An Optimization Framework for Cryptographic Hash Function

Qiushi Xu, Taoyi Gong
Department of Electrical Engineering
University of Southern California, Los Angeles, CA, USA
{qiushixu, taoyigon}@usc.edu

Abstract- In order to satisfy the growing information security demand, encryption algorithms that are more rapid and more efficient need to be developed. In this project, we first analyze the motivation to develop encryption algorithms, and go through some previous work in this field. Then, we attempt to use load balancing framework to accelerate encryption algorithms and finally, we will analyze the performance improvement brought by this new method.

Ⅰ. Introduction and motivation

With the rapid development of information and telecommunication technology, the significance of information security has been increasing continuously. As one of the mathematical algorithms of encryption technology, cryptographic hash function has been widely used in e-commence, cyber-security and data storage. However, while the amount of information data and the speed of network are both in high-speed growth, the calculation throughput of encryption algorithms are unable to meet the demand in the high-density data environment. Under this circumstance, our project aims to implement a novel methodology to accelerate the speed of encryption hash functions in order to achieve the goal of performance optimization. Inspired by a complex clustered network-based load balancing framework,[1] we propose the optimized application of both MD5 and SHA algorithm.

Ⅱ. Related work

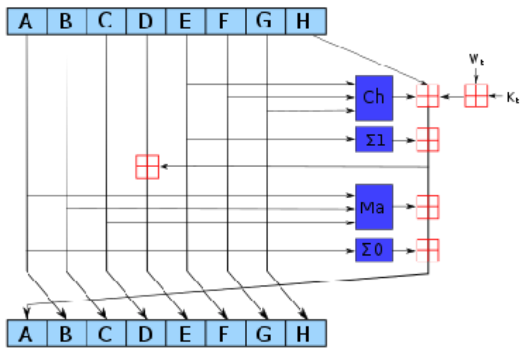
The encryption algorithm is an important technique to protect information data, which can be classified into three types based on whether secret key is used or the usage of secret key:

A. Private Key Algorithm, where an identical key is kept by both sides of transmission, is also called symmetric encryption algorithm. It has the advantage of high speed, but the disadvantage lies in poor confidentiality of key. Common private key algorithms include DES, 3DES and AES.[2]

B. Public Key Algorithm, where sender and receiver have their own private keys and one public key, is also called asymmetric encryption algorithm. It is good for easiness of keys' distribution and preservation, but is in lower performance compared with private key algorithm. Common public key algorithms include RSA, DSA and ECC.[3]

C. Cryptographic Hash Function, or message digest function, is different from previous two algorithms, as its lack of utility of secret key. Hash function is a one-way function to map data of an arbitrary size to a bit array of a fixed size, for which it is practically infeasible to invert or reverse the computation. Therefore, applications of cryptographic hash function are notably in digital signatures, forms of authentication and checksum for corruption detection. Common hash functions include MD5, SHA-1 and SHA-256.

MD5, that is Message-Digest Algorithm 5, converts message with arbitrary size into a hash value with 128 bits. It was designed by Ronald Rivest in 1991 to replace an earlier hash function MD4,[4] and was specified in 1992 as RFC 1321. Since it was first developed, it has been widely applied for long period of time for no severe issue found, despite the appearance of progressive results as regards attacks on MD5 in recent years.



SHA, that is Secure Hash Algorithms, is a family of cryptographic hash functions published by the National Institute of Standards and Technology (NIST),[5] including SHA-0, SHA-1, SHA-2 and SHA-3, each of which was successively designed with increasingly stronger encryption in response to hacker attacks. In our project, SHA-256, that is part of the SHA-2 family, is chosen to be analyzed, as it's the most widely used and best hashing algorithm, often in conjunction with digital signatures.

With respect to speed up on calculation of encryption algorithm, there exist some productive and inspiring researches. Ye Tian et al. [6] proposed the experimental results, in which the MD5 algorithm based on FPGA implementation has high processing speed and less resource usage. Similarly in the hardware direction, taking advantage of the GPU's powerful parallel computing capability, Thuong Nguyen Dat et al. [7] implemented Keccak-512 hash algorithm with the integrated development environment CUDA for GPU in the heterogeneous GPU+CPU system. At the principal design level, Soufiane Oukili et al. [8] accelerated AES algorithm at the stage of the substitution box (S-box) transformation by using pipeline technique to allow a parallel processing in order to obtain high throughput.

