Discussion on Runtime Analysis of Pennylane-Lightning Benchmarks

Md Mazder Rahman

January 2023

1 Page 1:

- Did you use clang 14 or 15? Both are mentioned in the document.
  That is an error. I used clang 15.0.6 for experiments.

- You mention using Ubuntu 2022 on the server. Is this Ubuntu 22.04, or 20.04? Also, the docker container uses Focal for the first build, which is 20.04, but the 2022 LTS release Jammy (22.04) is used for the second build.

  In bare-metal for both compiling and runtime environment is Ubuntu 22.04.

  Yes the docker container uses Focal for the first build, which is 20.04, but the 2022 LTS release Jammy (22.04) is used for the second build.

  Why I did this at the first time? I was having trouble to compile in the container with ubuntu:latest, that is, 2022 LTS release Jammy (22.04) using spack like the same way I did in bare-metal. It shows me I am running out of my time and deadline. I found ghcr.io/spack/ubuntu-focal:latest, that is, Ubuntu 20.04 where spack environment is already there. Then I mvoed forward and got success to compile. Then I copied binary and dependecies to ubuntu:latest (22.04) for running.

  That might be misleading in results comparison since in the docker container, Ubuntu 20.04 used for compiling environment and Ubuntu 22.04 used for runtime environment. However, the performance of docker container is very established, and in IBM research it has been shown that Docker is nearly identical to native performance as it has negligible impacts on performace of execution of application comparing to native performance even better than VM.
  According to docker performance in others research I should expect negligible (like 0.1%) impacts of container in both cases for gcc and clang

when running container in Bare-metal. TODO:build and test in container in bare-metal and VM in future.

2 Page 3:

2.1 Section 3.1: Load average

- What can the load average tell us about the overall performance?
  In a short, load average can tell the current CPU utilization. For example, a load of 1.00 is 100% CPU utilization on a single-core system.

  In general, a high CPU load doesn't mean that it negatively impacts a system's performance if it is a short-term occurrence. However, running a CPU at 100% CPU utilization for long periods may have severe impacts on system performance, moreover, a new multithreaded and compute-intensive program is likely to be in waiting queue, may results in high response time and high turnaround time. That is, CPU throughput and perfomance will be low.

  So, it is benificial to debug CPU load average before running a program (multithreaded, may have thread switching, thread synchronization), otherwise, the measurement of overall performance of the program may be inaccurate.

- What should we aim for load-average to be?
  Load average is considered to be ideal if its value is lower than the number of CPUs or cores in the Linux system. Usually,

  Load Average < #CPU or cores *0.7

- When combined with overall wall-clock time, are there any additional metrics we can use to better inform us and compare the performance?
  Yes, I think we should capture serialized (single-thread) runtime for analyzing sacalability, speed up with resource scalling with multithread.

  - **Complexity analysis** of the algorithms to predict runtimes and memory requirements. Analyze the lower and upper bounds to compare the optimality of the algorithm.
  - **Granularity** fine-grain time information (measure execution time of a loop, small codesegment, or even a single instruction) can be added to compare.

- Are there any additional tooling suites we could use for better performance metrics and analysis
  I think memory usage can be added.

  Moreover, since the efficiency of multithread management and synchronizations impacts of the performance, therefore, profiling tools can be used to varify parallel overhead and to pinpoint areas of the program where concurrency happening unexpectedly like threads idleness or overowrking.

- Some profiling tools: Arm-Map, Score-p Scalasca, TAU etc.

2.2 Section 3.1.1:

Total runtime: Before answering the questions in this section, let's first take a look to the original experiment#2 for GCC Vs Clang in **ONLY BARE-METAL** as shown here in Table 1(a). It has been shown clang gets 37.8% speed against gcc. I have choosen this data becasue of low load average, and it can be considered as ideal load average.

Table 1: Experimental Results GCC VS CLANG in Bare-Metal.

