Heterogenous Inferencing

Optimierte Verteilungen der Operationen auf verschiedene Beschleuniger

Martin Barthel
Magdeburg, Germany
martin.barthel@st.ovgu.de

Nils Treuheit
Magdeburg, Germany
nils.treuheit@st.ovgu.de

***Zusammenfassung*—Der Aufwand soll sowohl beim Training als auch bei Inferenzen auf künstlichen Neuronalen Netzen soweit wie möglich minimiert werden, um Ressourcen zu schonen. Insbesondere bei Heterogeneous Inferencing Setups muss die Performance der einzelnen Geräte bestimmt werden, damit die Ausführung der Inferenzoperationen der Netzwerk-Layer pareto-optimal auf die verschiedenen Hardwarekomponenten verteilt werden kann. Dazu misst man beispielsweise Energieverbrauch und Durchsatz und kann so bestimmen, welche Geräte sich besonders für spezifische Einsatzzwecke eignen. Innerhalb bestehender Arbeiten wurden vor allem Benchmarks für komplette Netze durchgeführt, ohne dabei Messungen auf Layer-Ebene anzustellen, sodass Aussagen über die Eignung der Geräte für spezifische Operationen nur schwierig möglich sind. Wir haben im Folgenden die Leistung von Standard-Layern auf gewöhnlicher Computerhardware (CPU und GPU) und "Low-Power" Beschleunigern untersucht. Wir stellten fest, dass USB-Beschleuniger insbesondere eingesetzt werden sollten, wenn ein möglichst geringer Energieverbrauch einen besonderen Vorrang besitzt oder andere Hardware bereits von anderen Prozessen ausgelastet wird. In den meisten Fällen sind die Laufzeiten der Inferenzen auf leistungsstärkeren Standard-Computerkomponenten deutlich geringer als die auf "Low-Power" Beschleunigern. Deshalb ist es sinnvoll ein Heterogeneous Inferencing Setup zu implementieren, welches auf Grund der aktuellen Hardware-Auslastung und einer Kompatibilitätsmatrix Inferenzoperationen auf unterschiedliche Hardwarekomponenten des Systems verteilt.**

***Index Terms*—Heterogenous Inferencing, Neuronale Netze, AI, Hardware Beschleuniger, TPU, MYRIAD, Layer Approach, Performance Benchmarks**

I. Einleitung

Heutzutage werden künstliche Intelligenzen immer häufiger eingesetzt um komplexe Daten getriebene Probleme zu lösen.

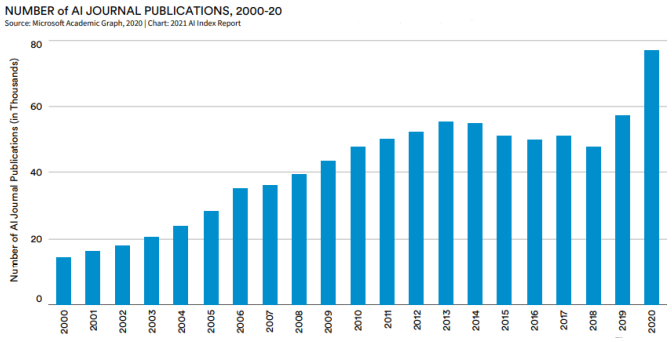

Abbildung 1. Statistik über globale akademische Publikationen in künstliche Intelligenz aus dem Artificial Intelligence Index Report 2021 [18]

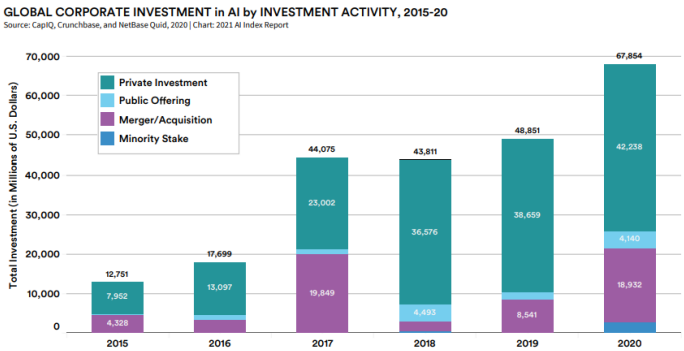

Abbildung 2. Statistik über globale Investitionen in künstliche Intelligenz aus dem Artificial Intelligence Index Report 2021 [18]

Das möglichst effiziente Bewerkstelligen von schwierigen Aufgaben wie die Klassifizierung und Detektion von Objekten oder aber auch die Mustersuche in Daten und das Clustern von Objekten motivieren die ständige Weiterentwicklung von angelernten künstlichen Intelligenzen. Zhang et al. zeigten in ihrem jährlichen Report Ende 2021 das stetig weiterwachsende Interesse an künstlich intelligenten Systemen [18]. Jahr für Jahr steigt die akademische und ökonomische Popularität künstlich intelligenter Systeme dargestellt in Abbildung 1 und Abbildung 2. In den letzten Jahren gewann vor allem die Forschung im Bereich Deep-Learning immer mehr an Bedeutung [18]. Somit liegen neuronale Netze seit einigen Jahren wieder im Fokus der Forschung und Entwicklung im Bereich des maschinellen Lernens. Allerdings ist das Training von neuronalen Netzen ein oft sehr langwieriger und vor allem rechenintensiver Prozess. Dies ist einer der Gründe, warum man im Anschluss die Inferenzen auf dem trainierten Netzwerk möglichst effizient ausführen möchte. Dafür ist es wichtig die Laufzeit und den Stromverbrauch so gering wie möglich zu halten. Inferenzen sind sehr viel weniger rechenintensiv als das Training, da es sich hier nur um einen einfachen Durchlauf durch das neuronale Netz handelt und keine "Error Backpropagation" und Anpassungen an den Gewichten des neuronalen Netzes nötig sind. Aus diesem Grund ist es auch möglich Hardware-Beschleuniger für die Inferenz über neuronale Netze zu entwickeln. Da maschinelles Lernen ein Teil vieler großer finanziell lukrativer Anwendungen und Prozesse in der Industrie ist, investieren große Unternehmen wie Intel, Google, Qualcomm, Amazon, Apple, Facebook, AMD und Samsung viel Geld

in die Entwicklung von möglichst effizienten Hardware-Beschleuniger für künstliche Intelligenz [14], [15]. Diese Vielzahl Beschleuniger weisen jedoch unterschiedliche Charakteristiken auf. Die Frage bleibt aber in wie weit es sich lohnt speziell optimierte Hardware-Beschleuniger anstelle von konventioneller "general purpose" Hardware (CPU, GPU) für die Inferenz zu benutzen. Wir wollen in diesem Projekt herausfinden, welche Hardware-Beschleuniger sich am besten für spezielle Operationen einer Inferenz eignen und eine Empfehlung für ein effizientes Heterogenous Inferencing Setup erstellen. Dazu schauen wir uns Coral's TPU [16] und Intel's VPU in Form von "Low-Power" USB Accelerators genauer an.