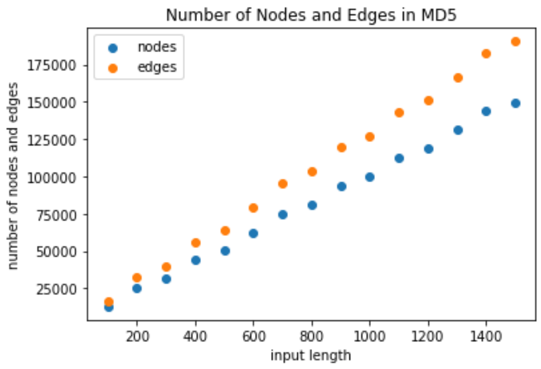
In particular with SHA-256, Hachem Bensalem et al. [9] demonstrated an OpenCL-based optimization methodology of SHA-256 implementation, among which the best reported optimized implementation achieves a throughput of 3973 Mbps, which is 4.3 times higher than the best previously published HLS-based SHA-256 implementation and even higher than the previously published implementations using a hardware description language. At the system on chip (SoC) level, Thi Hong Tran et al. [10] solve the performance problem with a novel SHA-256 architecture named the multimem SHA-256 accelerator which achieves significantly better processing rate and hardware efficiency than previous works. Notably, the accelerator employs three novel techniques, the pipelined arithmetic logic unit (ALU), multimem processing element (PE), and shift buffer in shift buffer out (SBi-SBo), to reduce the critical path delay.

III. Preliminary results in LLVM

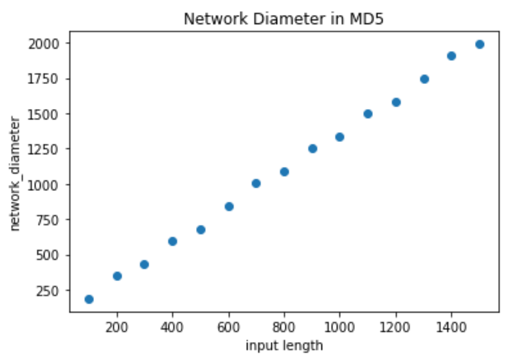
In our first stage of experiment, we implemented MD5 Algorithm and analyzed it with LLVM and Gephi tool.

In order to figure out how the size of the input influence properties of the algorithm, we run the MD5 program with various sets of input, whose lengths are distributed between [100, 1500], under LLVM scripts and checked statistical results of graphs generated.

