BENCHMARKING MATRIX MULTIPLICATIONS FOR VARIABLE QUBIT SIZE AND DEPTH

ABSTRACT

To emulate a quantum computation on a classical computer i.e. the evolution of the unitary operations on the wave function of the particle in quantum mechanics, we have to perform unitary matrix and normalized vector multiplications in the high-level programming languages of Python, C++, Java, etc. Quantum Libraries already available perform the matrix-vector multiplication in the backend using the numpy libraries of Python like Qiskit or use a C++ wrapper to further optimize the runtime it as in Qiskit-Aer Simulators. Since a fully functioning fault-tolerant computer is decades away, it is in our best interest to design new quantum algorithms and develop accelerator test beds for Quantum Emulations. All the quantum computer operations can be emulated on a classical computer, with the only downside being that the matrix multiplications scale up as $O(N^3)$ in runtime. In contrast, the quantum computer scales it up as $O(n)$, where $N = 2^n$, where N is the matrix dimension, where n is the number of qubits, so the runtime for quantum emulations on the classical computer increases exponentially with increase in number of qubits and increases linearly with increase in number of depths, complexity wise. Though it is not possible to change the exponential index, it is possible to reduce the runtime for quantum emulations on classical computers by use of GPU and Alveo Accelerator Cards, and also code optimization on the software side like using a C++ wrapper. In this paper, we will benchmark the matrix-matrix multiplications on HPC Accelerator Cards varying the qubit size and the depth of the quantum circuit and provide a universal mathematical equation for the runtime on the GPU and Alveo Vector Cards for two variables of qubit size and quantum circuit depth. So a theoretical limit on qubit size and qubit depth exactly can be established for quantum emulations on present classical supercomputers.

Keywords - Qiskit, Qubit, GPU, Alveo Accelerator Cards

1. INTRODUCTION

What is a Qubit? A Qubit is a quantum binary unit that can take any normalized amplitude and the state for that can be '0', '1', or '0 + 1' or '0 - 1', on a 0 and 1 basis, These basis can be represented in the form of a column vector with the size as $[2^n, 1]$, where n is the number of qubits. Now any quantum algorithm can be decomposed into a set of universal quantum gates of $U(\theta, \phi, \psi)$ and CNOT, which can be represented as unitary matrices of 2x2 and 4x4 sizes respectively. Now it is up to the coder on how one will code the operations of quantum emulations as, i.e it can be done as Kronecker Product between the 2x2 unitary gates, 4x4 CNOT gates matrices for all qubits at a particular depth and then multiply the resultant unitary with the input column state vector, or one can also follow the approach of unitary matrix - Kronecker product and then state vector multiplication for each depth. But here we are following the prior approach of performing the matrix-matrix multiplications first and then matrix-vector multiplications. Here the Kronecker product requires the same time irrespective of the hardware architecture, we will not perform the Kronecker product on the CPU, GPU, and ALVEO, but rather perform and benchmark the time for matrix-matrix multiplications by varying the qubit size and the depth(1).

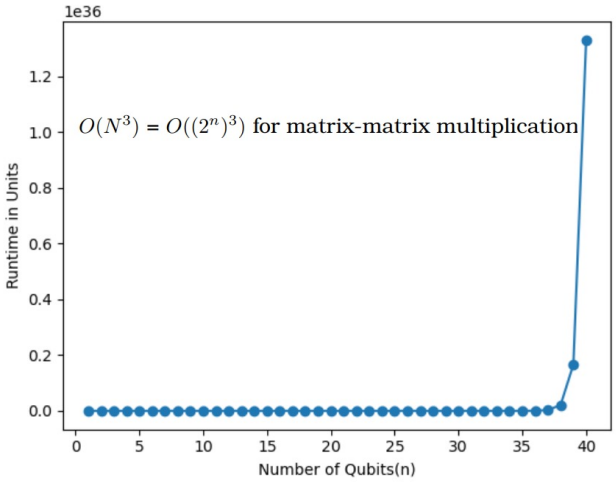

Figure 1 – Cubic Exponential Curve for Runtime as the number of qubits is increased for Quantum Emulations

Emulation of Quantum Algorithms using FPGAs has been done in (2) but in our work, Alveo card combines three essential things: a powerful FPGA for acceleration, which has high-bandwidth DDR4 memory banks, and connectivity to a host server via a high-bandwidth PCIe Gen3x16 link. This link is capable of transferring approximately 16 GiB of data per second between the Alveo card and the host.