II. Grundlagen

Neuronale Netze werden in verschiedene Kategorien unterteilt, wie zum Beispiel Multilayer Perceptrons [4]. Diese besitzen viele aufeinander folgende Hidden Layer. Als Hidden Layer werden Operationen auf den ihnen übergebenen Daten bezeichnet, deren Ergebnis nur einen Zwischenschritt darstellt. Dem Output Layer kann dann die Ausgabe des Netzes entnommen werden. Berechnungen der Layer nutzen nur Daten von vorherigen Layern, die näher an der Eingabe liegen als sie. Die Ausgabe wurde vom Netz den Eingabedaten zugeordnet, die dem Input Layer übergeben wurden. In Abbildung 3 sieht man den Aufbau eines einfachen Multi Layer Perceptron nach Krenker et al [7]. In diesem Fall ist gibt es nur einen Hidden Layer, wie ihn Goodfellow et al. beschreibt [4]. Die Daten aus dem dreielementigen Eingabevektor werden an diesen weitergegeben. Veranschaulichend kann gesagt werden, dass es in einer Schicht Neuronen gibt, die jeweils jeden Eingabewert gewichten. Diesen Layertyp nennen wir Vollverbundenen oder Dense Layer. Außerdem besitzen die Neuronen eine Aktivierungsfunktion wie ReLU, die auf die Summe der gewichteten Inputs angewendet wird. Im Folgenden wird auch eine alleinstehende Aktivierungsfunktion als Layer angesehen. Im Beispielnetz existiert ein Neuron im Output Layer, mithilfe dessen das Ergebnis bestimmt werden kann.

Eng verwandt mit Multilayer Perceptrons sind Convolutional Neuronal Networks. Diese werden vor allem bei der Bilderkennung eingesetzt. Außerdem können Convolutional

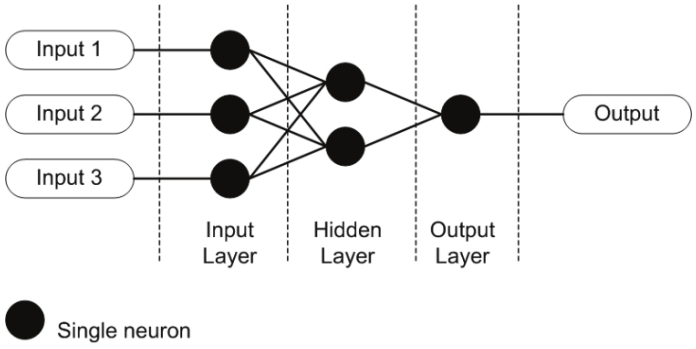

Abbildung 3. Schema einer Neuronalen Netzwerkschicht [7]

Neuronal Networks auf wiederkehrenden Messergebnissen eingesetzt werden, die dem Netzwerk dann zusammen als Array übergeben werden. Der Name dieser Netze leitet sich von der Convolutions-Funktion ab. Diese Operation wird bei diesen dort durchgeführt, wo sonst die Gewichtung und Summierung der Eingaben innerhalb der Neuronen in einem Vollverbundenen Layer erfolgte, falls es sich bei der Schicht um einen Convolutional Layer handelt. Anhand der Ergebnisse der später beschriebenen Convolution wird daher auch meist eine Aktivierung berechnet, anstatt das Ergebnis direkt weiter zu verwenden. Außerdem kann deren Rückgabewert dann durch Pooling verarbeitet werden. Unter Pooling-Operationen fallen unter anderem die Rückgabe des Durchschnitts von Werten oder des höchsten Wertes jeweils auf einen Bereich des Eingabearrays bezogen. Convolutional Neuronal Networks sind in gewisser Weise an Forschungsergebnisse im Bereich der Bilderkennung innerhalb von Gehirnen angelehnt.

Wichtige Beispiele für Operationen, die in Neuronalen Netzen genutzt werden, werden im Folgenden aufgelistet.

A. Vollverbundene Layer

Wichtiger Bestandteil von Multi Layer Perceptrons sind Layer, bei denen wie im Beispiel jedes Neuron mit jeder Eingabe verbunden ist. Sie werden durch eine Matrix-Vektor- beziehungsweise eine Matrix-Matrix-Multiplikation realisiert. Die von Neuronen abgeleiteten Elemente des Layers können bei diesem auf jeweils alle Werte des Eingabearrays zugreifen. Jedes Element von dessen Elementen multiplizieren sie mit einem Gewicht und bilden die Summe aus diesen und einer Zahl, dem Bias. Letzterer und die Gewichte können pro Neuron verschieden sein.

B. Aktivierungsfunktion

In den Layern wird auf die Ergebnisse der Matrix-Multiplikation der vollverbundenen Layer eine Aktivierungsfunktion angewendet. Heutzutage wird bei diesen oft auf ReLU zurückgegriffen. ReLU steht für Rectified Linear Unit. Für jeden negativen Eingabewert gibt diese null zurück, bzw. den Wert selbst, falls dieser größer null ist. Eine weitere, verwandte Aktivierungsfunktion ist Leaky ReLU. Diese Operation wird definiert mit $f(x) = c \cdot min(x, 0) + max(x, 0)$. Als Beispiel für c wird bei Goodfellow et al 0.01 angegeben. Durch die Erweiterung von ReLU um das gewichtete Minimum verändert sich die Ableitung der Aktivierungsfunktion. Dadurch können während des Trainings Gewichte vorhergehender Operationen auch in Fällen angepasst werden, wo dies mit ReLU unterbleibt. Bei ReLU ist dies bei Werten kleiner gleich null nicht möglich.

