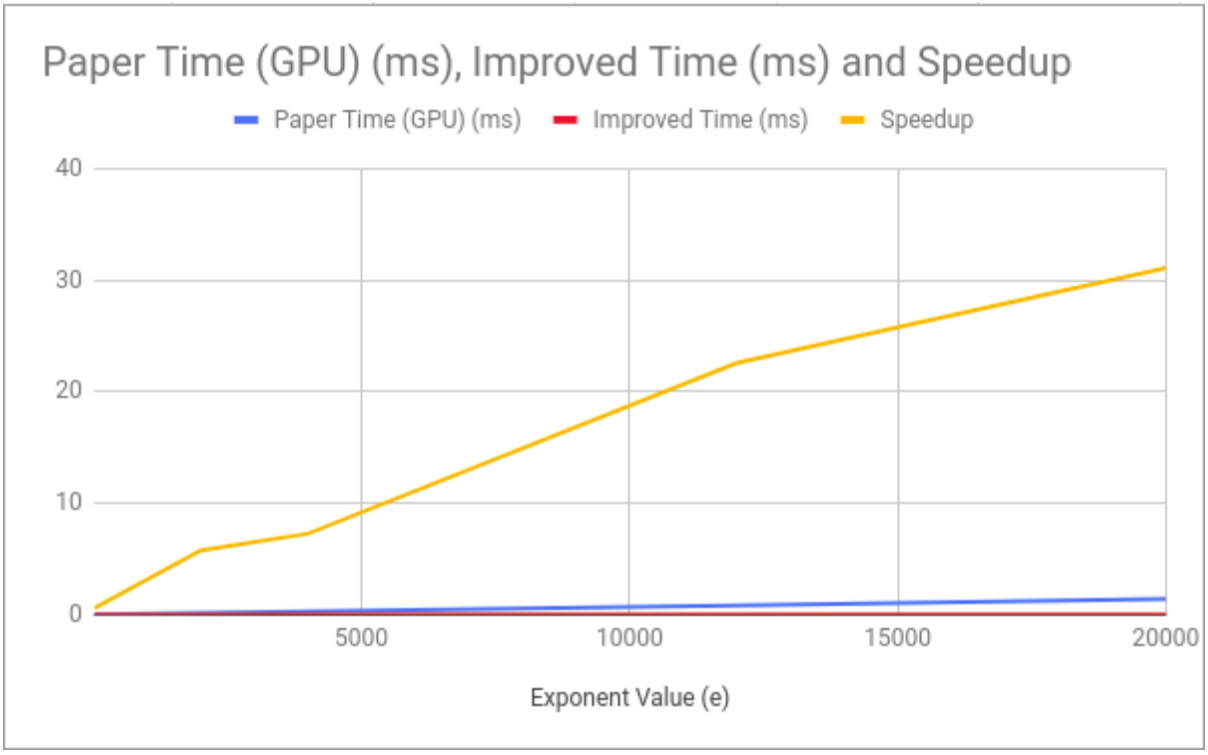

Fig. 2. Speedup vs Exponent for Increasing Values of Exponent. The Speedup is calculated as 'Paper time' divided by 'Improved time'

5.2 Fermat Attack

We tried to check the serial vs parallel performance for different values of the modulus n. For the purpose of timing the execution, we check the timing on the worst case exploratory situation on both the cpu and gpu version of the code.

Table 2. CPU and GPU Times for Fermat Attack for various modulus values

Modulus Value - n	**Serial Time (ms)**	**Parallel Time (ms)**	**Speedup**
18512544	0.002	1.4569	0.001372
4335743309	39420.6406	87.2057	452.0420179
69417725381	643764.3125	1214.2948	530.1548788

5.3 Pollard P-1 Attack

For this, we evaluated the relative speedup between increasing modulus sizes for the worst case exploration scenario, since we couldn't evaluate on large enough values of the modulus and Smoothing factor. We define the relative speedup as the ratio of the time taken between consecutive (increasing) values of B for CPUs and GPUs separately. If the GPU is able to have a small relative speedup compared to the CPU, then for large values of B,