How does a qubit size relate to matrix size in emulations on a classical processor using a quantum library(8)? The answer is $(2^n * 2^n)$ where n is the number of qubits, to represent the superposition on a qubit or a unitary gate operation on a qubit, we require a matrix size of 2x2 and vector size of 2x1, which scales up exponentially as we increase the number of qubits i.e the matrix size $(2^n * 2^n)$ and state vector size $(2^n * 1)$. Once the code optimization is done on the backend, then it is a hardware architecture of the classical computation that needs to be changed to get the

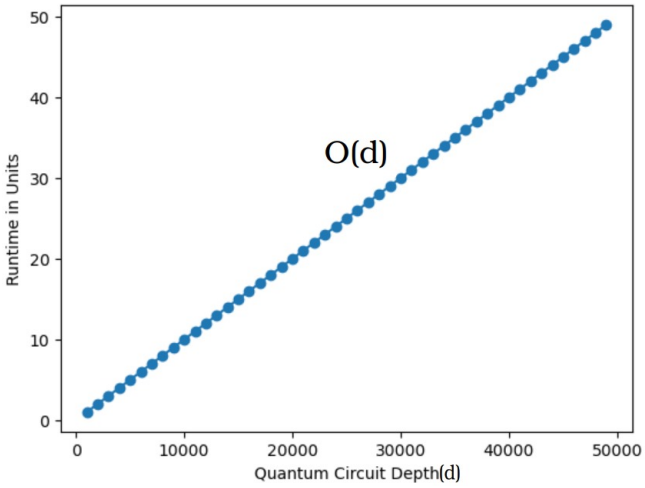

Figure 2 – Runtime in units vs Quantum Circuit Depth Complexity

performance enhancement, i.e. the GPU and the ALVEO Cards i.e. HPC Cards on our end(3). In other research works, CMOS circuit emulators for quantum computing have been proposed (4), but these hardware implementations are not yet commercially available. In the near term, using classical computing hardware to emulate quantum computation remains a viable solution.

The depth of the quantum circuit is equivalent to the number of matrices used in the multiplications in quantum emulations. The complexity of increasing the quantum circuit depth i.e matrix multiplication depth is linear for CPU, but how much is the slope of the runtime with increasing depth lesser in the case of the GPU and ALVEO, Also how does the complexity of the runtime vary with increasing the number of qubits on ALVEO and GPU, which is $O(N^3)$ for CPU? Even if the complexity of the runtime remains the same on the accelerator cards the exact equation of the complexity will have lower values on GPU and ALVEO Cards owing to the customization of the hardware architecture as per the application on ALVEO Cards and parallelism on GPU Cards. This paper aims to benchmark the how and exact mathematical equation for variable qubit size and the quantum circuit depth, which can be further used to establish a bottleneck for the qubit size and the quantum circuit on a present supercomputer for quantum emulations. Now let's move on to the exact dataflow for matrix multiplications on CPU, GPU, and ALVEO Cards(5).

A clear pictorial representation between the quantum emulations and actual quantum circuits is shown in Fig. 3.

2. MATRIX MULTIPLICATION ON CPU

In CPU, first the data (Matrix elements) is loaded to the cache L1 from CPU RAM(Random Access Memory) by using multiple data buses for efficient parallel computation. Once the required matrix elements are in the cache, they can be loaded into CPU registers, which are small, fast storage locations directly accessible by the CPU cores. The CPU's instruction decoder and execution units handle the

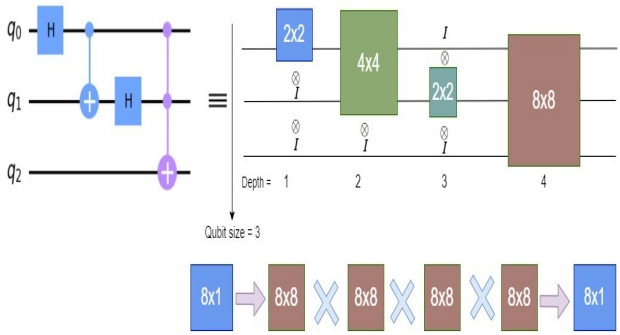

Figure 3 – Quantum Circuit and its Quantum Emulations are Matrix-Vector and Matrix Matrix Multiplications