Alternativ kann tanh auf den Ergebnisse eines Vollverbundenen Layers angewendet werden. Die Änderungsrate von $f(x) = tanh(x)$ ist an den meisten Punkten gering, nur wenn x einen geringen Betrag hat, ist diese hoch. Diese Eigenschaft erschwert die Anpassung der Gewichte des Netzes, in dem sich tanh befindet. Allerdings gibt die Aktivierungsfunktion beim Koordinatenursprung annähernd die gleichen Werte zurück,

wie ihr übergeben wurden, was die Anpassung der Gewichte des Netzes dort unterstützt.

Als letztes soll noch auf die Sigmoid Aktivierungsfunktion eingegangen werden. Die Formel zur Berechnung dieser Aktivierungsfunktion ist $\frac{1}{1+e^{-x}}$. Genauso wie die tanh Aktivierungsfunktion hatte sie eine hohe Verbreitung, wurde aber von ReLU abgelöst. Auch hier gelten die Bemerkungen wie bei tanh zur Änderungsrate, ohne das sigmoid Eigenschaften vergleichbar günstig an die von $f(x) = x$ heran reichen wie bei tanh. Einsatzgebiete sind eher im Bereichen wie Rekurrente Netze zu finden.

C. Convolutional Layer

Im Namen der Convolutional Neural Networks ist bereits die eingehend angesprochene Convolution enthalten. Mathematisch ist diese durch $k(x) = \int g(x-y) \cdot f(y)dy$ definiert, wenn g die Gewichtung darstellt und f die Eingabedaten an der übergebenen Position angibt. Die tatsächlich genutzte Funktion arbeitet jedoch meist mit Summen, da die an den Layer übergebenen Daten und die entsprechenden Gewichte nur eine bestimmte Anzahl von Werten umfassen. Nur dort sind f und g ungleich null. Es wird ein sogenannter Kernel verwendet, der die Gewichte beinhaltet. Dieser enthält meistens weniger Werte als das Array, auf das er angewendet wird. Die Gewichte des Kernels werden mit einer Untermenge der Eingabedaten multipliziert, um an das Ergebnis der Convolution an einem Punkt zu gelangen. Dazu wird folgende Formel verwendet: $k(x) = \sum g(x-y) \cdot f(y)$. Theoretisch müsste dabei y alle Werte aus $\mathbb{R}$ annehmen. Durch die eben vorgestellte Beschränkungen der Zahl der Gewichte ist es aber ausreichend, sich auf diese zur Berechnung des Ergebnisses zu beschränken.
Es gibt verschiedene Möglichkeiten, das Verhalten der Convolution anzupassen. In den meisten Fällen wird nicht nur eine Gewichtsmatrix an die Eingabe des Layers angelegt, sondern mehrere pro Layer. In praktisch eingesetzten Netzen enthalten so die Ergebnisse mehr Informationen.

Außerdem ist es möglich, den Kernel nicht an alle möglichen Positionen an den Eingabearray anzulegen. Dann wird für die nächste Berechnung eines Wertes der Ausgabe der Kernel nicht um 1 sondern n in die jeweilige Richtung verschoben. Die gewählte Zahl gibt dann den Stride an. In diesem Fall wäre dies n.

III. Aufbau des Experiments

A. Modelle

Alle der oben genannten Aktivierungsfunktionen setzten wir in Tensorflow-Modelle um. Diese Modelle sind folglich ReLU, Leaky ReLU, tanh und sigmoid, wie in der Modellübersicht zu sehen ist. Diese erhielten alle pro Inferenz einen 128 elementigen Eingabevektor. Zusätzlich können auch andere Funktionen in Neuronale Netze integriert werden, wie in unserem Fall das Modell Scalar Multiplication, welches alle Eingaben jeweils mit fünf multipliziert. Als Eingabe wurde auch hier ein eindimensionaler Array aus 128 Werten genutzt.

Layer Network	Operation	Eingabe
ReLU	ReLU	128
Leaky ReLU	Leaky ReLU	128
tanh	tanh	128
sigmoid	sigmoid	128
Scalar Multiplication	Lamda ($f(x) = 5x$)	128
Small Dense	Vollverbundener Layer mit 8 Neuronen	128
Big Dense	Vollverbundener Layer mit 512 Neuronen	128
Simple 2D-Conv.	Convolution mit 12 Filtern und Kernelgröße 3x3	128x128x3
Dilated 2D-Conv.	Convolution mit 12 Filtern, Kernelgröße 3x3 und einer Dilation von 2	128x128x3
Strided 2D-Conv.	Convolution mit 12 Filtern, Kernelgröße 3x3 und einem Stride von 5	128x128x3
Small Kernel 2D-Conv.	Convolution mit 9 Filtern und Kernelgröße 3x3	128x128x3
Big Kernel 2D-Conv.	Convolution mit 9 Filtern und Kernelgröße 7x7	128x128x3
Few Filter 2D-Conv.	Convolution mit 3 Filtern und Kernelgröße 3x3	128x128x3
Many Filter 2D-Conv.	Convolution mit 256 Filtern und Kernelgröße 3x3	128x128x3

Tabelle I
Verwendete Modelle

Vollverbundene Layer werden durch Small Dense und Big Dense repräsentiert. Small Dense besitzt 8 Neuronen, Big Dense 512, um festzustellen, welchen Effekt die Veränderung der Anzahl dieser hat. Auch diesen wird ein Array mit der gleichen Größe wie das vorherige übergeben. Im Fall der Layer, die zum Testen der verschiedenen Konfigurationen der Convolution verwendet werden, wurden Eingaben, die einem typischen Bild ähneln, verwendet. Diese waren 128x128x3 große Arrays. Eines der Convolutions-Modelle ist Simple 2D-Conv.. Dieses nutzt 12 Filter und einen zweidimensionalen Kernel mit jeweils drei Elementen pro Zeile in allen drei Spalten. Strided 2D-Conv. nutzt die gleiche Kerneanzahl und -größe, wie alle hier verwendeten Convolutional Networks mit Ausnahme von Big Kernel 2D-Conv., jedoch wird der Kernel um jeweils fünf in x- und y-Richtung verschoben. Dilated 2D-Conv. sticht dadurch hervor, dass das Parameter Dilation auf zwei gesetzt wurde. Bei Small Kernel 2D-Conv. und Big Kernel 2D-Conv. war das Parameter von Interesse die Kernelgröße, die im Fall von Big Kernel 2D-Conv. auf 7x7 erhöht wurde. Diese beiden Modelle setzten nur neun Filter ein. Abschließend wurden Few Filter 2D-Conv. und Many Filter 2D-Conv. dazu genutzt, den Einfluss der Anzahl der Filter auf die Messgrößen zu untersuchen. Modelle und Benchmarkskripte sind auf GitHub [17] abgelegt wurden.