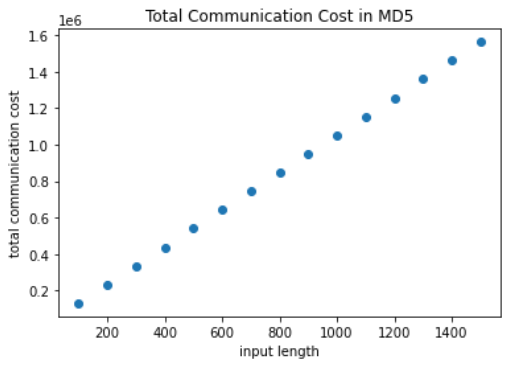
A. Number of Nodes and Edges



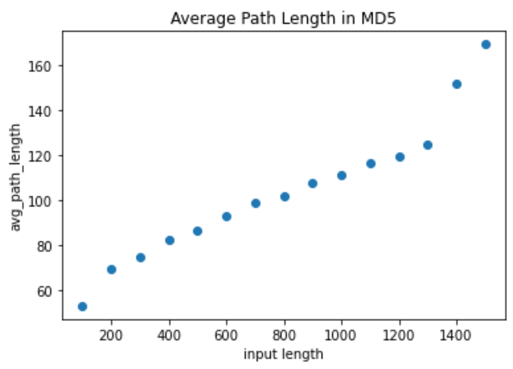
B. Network Diameter



C. Total Communication cost

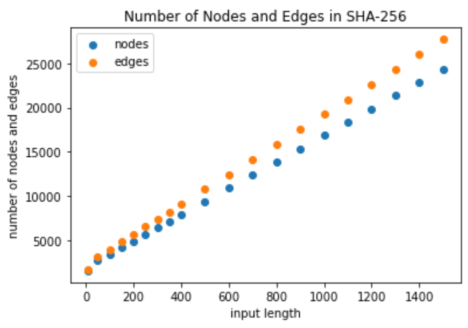


D. Average Path Length

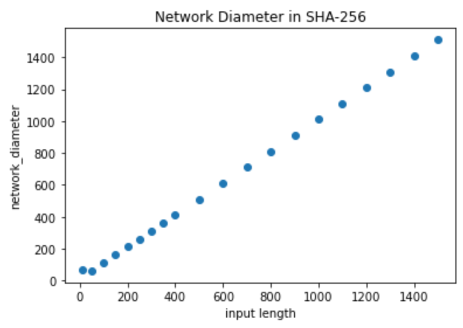


At the second stage of our experiment, for comparison purpose, we implemented SHA-256 Algorithm and analyzed it in the identical method.

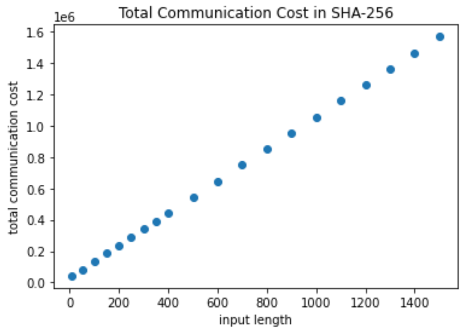
A. Number of Nodes and Edges



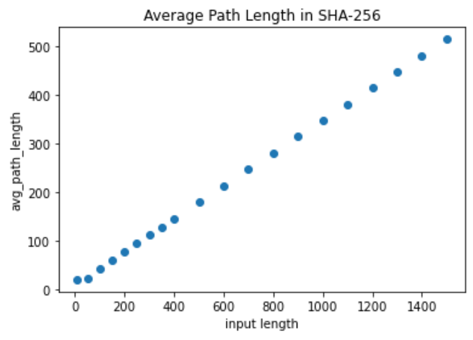
B. Network Diameter



C. Communication cost



D. Average Path Length

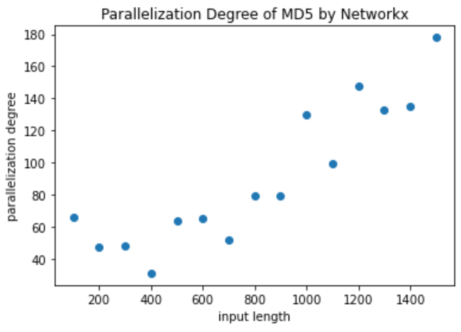


In our experiments, we tested the performance of these two algorithms. The result shows that the length of input data is linear with the number of nodes, edges, network diameters, average path length and communication costs.

In addition, we tried to explore parallelization degree and its relationship with input with fixed parameters of Quality Function in thesis (N=100, λ_1 =0, λ_2 =0.5).[1]

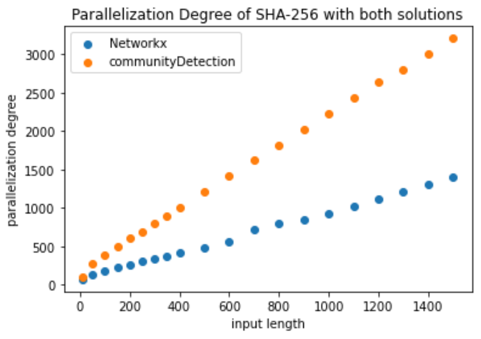
For MD5 Algorithm, we have difficulties on first solution, which is to count the number of files generated by performing Community

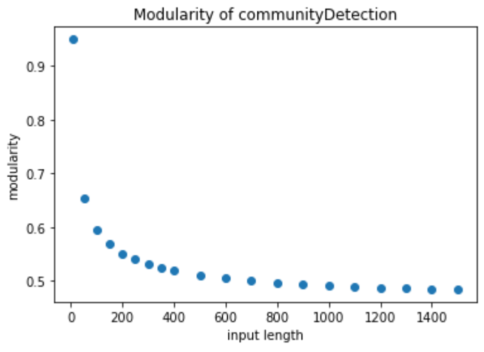
Detection and Clustering, due to extremely huge volume of data generated with MD5's high computation complexity.