GCC vs CLANG in Bare-Metal					
Compilers	Runtime Env	Load Average	Real Time(ns)	CPU Time(ns)	Iterations
GCC	BareMetal	0.69, 0.60, 0.70	1336008819551	1335759811010	55934441
CLANG	Bare-Metal	0.52, 0.65, 1.03	830112010356	829955389480	62061339
100*(gcc-clang)/gcc			37.8663%	37.8664%	-10.9537%
That means runtime 37.8% more when using gcc in comparing to clang. That is clang gets speed up.					

Table 1(a)

Table 2: Experimental Results GCC VS CLANG in Bare-Metal.

GCC (Float Vs Double) Load Average 0.69, 0.60, 0.70			
F	692786995833	692662712386	29871826
D	643221823718	643097098624	26062615
F+D	1336008819551	1335759811010	55934441
100*(D-F)/D	-7.70577%	-7.70733%	-14.6156%

Table 2(a)

CLANG (Float Vs Double) Load Average 0.52, 0.65, 1.03			
F	396967258978	396897660374	34146522
D	433144751378	433057729106	27914817
F+D	830112010356	829955389480	62061339
100*(D-F)/D	8.35229%	8.34994%	-22.324%

Table 2(b)

- Since this is coarse grained across all gate-calls, would it be useful to understand differences on a gate-by-gate basis? Yes, it has been discussed subsequently in the answer of the next question.

- How does the result for 32-bit and 64-bit floats affect the runtime? The Table 2(a) shows runtime of flaot and double for GCC and CLAGN results in the following equations.

$$T_{clang\_double} < T_{gcc\_double} \tag{1}$$

$$T_{clang\_float} < T_{gcc\_float} \tag{2}$$

$$T_{clang\_double} + T_{clang\_float} < T_{gcc\_double} + T_{gcc\_float} \tag{3}$$

$$T_{clang} < T_{gcc} \tag{4}$$

However, look at the following equations (5) and (6). For GCC, the runtime of double < float whereas for CLANG, the runtime of double > float that is usually.

$$T_{gcc\_double} < T_{gcc\_float} \tag{5}$$

$$T_{clang\_double} > T_{clang\_float} \tag{6}$$

So, it implies that few or more or maybe all of gate's float operations take longer times for gcc. There might have several reasons depends on data alignment, data load in register for SIMD, in case multithreading, it is also possible processes and/or threads switching; all are depends on the program implementaion details and how a compiler optimizes and genarates codes.

Before going to see the runtime distrubutions of gate-gate, let's see this situation whether it is consistent in different runs. In the Table 3, you can find that for GCC, the runtime of double < float for all different runs in bare-metal even in container. Whereas for CLANG, the runtime of double > float in both bare-metal and container. That is, the situation is consistent in all experiments.

Now let's dig into gate-gate results.

Table 3: Experimental Results GCC VS CLANG in Bare-Metal.

GCC (Float Vs Double) Load Average: 1.64, 1.64, 1.47			
F	714293711519	714249170357	29385479
D	648118576008	648034929508	25870911
F+D	1362412287527	1362284099865	55256390
100*(D-F)/D	-10.2103	-10.2177	-13.585
CLANG (Float Vs Double) Load Average: 2.37, 1.92, 1.59			
F	394339740673	394313122040	34227054
D	432223408998	432182274639	28513165
F+D	826563149671	826495396679	62740219
100*(D-F)/D	8.76483	8.76231	-20.0395

Table 3(a)

Note that this is new runs gcc followed by clang Intension was to change the load			
GCC (Float Vs Double) Load Average: 5.90, 5.86, 5.84			
F	731241386397	731188250273	30068240
D	654910093391	653388492931	25617506
F+D	1386151479788	1384576743204	55685746
100*(D-F)/D	-11.6552	-11.9071	-17.3738
CLANG (Float Vs Double) Load Average: 6.83, 6.42, 6.08			
F	365084269092	365047471117	35497385
D	444406265327	443711912395	28377947
F+D	809490534419	808759383512	63875332
100*(D-F)/D	17.849	17.7287	-25.0879

Table 3(b)

Experimental Results GCC VS CLANG in Container.