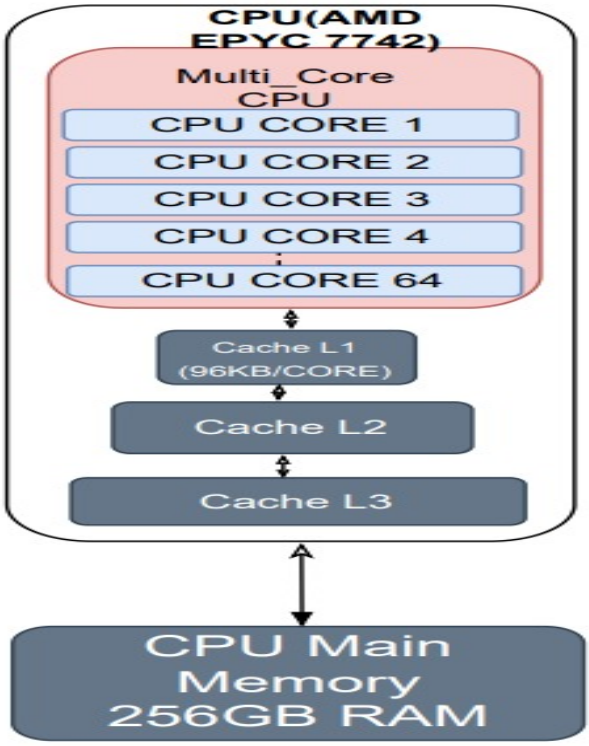

Figure 4 – Internal Architecture of CPU

loading of data from cache into registers. This process is typically controlled by assembly-level instructions generated by the compiler or software. Once the matrix elements are loaded into CPU registers, the actual multiplication operation can begin. The EPYC 7742 CPU features multiple cores, each capable of executing instructions independently. These cores can work in parallel, allowing for efficient processing of matrix multiplication tasks. The CPU's SIMD(Single Instruction, Multiple Data) units can be leveraged for parallel computation. SIMD instructions enable the execution of the same operation on multiple data elements simultaneously, which is beneficial for matrix multiplication. The CPU executes instructions generated by the software or compiler to perform the matrix multiplication algorithm. This involves a combination of arithmetic operations(e.g., Addition and Multiplication) and data movement instructions. Once the matrix multiplication is complete, the result matrix is stored

back into CPU registers. Depending on the application's requirements, the results may need to be written back to RAM. This involves a similar process to loading data from RAM, where the CPU's cache hierarchy is utilized to manage data movement efficiently. For these operations, we use the PYTHON library(numpy).

3. MATRIX MULTIPLICATION ON GPU

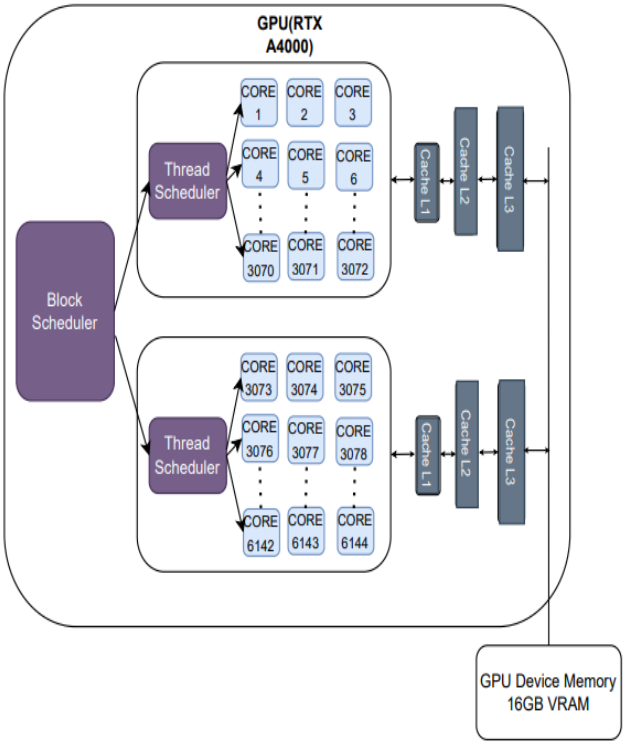

Figure 5 – Internal Architecture of GPU

We are using GPU RTX A4000 series in this series first, we need to transfer the matrices from the system's RAM(main memory) to the GPU's dedicated memory known as Video RAM (VRAM). This transfer typically involves using functions provided by GPU-accelerated libraries like CUDA(Compute Unified Device Architecture) or cuBLAS(CUDA Basic Linear Algebra Subprograms). The CUDA programming model, for instance, provides functions like 'cudaMemcpy()' to transfer data between the HOST(CPU) and device (GPU) memories efficiently. Matrix multiplication on a GPU is typically implemented as a kernel function, which is a small program executed in parallel by many threads on the GPU(6). CUDA kernel function specifically designed to perform matrix multiplication is being used. Each thread in the GPU executes a portion on the matrix multiplication operation in parallel. CUDA provides grid and block structures to organize these threads into a grid of thread blocks, which are then executed concurrently on the GPU's streaming multiprocessors(SMs). Once the matrices are loaded into GPU memory and the matrix multiplication kernel is working, the kernel is launched from the CPU. This initiates parallel execution of the matrix multiplication operation on the GPU. The number of threads per block and the number of blocks per grid are determined based on the size of the matrix and the GPU's architecture to achieve optimal parallelism. Each thread block is scheduled onto an SM for execution. Within each SM, multiple thread blocks can be processed concurrently, with each block utilizing the SM's resources efficiently, The CUDA runtime manages the scheduling and execution of thread blocks across the GPU's SM, maximizing parallelism and throughput. Once the matrix multiplication is complete, the result matrix is transferred from GPU memory back to CPU memory using functions similar to cudaMemcpy(). Overall, performing matrix multiplication on an RTX A4000 GPU involves leveraging its massively parallel architecture, utilizing the CUDA programming model for kernel execution, and efficiently managing data movement between the CPU and GPU memories. This approach allows for significant acceleration of matrix operations compared to traditional CPU-based computations, especially for large matrices