As it turns out, MD5 has unpredictability on parallelization degree to some extent compared with linearity

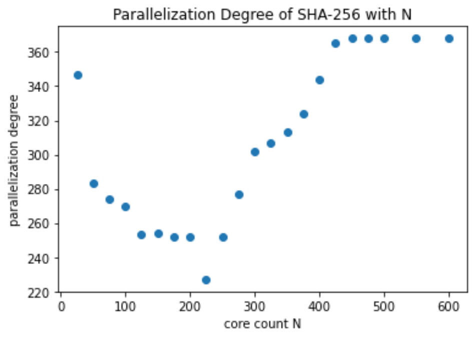
Therefore, we focus our further research of parallelization degree and modularity on SHA-256, for which we applied both solutions mentioned in professor's instructions.

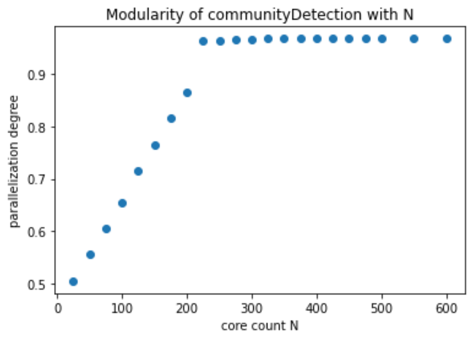




It turns out that, for both algorithms, parallelization degree rises with the increment of input size, but the performance of modularity is getting worse.

Then we fixed input length to 50, and explore how the parameter N influence parallelization degree and modularity.





$$R_2 = \frac{\lambda_2}{n_c^2}(n_c - N)^2 H(n_c - N)$$

From these two graphs, we can tell that when the core number N is smaller than 225, the parallelization degree decreases as the N increase. When N is larger than 225, the parallelization degree increases and has a up limit of 368. This is because in order to ensure the number of clusters doesn't exceed the core count, when n_c is larger than N, the quality function will subtract R_2. The modularity value shows the same thing: when n_c is larger than N, the value is relatively small and increases with N. When n_c is smaller than N, modularity reaches a high value and keep stable.

Ⅳ. Optimization with Gem5

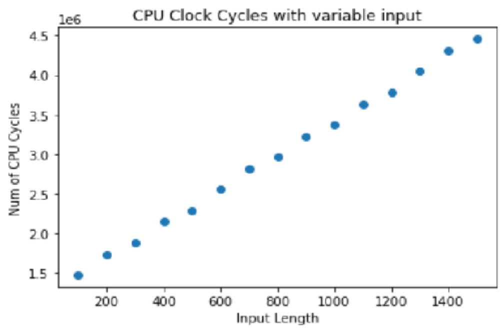
In the second stage of our experiment, we try to use Gem5 to simulate the execution time in terms of clock cycles and optimize with the low-level clusters generated by Community Detection Algorithm during LLVM experiment.

In the simulation of Gem5, we used the default configuration script se.py, which is System Call Emulation Mode (SE). For this mode, binary executable file must be statically compiled (In the first experiment we used binary executable files whose input size are from 100 to 1500, but after that we only focus on executable file with input=50). To specify a few important options of the simulation, we use the following command:

build/X86/gem5.opt ./configs/example/se.py -c ./sha256 --cpu-type=TimingSimpleCPU --caches --l2cache --l1d_size=128kB --l1i_size=128kB --l2_size=1MB --l1d_assoc=2 --l1i_assoc=2 --l2_assoc=2 --cacheline_size=64

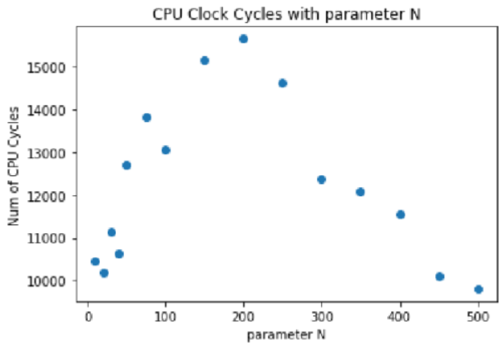
This defines a L1 Data cache, 2-way set-associative, with a 64-byte block, 1024 sets (2*64*1024=128KB). It also defines a similar Instruction cache, and a unified L2 1MB cache directed mapped, with 64-byte block, 8192 sets (2*64*8192=1MB). With the defined options and the SE script, a microprocessor architecture configuration is established.