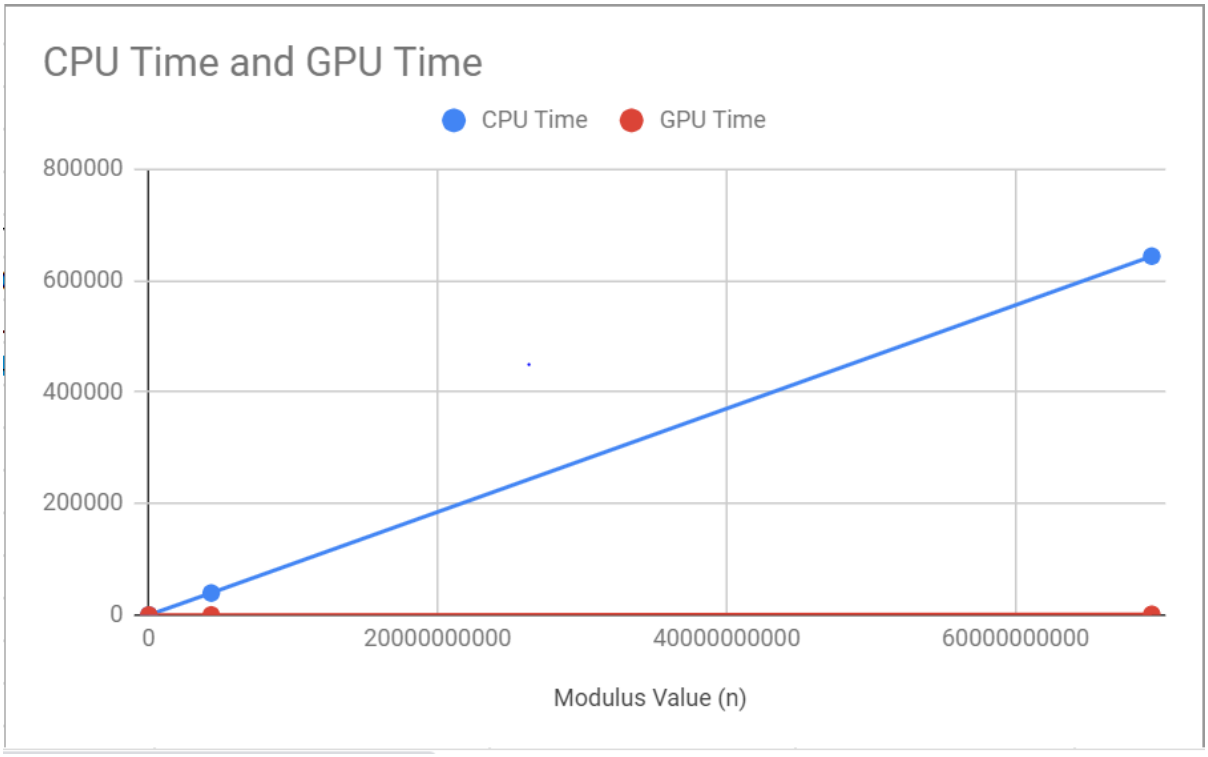

Fig. 3. CPU Time vs GPU Time for the execution of Fermat Factorization attacks on different Modulus Values

the GPU would be much faster. We also explored the resource utilization in the GPU implementation, basically evaluating how different block sizes affected the runtime of the algorithm. We present results for both.

Table 3. Division of Blocks per grid and Threads per block for a fixed total

Blocks per Grid	**Threads per Block**	**Time Taken (ms)**
32	32	46.37
100	10	47.29
200	5	67
10	100	71.74
250	4	75.65
1	1000	306.49

6 DELIVERABLE COMPONENTS

(1) Optimized RSA Encryption - Improvement on the reference paper through various optimization techniques.
(2) Fermat Attack - CPU and GPU
(3) Pollard Attack - CPU and GPU

Table 4. Statistics for Pollard Attack

Values for B	**CPU Time (ms)**	**GPU Time (ms)**	**Launch Config**	**CPU Relative Speedup**	**GPU Relative Speedup**
100	0.2	9.94	(10,10)	1	1
300	2.4	13.59	(20,10)	12	1.36
500	2.8	19.59	(30,10)	1.16	1.44
1000	11	47.29	(100, 10)	3.92	2.47

7 GITHUB

The code can be found at this link

REFERENCES

[1] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, February 1978.

[2] Sonam Mahajan and Maninder Singh. Analysis of RSA algorithm using GPU programming. *CoRR*, abs/1407.1465, 2014.