4. MATRIX MULTIPLICATION ON ALVEO ACCELERATOR CARD

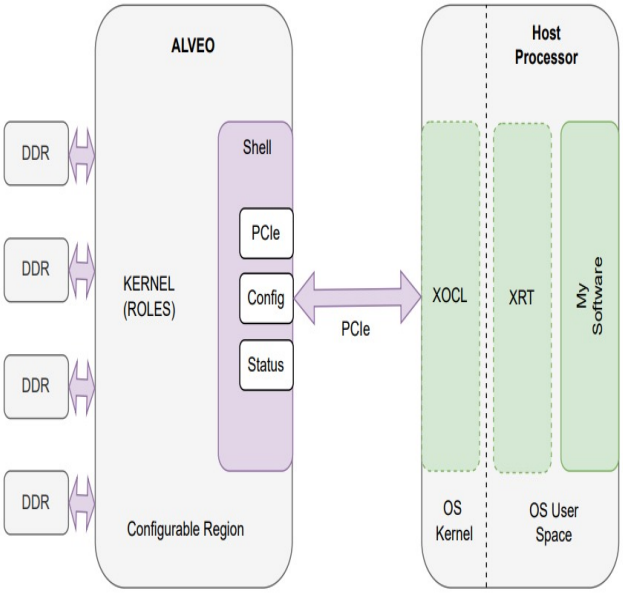

Figure 6 – Internal Architecture of ALVEO

We have used the Alveo U55C accelerator card, this accelerator card contains 16GB of High Bandwidth Memory(HBM) of second generation to perform the required mathematical operations. The Alveo can directly access this memory for fast-chain command executions. The shell sections shown in the figure is reserved by the Alveo card to host functionalities like: PCIe control kernel, Xilinx Run Time (XRT) driver, Status registers. The rest of the Alveo outside shell is the user programmable region. In OS userspace has two sections

1. 'My Software' sections where we make or store the software.
2. XRT section is the communication link between the Host

and the Alveo card.
The OS kernel has Xilinx Open Computing Language(XOCL) for parallel programming on Alveo(FPGA). The matrix multiplication is written in a .cpp file, in a high-level programming language of C++. This file is then compiled with the help of Vitis compiler(V++) and after compilation, we get two files mmult.xo (matrix multiplication) and vadd.xo (vector addition) which are the kernels or functions defined in the .cpp file, through which we create xclbin, which is a lower-level machine language file, that consists of the two definitions of mmult and vadd. This xclbin file is used as an overlay file using the PYNQ library in Python language where the matrices to be multiplied are defined and the matrices to be multiplied are first reshaped as per the shapes defined in .cpp file called as tiling of matrices and then the matrices are ported on the ALVEO card using the sync.to.device() function of the pynq library. Using the mmult and vadd functions the matrix multiplications are performed on the ALVEO card and then the result is ported back to the Host Processor using the sync.from.device() function of the overlay method of PYNQ library(7).

5. ARCHITECTURE SPECIFICATIONS AND THEORETICAL EQUATIONS

5.1 CPU Architecture Specification

AMD EPYC 7742
Number of Cores - 64
RAM of Processor - 256 GB
Number of Threads - 64 cores * 2 threads/core = 128 threads
Base Clock Speed - 2.25GHz
FLOPS = 4608 GFLOPS (double-precision performance)

$$t_{CPU} = (2.1701 * 10^{-4} * 2^{3n} + (2^{2n} * (2^n - 1))) * d \quad (1)$$

Here, n is the number of qubits (i.e the matrix size will be 2^n), and d is depth (number of matrices). The t_{CPU} gives the computational runtime for CPU at depth 1 and at any matrix size.