GCC (Float Vs Double) Load Average: 0.62, 0.58, 0.72			
F	694928601793	694822264959	26699401
D	632683116252	632593302736	23985045
F+D	1327611718045	1327415567695	50684446
100*(D-F)/D	-9.83834	-9.83712	-11.3169
CLANG (Float Vs Double) Load Average: 0.70, 0.56, 0.55			
F	429936107415	429856894524	28207497
D	466530716887	466422965889	23409552
F+D	896466824302	896279860413	51617049
100*(D-F)/D	7.84399	7.83968	-20.4957

Table 3(c)

GCC (Float Vs Double) Load Average: 1.00, 0.63, 0.57			
F	704014976046	703887610677	26426472
D	635258782941	635141999001	23490451
F+D	1339273758987	1339029609678	49916923
100*(D-F)/D	-10.8233	-10.8237	-12.4988
CLANG (Float Vs Double) Load Average: 0.89, 0.80, 0.63			
F	435856931533	435800254231	27856599
D	476286447337	476099561948	22908871
F+D	912143378870	911899816179	50765470
100*(D-F)/D	8.48849	8.46447	-21.5974

Table 3(d)

2.3 Gate-Gate

For analyzing of gate-gate results, the total runtime of all 10 operations on a gate is considered. The total raw data size is 3020, totaling 10 operations on each gate results in data set of size 302. Out of 302, one half for float and other half for double of data size 151.

For Gate-Gate analysis, considering that gcc vs clang in bare-metal where ideal load average can be considered as shown in Table 1 and Table 2. On that data set, gate-gate analysis of runtimes for float vs double are shown in Figure 1 and Figure 2. For gcc in Figure 1, the runtimes of 39 gates<float> are higher than that of gates<double>. Some of those are significantly high. For the highest one, the total runtime of SingleQubitOp<float>/PI/6-24/1 is 70% higher than the total runtime of SingleQubitOp<double>/PI/6-24/1.

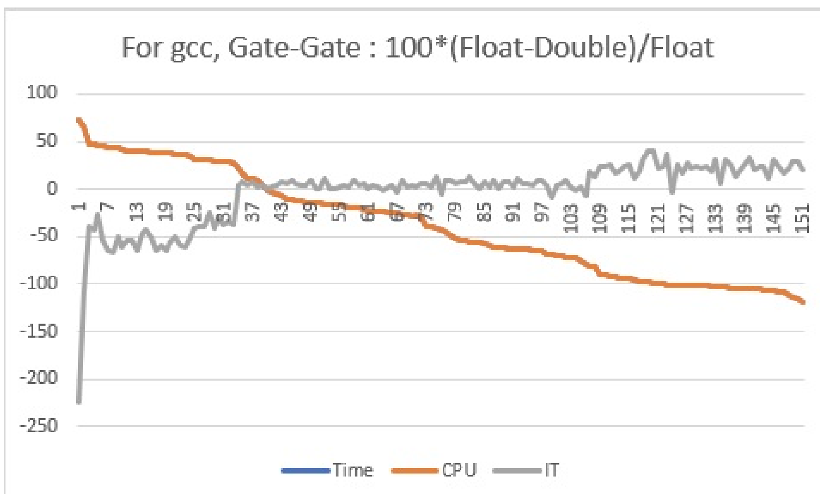

Figure 1: For gcc, Gate-Gate, Float Vs Double.

Whereas, for clang in Figure 2, the runtimes of 20 gates<float> are higher than that of gates<double>. For the highest one, the total runtime of CZ<float>/PI/6-24 is 31% higher than the total runtime of CZ<double>/PI/6-24.

