**Execution Profile of Post-Quantum Cryptography Algorithms**

**Abstract:**

There are many candidate algorithms being evaluated for the standardization of the next-gen post-quantum cryptography. Currently, there are several candidate algorithm being presented under round 3. In this project, we will review a few selected algorithm and conduct performance profiling and report our findings. Based on these findings, we will come up with recommendations for parallel implementation on multicore CPU or GPU.

**I. Introduction:**

Quantum computers are used for solving complex and difficult mathematical problems compared to that of conventional computers. If the production of quantum computers can be done in bulk, all the communications in the world pose a threat of being compromised. This is because quantum computers can break many frequently used public-key cryptosystems. Thus, confidentiality and integrity of every digital communication is at stake.

This situation asks for immediate development of post-quantum cryptography or quantum-resistant cryptography. These algorithms must be compatible with the current communication protocols and be resistant to quantum and classical computers at the same time.

NIST finalized the round 3 candidates on 22 July, 2020 (list of candidates is available at [2]). We chose FrodoKEM [3] which is an alternate

candidate for Public-key Encryption (PKE) and Key-Establishment Algorithm.

As stated in [3], FrodoKEM is a family of Key-encapsulation Mechanism (KEM) whose security is derived from learning with errors problem [4]. Learning with errors is closely related to the algebraically unstructured lattices.

FrodoKEM’s implementation is available at [5] which supports Windows, Linux and macOS devices. We chose FrodoKEM-640 which targets Level-1 in the NIST call for proposals. This algorithms has equivalent or more brute-force security of AES-128.

**II. Parallelism**

As stated on [8], many calculations or processes are carried out simultaneously in parallelism. There are several types of parallelism:

1. Bit-Level Parallelism: here the reason for speed-up is the increase in computer word-size. Computer word-size is the amount of information that can be manipulated by the processor per cycle. For example, adding two 32-bit integers with a 16-bit processor takes 2 instructions. The instructions are addition of 16 lower bits using standard addition instruction, and addition of 16 higher bits using add-with-carry instruction. But, if we had a 32-bit processor, this operation could have been done in one instruction.
2. Instruction-Level Parallelism: here, instructions are re-ordered and combined into groups. These groups are then executed in parallel without deviating from the expected result of the program.
3. Task Parallelism: here, different calculations are performed on same or different sets of data. Tasks are decomposed into sub-tasks which are then allocated to processor for execution.
4. Superword Level Parallelism: it is based on loop unrolling and basic block vectorization.

In Data-level parallelism [9] involves single stream of instructions being operated concurrently on different data. It is limited only by non-regular data manipulation methods and memory bandwidth.

**III. Threading**

For the task of multi-threading, we have used OpenMP framework from the given list shown in Figure 1.

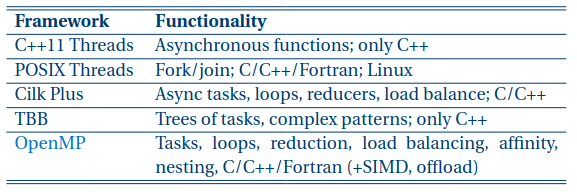


Figure 1 : Threading Frameworks[9]

**IV. Single Instruction Multiple Data (SIMD)**

Vector instructions are one of the implementations of SIMD. There are two approaches for using vector instructions:

1. Automatic Vectorization and
2. Explicit Vectorization.

Though Automatic Vectorization of Loops sounds great, it has its own fair share of limitations. There are 4 limitations:

1. The number of iterations of the loop must be defined beforehand. That is, the number of iterations cannot be changed when the loop is being executed.
2. There should be no vector dependence in the loop. True vector dependence like shown in Figure 2a cannot be vectorized. This is because the value of a[i] depends on the value of a[i-1] (previous element). Figure 2b shows a case where the value of a[i] depends on a[i+1] (not previous element), in this case vectorization is possible. Figure 2c shows another case where vector dependence is present, but it may be safe to vectorize as a[i] is dependent on a[i-16]. This condition has no vector dependence if the vector length used for vectorization is less than or equal to 16.
3. The function which contains the loop must be SIMD enabled to allow Automatic Vectorization.
4. If innermost loops (loops inside loops) needs vectorization, we have to override using #pragma omp simd directive.

Functions themselves can be made SIMD-Enabled Functions by using the #pragma omp declare simd directive.

