Optimization & Boosting Performance of Running Complicated Codes like **AI** & **Cryptographic** on **CPU** (± Intel oneAPI) & **GP-GPU**

A Topic in Parallel Processing

By: Pouya Ebadollahyvahed

1- The Goal: Writing Fastest Possible Executable Code

In the previous post (article) we had performance benchmarking on symmetric encryption (AES-GCM), hashing (SHA-256), asymmetric encryption and digital signature (ECC / ECDSA) operations according to DLMS / COSEM security suites 0, 1 and 2. The conclusion was that for IoT communication (like smart meter to AHE / MDC / Head-End) the most suitable cryptography algorithm (in general) can be AES-GCM.

In this article we want to optimize implementation of AES-GCM code (this also can be applied to any **mathematically complicated algorithm implementation** like **AI** and **Neural Network calculations**) to reach to the **FASTEST EXECUTION** through:

- Code & Compiling Optimization (also Benchmarking **C**/**C++**, **C#**, **Python** and **Java**)
- **Parallel Processing** through Multi-Threading
- Using **CPU Special Instructions** and Intel oneAPI Library
- Running code on **GPU**

2- Which Language Does Generate the Fastest Executable Code?

The first question needs to be answered here is that which computer language is the fastest one? In other words which language can produce the fastest executable (or intermediate) code?

To answer this question, a simple interpolation algorithm (with mix integer and floating-point calculations with no disk transactions)[1] has been implemented in several languages and then their execution time is being measured programmatically. Following is the result:

Chapter 8 of this article contains specification of the test platform (all under Windows & on CPU):

Language	Execution Time (ms)	Normalized Performance (%)
Python	309666	1
Java	7550	56
C# (Debug)	10474	40
C# (Release)	5392	78
C++ (Debug)	8269	51
C++ (Release)	4255	99
C++ (Release, Optimized for Speed)	4200	100

Note: the formula for calculating "Normalized Performance" is $= 100 * \frac{\text{Shortest Measured Time (the Best)}}{\text{Time Measured in This Test}}$

Since C++[2] proves the best performance, all subsequent tests and benchmarks are done based on **C++**.

[1] These benchmarking code snippets are originally developed in 2013 to test VEE module of MDM system but I used them again for making a benchmark for this article

[2] C++20 in this test

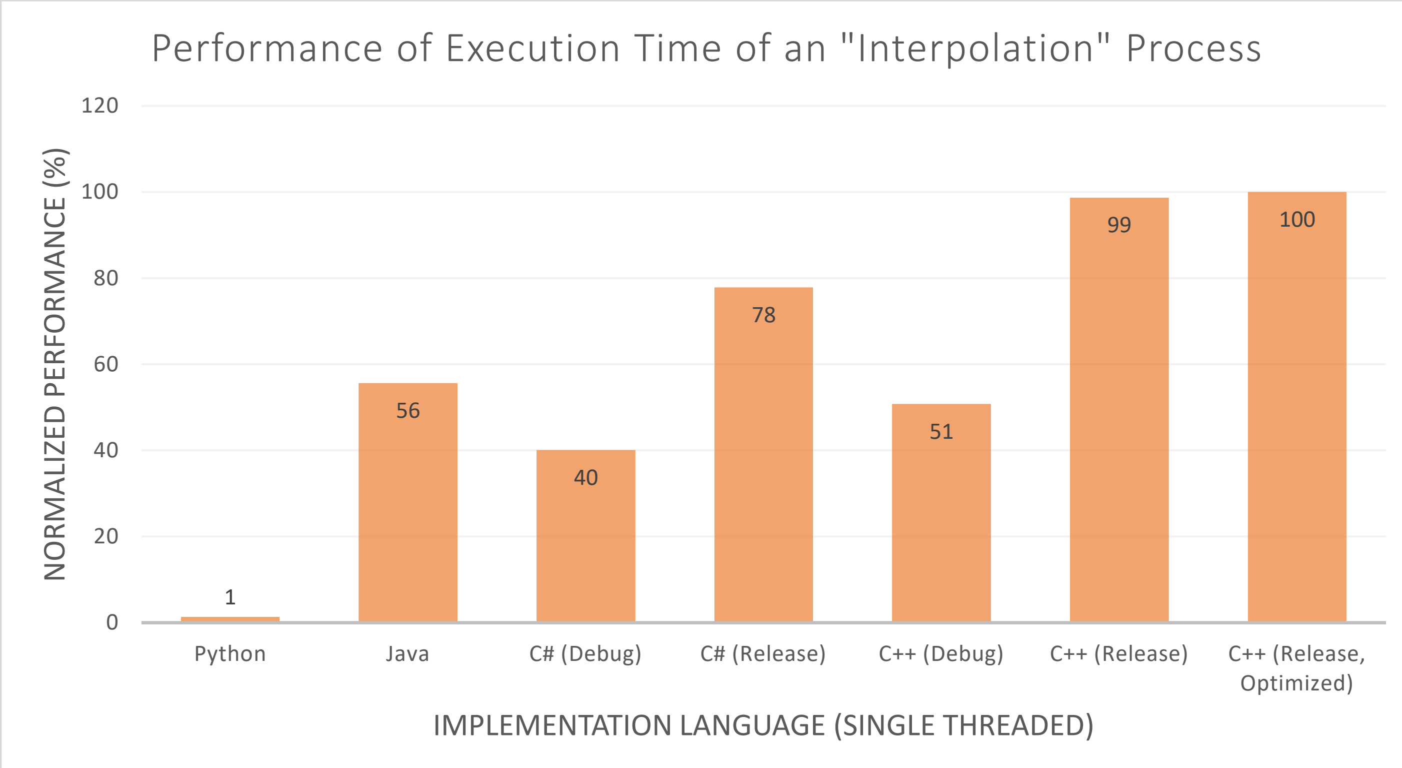

Figure 1 Performance Benchmarking of Languages, Running Code on 1 Core of CPU under Windows

The question comes next is that, is there any performance difference between running codes under Windows, Linux or other OSes and using different compilers?

Following table shows performance result of running a matrix calculation code under Windows and Linux.

Specification of the test platform can be found in chapter 8 of this article.

Threads Count	Time for Execution (s) Windows Release Optimized	Normalized Performance (%) Windows Release Optimized	Time for Execution (s) Linux Release Optimized	Normalized Performance (%) Linux Release Optimized	Time for Execution (s) Linux Release Not Optimized
1	**215**	7	**176**	8	**240**
10	24	58	25	56	
20	14	100	15	93	
30	15	93	15	93	

Although for Threads = 1, the performance difference between **gcc** (with & without **Ofast** switch) under Linux Debian and MSVC under Windows is considerable but in multi-threaded execution, no significant performance difference is observed, so all subsequent tests are done under Windows (Optimized).