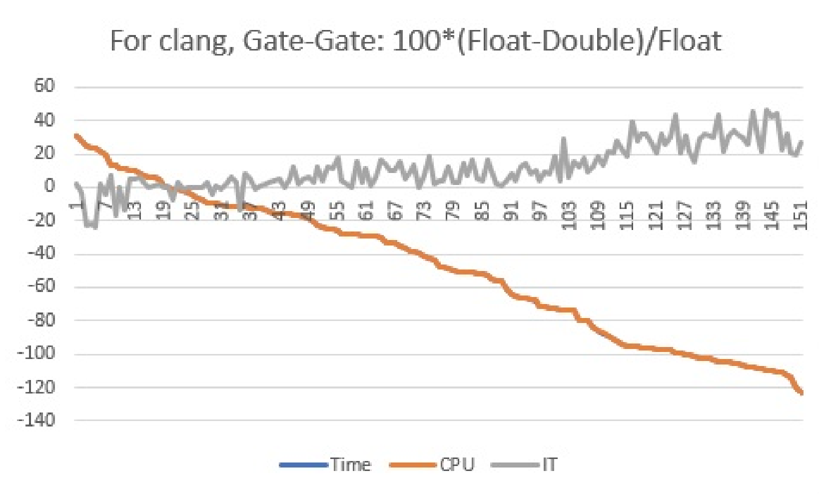

Figure 2: For clang, Gate-Gate, Float Vs Double.

The Table 4 shows the comparison of 2 gates. This is very interesting that gcc and clang act opposite for those two gate operations. At this point, I am wondering that the implementations of some gates are in such a way that gcc and clang generate codes, one is more optimized than the other. It is also possible that gcc and clang are using different optimization level either O2 or O3. Or both are using the same optimization level either O2 or O3 but one is optimizing and the other one is not. Maybe I am worng to complain compilers rather thinking this situation is obvious since I don't know the details of code implementations.

Table 4: GCC VS CLANG in Bare-Metal.

gcc	SingleQubitOp<float>/PI/6-24/1	>	SingleQubitOp<double>/PI/6-24/1
	CZ<float>/PI/6-24	<	CZ<double>/PI/6-24
clang	CZ<float>/PI/6-24	>	CZ<double>/PI/6-24
	SingleQubitOp<float>/PI/6-24/1	<	SingleQubitOp<double>/PI/6-24/1

If this is completely unexpected then those gate's operations can be used as a variant for gcc and clang performance analysis, considering that minimal optimal algorithm is used to implement those gates.

- Are there any challenges with vectorization of complex numbers in single or double precision? (e.g. how do the AVX2/AVX512 kernels differ from the default kernels, and why do they differ?)
  It seems all of the concerns are hidden in these questions. Yes, vectorization of complex numbers are challenging. I have tried to explain very briefly in the subsequent sections.

2.4 Vectorization, AVX2 Vs AVX512, Single Vs Double precision complex number

AVX2 and AVX512 are extensions to the x86 instruction.
32 bits for single precision floating-point number
64 bits for double precision floating-point number

2.4.1 AVX2

AVX2 works with 256 bits register that is, we can load in a register
$256/32 = 8$ single precision floating-point numbers or
$256/64 = 4$ double precision floating-point numbers

A complex number has two parts: real and imaginary, so we can load in a register
$8/2 = 4$ single precision complex numbers or
$4/2 = 2$ double precision complex numbers.

The Table 5 shows for AVX2, how data can be vectorized in array[] and loaded in a register for SIMD.

2.4.2 AVX512

AVX2 works with 512 bits register that is, we can load in a register
$512/32 = 16$ single precision floating-point numbers or
$512/64 = 8$ double precision floating-point numbers

A complex number has two parts: real and imaginary, so we can load in a register
$16/2 = 8$ single precision complex numbers or
$8/2 = 4$ double precision complex numbers.

The Table 6 shows for AVX512, how data can be vectorized and loaded in register for SIMD.

So with a single instruction, 2x data load and process in AVX512 comparing to AVX2.

Table 5: Vectorization of single or double precision complex number in AXV2.