At the beginning of the simulation, we ran the simulation and the result indicates that the initial number of CPU Clock Cycles is 1340505. In addition, like what we did in LLVM experiment, we tried to find out how the input size of the program influence the execution time in terms of CPU Clock Cycles. We ran the simulation of SHA-256 program with 15 sets of sample inputs, whose lengths are distributed between [100, 1500].

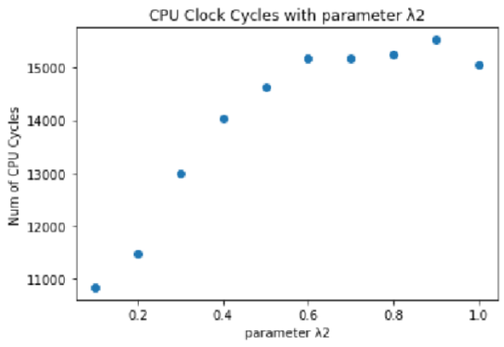


This image shows a prefect linear relation between the input size and the number of clock cycles. In addition, the clock cycles are about 1 million to 4 million. In the following experiment, we will show the significant improvement brought by the optimization algorithm.

After that we optimize the Gem5 simulation by applying LLVM Community Detection Cluster to parallelization. We want to know how the three parameters (N, λ1, λ2) in the Community Detection Algorithm affect the optimization performance.

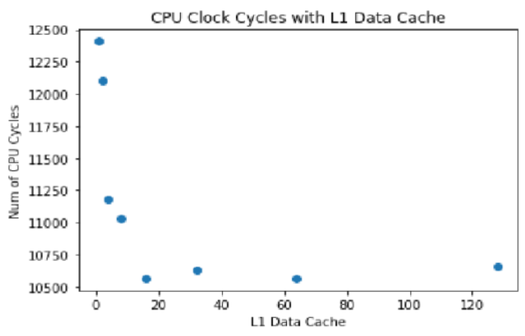


This graph shows the relationship between parameter N with CPU Clock Cycles. We can see that the CPU clock cycles is around 10K to 15K, which is far less than the previous simulation result.

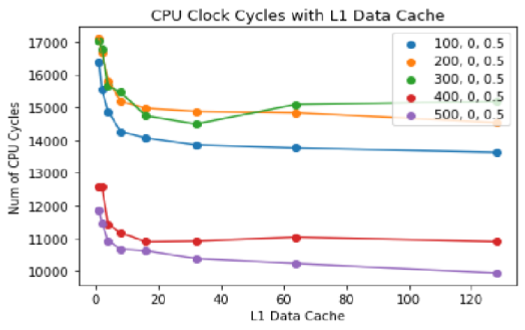


This image shows the relationship between parameter λ2 with CPU Clock Cycles. We notice that the clock cycles increase as the L1 increases, but it reaches an up limit at around λ2 = 0.6.

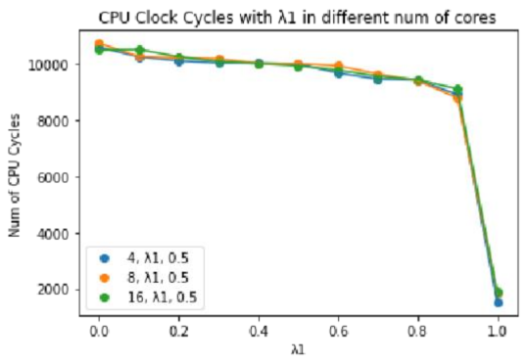
Moreover, we choose some sets of LLVM clusters and explore how the Layer1 Data Cache will influence the results.