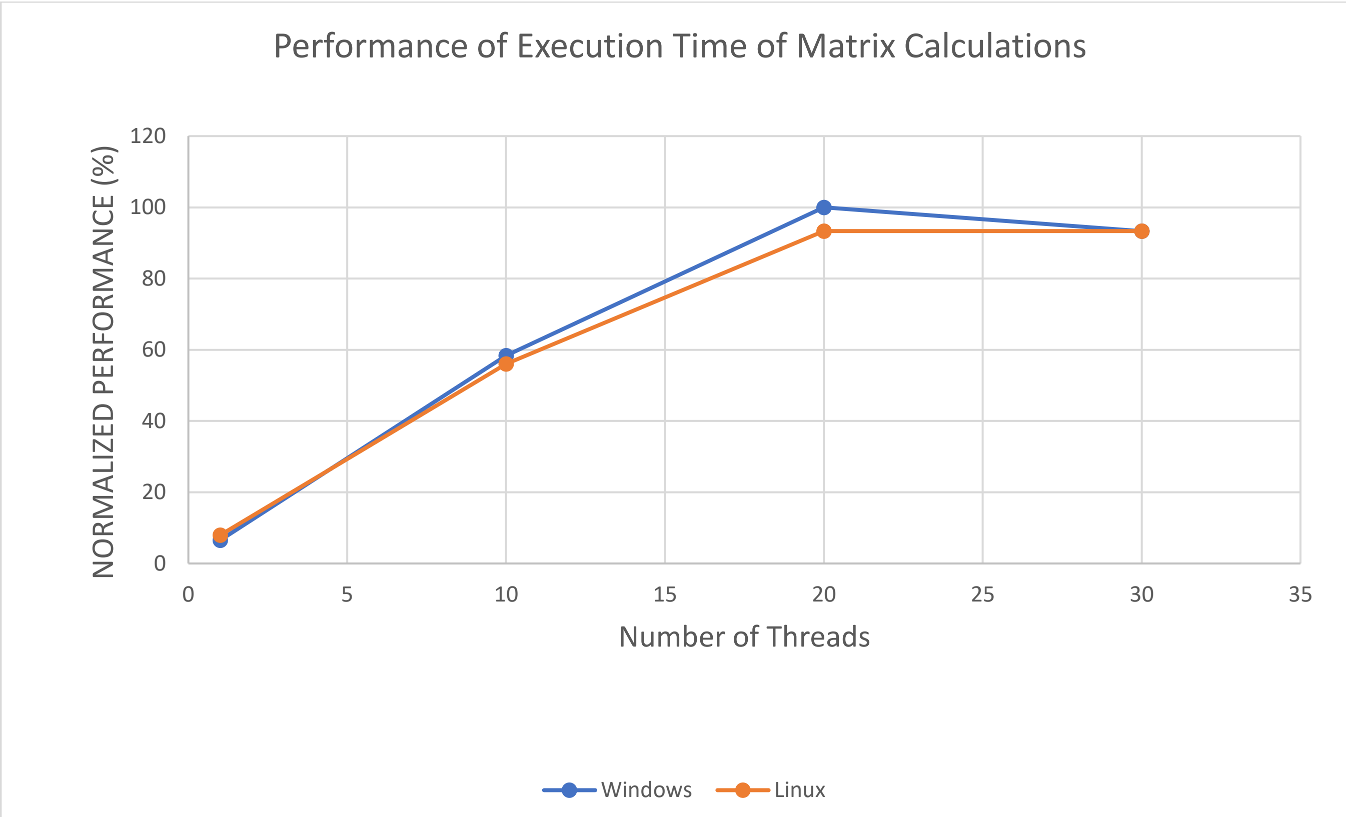

Figure 2 Performance Benchmarking of gcc (under Linux Debian) and MSVC (under Windows). CPU has 20 Actual Cores in Total

3- Benchmarking Data Types for Processing Time on CPU & GPU

The other factor which heavily impacts execution performance and sometimes programmers ignore it is variables' data types (like int, long, float and double) which are used in calculations. Sometimes this factor is not visible to programmers of certain high-level languages and they don't have any control over choosing type of data for variables.

Depends to processor architecture (CPU / GPU / NPU) and width of its data bus, execution time of mathematical operations on different types of data can be radically different; for example, as it is seen in the result of the following benchmarking tests, on Intel Core i7 CPU, calculations with 4-Byte and 8-Byte integer have almost the same performance, while 4-Byte float is almost 30% slower. Specification of the test platform can be found in chapter 8 of this article.

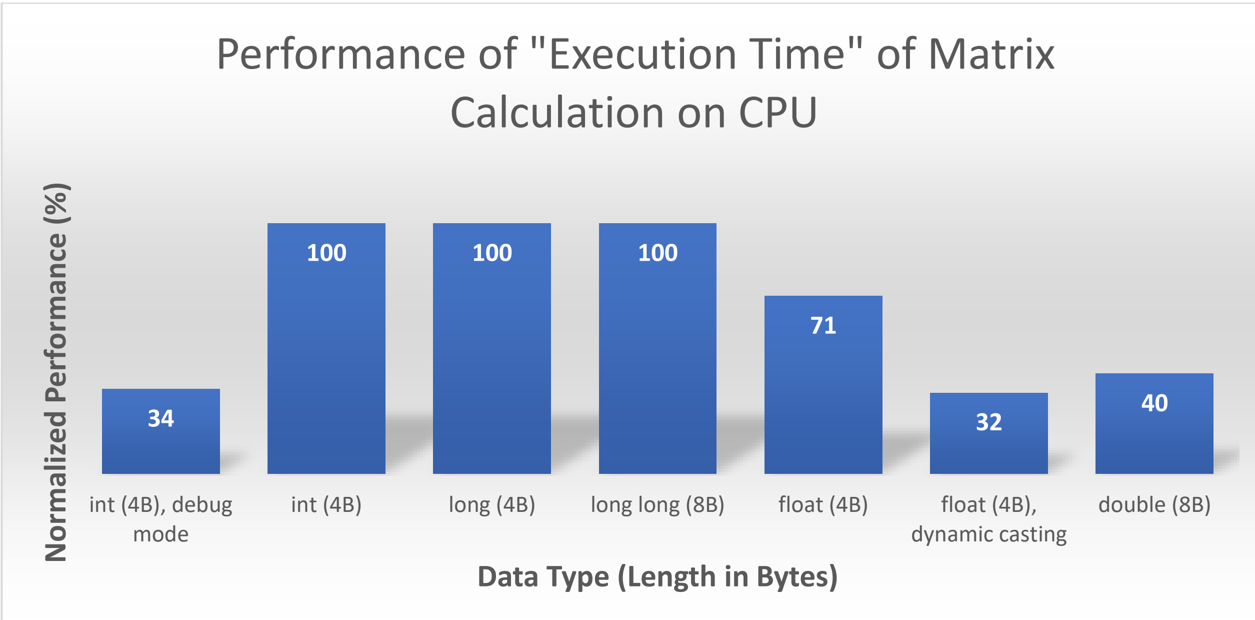

Figure 3 Performance Benchmarking of Data Types when Running on CPU (Single Threaded)