5.2 GPU Architecture Specification

RTX A4000
Number of Cores - 6144
RAM of Processor - 16 GB
Number of Threads - 6,144 cores * 32 threads/core = 196,608 threads
Base Clock Speed - 1420MHz
FLOPS = 299.5 GFLOPS (double-precision performance)

$$t_{GPU} = (3.3388 * 10^{-3} * 2^{3n} + (2^{2n} * (2^n - 1))) * d \quad (2)$$

5.3 Alveo Architecture Specification

ALVEO U55C
Look up tables (LUTs): 1304K LUTs.
Registers: It features 2,607K registers.
DSP Slices: The card includes 9,024 DSP slices.
RAM of Processor:16GB High Bandwidth Memory (HBM2).
Base Clock Speed - 300MHz
FLOPS = 500-600 GLOPS/SECOND (approximately)

$$t_{Alveo} = ((1.6666 * 10^{-3} - 2 * 10^{-3}) * 2^{3n} + (2^{2n} * (2^n - 1))) * d \quad (3)$$

6. RESULTS

Experimental findings:
Runtime equations for 3architectures from curve fit

$$t_{CPU} = (3.1181 * 10^{-9} * 2^{3n} + (2^{2n} * (2^n - 1))) * d \quad (4)$$

The coefficient a is independent of the matrix size and depth in the case of CPU. This is evident from Figure 8, we plot the runtime vs number of qubits for depth 11 and found that the value of a is $3.12324502 * e^{-09}$ which is almost equal to the value obtained at depth 1.

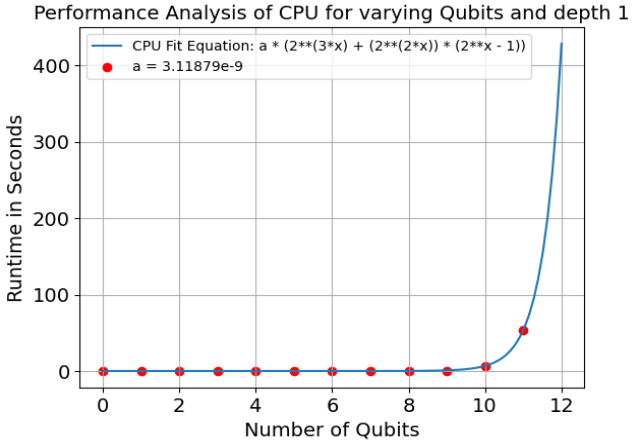

Figure 7 – Performance of CPU

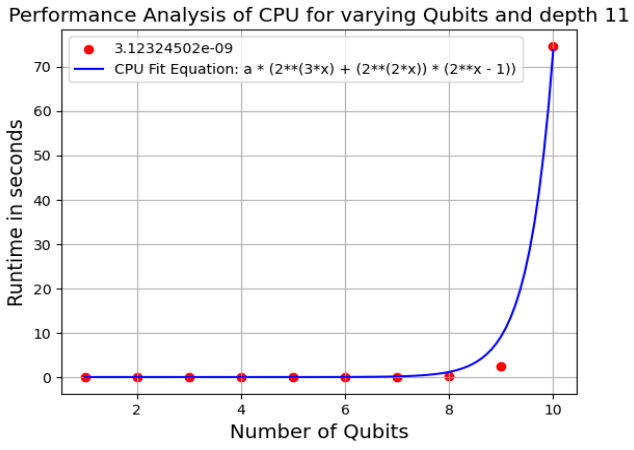

Figure 8 – Performance of CPU

The coefficient a in this case of GPU is dependent upon the matrix size and depth in the case of GPU. At a particular depth, the coefficient is different.
t_{GPU} from 0 to 8 Qubits

$$t_{GPU} = (5.44 * 10^{-8} * 2^{3n} + (2^{2n} * (2^n - 1))) * d \quad (5)$$

t_{GPU} from 9 to 16 Qubits

$$t_{GPU} = (3.82*10^{-14}*2^{3n}+(2^{2n}*(2^n-1)))*d+1.8545 \quad (6)$$

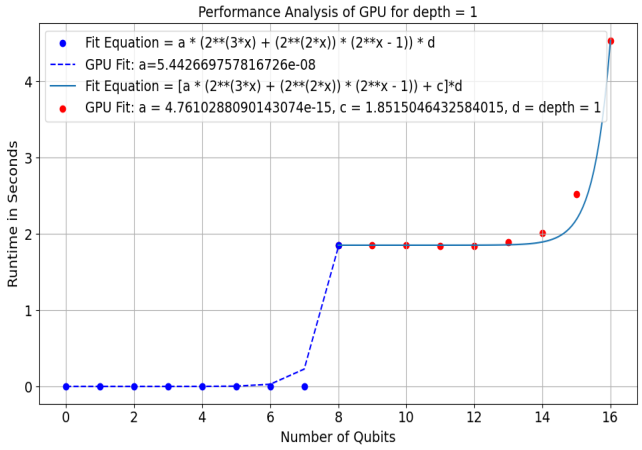

Figure 9 – Performance of GPU

$$t_{Alveo} = (1.4093 * 10^{-10} * 2^{3n} + (2^{2n} * (2^n - 1))) * d \quad (7)$$

The coefficient a in this case is independent of the matrix size and depth in case of Alveo as well.