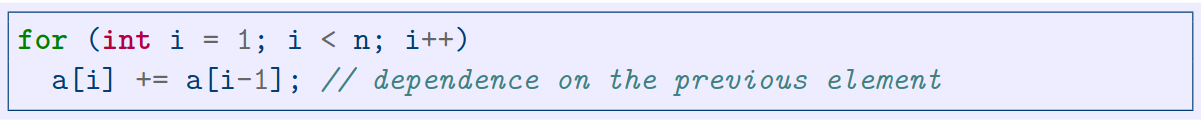


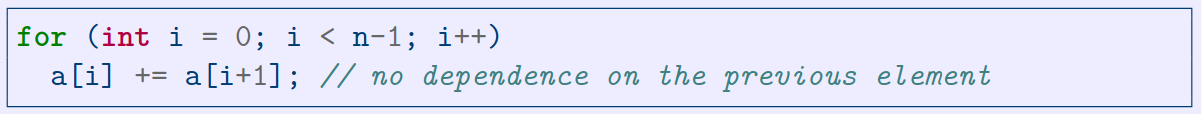
Figure a : Vectorization impossible[6] 

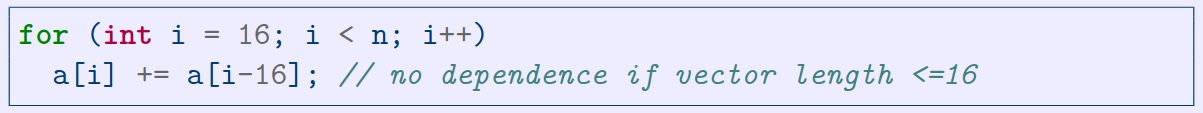
Figure 2b : Vectorization possible[6] 

Figure 2c : Vectorization may be possible[6]

**V. FrodoKEM-640**

We have tried optimizing the Reference Implementation of FrodoKEM-640 which can be found in the folders extracted from the zip file provided on [2].

We used GNU gprof for profiling the executable named test\_KEM which is used for running the tests. Using gprof, we determined which functions were taking the most execution time. These functions were our target for optimization. Figure 3 shows the generated output when test\_KEM was executed. It also contains the gprof flat profile generated with -p (for printing the flat profile) and -b (for not printing the verbose blurbs) flags.

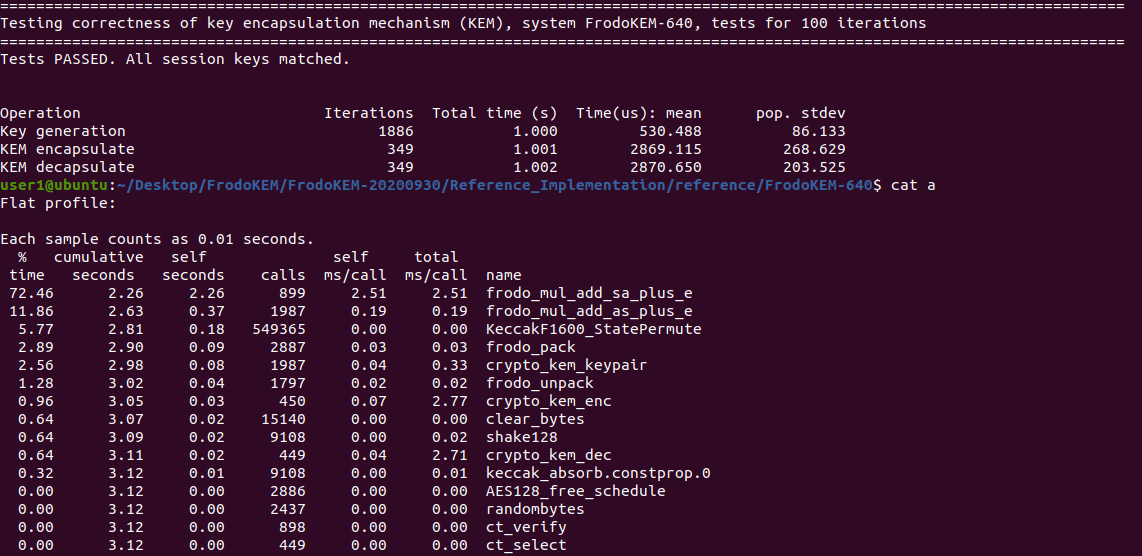


Figure 3 : test\_KEM output and gprof flat profile

An alternate profiler named Intel Advisor was used as an analysis tool for achieving high application performance. Intel compiler (icc used here) auto-vectorizes potential loops and functions for better performance.