<— 256bits Register —>							
4 single precision floating-point complex numbers can be loaded							
$32bits$	$32bits$	$32bits$	$32bits$	$32bits$	$32bits$	$32bits$	$32bits$
$complex_1$		$complex_2$		$complex_3$		$complex_4$	
re	im	re	im	re	im	re	im
$d[i=0]$	$d[i=1]$	$d[i=2]$	$d[i=3]$	$d[i=4]$	$d[i=5]$	$d[i=6]$	$d[i=7]$
2 single-qubit states or 1 2-qubit state can be loaded							
$single-qubit\_state_1$				$single-qubit\_state_2$			
$two-qubit\_state$							
2 double precision floating-point complex numbers can be loaded							
$64bits$		$64bits$		$64bits$		$64bits$	
$complex_1$				$complex_2$			
re		im		re		im	
$d[i=0]$		$d[i=1]$		$d[i=2]$		$d[i=3]$	
1 single-qubit states can be loaded							
$single\_qubit\_quantum\_state$							

Table 6: Vectorization of single or double precision complex number in AXV512.

<— 512bits Register —>							
8 single precision floating-point complex numbers can be loaded							
$64bits$	$64bits$	$64bits$	$64bits$	$64bits$	$64bits$	$64bits$	$64bits$
$complex_1$	$complex_2$	$complex_3$	$complex_4$	$complex_5$	$complex_6$	$complex_7$	$complex_8$
$re.im$	$re.im$	$re.im$	$re.im$	$re.im$	$re.im$	$re.im$	$re.im$
$d[i=0,1]$	$d[i=2,3]$	$d[i=4,5]$	$d[i=6,7]$	$d[i=8,9]$	$d[i=10,11]$	$d[i=12,13]$	$d[i=14,15]$
4 single-qubit states or 2 two-qubit states or 1 four-qubit state can be loaded							
$single-qubit\_state_1$		$single-qubit\_state_2$		$single-qubit\_state_3$		$single-qubit\_state_4$	
$two-qubit\_state_1$				$two-qubit\_state_2$			
$four-qubit\_state$							
4 double precision floating-point complex numbers can be loaded							
$128bits$		$128bits$		$128bits$		$128bits$	
$complex_1$		$complex_2$		$complex_3$		$complex_4$	
$re.im$		$re.im$		$re.im$		$re.im$	
$d[i=0,1]$		$d[i=2,3]$		$d[i=4,5]$		$d[i=6,7]$	
2 single-qubit states or 1 two-qubit state can be loaded							
$single-qubit\_state_1$				$single-qubit\_state_2$			
$two-qubit\_state$							

2.5 AVX2/AVX512 kernels Vs Default Kernels with GCC Vs Clang

2.5.1 GCC AVX2/AVX512 kernels Vs Clang AVX2/AVX512 kernels

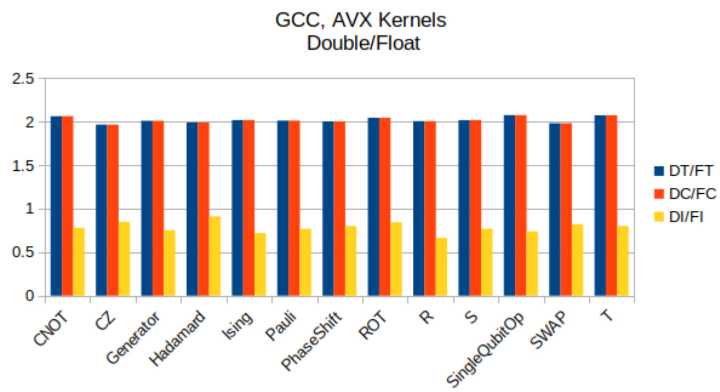

Figure 3: For GCC, AVX2/AVX512 kernels (Float Vs Double).

