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), 4 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, Maninder, Sonam, et al. 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.

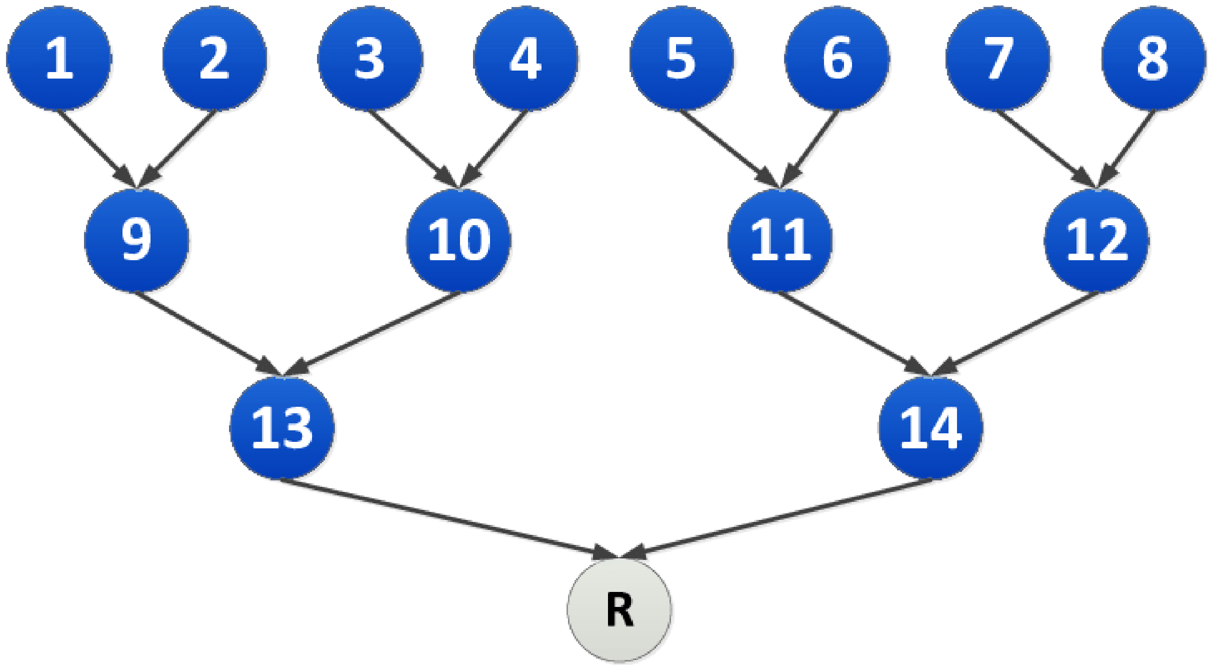

Fig. 1. Task Dependency Graph For Modular Exponentiation Algorithm Mentioned Above. The tasks one through eight have the value m^2 stored with them and they multiply it further to form multiples of 2 as we go lower in the graph. Finally, R contains the value m^e

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.

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. Further, we plan to implement certain bit level optimizations for bringing down compute time of exponents. We can plan to improve our reduction algorithm by implementing the algorithms discussed in class (sequential addressing, unrolling etc.,).

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. The implementation of Big Integer operation on cuda isn't trivial, and we have reserved this for our final deliverables. For the mid-review, we have implemented a parallelized version of this attack for values of p and q in the range of 30 bits as our value of n must be less than 64 bits. To this end we have implemented a naive code that will explore values of a that satisfy these conditions without any optimizations. This of course is done in parallel. Future optimizations will include the usage of shared memory and warp memory to make the calculations of squares of large numbers efficient by utilizing normal math identities such as $(a+b)^2$ to store intermediate square values that can be reused to calculate $(a+i)^2$ from $(a+(i-1))^2$

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.