Figure 4 shows initial flat profile of test\_KEM containing 6 loops which were auto-vectorized.

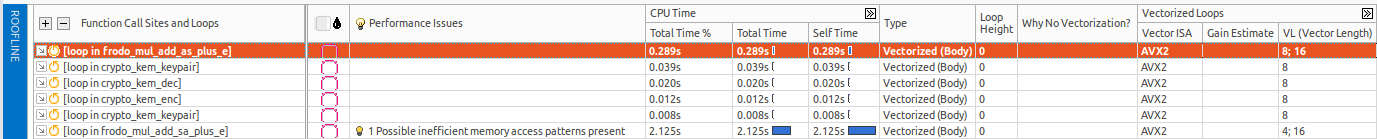


Figure 4 : Intel Advisor summary and roofline for vectorized loops

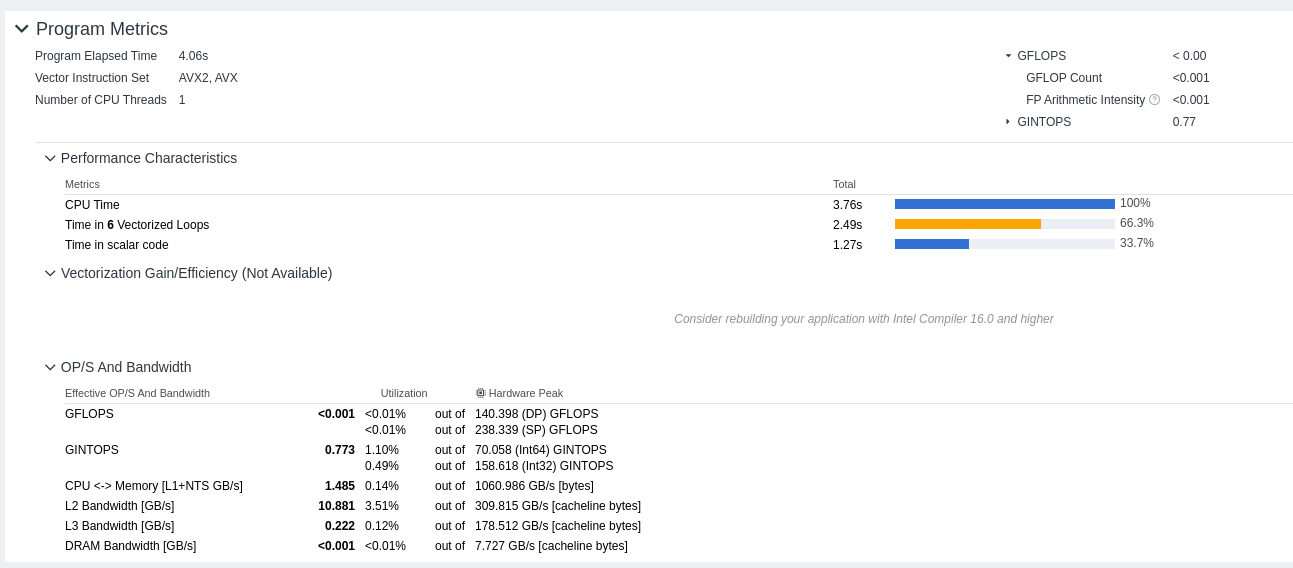


Figure 5 : Intel Advisor summary

Figure 5 shows the Summary and Roofline report produced by Intel Advisor. In Program Metrics section, we can see that total elapsed time is 4.06s with 1 CPU thread. AVX2 and AVX are the Instruction Sets being used. Advanced Vector Extensions is used for Single Instruction Multiple Data (SIMD) operations.

Following sections present Vectorization updates made to the original program and their gain speedups.