B. Verwendete Geräte

Für Experimente wurden unter anderem ein Intel Neural Compute Stick 2 verwendet. Weiterhin kam ein Google Edge TPU ML Accelerator zum Einsatz. Der Rechner, an den die USB-Beschleuniger angeschlossen waren, besaß 16 Gigabyte Arbeitsspeicher. Der Prozessor war ein Intel Core I5-7500, welcher eine Intel HD-Grafik 630 als integrierte Grafikkarte für die Messungen bereitstellte, die unter Ubuntu 20.04 stattfanden. Den USB-Beschleunigern stand bei diesen ein USB 3.0 Anschluss zur Verfügung. Zum Auslesen der Wattstunden wurde ein USB Safety Tester genutzt. Dessen Messwerte wurden mit der OpenKamera App auf einem Tablet und ffmpeg 4.3.1 verarbeitet. Für Inferenzen auf der Grafikkarte, dem Prozessor und dem Intel Neural Compute Stick 2 wurde der Openvino-Toolkit in Version 2021.4 verwendet. Für den Edge TPU Beschleuniger dagegen wurde TensorFlow Lite in Version 2.5.0 und das libedgetpu1-std Paket in Version 16.0 eingesetzt.

C. Ablauf des Experiments

Zur Durchführung der Benchmarks wurde zuerst TensorFlow Modelle der Layer aus Tabelle 1 auf die Festplatte geschrieben [2]. Diese mussten anschließend, um auf die Edge TPU geladen zu werden, einer sogenannten Quantisierung unterzogen werden und als TensorFlow Lite Modell gespeichert werden. Danach wird der Edge TPU Compiler auf ihnen ausgeführt. Für Grafikkarte, Prozessor und Intel Neural Compute Stick 2 musste Modelle im richtigen Format für Openvino vom von diesem Toolkit mitgelieferten Compiler generiert werden [11]. Dort wurde der Datentyp auf FP16 gestellt, da dieser auf allen Geräten unterstützt wurde [13], und im Sinne von vergleichbareren Messergebnissen. Openvino unterstützt laut Dokumentation neben synchronen auch asynchrone Inferenzen [12], die auf anderen Plattformen nicht verfügbar zu sein scheinen, sodass wir für die im Paper beschriebenen Messungen nur synchrone Inferenzen nutzen.

1) Overhead: Um den Overhead bzw. die Latenz, die das Laden des Netzes braucht, zu bestimmen, ist pro Layer jeweils ein zusätzliches Modell mit drei Schichten der gleichen Operation hinzugefügt wurden. Durch Hintereinanderkoppelung der Layer sollte die Zeit ohne Overhead, zumindest bei den Modellen mit dem 128 Element Array als Eingabe, jeweils linear mit der Layeranzahl wachsen. Die Zahl, um die diese abweicht, kann anschließend zur Berechnung des Overheads genutzt werden. Bis auf Strided 2D-Conv. wurde auch überall ein drittes Modell mit acht Layern ergänzt. Auch auf diesen wurden die im Nachfolgenden beschriebenen Durchläufe ausgeführt, nachdem sie wie oben beschrieben gespeichert und für Edge TPU und Intel Neural Compute Stick 2 übersetzt wurden. Um den Overhead, genannt o, zu berechnen wurde folgende Formel verwendet:

$$f(t_1, t_3, t_8) = \frac{(((t_1 - \frac{t_8}{8}) \cdot \frac{8}{14} + (t_1 - \frac{t_3}{3}) \cdot \frac{3}{4}) + (8t_1 - t_8) - (3t_1 - t_3) \cdot 3)}{2}$$

.

Dabei gibt t_1 die Laufzeit für ein Modell mit einem Layer der gewählten Operation an. t_3 und t_8 stellen die Zeiten der Modelle mit mehr Layern da. Um die in jedem Fall notwendige Latenz für eine Inferenz, d.h. den Overhead, zu berechnen, wurde an dieser Stelle der Mittelwert zwischen zwei Zwischenergebnissen bestimmt. Das Erste setzt sich aus den Differenzen zwischen der für einen Layer nötigen Zeit und der Dauer, die man ausgehend von den Messungen für drei- bzw. achtschichtige Modelle für nur einen Layer erwarten würde ($(((t_1 - \frac{t_8}{8}) \cdot \frac{8}{14} + (t_1 - \frac{t_3}{3}) \cdot \frac{3}{4})$) zusammen. Diese beiden Differenzen wurden danach gewichtet addiert. Für das zweite Teilergebnis wurde, ausgehend von einschichtigen Modellen, die erwartete Zeit für mehrschichtige Modelle berechnet ($(8t_1 - t_8) - (3t_1 - t_3) \cdot 3$). Dann wurde die Differenz zwischen den gemessenen Werten gebildet.
Der Overhead wurde nicht für Convolutional Layer bestimmt. Außerdem wurde dieser nicht für die beiden Vollverbundenen Layer berechnet. Bei genannten Operationen unterscheiden sich die Anzahl der nötigen Operationen pro Layer, was die Berechnungen schwierig macht. Bei Small Dense und Big Dense tritt dies dadurch auf, dass sich Zahl der Eingabewerte und der Neuronen im zweiten Layer unterscheiden. Die Laufzeiten dieser gestapelten Modelle liegen trotzdem vor. Im Folgenden werden Overhead und Latenz synonym verwendet.

2) Laufzeit einer einzelnen Inferenz: Eine weitere zu bestimmende Größe war die Laufzeit einer Inferenz. Während des Experiments wurde jedes Neuronale Netz 32 Mal auf ein Gerät geladen. Nach allen Ladevorgängen wurden immer 64 einzelne Inferenzen ausgeführt. Vor und nach den Inferenzen

wurde jeweils die Zeit festgehalten, sodass die Differenz die benötigte Dauer zum Berechnen des Ergebnisses aus den Eingaben ergab.