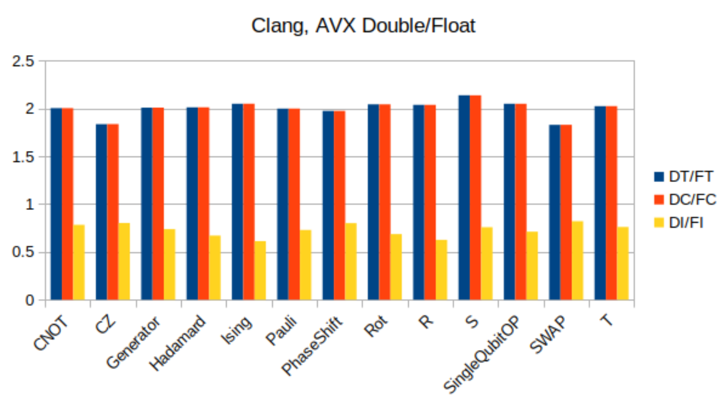

Figure 4: For Clang, AVX2/AVX512 kernels (Float Vs Double).

From Figure 3 and Figure 4, it can be realized that the runtimes of Double is on average 2x of the runtime of float. This is consistent for both $Time$ and CPU (Note that $Time$ and CPU refers to the Time and CPU in origin data and if I am not worng Time=Real Time and CPU=CPU usage, also in legend DT=$Time$ for double, FT=$Time$ for float, DC=CPU for double, FC=CPU for float). So for AVX2/AVX512 kernels, both compilers have similar performance with the exception of CZ, S and SWAP gates.

2.5.2 GCC Default kernels Vs Clang Default kernels

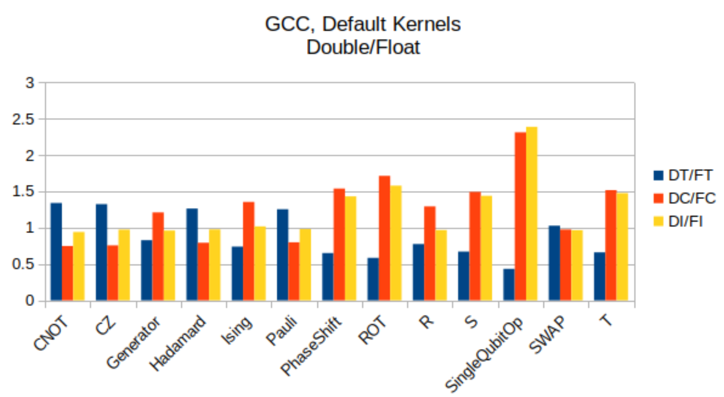

Figure 5: For GCC, Default kernels (Float Vs Double).

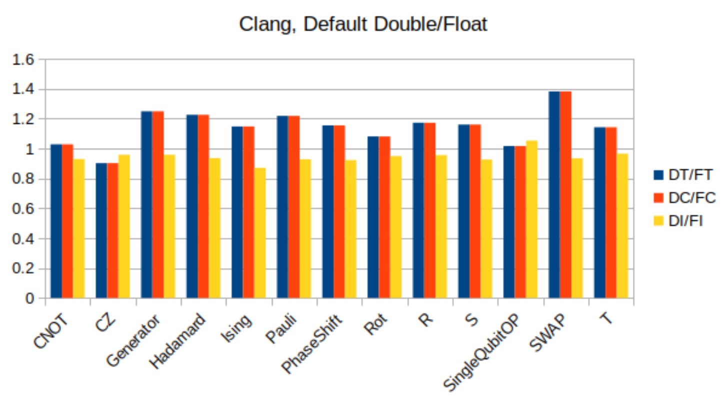

Figure 6: For Clang, Default kernels (Float Vs Double).

From Figure 5, it can be realized that for gcc the runtime of double and float are high and low for both $Time$ and CPU. Out of 13 cases, only 4 cases the runtime $Time$ double > float, however, in 8 cases, the runtime CPU double > float. That means actual data computaton is interrupted, maybe, for more data load/store access than computaton since data are not aligned.