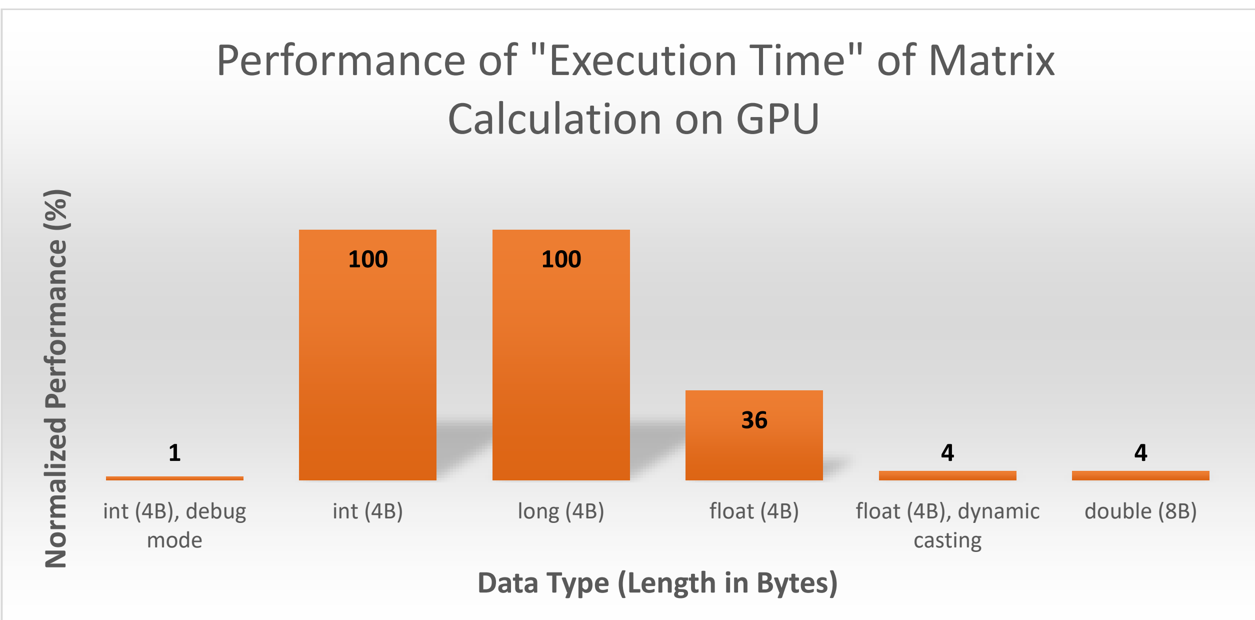

Figure 4 Performance Benchmarking of Data Types when Running on GPU (Single Threaded)

Note: dynamic casting is writing codes like: `float f { 1.3 };` so, in runtime the number 1.3 will be casted to float. While equivalent static casting for above line will be `float f { 1.3f };`

4- Benchmarking Parallel Processing on CPU (with and without oneAPI)

The classic and the straightest way to increase execution performance and to decrease the execution time is breaking a task to smaller sub-tasks and run them in parallel which is called **parallel processing**. **Amdahl's law** (or Amdahl's argument) shows us that by running tasks (or processes) in parallel, the execution performance can improve, but how much in reality? What is the cap?

In this chapter we will run identical tasks in threads (standard C++17 threads), to check how much we can improve execution performance? In other words, we want to check the linearity between "**increasing number of threads**" and "**increase in execution performance**" in result.

The other way to increase performance of some certain algorithms like AES-GCM (a method of symmetric encryption) or hashing is to use **Intel OneAPI** library (the IPP module). Since functions in this library use special instructions of CPUs and pretty much optimized for performance, by using these functions, overall performance also can be increased. Some of functions of Intel IPP library run algorithms directly on CPU by using their special instructions.

In the 2nd test in this chapter, AES-GCM operations are implemented using Intel IPP library and their execution result is compared to the previous test which was normal implementation of AES-GCM based on ordinary calculations.

Specification of the test platform can be found in chapter 8 of this article.