3) Messung über mehrere Inferenzen hinweg und Durchsatzbestimmung: Ein weiterer Messwert ist der Durchsatz gewesen. Diesen haben wir indirekt aus der Zeit zwischen Anfang und Ende von 63 Inferenzen ermittelt. Auch in diesem Fall wurden 32 Durchgänge pro Netzwerk genutzt. Der Durchsatz kann anschließend mit der Formel

$$f(t,l) = \frac{1}{\frac{8t_1 + \frac{t_3 - l}{3} + \frac{t_8 - l}{8} + 2l}{10}}$$

berechnet werden, wobei l die vorhergehend bestimmte, immer vorhandene Latenz angibt. Weiterhin führten wir vor diesen Inferenzen pro Durchgang jeweils eine Inferenz durch. Zumindest im Fall der Edge TPU dauert diese laut Dokumentation besonders lange [1]. Um festzustellen, ob dies auch auf Openvino Geräten auftritt und welche Größenordnung diese aufweist, wurde auch deren Länge erfasst.

4) Messungen des Energieverbrauchs: Energiemessungen für den Neural Compute Stick 2 und den Google Edge TPU ML Accelerator erfolgten, indem diese mit dem Rechner über den USB Safety Tester verbunden wurden. Danach wurde die Aufnahme mit dem Tablet gestartet. Der Zeitpunkt des Beginns des Videos wurde festgehalten. Danach wurden die Start- und Endzeiten der Blöcke von Inferenzen erfasst. Relativ zu diesen wurde der Zeitpunkt im Video ermittelt, und die Bilder mit den Wattstunden am Anfang und Ende extrahiert, und dann die Differenz daraus gebildet. Hierbei wurden die Netze jeweils acht Mal auf die beiden USB Beschleuniger geladen. Um die Differenz zu messen, wurden 8192 einzelne Inferenzen zwischen der Ermittelung des Anfangs- und Startwert ausgeführt. Im Fall des Intel Neural Compute Stick 2 wurde für jedes Netz nachträglich ein zusätzlicher Durchgang mit 8192 Inferenzen durchgeführt. Diese wurden genutzt, falls Messwerte, zum Beispiel durch Überschreiten der maximal möglichen Videogröße, verloren gingen.

IV. Evaluation

Zu allen Messungen im Rahmen unserer Benchmarks haben wir im Anhang die Ergebnisse in Form von Violin-Plots bereit gestellt. Darüber hinaus sind detaillierte Ergebnisse und Messprotokolle unserer Evaluation in unserem GitHub Repository zu finden [17].

A. Laufzeitmessungen für einzelne Inferenzen

In Tabelle 2 und Tabelle 3 haben wir die Resultate der Single-Operation Layer Messungen aus Abbildung 197 zusammengefasst. Betrachtet man Tabelle 2 lässt sich für einzelne Inferenzen bei Single-Operation Layern sagen, dass Coral's TPU ungefähr 6 bis 10 mal schneller als Intel's

Layer Network	CPU	GPU	TPU	MYRIAD
ReLU	18587 fps	5483 fps	9183 fps	533 fps
Leaky ReLU	18595 fps	5446 fps	-	549 fps
tanh	18167 fps	4852 fps	3356 fps	531 fps
sigmoid	18502 fps	5327 fps	3951 fps	538 fps
Scalar Multiplication	18582 fps	5184 fps	4270 fps	562 fps

Tabelle II
Durchsatzraten für einzelne Inferenzen bei Single-Operation Layern

Layer Network	CPU	GPU	TPU	MYRIAD
ReLU	53 µsec	181 µsec	101 µsec	1768 µsec
Leaky ReLU	53 µsec	182 µsec	-	1714 µsec
tanh	54 µsec	204 µsec	258 µsec	1821 µsec
sigmoid	53 µsec	188 µsec	217 µsec	1836 µsec
Scalar Multiplication	53 µsec	183 µsec	198 µsec	1768 µsec

Tabelle III
Latenzen für einzelne Inferenzen bei Single-Operation Layern

Neural Compute Stick 2 ist. Die integrierte GPU ist in den meisten Fällen nur minimal schneller als die TPU. Aber die TPU scheint für ReLU-Operationen optimiert wurden zu sein, da sie hier fast 2 mal schneller ist als die GPU. Die CPU ist in allen Fällen deutlich mehr als 3 mal schneller als alle anderen getesteten Geräte. Allerdings gilt es hier zu beachten, dass die abgeschätzte Latenz bei allen Geräten sehr nahe an der Laufzeit für eine einzelne Inferenz liegt. Die Latenzen waren, wie man in Tabelle 3 sehen kann, bei dem Neural Compute Stick 2 ungefähr 10 mal höher als bei der GPU und TPU. Die Inferenzen auf der CPU bieten eine mehr als 3 mal geringere Latenz als alle anderen Geräte. Die Latenzen der Hardware Verbindungen beeinflussen bei diesen Messungen also maßgeblich unsere Ergebnisse. Da die

CPU und GPU auch von anderen Systemprozessen genutzt werden, sind die Inferenzlaufzeiten nicht stabil. Das bedeutet die Inferenzlaufzeiten siedeln sich zwar vorrangig um den Mittelwert an, es gibt aber auch massive Ausreißer, die in den Violin-Plots im Anhang zu sehen sind. Außerdem ist es wichtig hervorzuheben, dass die TPU keine Unterstützung bietet um eine "Leaky ReLU"-Operation auszuführen, weshalb unsere Ergebnisse aus dem Violin-Plot nicht repräsentativ für einen Inferenzdurchlauf auf der TPU sind.
Die Auswertung der Fully-Connected Layer Messungen aus Abbildung 198 ergaben folgende Resultate:

Layer Network	CPU	GPU	TPU	MYRIAD
Small Dense	13630 fps	3869 fps	3625 fps	528 fps
Big Dense	8338 fps	2994 fps	-	487 fps