This image is the relationship between L1 Data Cache with CPU Clock Cycles. As the size of the data cache increases, we only need less clock cycles, and it reaches a down limit when data cache is larger than 20KB.



We also tested other parameters. We change the value of N from 100 to 500 and test the clock cycles. The result of each group is very similar, which means that for our input parameter, 20KB of data cache is enough for this algorithm to reach the highest efficiency.



After milestone3, professor suggested us to limit the n value and try to adjust other parameters. Therefore, we changed the value of lamda1 from 0 to 1. Notice that the number of CPU cycles decrease slowly and drop tremendously when lamda1 = 1.

Ⅴ. Algorithms of Communities in Networkx

In addition to the studies on the Community Detection Algorithm. In the last stage of our project, we also tried other Communities Algorithms in the Networkx Library, such as Bipartition, K-Clique and Greedy Modularity Communities.

And we write the python script to transform the output of the algorithms into the LLVM instruction clusters that we can execute.

A. Bipartition

A graph is partitioned into two blocks using the Kernighan–Lin Algorithm. This algorithm partitions a network into two sets by iteratively swapping pairs of nodes to reduce the edge cut between the two sets. The pairs are chosen according to a modified form of Kernighan-Lin, which moves node individually, alternating between sides to keep the bisection balanced.

By applying bipartition, we separate the LLVM instructions into two clusters. After running the execution, the result is about 0.8million, reducing about 35% of execution time compared with the initial number of CPU cycles (table shown below).

The middle one is the bipartition created by our own by simply cutting the LLVM in half. And we want to compare it with the Kernighan-Lin Algorithm to see if it has better performance. As it turns out, the algorithm is indeed better, but the effect is not very significant, which is just 2% faster than our bipartition method.

Execution time	Number of CPU Cycles
Initial	1340505.0
Bipartition	862565.1138195811
kernighan_lin_bisection	840307.354369666

B. K-Clique

Second, we try to find k-clique communities in graph by using the percolation method, where a k-clique community is the union of all cliques of size k that can be reached through adjacent (sharing k-1 nodes) k-cliques. It requires one more parameter k, which means in every cluster, the number of nodes should be no less than k. The interesting thing is that we actually tried the bigger k, but the K-Clique cannot successfully partition the clusters starting from 3. Therefore, the only feasible partition is 16(when k=2).

From the table shown below, we can see that the result of k-clique is significant, which is about 90thousand, reducing about 93% execution time. And similarly, we did our 16-partition by equally dividing the LLVM into 16 parts. It found out that K-Clique is also doing better performance.

Execution time	Number of CPU Cycles
Initial	1340505.0
16-partition	98989.948127693364
k_clique_communities	91424.8186781547

C. Modularity-based Communities

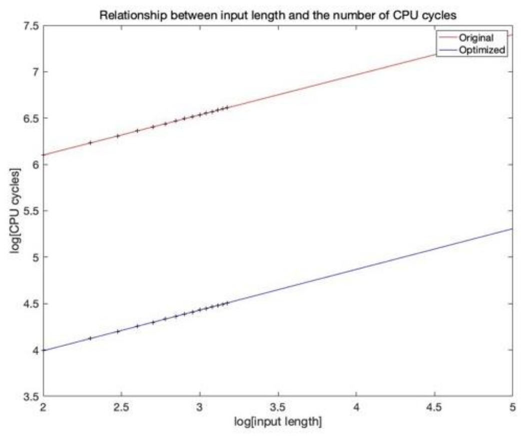
The last one is greedy modularity communities, which we used to calculate parallelization degree on the previous stage, but this time we get the concrete execution result by simulation.

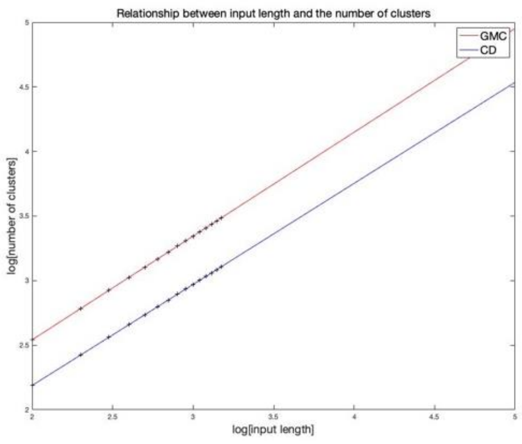
This function uses Clauset-Newman-Moore greedy modularity maximization to find the community partition with the largest modularity. Greedy modularity maximization begins with each node in its own community and repeatedly joins the pair of communities that lead to the largest modularity until no further increase in modularity is possible (a maximum).