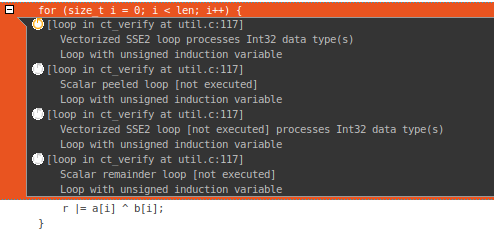
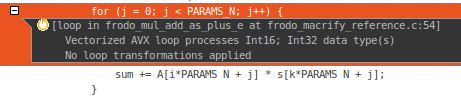
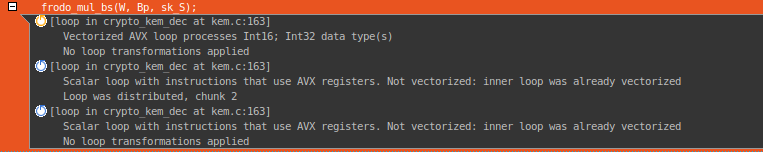
Section 1: Figure 6 shows function named frodo\_mul\_bs which is, in simple terms matrix multiplication. Vector ISA used is AVX, Efficiency is 100% and Gain Estimate is 10.94x. Vector Length (VL) for this function is 8 and has 80 Average Trip Counts.

Figure 6 : Section 1

Figure 8 : Section 3

Figure 7 : Section 2

This loop is present in crypto\_kem\_dec function of kem.c program.

Section 2: Figure 7 shows a for loop. This loops is vectorized with AVX Vector ISA, and has 81% Efficiency. This loop has Gain Estimate of 6.45x and Vector Length 8. It is very easy to understand that why this loop was vectorized, reason being it satisfies all the conditions presented in IV. Most importantly, it has no vector dependence. It has 80 Average Trip Counts.

This loop is present in frodo\_mul\_add\_plus\_e function of frodo\_macrify\_reference.c.

Section 3: Figure 8 shows a for loop which is vectorized with SSE2 Vector ISA. It has 324 Average Trip Counts and Estimated Gain of 5.93x. Estimated Efficiency of this loop is 74% with Vector Length being 8.

This reason for this loop being vectorized was #pragma omp declare simd declared above the function ct\_verify, which contains this for loop.

Section 4: This section involves same function being called within 2 different functions of the same program with 80 Average Trip Counts.

Function frodo\_mul\_add\_sa\_plus\_e has a loop within it which is being vectorized in crypto\_kem\_enc and crypto\_kem\_dec functions. This function call is present in kem.c and is vectorized with 47% Estimated Efficiency at both the sites. Vector Length is 8 and Gain Estimate is 3.80x.

Section 5: Figure 9 shows a nested for loop in function named frodo\_sample\_n. This function is called three times in different functions of different programs. All of the three functions had 320 Average Trip Counts.

1. The first site has function declaration in noise.c. AVX Vector ISA is used with 23% Estimated Efficiency and 1.84x Gain Estimate. The for loop was unrolled and jammed by 2.

Loop unrolling is the optimization involving loops to reduce their frequency of branches and loop maintenance instructions.

Loop jamming reduces the time taken to compile many number loops. Here, more than one loop is combined into a single loop.

Now, the whole process of unrolling and jamming is a loop transformation technique, where the size of the inner loop body is increased by unrolling outer loops multiple times.

1. Same results are for the function call in kem.c, but has 22% Estimated Efficiency and 1.73x Gain Estimate.
2. Another function call in kem.c has same results, but has 21% Estimated Efficiency and 1.66x Gain Estimate.

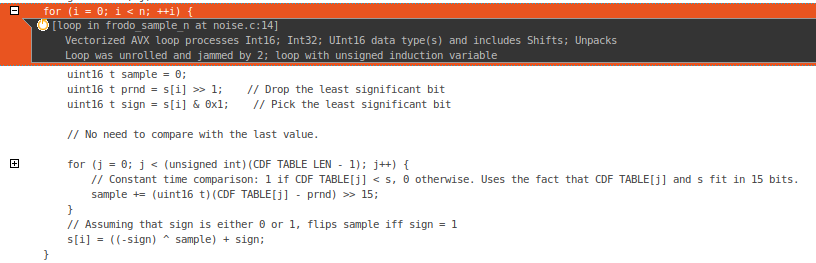
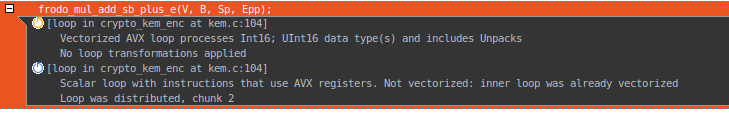


Figure 9 : Section 5



Section 6: Figure 10 shows a function named frodo\_mul\_add\_sb\_plus\_ewhich is present in crypto\_kem function of kem.c.

Figure 10 : Section 6

The Gain Estimate for its optimization is just 1.11x which is very insignificant. AVX Vector ISA is used which gives 14% Estimated Efficiency and has Vector Length as 8.