Tabelle IV
DURCHSATZRATEN FÜR EINZELNE INFERENZEN BEI FULLY-CONNECTED LAYERN

Für einzelne Inferenzen bei Fully-Connected Layern lässt sich Tabelle 4 zusammenfassend sagen, dass Coral's TPU ungefähr 6 mal schneller als Intel's Neural Compute Stick 2 bei Inferenzen über kleine Fully-Connected Layer ist. Die integrierte GPU ist bei Inferenz über den selben Layer nur minimal schneller als die TPU. Die TPU weist sehr interessante Ergebnisse für größere Fully-Connected Layer auf. Hier brauchte die TPU im Durchschnitt in etwa 8.46 ms für eine einzelne Inferenz über einen einzelnen Fully-Connected Layer der Größe 512. Für eine einzelne Inferenz über ein Netz aus 3 dieser Layer braucht die TPU aber erstaunlicherweise nur 0.30 ms. Dies legt eine Optimierung für gestackte Fully-Connected Layer nahe. Für eine einzelne Inferenz über ein Netz aus 8 der selben gestackten Layer war die TPU jedoch deutlich langsamer mit etwa 24.20 ms im Durchschnitt, dies hängt höchstwahrscheinlich mit der Netzwerkgröße zusammen. Angesichts dieser Ergebnisse raten wir beim Einsatz der TPU auf die Größe der Fully-Connected Layer zu achten und hier mehr Tests für den speziellen Anwendungsfall durchzuführen. Die CPU ist in allen Fällen in etwa 3 mal schneller als alle anderen getesteten Geräte. Da es sich nicht um eine lineare Skalierung von Operationen zwischen Netzen aus einzelnen und aus mehreren gestakten Fully-Connected Layern handelt, verzichten wir hier auf abgeschätzte Latenzen. Jedoch sollten die Latenzen mit steigender Netzwerkgröße und -komplexität auch zu nehmen. Unsere wenig repräsentativen Ästimationen spiegeln dies zumindest für die Hardware Beschleuniger wieder. Allerdings ergeben unsere von Abstraktionsfehlern behaftete Abschätzungen auch, dass für die CPU und GPU mit steigender Netzwerkgröße und -komplexität die Latenz leicht abnimmt. Diese Ergebnisse sind aber höchstwahrscheinlich nicht real und lassen sich auch theoretisch nicht belegen. Wir können aber zumindest annehmen, dass die Latenzen für kleine Fully-Connected Layer in etwa genauso hoch sind wie bei den Single-Operation Layern und dass mit einem ähnlichen Verhältnis zwischen Latenzen und Laufzeiten wie bei den Single-Operation Layern zu rechnen ist.
Die Auswertung der Convolutional Layer Messungen aus Abbildung 199 ergaben folgende Resultate:

Layer Network	CPU	GPU	TPU	MYRIAD
Simple 2D-Conv.	990 fps	788 fps	93 fps	94 fps
Dilated 2D-Conv.	1079 fps	829 fps	100 fps	73 fps
Strided 2D-Conv.	3880 fps	1932 fps	292 fps	357 fps
Small Kernel 2D-Conv.	1014 fps	913 fps	93 fps	105 fps
Big Kernel 2D-Conv.	366 fps	598 fps	95 fps	88 fps
Few Filter 2D-Conv.	1833 fps	1312 fps	184 fps	148 fps
Many Filter 2D-Conv.	10 fps	13 fps	5 fps	4 fps

Tabelle V
DURCHSATZRATEN FÜR EINZELNEN INFERENZEN BEI 2D-CONVOLUTIONAL LAYERN

Die Resultate für einzelne Inferenzen bei zweidimensionalen Convolutional Layern in Tabelle 5 zeigen, dass Coral's TPU ungefähr genauso schnell ist wie Intel's Neural Compute Stick 2. Die TPU ist minimal schneller als der Neural Compute Stick 2 für einzelne Inferenzen bei dilated Convolutions und bei Convolutions mit wenigen Filtern. Dafür ist der Neural Compute Stick minimal schneller für einzelne Inferenzen bei strided Convolutions und bei kleinen Convolutions. Die integrierte GPU und die CPU sind bei den einzelnen Inferenzen über die selben Layer deutlich schneller als die Hardware Beschleuniger. Die CPU ist fast immer schneller als die GPU und die GPU ist in der Regel 6 bis 8 mal schneller als die Hardware Beschleuniger. Auffällig sind hier aber die Convolutions mit vielen Filtern und der große Convolution

Layer, bei diesen war die integrierte GPU schneller als die CPU und bei der Convolution mit vielen Filtern hat die gesamte Hardware ausgesprochen lange für eine einzelne Inferenz über einen Layer gebraucht. Das bedeutet bei der Implementation eines neuronalen Netzes ist bei Convolutional Layern auf möglichst wenige Filtern zu setzen, besonders wenn man Echtzeit Performance erreichen will. Kleine Convolutional Layer mit wenig Filtern und einem geringen Stride sollten die beste Inferenz-Performance erzielen. Für Latenzen gilt hier das Selbe wie auch für die Fully-Connected Layer.

B. Laufzeitmessungen für Inferenz-Batches

Zunächst betrachten wir die jeweils erste Inferenz in einer Batch aus 64 Inferenzen. Hier hat die Auswertung der Single-Operation Layer Messungen aus Abbildung 203 folgende Resultate ergeben:

Layer Network	CPU	GPU	TPU	MYRIAD
ReLU	4380 fps	2084 fps	8887 fps	472 fps
Leaky ReLU	4377 fps	2106 fps	-	485 fps
tanh	4461 fps	2090 fps	3474 fps	496 fps
sigmoid	4355 fps	2148 fps	3926 fps	497 fps
Scalar Multiplica-tion	4364 fps	2133 fps	4227 fps	497 fps

Tabelle VI
DURCHSATZRATEN FÜR DIE ERSTE INFERENZ EINES BATCHES BEI SINGLE-OPERATION LAYERN

Für die ersten Inferenzen eines Batches bei Single-Operation Layern lässt sich Tabelle 6 zusammenfassend sagen, dass die TPU in allen Fällen mindestens 8 mal schneller als Intel's Neural Compute Stick 2 ist. Die integrierte GPU ist ca. 5 mal so schnell wie Intels Neural Compute Stick 2, aber immer mehr als 100 fps langsamer als die TPU. Die CPU ist in fast allen Fällen am schnellsten. Nur bei der ReLU-Operation ist die TPU durch ihre Optimierung deutlich schneller als alle anderen Geräte. Hier sticht die TPU mit einem mehr als doppelt so hohen Durchsatz hervor. Allerdings bietet die TPU keinen Support für die "Leaky ReLU"-Operation, weshalb wir hier keine repräsentativen Ergebnisse für die Durchsatzrate und Latenz präsentieren können. Dies gilt es auch zu beachten bei der Betrachtung der Violin-Plots, die dort ersichtlichen Ergebnisse entsprechen zwar reellen Messungen, aber bei diesen wurde statt der

Layer Network	CPU	GPU	TPU	MYRIAD
ReLU	228 µsec	468 µsec	111 µsec	2045 µsec
Leaky ReLU	228 µsec	471 µsec	-	2020 µsec
tanh	221 µsec	477 µsec	264 µsec	1940 µsec
sigmoid	227 µsec	451 µsec	226 µsec	1952 µsec
Scalar Multiplica-tion	229 µsec	465 µsec	204 µsec	2002 µsec