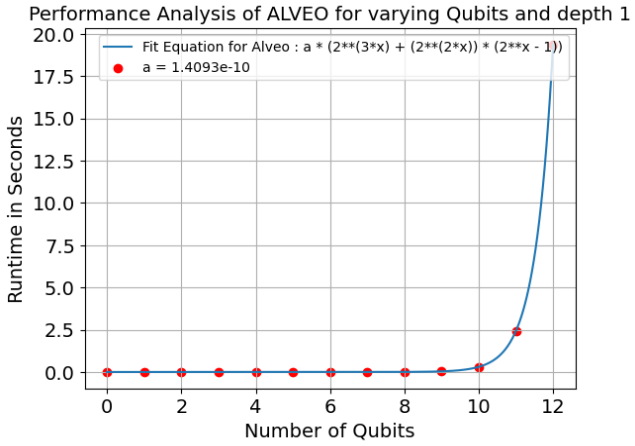

Figure 10 – Performance of Alveo card

In Table 1, we have given the performance sequence of all three architectures. This table shows the comparison of runtimes at a particular matrix size (i.e. at a given number of qubits) at depth 1. Here, depth means the number of times the matrices are getting multiplied together. We observe that GPU is better in most of the cases and Alveo is better at a few sizes (i.e. 8, 9, and 10). It is also worth mentioning that at lower sizes CPU runtime is almost equal to GPU. In Figure 11, we have plotted the computational runtime vs increasing matrix size using the same data used in Table 1 to establish the performance sequence. Similarly, Figure 12 shows the computational runtime for each matrix size and with increasing depth (from 1 to 21).

In Table 2, the performance sequence is given with increasing depth. It is observed that in most of the cases (i.e sizes) the

Table 1 – Performance of CPU, GPU, and Alveo with increasing qubit size for the fixed depth of 1

Qubits (n)	**Performance Sequence**
1	GPU, CPU, Alveo
2	GPU, CPU, Alveo
3	GPU, CPU, Alveo
4	GPU, CPU, Alveo
5	GPU, CPU, Alveo
6	CPU ≈ GPU, Alveo
7	GPU, CPU, Alveo
8	Alveo, CPU, GPU
9	Alveo, CPU, GPU
10	Alveo, GPU, CPU
11	GPU, Alveo, CPU
12	GPU, Alveo, CPU
13 onwards	GPU, Alveo, CPU

GPU performs better than the other two architectures (CPU and Alveo). But it is also worth mentioning the performance behaviour at some sizes, for example at qubit size 8 (Table 2), Alveo performed the best for the first time and CPU performed better than GPU.

Now, let us look at Figure 13, it shows the computational runtime versus depth of GPU and CPU (the Alveo runtime has to be excluded because its runtime was in a different range). At this qubit size (i.e 7), both of the architectures performed equally good (note that the scaling of y-axis i.e the time axis has been done because the runtime range is of order 10^6).

In Figure 14, we have plotted the performance of the architectures vs depth, the Alveo is again better than CPU and GPU, also GPU is better than CPU after 7 depths. But at the same time, this trend will not followed after further increasing the depth. This is evident if one extrapolates the lines further.