Performance Issues section states underutilization of FMA Instructions. It suggests the use of AVX2 which supports Fused Multiply-Add (FMA) for improving performance.

Section 7: Figure 11 shows a section where the performance is being decreased.

The loop is mostly compute bound and Unpacks and Shifts instructions might degrade the performance. AVX Vector ISA is used which gives 7% Estimated Efficiency and 0.57x Gain Estimate. Since the gain is less than one, this implies the loop will become slower.

Vector Length was 8 and had just 12 Average Trip Counts.

Now let’s see Threading updates.

Section 8: Figure 12 shows Threading with OpenMP. Using #pragma omp parallel for num\_threads(2), we were using 2 threads for parallelizing the for loop.

It was expected to have benefitted the execution time, but to our surprise, this update increased the cumulative seconds in Flat Profile. The overhead of managing the threads during and after execution is high.

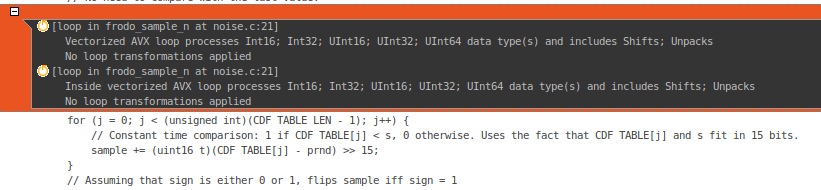


Figure 11 : Section 7

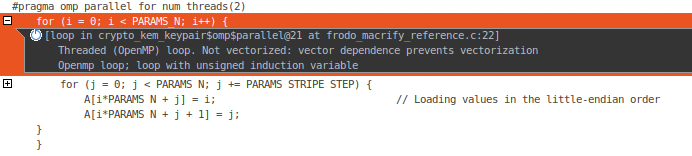
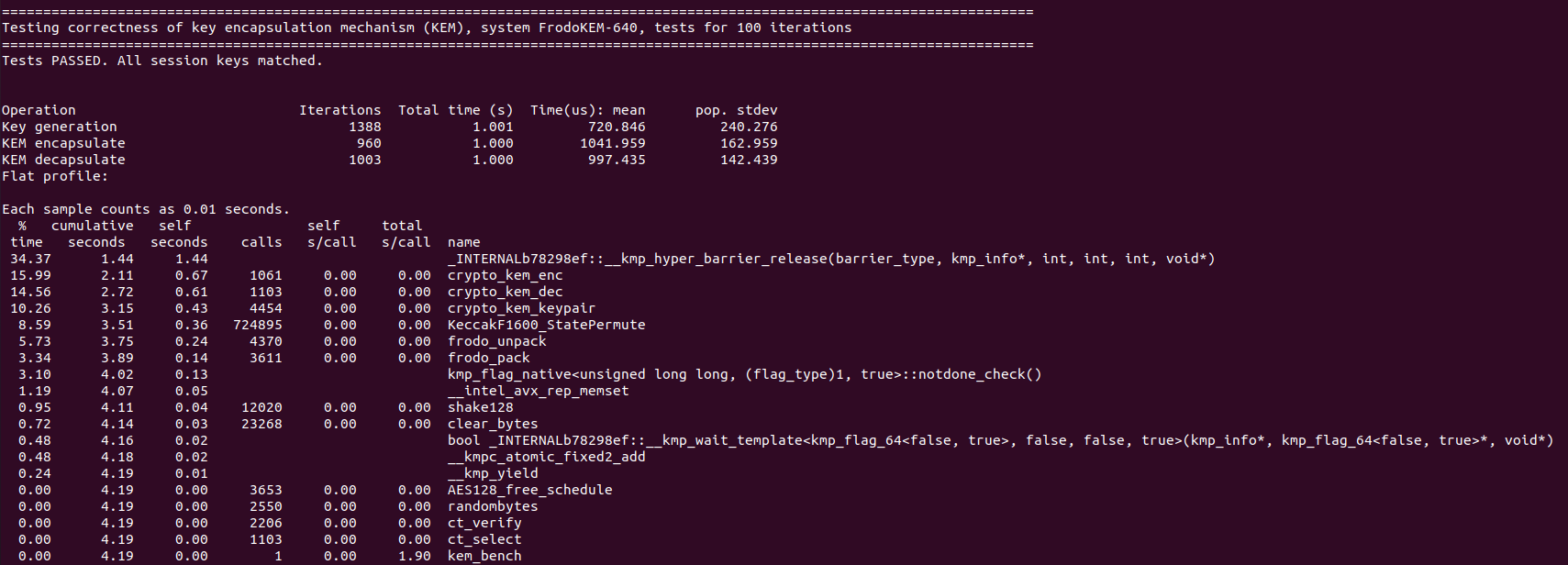