Tabelle VII
LATENZEN FÜR DIE ERSTE INFERENZ EINES BATCHES BEI SINGLE-OPERATION LAYERN

TPU die native Systemhardware benutzt. Betrachtet man Tabelle 7, so stellt man fest, dass die Latenzen der TPU und CPU am geringsten waren. Die CPU weißt gegenüber der TPU eine etwas kleinere Latenz für tanh auf, während die TPU eine etwas geringere Latenz gegenüber der CPU für die Skalarmultiplikation und eine nur halb so große Latenz für die ReLU-Operation aufweist. Die integrierte GPU weist ungefähr doppelt so hohe Latenzen auf wie die CPU. Die Latenzen von dem Neural Compute Stick 2 sind verglichen mit der integrierten GPU zwischen 4 bis 5 mal so hoch.
Die Auswertung der Fully-Connected Layer Messungen aus Abbildung 204 ergaben folgende Resultate:

Layer Network	CPU	GPU	TPU	MYRIAD
Small Dense	3694 fps	1714 fps	3492 fps	486 fps
Big Dense	2307 fps	1426 fps	-	462 fps

Tabelle VIII
DURCHSATZRATEN FÜR DIE ERSTE INFERENZ EINES BATCHES BEI FULLY-CONNECTED LAYERN

In Tabelle 8 sehen wir bei den Durchsatzraten für die erste Inferenz eines Batches bei Fully-Connected Layern eine ähnliche Relation zwischen den Geräten wie bei den Single-Operation Layern. Die TPU und die CPU sind in etwa gleich schnell bei der Inferenz über einem kleinen Fully-Connected Layer. Die integrierte GPU ist immer etwas mehr als halb so schnell wie die CPU und mehr als 3 mal schneller als Intel's Neural Compute Stick 2. Die TPU zeigte jedoch interessante Ergebnisse für die Inferenz über

einen großen Fully-Connected Layer. Hier erreichte die TPU für ein Netz aus nur einem Layer eine durchschnittliche Geschwindigkeit von 8,47 ms pro Inferenz, während die TPU für ein Netz aus 3 der selben Layer nur 0.47 ms pro Inferenz braucht. Für die Inferenz über ein Netz aus 8 dieser Layer brauchte die TPU 24.64 ms. Aus diesen Ergebnissen lässt sich schlussfolgern, dass die TPU für gestackte Fully-Connected Layer optimiert ist, allerdings ab einem bestimmten Parameterumfang überladen ist und deshalb wieder deutlich länger braucht. Latenzmessungen sind auch hier nicht wirklich möglich, da die Parameter pro Layer nicht linearen mit der Anzahl der Layer skalieren.
Die Auswertung der Convolutional Layer Messungen aus Abbildung 205 ergaben folgende Resultate:

Layer Network	CPU	GPU	TPU	MYRIAD
Simple 2D-Conv.	802 fps	525 fps	93 fps	105 fps
Dilated 2D-Conv.	864 fps	515 fps	100 fps	79 fps
Strided 2D-Conv.	2266 fps	946 fps	354 fps	310 fps
Small Kernel 2D-Conv.	830 fps	566 fps	94 fps	117 fps
Big Kernel 2D-Conv.	338 fps	436 fps	97 fps	96 fps
Few Filter 2D-Conv.	1299 fps	732 fps	183 fps	166 fps
Many Filter 2D-Conv.	10 fps	13 fps	5 fps	4 fps

Tabelle IX
DURCHSATZRATEN FÜR DIE ERSTE INFERENZ EINES BATCHES BEI 2D-CONVOLUTIONAL LAYERN

Betrachtet man Tabelle 9 lässt sich zusammenfassend für die erste Inferenz eines Batches bei 2D-Convolutional Layern sagen, dass die TPU und Intel's Neural Compute Stick 2 ungefähr gleich schnell waren. Die TPU war leicht schneller als Intel's Neural Compute Stick 2 bei dilated und strided Convolutions. Wohingegen Intel's Compute Stick 2 leicht schneller als die TPU bei kleinen Convolutions war. Die integrierte GPU war in den meisten Fällen mehr als 5 mal schneller als die TPU. Die Ausnahmen sind hier strided Convolutions, wo die GPU nur zwischen 2 bis 3 mal schneller als die TPU war und die Convolutions mit wenigen Filtern wo die GPU ungefähr 4 mal so schnell wie die TPU ist. Die CPU war in den meisten Fällen mehr als 8 mal so schnell wie die TPU. Bei strided Convolutions war die CPU sogar doppelt so schnell verglichen mit der GPU. Dahingegen war die CPU nur etwa 7 mal schneller als die TPU bei Convolutions mit wenig Filtern und nur 3 mal so schnell wie die TPU bei großen Convolutions, die GPU war hier sogar fast 100 fps schneller als die CPU. Für Convolutions mit vielen Filtern hat die gesamte Hardware schlechte Durchsatzraten erzielt. Das bedeutet bei der Implementation eines neuronalen Netzes ist bei Convolutional Layern auf möglichst wenige Filtern zu setzen, besonders wenn man Echtzeit Performance erreichen will. Kleine Convolutional Layer mit wenig Filtern und einem geringen stride sollten die beste Inferenz-Performance erzielen. Für Latenzen gilt hier das Selbe wie auch für die Fully-Connected Layers.

Als nächstes betrachten wir der Messergebnisse für die 63 folgenden Inferenzen jeweils zusammengefasst in einer Batch. Zuerst zeigen wir die Resultate der Auswertung der Single-Operation Layer Messungen aus Abbildung 200:

Layer Network	CPU	GPU	TPU	MYRIAD
ReLU	20080 fps	5623 fps	9232 fps	537 fps
Leaky ReLU	19963 fps	5526 fps	-	540 fps
tanh	19465 fps	4985 fps	3769 fps	531 fps
sigmoid	19921 fps	5302 fps	4030 fps	551 fps
Scalar Multiplication	20251 fps	5472 fps	4338 fps	577 fps

Tabelle X
DURCHSATZRATEN FÜR INFERENZ-BATCHES BEI SINGLE-OPERATION LAYERN

Zusammenfassend lässt sich angesichts Tabelle 10 für Inferenz-Batches bei Single-Operation Layern sagen, dass Coral's TPU mehr als 7 mal schneller als Intel's Neural Compute Stick 2 ist. Die integrierte GPU ist in den meisten Fällen mehr als 1000 fps schneller als die TPU. Aber die TPU scheint speziell für ReLU-Operationen optimiert wurden zu sein, da sie hier fast 2 mal schneller ist als die GPU und fast 20 mal schneller als Intel's Neural Compute Stick 2. Die CPU ist in allen Fällen fast 4 mal schneller als die GPU und auch für ReLU-Operation ist die CPU mehr als doppelt so schnell als die TPU. Allerdings muss man, wie in Tabelle 11