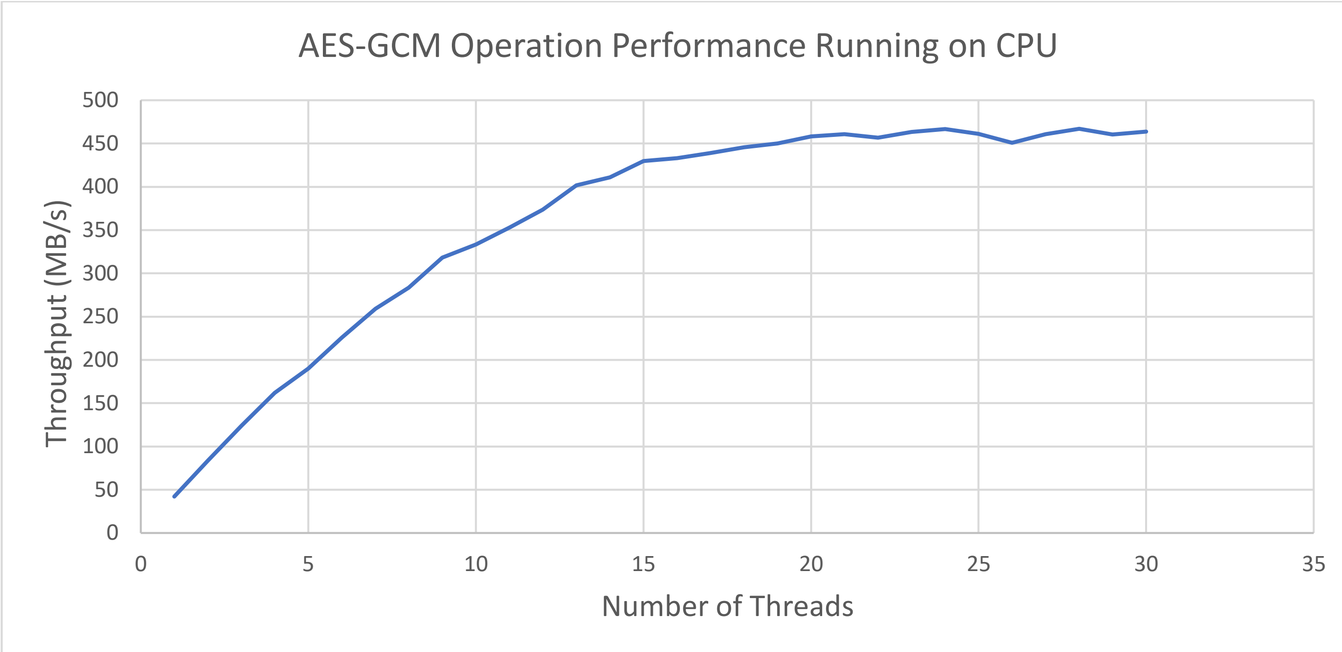

Figure 5 Performance Benchmarking. CPU has 20 Actual Cores in Total

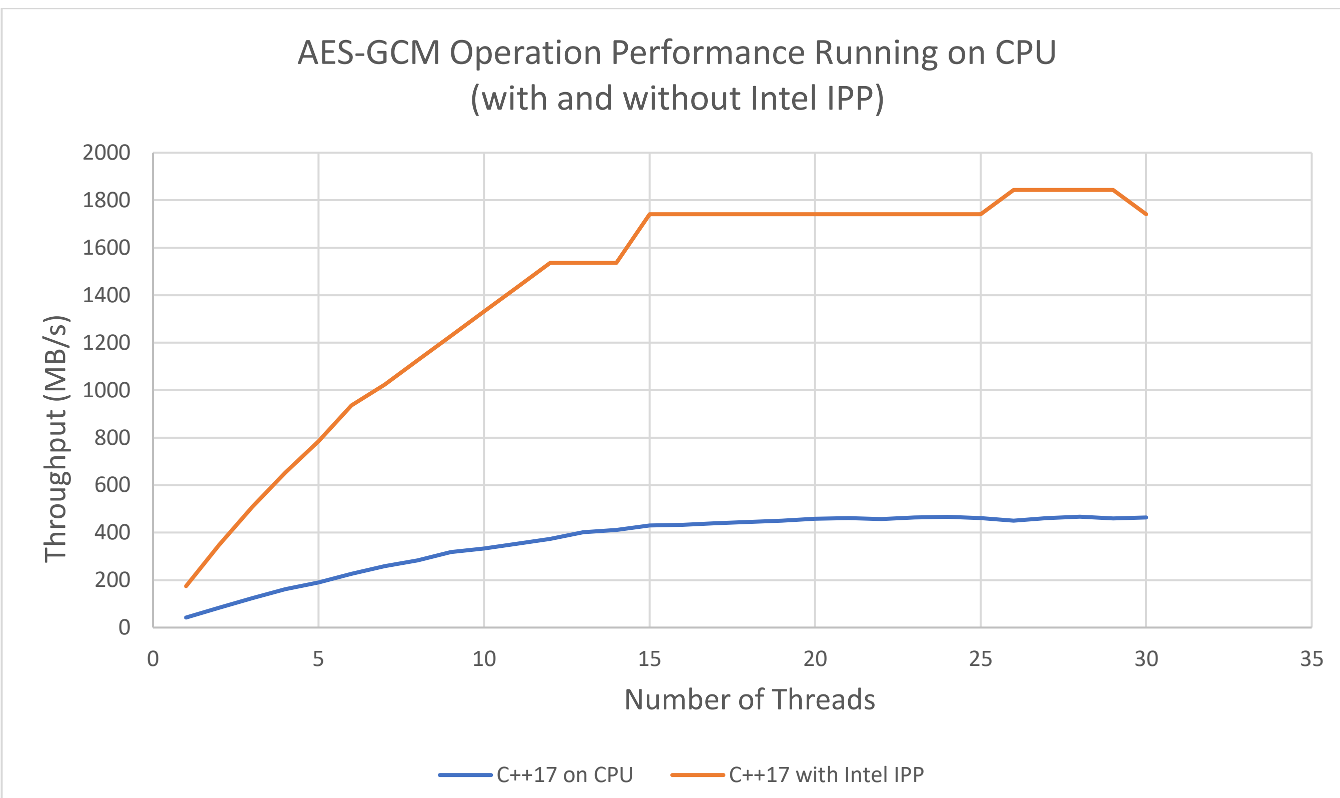

Figure 6 Performance Benchmarking (with and without IPP). CPU has (8 + 6 + 6) Actual Cores in Total

5- Benchmarking Parallel Processing on CPU and GPU

Since years ago, GPUs are getting more and more attention to run heavy calculations like for AI algorithms, Neural Network operations and Matrix calculations. The main reason is that GPUs have much more cores in comparison to CPUs which makes them more suitable for being used in Parallel Processing.

The following benchmark is running the AES-GCM test (the same as chapter 4) on GPU.

Specification of the test platform can be found in chapter 8 of this article.

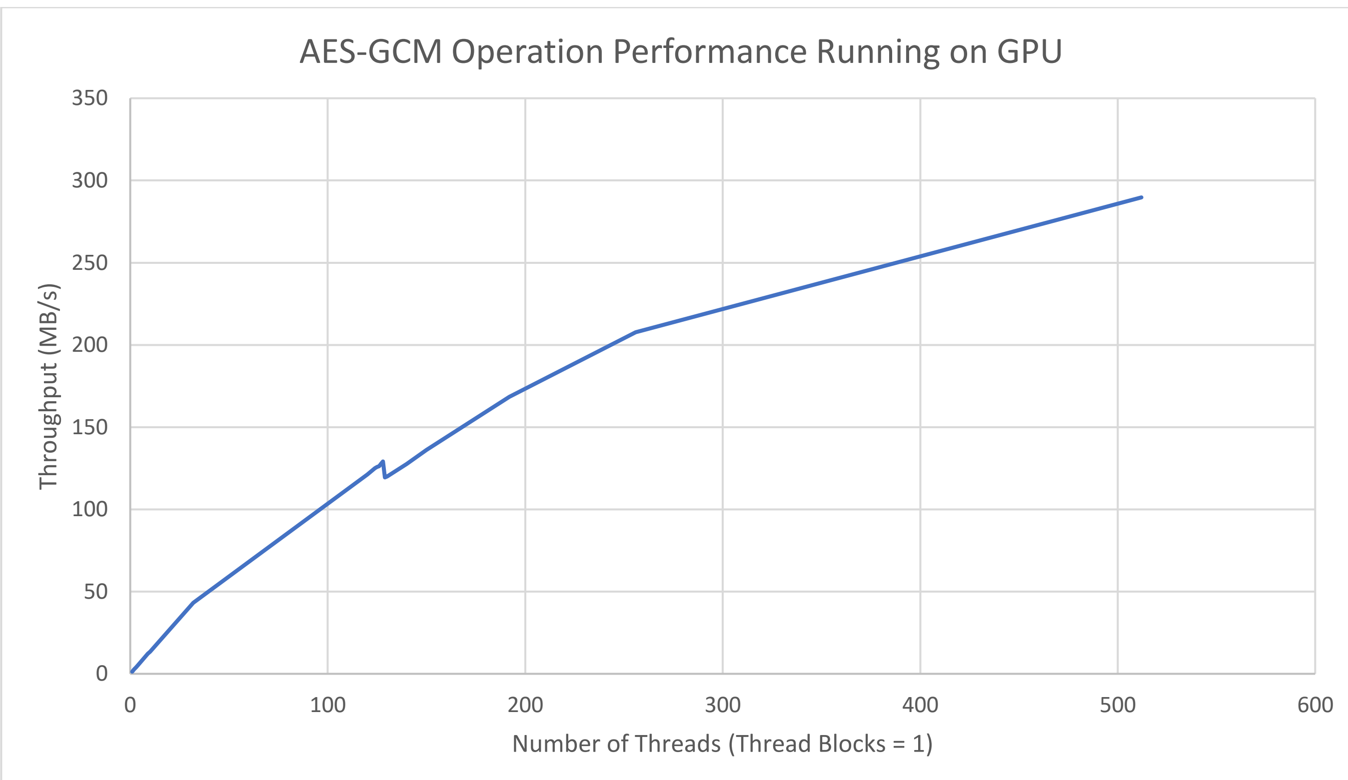

Figure 7 Performance Benchmarking. GPU has 128 Cores per SM. the break on the line is at number 128