Figure 12 : Section 8



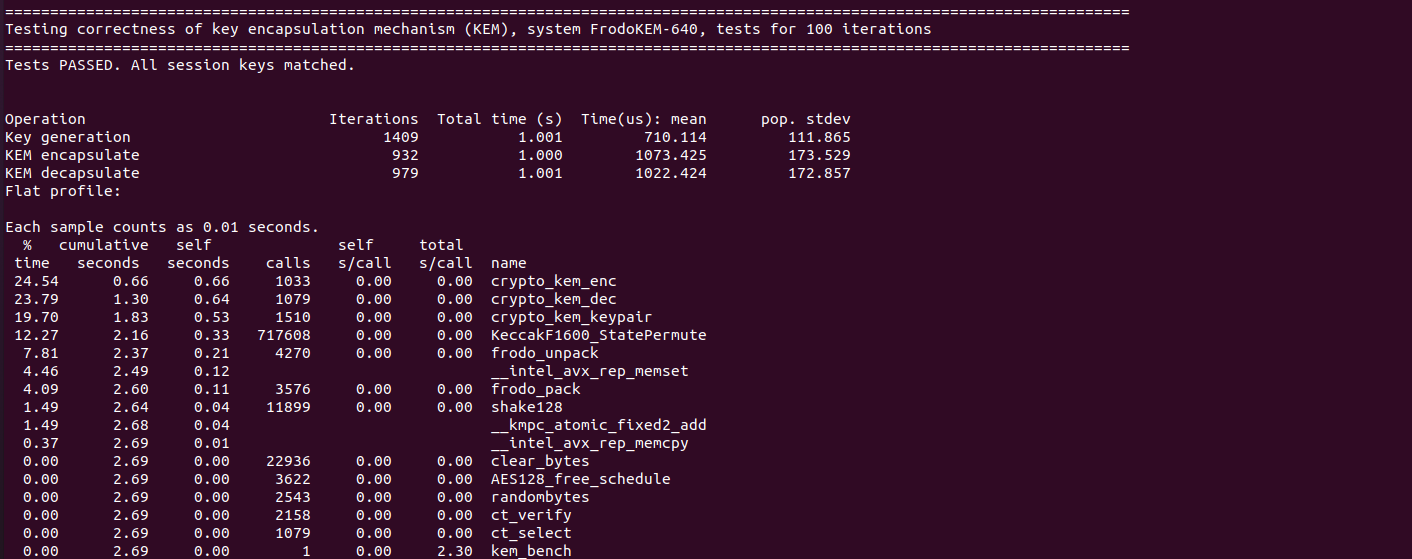
Figure 1a : with threading

Figure 13b : without threading

The changes can be observed by the gprof Flat Profiles as show in Figure 13a and Figure 13b.

The gprof Flat Profile of Optimized Implementation of FrodoKEM-640 is present in Figure 14.

Though the number of iterations are higher, this executable is highly optimized as the cumulative seconds is much smaller than ours. Also, the total time column is comparatively better than our test\_KEM output.