On the other hand From Figure 6 it can be realized that for clang the runtime of double > float for almost all the cases. This is consistent for both $Time$ and CPU. Let's see actual runtimes in the next page.

In comparing Figure 7 and Figure 8 (for each gate read from left as gcc-clang in legend F=float, D=double), it can be realized that for both GCC and Clang, the runtimes for some gates are significantly higher in Default kernels than that of AVX2/512 kernels. It is 10x for ROT and Single-QubitOp gates for GCC.

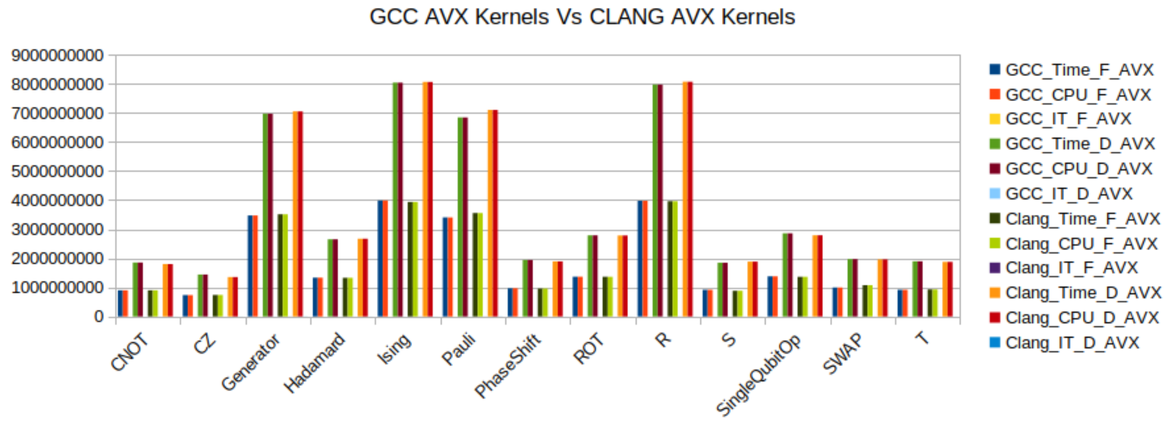

Figure 7: GCC AVX2/512 Vs Clang AVX2/512 Kernels.

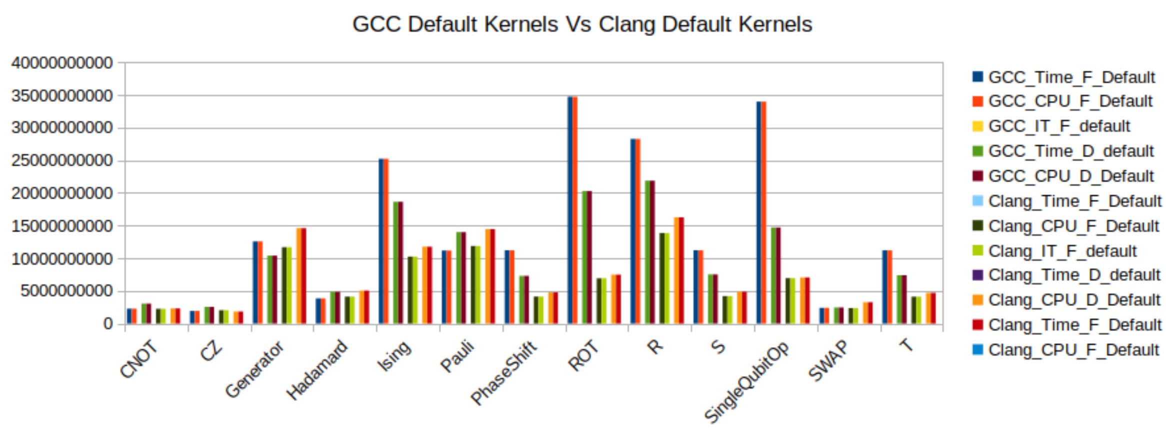

Figure 8: GCC Default Vs Clang Default Kernels.