The greedy modularity communities algorithm partitioned the instructions into 126 clusters. And it turns out that it has the best execution performance, which is about only 30thousand, reducing about 97% execution time (table shown below).

Execution time	Number of CPU Cycles
Initial	1340505.0
greedy_modularity_communities	29766.8110778848

Ⅵ. Performance prediction





Based on the professor's advice, we draw these two graphs to analyze and predict the performance of the algorithm.

For the first graph, we take the logarithm of the input length and the number of clusters, then do linear fitting with least square method to generate slope and intercept. Notice that community detection algorithm is significantly better than greedy modularity communities algorithm.

For the second graph we do the same operation to the input length and the CPU cycles. The red line represents the CPU cycles without any optimizations. This figure shows the performance improvement brought by the optimization algorithm.

Ⅶ. Conclusion and further research plan

To sum up, our project's goal is the optimization of the Cryptographic Hash Function, and we did achieve the objective.

Through all the four milestones of this

project, what we did is we implemented MD5 and SHA-256 Algorithm in C language, used script to translate them into LLVM instructions, had a preliminary statistics analysis based on Gephi Graph. We used Community Detection Algorithm to optimize the parallelization and explored it further by changing three parameters(N, λ1, λ2) to see how they influence the results. We simulated the execution of the previous results on Gem5. In addition, we do the similar experiments on other Communities Algorithms in Networkx library.

In the future, if we can do further research, we can explore more communities Algorithms to achieve even better optimization. On the other hand, in the hardware level, we can use Verilog or VHDL to design System-on-Chip architecture for the application.

REFERENCES

[1] Y. Xiao, Y. Xue, S. Nazarian, P. Bogdan. "A Load Balancing Inspired Optimization Framework for Exascale Multicore Systems: A Complex Networks Approach". In: ICCAD. 2017.

[2] Abd Elminaam, D. S., Abdual-Kader, H. M., & Hadhoud, M. M. (2010). Evaluating The Performance of Symmetric Encryption Algorithms. Int. J. Netw. Secur., 10(3), 216-222.

[3] Gustavus J. Simmons. 1979. Symmetric and Asymmetric Encryption. ACM Comput. Surv. 11, 4 (Dec. 1979), 305 – 330.

[4] Ciampa, Mark (2009). CompTIA Security+ 2008 in depth. Australia; United States: Course Technology/Cengage Learning. p. 290. ISBN 978-1-59863-913-1.

[5] "Secure Hash Algorithms" searched from Wikipedia https://en.wikipedia.org/wiki/Secure_Hash_Algorithms

[6] Ye Tian, Kun Zhang, Pu Wang, Yuming Zhang, Jun Yang, Add "Salt" MD5 Algorithm's FPGA Implementation, Procedia Computer Science, Volume 131, 2018, Pages 255-260, ISSN 1877-0509.

[7] T. N. Dat, K. Iwai, T. Matsubara and T. Kurokawa, "Implementation of high speed rainbow table generation using Keccak hashing algorithm on GPU," 2019 6th NAFOSTED Conference on Information and Computer Science (NICS), 2019, pp. 166-171.

[8] Hachem Bensalem, Yves Blaquiere, & Yvon Savaria (2021). Acceleration of the Secure Hash Algorithm-256 (SHA-256) on an FPGA-CPU Cluster Using OpenCL international symposium on circuits and systems.

[9] Thi Hong Tran, Hoai Luan Pham, & Yasuhiko Nakashima (2021). A High-Performance Multimem SHA-256 Accelerator for Society 5.0 IEEE Access.

[10] S. Oukili and S. Bri, "High speed efficient advanced encryption standard implementation," 2017 International Symposium on Networks, Computers and Communications (ISNCC), 2017, pp. 1-4, doi: 10.1109/ISNCC.2017.807197.