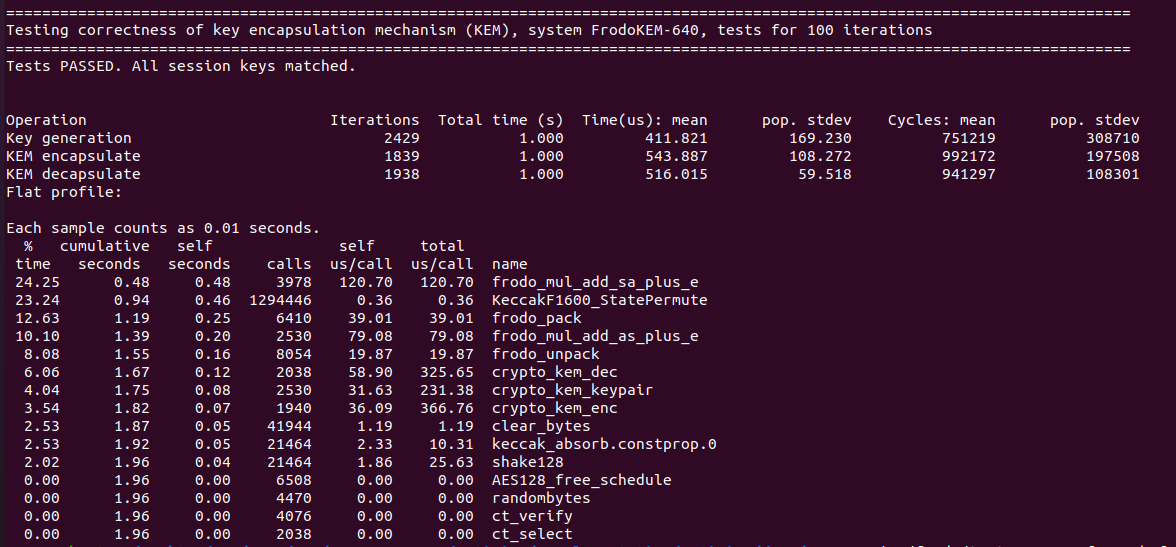


Figure 1 : Optimized Implementation of FrodoKEM-640, test\_KEM output and Flat Profile

All the above results were produced when the executable was running on host machine with 4 logical CPUs and Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz processor.

Coursera course at [10] provided the access to Colfax cluster [11]. This cluster had Intel Xeon Phi processor in the cluster. Intel Xeon Phi was not accessible from the login node.

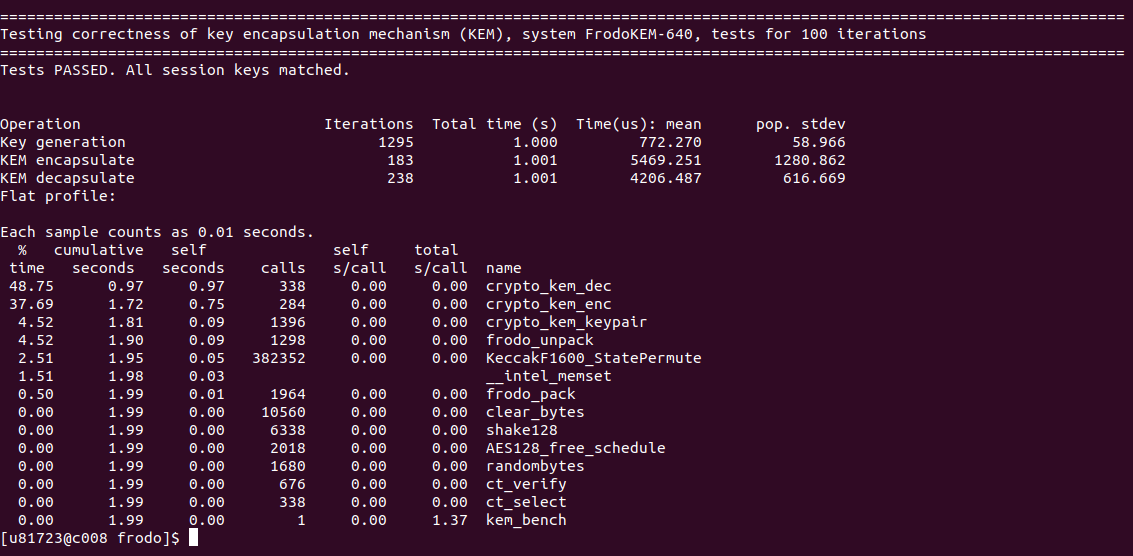


Figure 1 : Flat Profile when test\_KEM was ran on login node

Figure 15 shows the Flat Profile from gprof when the test\_KEM executable was ran on login node of Colfax Cluster.

Also, Figure 16 show the final summary and roofline window of the code optimized by us. Comparing this with Figure 5, we can see that the Number of CPU Threads are increased to 2. GINTOPS is increased from 0.773 to 1.185.Also, CPU <-> Memory [L1+NTS GB/s] is increased from 1.485 to 1.779.

It’s clear that Optimized Implementation of FrodoKEM-640 was better than our version of Optimization of Reference Implantation.

**VI. Conclusion and Future Work**

FrodoKEM-640 is an alternate candidate for PKE and KEA as defined by the NIST in Round-3 submissions of Post Quantum Cryptography.

In the above document, we analysed the sections of Reference Implementation of FrodoKEM-640 which were responsible for majority of time-taken in the executable.