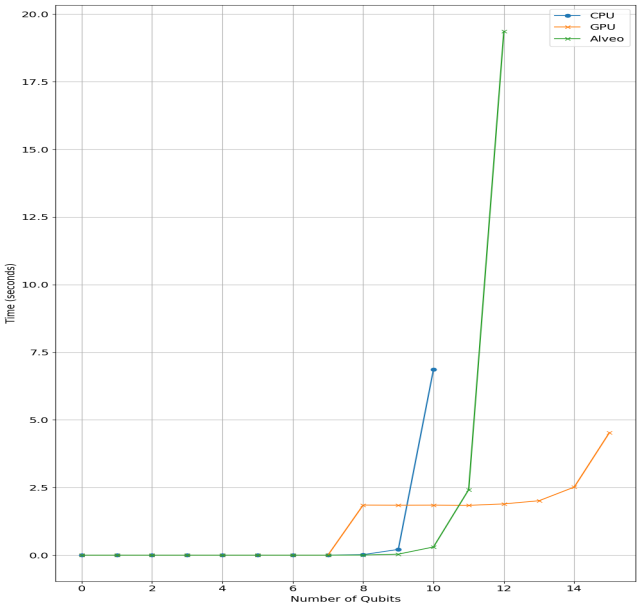

Figure 11 – Performance of CPU, GPU, and Alveo with increasing qubit at depth 1

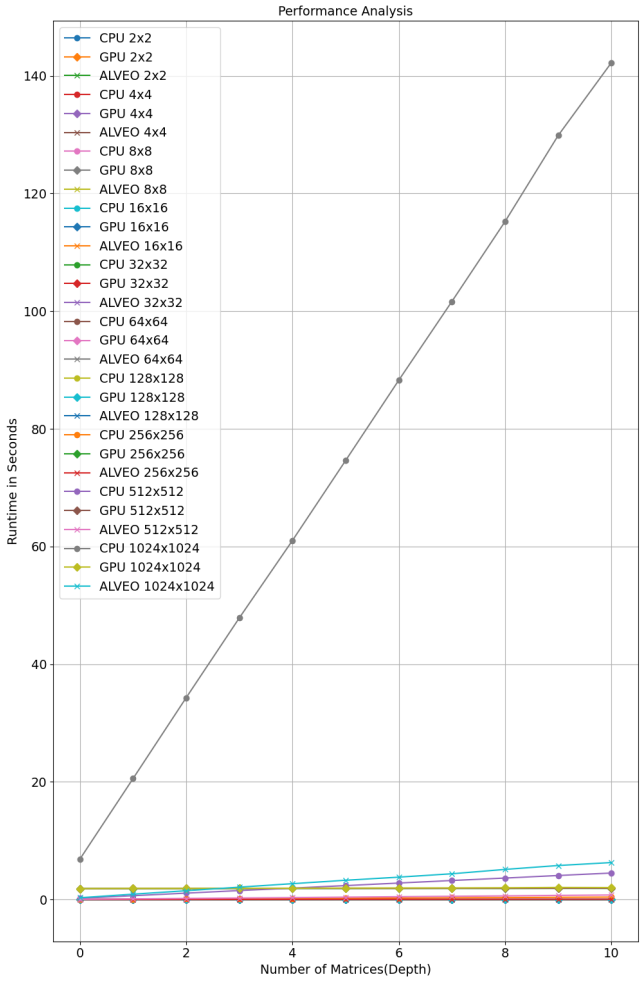

Figure 12 – Runtime comparison of different qubit sizes with depths (from 1 to 21)

Similarly, in Figure 15 we have plotted the performance of the architectures(only for GPU and Alveo) vs depth for 10 qubits, we have to exclude the CPU data from this plot to clearly show the crossover between Alveo and GPU. The performance crossover occurred after 5 depths.

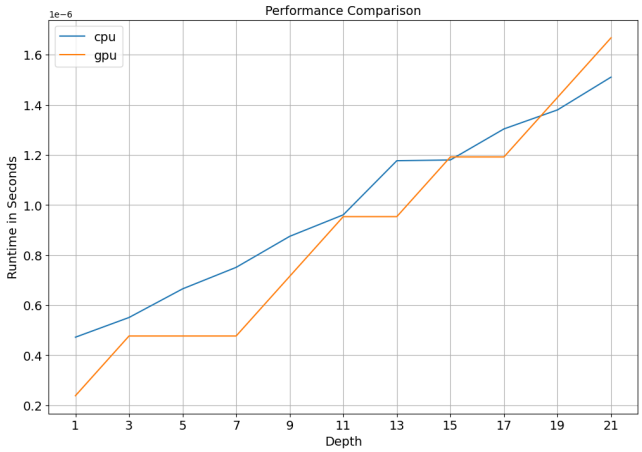

Figure 13 – GPU ≈ CPU (it is taking almost the same time) for 7 qubits

Table 2 – Performance of CPU, GPU and Alveo with increasing depth (from 1 to 21) for the given qubit size

Qubits(n)	Performance Sequence	Crossover depth of two systems
1	GPU ≈ CPU, Alveo	GPU ≈ CPU
2	GPU ≈ CPU, Alveo	GPU ≈ CPU
3	GPU ≈ CPU, Alveo	GPU ≈ CPU
4	GPU ≈ CPU, Alveo	GPU ≈ CPU
5	GPU ≈ CPU, Alveo	GPU ≈ CPU
6	GPU ≈ CPU, Alveo	GPU ≈ CPU
7	GPU ≈ CPU, Alveo	GPU ≈ CPU
8	Alveo, CPU, GPU	No crossover observed
9	Alveo, GPU, CPU	Crossover observed between CPU and GPU after depth 7
10	GPU, Alveo, CPU	Crossover observed between GPU and Alveo after depth 5