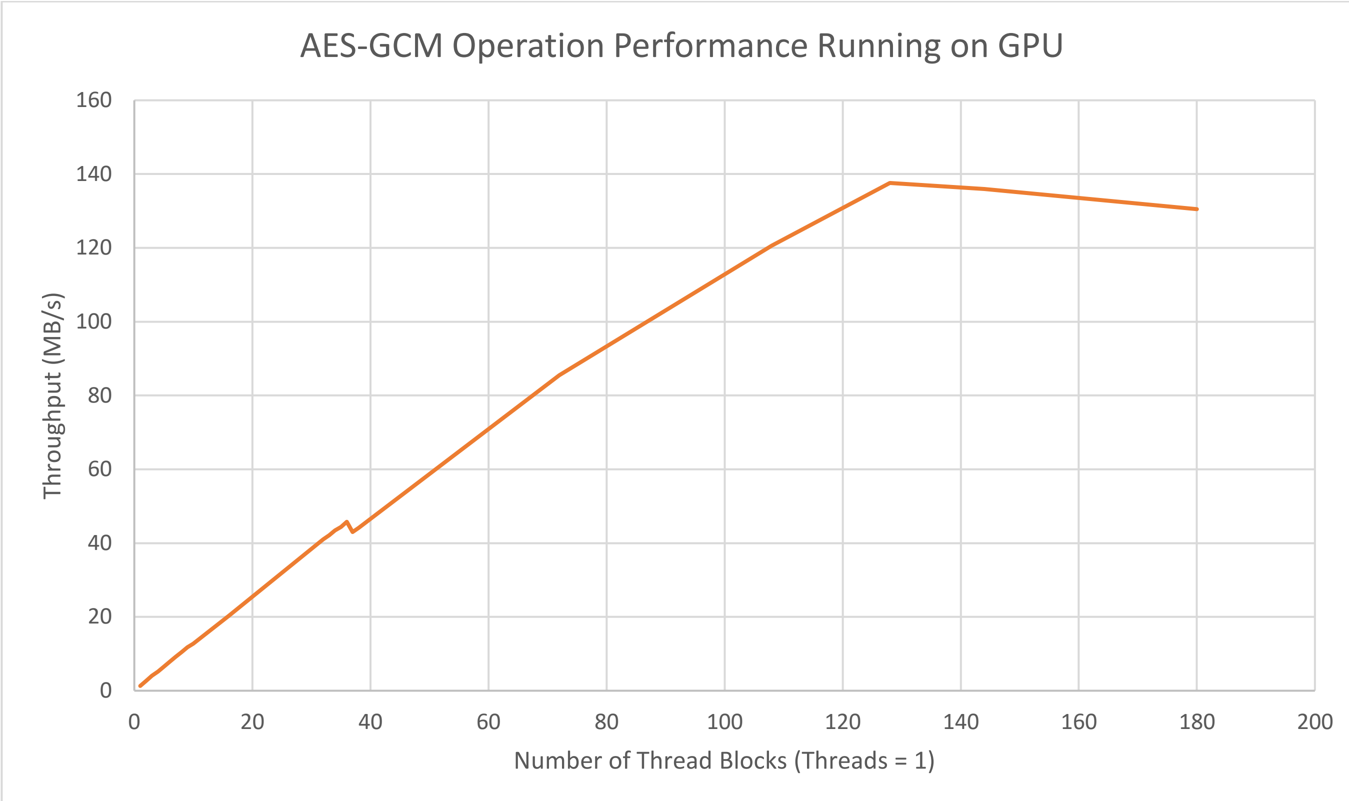

Figure 8 Performance Benchmarking. GPU has 36 SMs. the break on the line is at number 36

In the following test, number of both threads and thread-blocks are kept maximum or more than maximum[3] to check the performance result:

No of Thread Blocks	No of Threads	Throughput (MB/s)
36	128	4200
36	256	4500
36	512	2500
72	128	4500
108	128	3300
108	256	2900
256	512	2700

[3] Maximum for “number of threads” is “number of cores per SM” and for “number of thread blocks” is “number of SMs”

Following diagrams compare performance of CPU and GPU together:

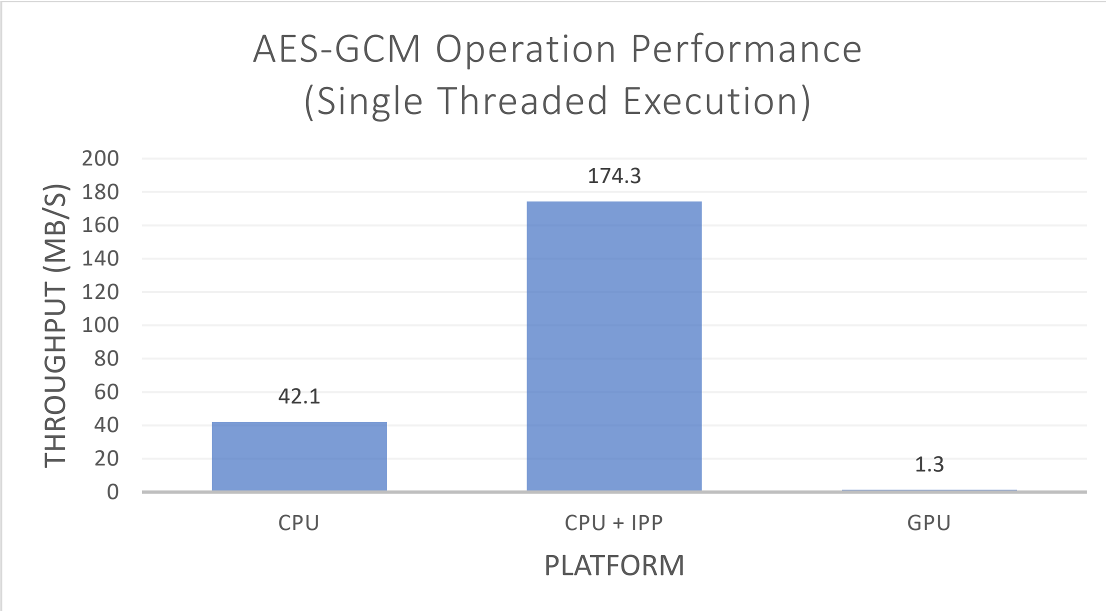

Figure 9 Comparison between a CPU core and a GPU core

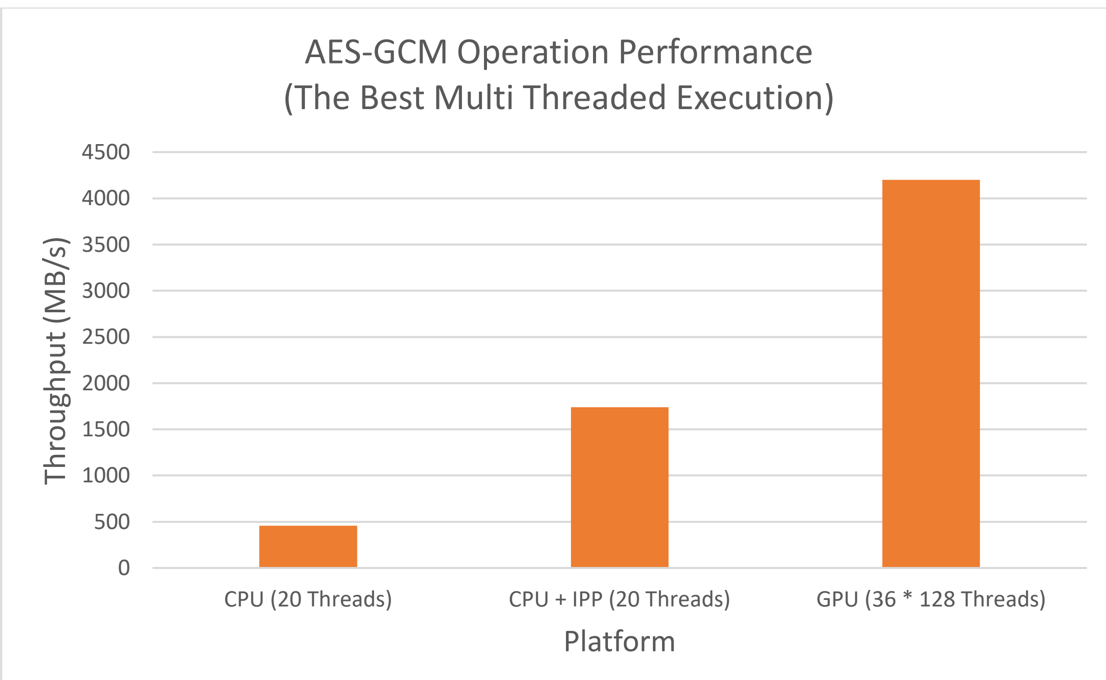

Figure 10 Comparison between CPU and GPU Throughputs

6- Benchmarking Hardware Architectures and CPU/GPU Generations

As the last test in this article, the same test-code of chapters 4 and 5 are being run on different generations of CPUs and GPUs to check the impact of architectural improvement on code execution performance.