Profiling was done using GNU gprof and in later stages Intel Advisor was used for better understanding. Vectorization and Parallelizing the code were our major updates which were responsible for the improved performance.

Most of the sections were sequential and had vector dependencies which the loops and functions from being vectorized.

Better memory management, using clusters and using higher ISA can further improve the performance.

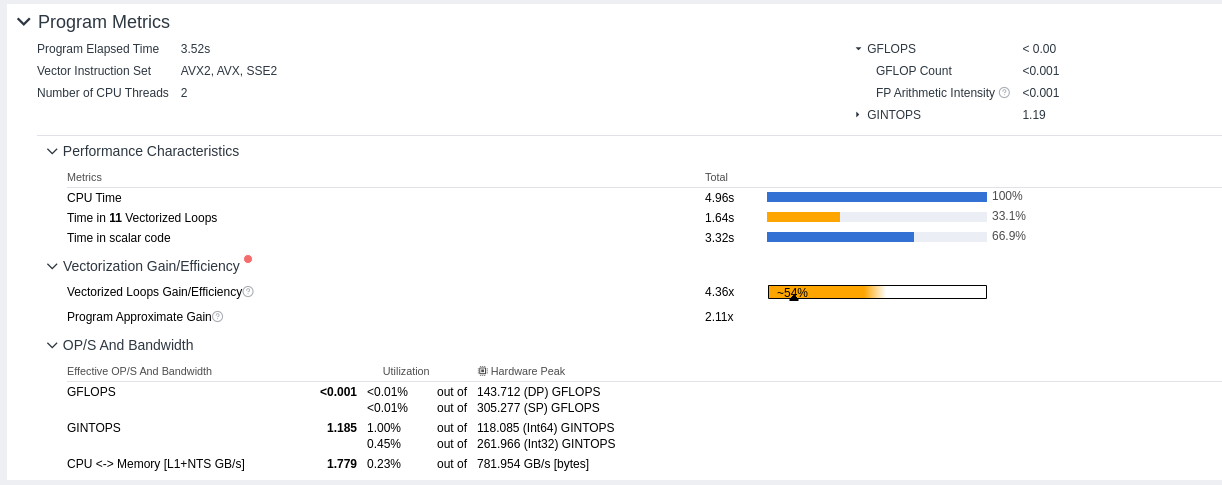


Figure 16 : Final Summary and Roofline output

**References:**

[1] <https://csrc.nist.gov/projects/post-quantum-cryptography>

[2] <https://csrc.nist.gov/Projects/post-quantum-cryptography/round-3-submissions>

[3] <https://frodokem.org/>

[4] <https://en.wikipedia.org/wiki/Learning_with_errors>

[5] <https://github.com/Microsoft/PQCrypto-LWEKE>

[6]<https://d3c33hcgiwev3.cloudfront.net/_1ac0ccc2c140116eae53c5ac789ddb30_Fundamentals_of_Parallelism_on_IA_02.pdf?Expires=1628467200&Signature=blt36HURJdL5vg5gbHPePDrFsKeesWe7iXmAaEuSrVRWIxA9Ouyfk8c3-l4Tki-fF33Slaq0wjp8Ph2TxJg1vKUOgSla23Mexn-0k4PE-rC5crzlaLgwDA~VIQ1WUh4eGvEczNySzuURJLVl2BsOi5lK2Qby2QFyZogDld7DAVY_&Key-Pair-Id=APKAJLTNE6QMUY6HBC5A>

[7] <https://en.wikipedia.org/wiki/Parallel_computing>

[8] <https://www.geeksforgeeks.org/introduction-to-parallel-computing/>

[9]<https://d3c33hcgiwev3.cloudfront.net/_1ac0ccc2c140116eae53c5ac789ddb30_Fundamentals_of_Parallelism_on_IA_03.pdf?Expires=1628553600&Signature=OI503dgwWk9F67GA~rdhPP01wBjnzWhpT5sVreW2CSxL2tdVbt8Or53zgeyjfhz2YdugJFRzyTha8nP33q2wYGonDZQG8D1x5m2UOeTjSTwJn5GroULcgbaCEPcwkIoyXTvgZ94i7vGLGyVappaBXgNjbgPou64j5IbaVpZYbQo_&Key-Pair-Id=APKAJLTNE6QMUY6HBC5A>

[10] <https://www.coursera.org/learn/parallelism-ia>

[11] <https://see.colfaxresearch.com/whale/>