So for AVX2/512 kernels both gcc and clang have equal performance. Whereas for default kernels clang performance is better than gcc. **I read in the documetation**
https://docs.pennylane.ai/projects/lightning/en/stable/avx_kernels/build_system.html

Default kernels and AVX2/512 kernels are build with different compiler options for a specific reason. That might be the root cause.

Runtime overhead:

- What may cause this overhead?
  The point in the original report was **"In bare-metal, the CPU load averages in both cases is greater than 1, therefore, both cases runtime incurs an extra overhead".** Yes I mean that, context switching between processes are happening, other processes affecting the results and incurring an extra overhead. See the results of Table 4(b) where the highest load average in this experiment shown and we can find the highest runtimes of these experiments. Maybe "extra overhead" is not good wording to use here.

- Is it likely that congestion from other processes can affect this result?
  Yes, I mean that.

- How can multi-threaded libraries affect this result?
  I think you mean external libraries like openMP, or others? It depends on which one is using and how threads are being spawn for loop parallizing, explicit synchronizations requiring in case of mutual exclusion of threads.

clang > gcc

- Will this be generally true? No, this is not. Depends on the programs and system architectures, it varies with optimiztion level -O2 and -O3 that I found in research.

- Could there be something else influencing these numbers?
  So in one of my experiments in bare-metal, it showed 39.33% speed up in total runtimes for clang compiler against gcc compiler. However, it is not true in general as I found it varies in others of my experiments.

- How can we tell and be certain that clang performance is generally better than gcc performance? It is very hard to be certain. It seems that it is all about vectorization and setting optimization level as well as Modern CPU Architecture. For gcc, vectorization is enable at level -02 so if gcc with optimization level -O3 (-O2 + other options), it may not improve much. However, if optimization level -O3, clang might further improve than -O2. So the chances for performance clang > gcc are on level -O3. Again no guarantee.

3 Page 4:

3.1 Section 3.1.2: Container runtime

- Container runtime has less overhead: Why is this the case?
  I thing this sentence should be "The runtime of program in container

has less overhead." This is becasue, when running in container, the CPU load average <1.00, 0.63, 0.57> which is less than the CPU load average <1.64, 1.64, 1.47> when running in bare-metal. Considering that less load average incurs less overhead in comparison.

4 Page 5:

4.1 Section 3.2:

- Is this a repeat of experiment 3.1 without any additional changes?
  Yes, it is a repeat of experiment without any modification

5 END

- For the Spack work, I am also curious about your thoughts about using it for end-to-end builds with specific compilers. Did you think it worked well, worked poorly, or something in between?

  First of all, for the assignment, it was the first time I used spack. So it was the battle place where I spent most of my time to figure out how spack can be automated to build a new package. Sometimes it takes long time to install gcc-11.3.0 and llvm-15.0.6, in my old system automating spack I saw like the followings, it seems too much time to install dependecies.

  gcc: Successfully installed gcc-11.3.0-nlewpm4kqv62wkfv7beakalrggnaqbm3
  Build: 55m 16.90s. Total: 56m 5.13s

  llvm: Successfully installed llvm-15.0.0-3lheniizmenuue36jhltgtlkxxcwdjll
  Build: 1h 51m 19.95s. Total: 1h 51m 27.22s.

  But since it was my first time, I feel like too complicated to use by end-users. Have you ever think of that you are compiling a program and all of /tmp/user_name directory space occupied and reporting low disk space? That's the reason I could not attempt to rebuild everything (as mention at the begining TODO) in the server since I have less freedom to use it.

  At this point, I would go in middle. It is a good HPC tools, so I think it is better to have more familiar, practice, use it to make the confident for efficiently use it.

My Comments: It is a great experience with a lot of clarity and learning curves. I would be very happy to discuss if any further.