		Operation Throughput (MB/s)		
CPU		**i7-11370H**	**i7-12700H**	**i7-13620H**
Cores		8	20	16
AES – GCM Test As in Chapter 8	Thread = 1	34.8	42.3	42.4
	Threads = cores	176	493	480
	Threads = cores, IPP	646	2000	1700

		Operation Throughput (MB/s)		
GPU		**RTX 3050**	**RTX 4070 (2023)**	**RTX 4060 (2024)**
Cores		16 * 128	36 * 128	24 * 128
AES – GCM Test As in Chapter 8	Thread = 1	1.2	1.4	1.5
	Threads = cores	903	4200	3100

7- Screen-shots of the **Benchmarking Software** and Tests

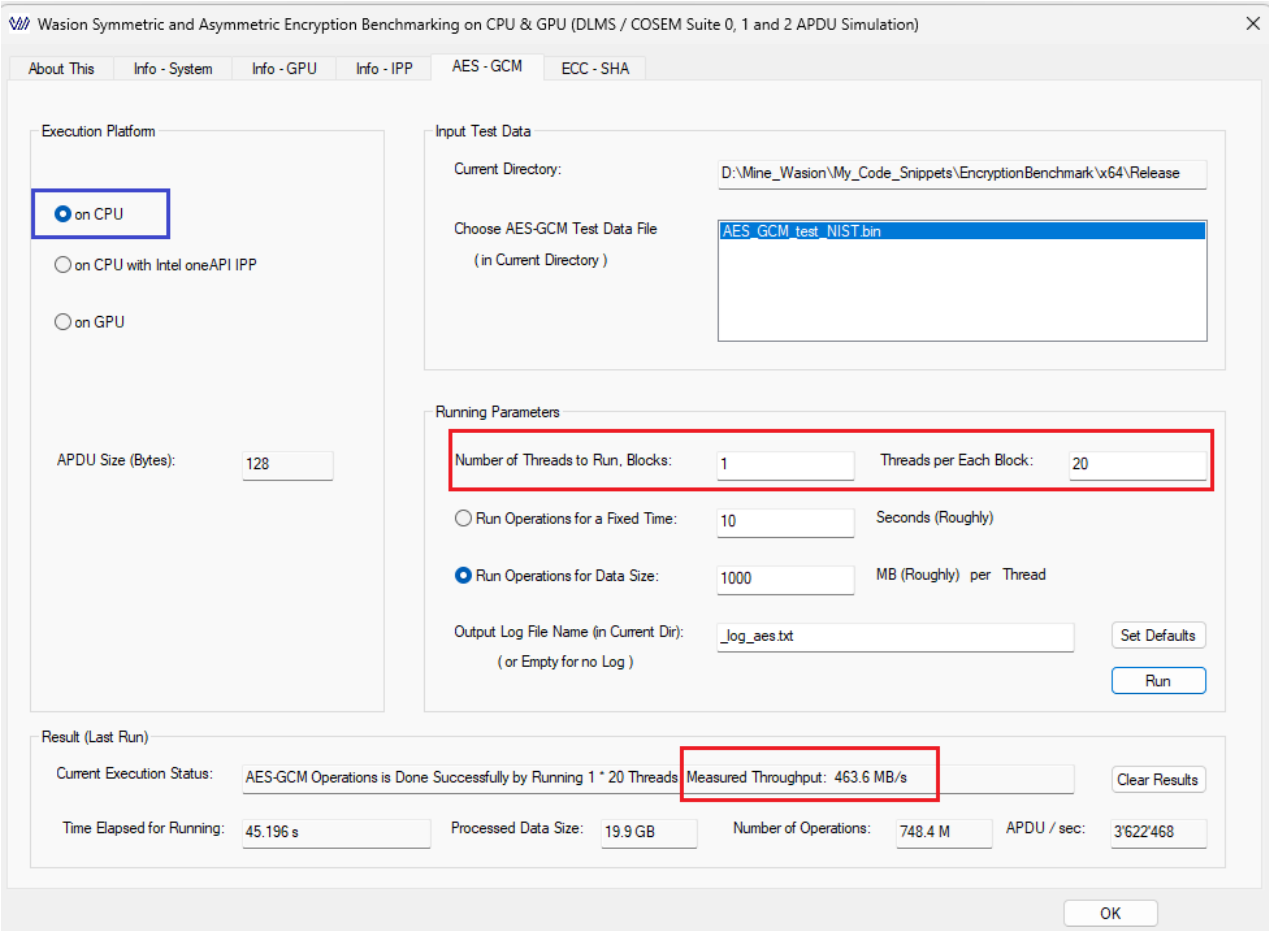

Figure 11 AES-GCM Test Screen (Running on CPU with 20 Thread) CPU has 20 Cores

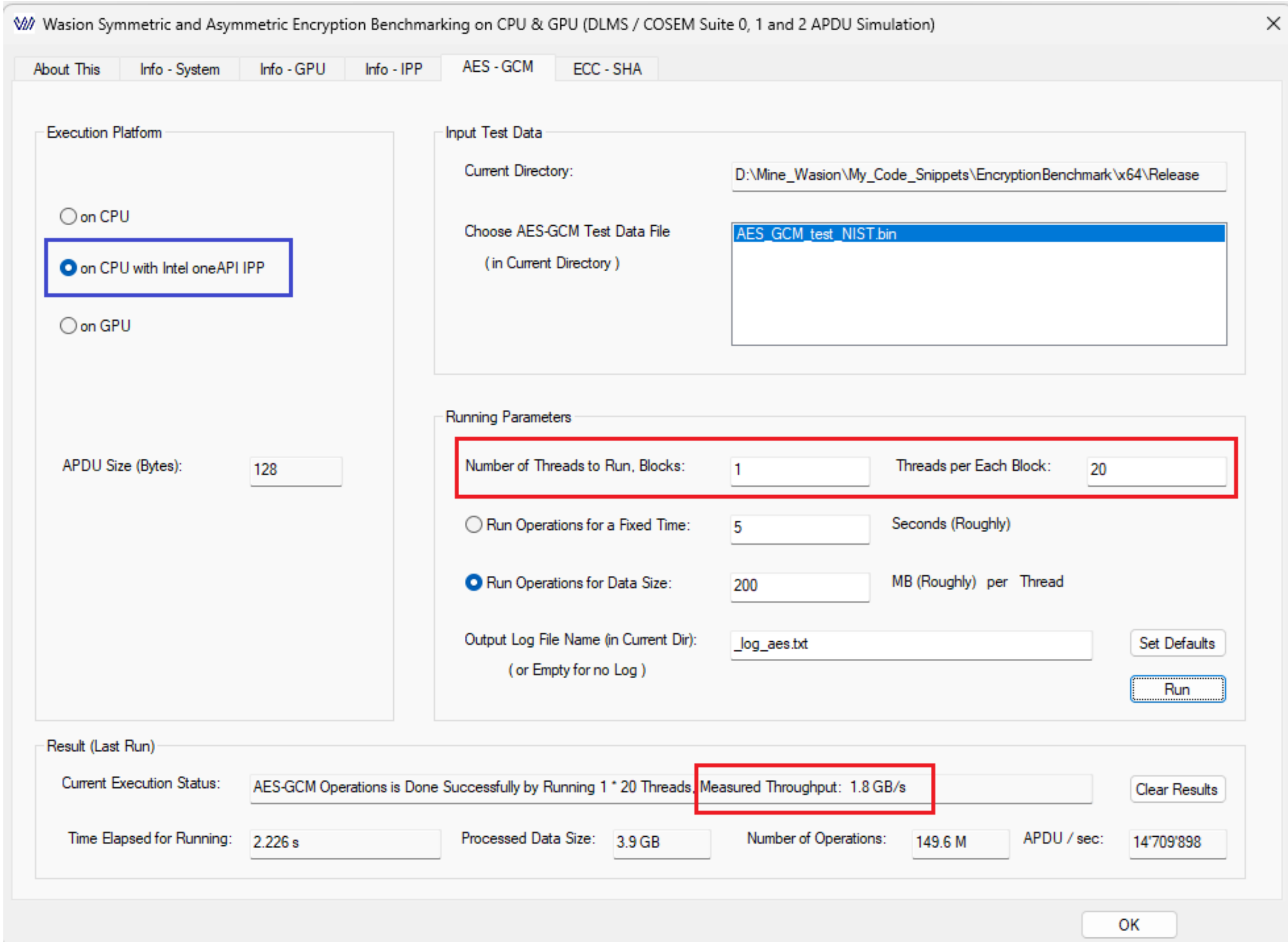

Figure 12 AES-GCM Test Screen (Running on CPU with 20 Thread, Using Intel IPP Lib) CPU has 20 Cores

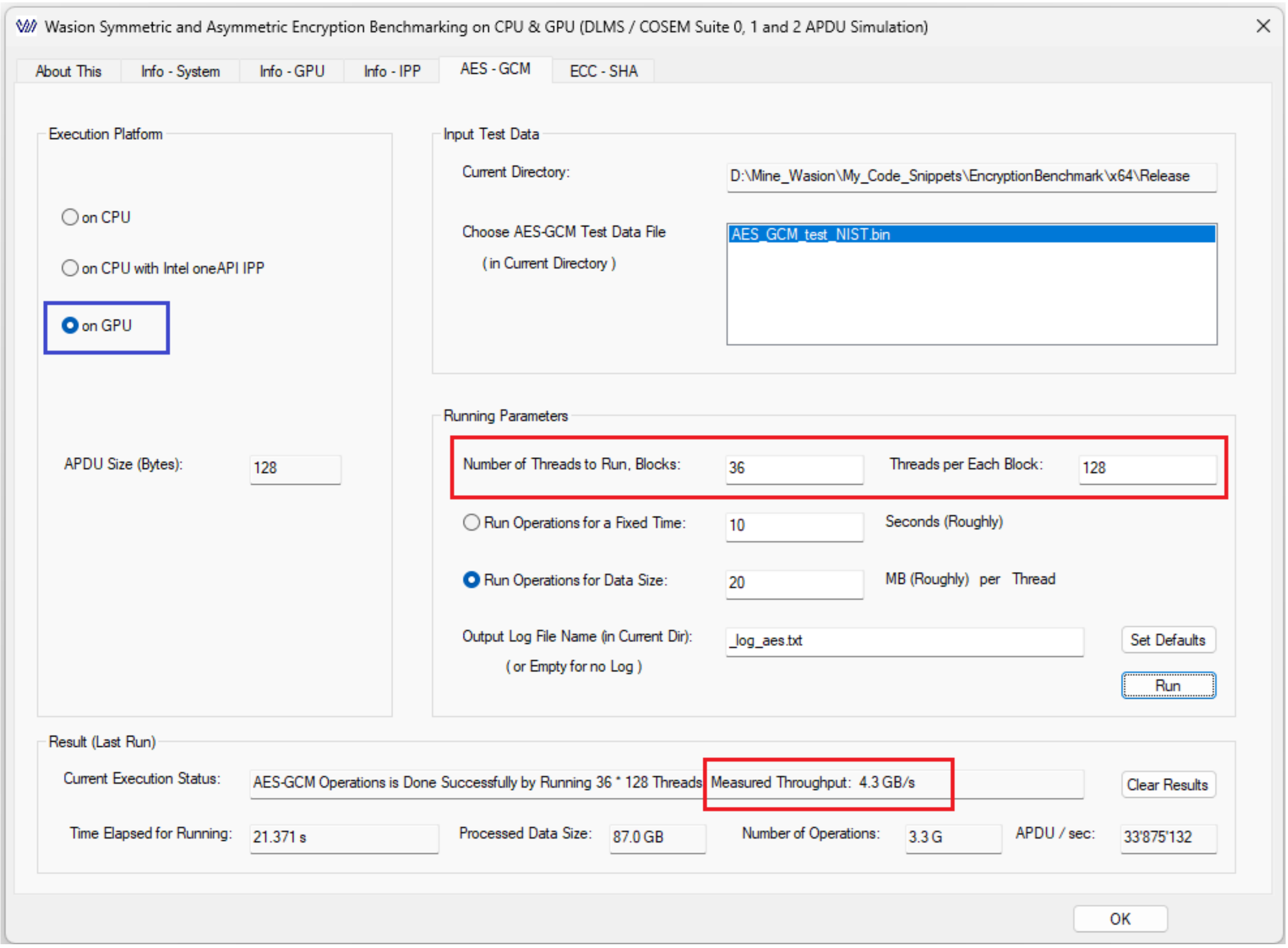

Figure 13 AES-GCM Test Screen (Running on GPU with 36 Thread Blocks and 128 Threads Per Block) GPU has 36 SMs and 128 cores per Each

```
Wasion America, Encyption Benchmarking App, Log File

Algorithm               : AES-GCM
Operations              : Encryption, Decryption and Authentication
Input Vector File Size  : 6'558'654 B

Executed Platform       : GPU, NVIDIA GeForce RTX 4070 Laptop GPU
Logging Time            : Tue Oct 15 15:55:14 2024



 Total Number of Encryption Operations                                  : 1.6 G
 Total Number of Decryption Operations with Successful Authentication   : 822.2 M