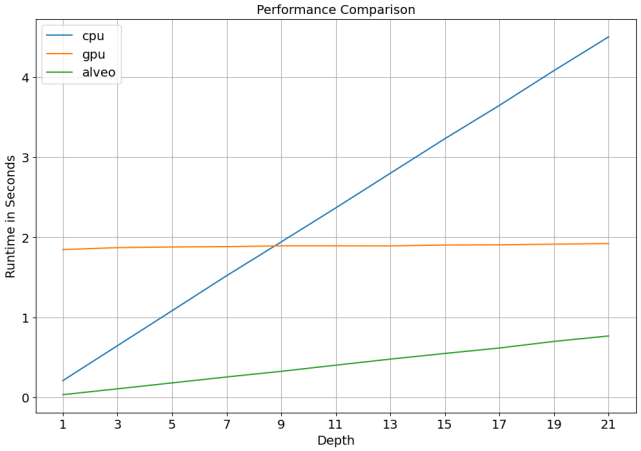

Figure 14 – Crossover observed between CPU and GPU after depth 7 and for 9 qubits

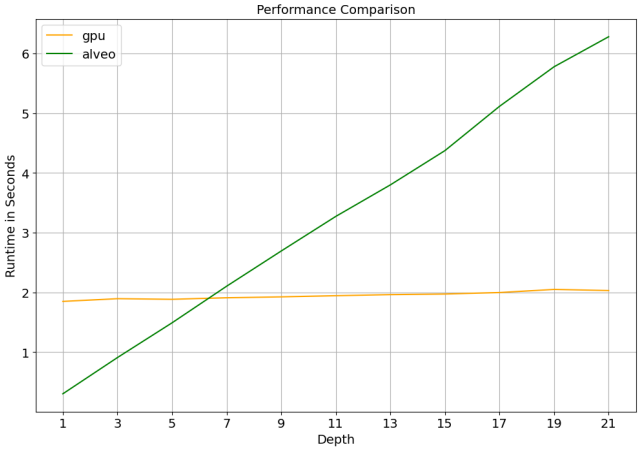

Figure 15 – Crossover observed between GPU and Alveo after depth 5 and for 10 qubits

7. CONCLUSION

In this paper, we were able to derive computation runtime equations for each architecture. The performance curves were obtained for CPU, GPU, and Alveo for any depth. This method of deriving computational run time equations can be useful in different applications such as building a quantum simulator and training deep learning and machine learning models. We also report that this kind of generalised equation which can give computational runtime for any matrix size and at any depth can be formulated in case of CPU and alveo but in case of GPU (as the coefficient is dependent upon the depth and size of the matrix multiplication). At each matrix size, the runtime behavior is almost similar if we keep increasing the depth. Similarly, if we fix the depth and keep increasing the matrix size the runtime behavior is different. Hence the universal computational runtime for any architecture can be formulated for matrix multiplication which can give experimental runtime. The crossover depths between CPU and GPU, as well as GPU and Alveo were identified and the performance sequence for different qubits was established.

REFERENCES

[1] Kestur, Srinidhi, John D. Davis, and Oliver Williams. "Blas comparison on FPGA, CPU and GPU." 2010 IEEE Computer Society Annual Symposium on VLSI. IEEE, 2010.

[2] M. Aminian, M. Saeedi, M. S. Zamani and M. Sedighi, "FPGA-Based Circuit Model Emulation of Quantum Algorithms," 2008 IEEE Computer Society Annual Symposium on VLSI, Montpellier, France, 2008, pp. 399-404, doi: 10.1109/ISVLSI.2008.43.

[3] Vestias, Mario, and Horácio Neto. "Trends of CPU, GPU and FPGA for high-performance computing." 2014 24th International Conference on Field Programmable Logic and Applications (FPL). IEEE, 2014.

[4] R. L. Smith and T. H. Lee, "Quantum Computing Gate Emulation Using CMOS Oscillatory Cellular Neural Networks," IEEE Transactions on Circuits and Systems II: Express Briefs, doi: 10.1109/TCSII.2024.3397846.

[5] Betkaoui, Brahim, David B. Thomas, and Wayne Luk. "Comparing performance and energy efficiency of FPGAs and GPUs for high productivity computing." 2010 International Conference on Field-Programmable Technology. IEEE, 2010.

[6] Asano, Shuichi, Tsutomu Maruyama, and Yoshiki Yamaguchi. "Performance comparison of FPGA, GPU, and CPU in image processing." 2009 international conference on field programmable logic and applications. IEEE, 2009.

[7] Thomas, David Barrie, Lee Howes, and Wayne Luk. "A comparison of CPUs, GPUs, FPGAs, and massively parallel processor arrays for random number generation." Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays. 2009.

[8] Arute, Frank, Kunal Arya, Ryan Babbush, Dave Bacon, Joseph C. Bardin, Rami Barends, Rupak Biswas et al. "Quantum supremacy using a programmable superconducting processor." Nature 574, no. 7779 (2019): 505-510.