5.2 Fermat Attack

We tried to check the serial vs parallel performance for a value of $n = 1103191240211$. We used Sage Math for the serial version, which is a highly optimized library for math operations. The attack took 5 milliseconds on Sage Math whereas on our parallel implementation, it took 10 microseconds. Since we were performing a sort of exploratory analysis, we had to terminate the kernels once any one thread found the correct value. To do this, we use the trap interrupt with asm assembly. However for the purpose of timing the execution, we use nvprof to gather the total time taken by memory operations, kernel calls, etc, and we subtract these from the wall clock time taken by the program. We had to resort to this method since we were unable to use cuda events (as the process was being terminated midway). We found that the kernel was only taking 0.01% of the total execution time, which amounted to 10 microseconds.

6 FINAL DELIVERABLES

(1) Optimized reduction for Encryption and Decryption, possibly bit level optimizations
(2) Support for larger numbers, 128 and 256 bit.
(3) Optimized Fermat attack
(4) Hastad attack.

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.

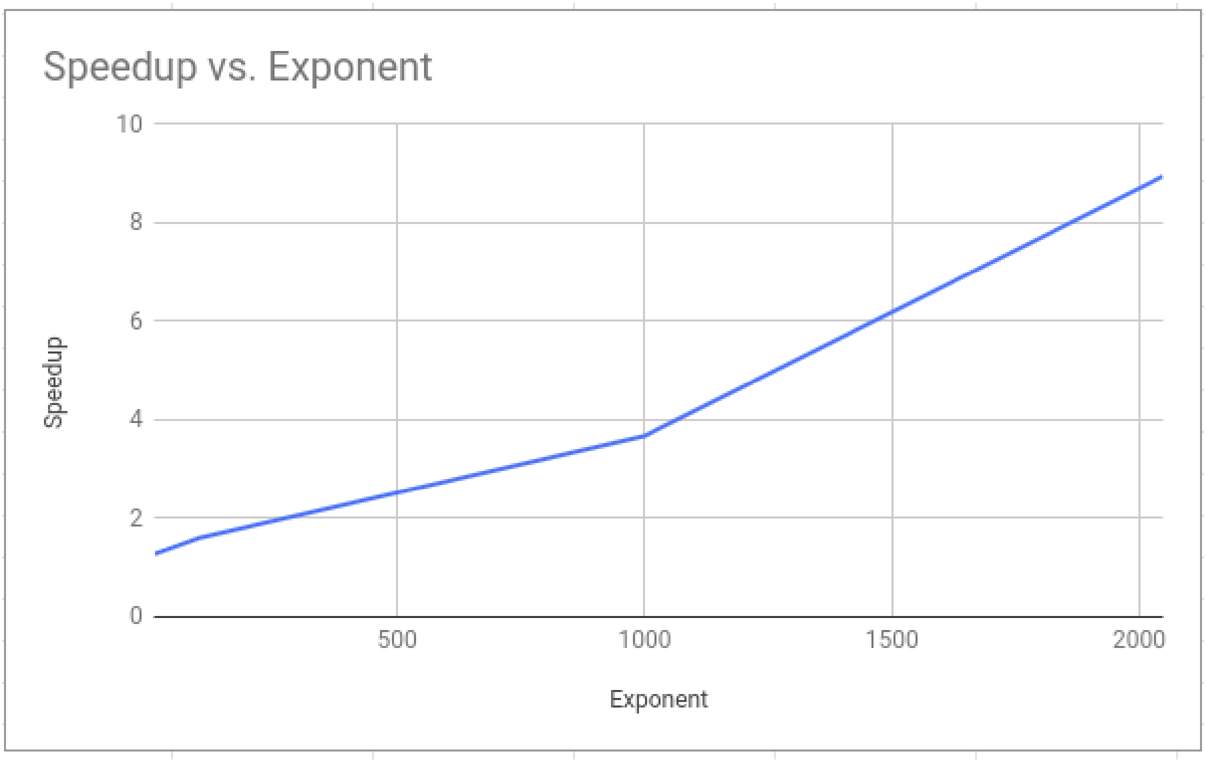

Fig. 2. Speedup vs Exponent for Increasing Values of Exponent

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