 Total Number of Decryption Operations with Planned Failed Authentication: 832.5 M
 Total Number of           Operations with  128-bit  key                : 1.1 G
 Total Number of           Operations with  192-bit  key                : 1.1 G
 Total Number of           Operations with  256-bit  key                : 1.1 G
 Total Volume of Messages Processed                                     : 87.0 GB



Operations Details:

Operation Encryption; Sequence No # 01 H
        Plain Text                  :    D5 DE 42 B4 61 64 6C 25    5C 87 BD 29 62 D3 B9 A2
        Key                         :    7F DD B5 74 53 C2 41 D0    3E FB ED 3A C4 4E 37 1C
        Initial Vector              :    EE 28 3A 3F C7 55 75 E3    3E FD 48 87
        Encrypted Text (Reference) :    2C CD A4 A5 41 5C B9 1E    13 5C 2A 0F 78 C9 B2 FD
        Encrypted Text (Calculated):    2C CD A4 A5 41 5C B9 1E    13 5C 2A 0F 78 C9 B2 FD

Operation Encryption; Sequence No # 02 H
        Plain Text                  :    00 7C 5E 5B 3E 59 DF 24    A7 C3 55 58 4F C1 51 8D
        Key                         :    AB 72 C7 7B 97 CB 5F E9    A3 82 D9 FE 81 FF DB ED
        Initial Vector              :    54 CC 7D C2 C3 7E C0 06    BC C6 D1 DA
        Encrypted Text (Reference) :    0E 1B DE 20 6A 07 A9 C2    C1 B6 53 00 F8 C6 49 97
        Encrypted Text (Calculated):    0E 1B DE 20 6A 07 A9 C2    C1 B6 53 00 F8 C6 49 97
```

Figure 14 Log File Generated by AES-GCM Test, Consists Details of all Encrypted / Decrypted Messages

8- Test platform (Hardware & Software Specification) [except for chapter 6]

Machine	Laptop Asus TUF Gaming F15
CPU	Intel Core i7 12700 (6 P-Core + 8 E-Core, in total **20 cores**) @ 4 GHz
RAM	32 GB @ 3.2 GHZ[4]
GPU	Nvidia RTX 4070 (36 SMs, each has 128 cores, in total **4608 CUDA cores**)
GRAM	8 GB @ 8 GHZ[5]
OS	Windows 11 Enterprise 64-bit (23H2) Linux Debain V12 64-bit, Kernel V6.6.13 *(bare metal installation)*
Language	ISO C++17 *(except when it is stated)* MFC for GUI
Compiler	*For Windows:* Visual Studio 2022 V17.10 (MSVC) nvcc 12.5 for CUDA (GPU) Intel IPP Library V2021.9 (2024) .NET 8.0.10 (for C#) JRE 21.0.1 (for Java) Python 3.12.7 *For Linux:* gcc/g++ V13.2
Compile Mode	64-bit Release Mode, optimized for speed *(except when it is stated)*
AES – GCM Test File (Data)	**Message Size:** 128 Bytes **Key Size:** 128-bit, 192-bit, 256-bit (each 33.3% of records) **Operations:** Encryption (50%), Decryption (25%), Authentication Only (25%)

[4] When running on CPU, all data (raw / plain data, encrypted data, keys and any other) are kept in, loaded from and stored in RAM so there were no disk operations during tests.
[5] When running on GPU, all data (raw / plain data, encrypted data, keys and any other) are kept in, loaded from and stored in GRAM so there were no disk operations (and negligible RAM operations) during tests.

9- Thermal Impact of Running Full Load on CPU / GPU

		Before Starting Tests	10s after Starting Tests
When Running on CPU (All Cores Busy)	**CPU Frequency**	2 GHz	4 GHz
	CPU Temperature	45^{o}C	95^{o}C
	CPU Load	0	100%
	CPU Voltage	800 mV	1279 mV
	CPU FAN Speed	0	2700 RPM[6]
When Running on GPU (All Cores Busy)	**CPU Frequency**	2 GHz	4 GHz
	CPU Temperature	45^{o}C	61^{o}C
	CPU Voltage	800 mV	1279 mV
	CPU Load	0	12%
	CPU FAN Speed	0	2200 RPM
	GPU Frequency	210 MHz	2.4 GHz
	G-RAM Frequency	405 MHz	8 GHz
	GPU Temperature	42^{o}C	51^{o}C
	GPU Voltage	620 mV	985 mV
	GPU Load	0	Max
	GPU FAN Speed	0	2700 RPM

[6] Also, GPU Fan runs fast like with 2700 RPM

10- Conclusion

I. CPU cores are generally faster than GPU cores (in our tests in chapter 5 almost 30 times)

II. Since GPUs typically have more cores in comparison to CPUs their overall performance can be better (in our tests in chapter 5 almost 9 times)

   a. **GPUs have better throughput rather than CPUs while CPUs have lower latency**

III. Increasing number of threads in CPUs (by factor of N while N < number of cores) increases the execution overall performance by factor of almost N/2 (in our tests) but the relation is not linear and increase of performance for greater values for N is less. For GPUs the relation between number of threads and the execution performance is more linear

IV. Increasing number of threads more than processor cores (both for CPUs and GPUs) doesn't much increase the performance. Increasing thread numbers much more than number of cores even decreases the performance, the reason can be wasted time in **context switching**.

V. Intel 12th generation CPUs (**Alder Lake**) have considerable performance improvement in comparison to 11th generation CPUs (**Rocket Lake**) while the performance difference between 12th and 13th generation CPUs (**Raptor Lake**) are not so much. For GPUs the performance difference between **Ampere** architecture (RTX 30xx) and **Ada Lovelace** architecture (RTX 40xx) is also considerable.

VI. Using different data types for variables has a great impact on execution performance. Based on the tests of chapter 3, "floating point" calculations (both 4-Byte and 8-Byte length data types) are way slower than integer data type calculations: both on CPU and GPU. On the tested CPU, calculations with 8-Byte integer and 4-byte integer have almost the same performance, the reason is that the tested CPU is a 64-bit CPU with 128-bit data bus.

VII. **Sometimes writing an optimized code is an art.** As it can be observed in chapter 3, the performance difference between dynamic and static casting for CPUs can be up to 120% and for GPUs can be up to 800%. Details can be found at the end of chapter 3[7].

VIII. For running specific algorithms like cryptographic codes Intel oneAPI (IPP module) proves a great boost in performance. Results printed at the end of chapter 4, shows a performance boost of around 450%. One of the reasons is that IPP uses special instructions of CPU to run codes directly on hardware.

   a. Developing code for IPP has some drawbacks like:

      i. It doesn't have so good documentation: Functions are not well explained and "structs" are not documented at all

      ii. There is very small amount of sample codes in the Internet for it

      iii. Intel's IPP forum is not active in a satisfactory level

[7] A great number of programmers do NOT pay attention to such details. They leave all casting to runtime hence the result is awful.

11- Table of Acronyms and Abbreviations

Acronym / Abbreviation	Stands For
AES	Advanced Encryption Standard
AES-GCM	Advanced Encryption Standard - Galois Counter Mode
AI	Artificial Intelligence
AHE	Advanced Head-End
APDU	Application Protocol Data Unit
API	Application Programming Interface
B	Byte
b	bit
COSEM	Companion Specification for Energy Metering
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DLMS	Device Language Message Specification
ECC	Elliptic Curve Cryptography
ECDSA	Elliptic Curve Digital Signature Algorithm
E-Core	Efficient Core (CPU)
G	Giga (almost 10 to the power of 9)
GB	Giga Byte
GB/s	Giga Bytes per Second
GCC	GNU C Compiler (GNU stands for GNU Not Unix)
GHZ	Giga Hertz
GPU	Graphics Processing Unit
GP-GPU	General Purpose Graphics Processing Unit
GUI	Graphical User Interface
H	Hexadecimal
IoT	Internet of Things
IPP	Intel Integrated Performance Primitives
ISO	Independent System Operator
JRE	Java Runtime Environment
KB/s	Kilo Bytes per Second (Kilo means 1024)

M	Mega (almost one million)
MB/s	Mega Bytes per Second (Mega means almost one million)
MDC	Meter Data Collector / Collection
MDM	Meter Data Manager / Management
MFC	Microsoft Foundation Class
MHz	Mega Hertz (Mega means million)
ms	milli second
MSVC	Microsoft Visual C
mV	milli Volt
NIST	National Institute for Standards and Technology
NPU	Neural Processing Unit
nvcc	Nvidia CUDA Compiler
Oct	October
OS	Operating System
P-Core	Performance Core (CPU)
R	Registered
RPM	Revolution Per Minute
RTX	Ray tracing Texel eXtreme - Ray Tracing eXperiance
s	Seconds
SHA	Secure Hash Algorithm
SM	Streaming Multiprocessors
TM	Trade Mark
Tue	Tuesday
V	Version
VEE	Validation, Estimation and Editing