Layer Network	CPU	GPU	TPU	MYRIAD
ReLU	49 µsec	177 µsec	108 µsec	1751 µsec
Leaky ReLU	50 µsec	179 µsec	-	1773 µsec
tanh	50 µsec	199 µsec	237 µsec	1800 µsec
sigmoid	49 µsec	182 µsec	218 µsec	1727 µsec
Scalar Multiplica-tion	49 µsec	182 µsec	209 µsec	1714 µsec

Tabelle XI
LATENZEN FÜR INFERENZ-BATCHES BEI SINGLE-OPERATION LAYERN

zu sehen, beachten, dass die abgeschätzte Latenz bei allen Geräten sehr nahe an der Laufzeit für eine einzelne Inferenz liegt. Bei dem Neural Compute Stick 2 waren die Latenzen mehr als 8 mal höher als bei der TPU und ca. 10 mal höher als bei der GPU. Die Inferenzen auf der CPU bieten eine mehr als 3 mal geringere Latenz als alle anderen Geräte. Die Latenzen der Hardware Verbindungen beeinflussen bei diesen Messungen also maßgeblich unsere Ergebnisse. Da die CPU und GPU auch von anderen Systemprozessen genutzt werden, sind die Inferenzlaufzeiten nicht stabil. Das bedeutet die Inferenzlaufzeiten siedeln sich zwar vorrangig um den Mittelwert an, es gibt aber eben auch massive Ausreißer, die in den Violin-Plots im Anhang zu sehen sind. Darüber hinaus ist es wichtig hervorzuheben, dass die TPU keine Unterstützung bietet um eine "Leaky ReLU"-Operation auszuführen, weshalb unsere Ergebnisse aus dem Violin-Plot nicht repräsentativ für einen Inferenzdurchlauf auf der TPU sind.

Die Auswertung der Fully-Connected Layer Messungen aus Abbildung 201 ergaben folgende Resultate:

Layer Network	CPU	GPU	TPU	MYRIAD
Small Dense	14434 fps	3893 fps	3616 fps	503 fps
Big Dense	8876 fps	3022 fps	-	465 fps

Tabelle XII
DURCHSATZRATEN FÜR INFERENZ-BATCHES BEI FULLY-CONNECTED LAYERN

Für Inferenz-Batches bei Fully-Connected Layern lässt sich in Tabelle 12 beobachten, dass Coral's TPU ungefähr 7 mal schneller als Intel's Neural Compute Stick 2 bei Inferenzen über kleine Fully-Connected Layer ist. Die integrierte GPU ist bei Inferenz über den selben Layer nur minimal schneller als die TPU. Die TPU weist sehr interessante Ergebnisse für größere Fully-Connected Layer auf. Hier brauchte die TPU im Durchschnitt in etwa 8.46 ms für eine Inferenz über einen einzelnen Layer. Für eine Inferenz über den selben dreifach gestackten Layer braucht die TPU aber erstaunlicherweise nur 0.47 ms. Dies legt eine Optimierung für gestackte Fully-Connected Layer nahe. Für eine Inferenz über den selben achtfach gestackten Layer war die TPU jedoch deutlich langsamer mit etwa 24.63 ms, dies hängt höchstwahrscheinlich mit der Netzwerkgröße zusammen. Angesichts dieser Ergebnisse raten wir beim Einsatz der TPU auf die Größe der Fully-Connected Layer zu achten und hier mehr Tests für den speziellen Anwendungsfall durchzuführen. Die CPU ist in allen Fällen in etwa 3 mal schneller als alle anderen getesteten Geräte. Wir verzichten hier auf abgeschätzte Latenzen, da es sich bei den gestakten Layer Netzen nicht, um eine lineare Skalierung von Operationen verglichen zu denen des einzelnen Fully-Connected Layer handelt. Latenzen sollten mit steigender Netzwerkgröße und -komplexität leicht zu nehmen. Unsere wenig repräsentativen Ästimationen weisen die selben Merkmale wie bei den einzelnen Inferenzen auf. Auch hier ist zumindest mit einem ähnlichen Verhältnis zwischen Latenzen und Laufzeiten zu rechnen wie bei den Single-Operation Layern.

Die Resultate der Auswertung der Convoutional Layer Messungen aus Abbildung 202 haben wir in Tabelle 13 zusammengefasst. Dort kann man für Inferenz-Batches bei zweidimensionalen Convolutional Layern erkennen, dass Coral's TPU ungefähr genauso schnell ist wie Intel's Neural Compute Stick 2. Die TPU ist minimal schneller als der Neural Compute Stick 2 für Inferenz-Batches bei dialated und strided Convolutions, sowie bei Convolutions mit wenigen Filtern. Dafür ist der Neural Compute Stick minimal schneller für Inferenz-Batches bei kleinen Convolutions. Die integrierte GPU und die CPU sind bei Inferenz-Batches über die selben Layer deutlich schneller als die Hardware Beschleuniger. Die CPU ist fast immer schneller als die GPU und die GPU ist in der Regel 6 bis 8 mal schneller als die Hardware Beschleuniger. Auffällig sind hier aber die Convolutions mit vielen Filtern und der große Convolution Layer, bei diesen war die integrierte GPU schneller als die CPU und bei der Convolution mit vielen Filtern hat die gesamte Hardware ausgesprochen lange für eine einzelne Inferenz über einen Layer gebraucht. Das bedeutet bei der Implementation eines neuronalen Netzes ist bei Convolutional Layern auf möglichst wenige Filtern zu setzen, besonders wenn man Echtzeit Performance erreichen will. Kleine Convolutional Layer mit wenig Filtern und einem geringen Stride sollten die beste Inferenz-Performance erzielen. Für Latenzen gilt hier das Selbe wie auch für die Fully-Connected Layer.

Layer Network	CPU	GPU	TPU	MYRIAD
Simple 2D-Conv.	993 fps	791 fps	93 fps	93 fps
Dilated 2D-Conv.	1087 fps	835 fps	101 fps	73 fps
Strided 2D-Conv.	3995 fps	1988 fps	347 fps	292 fps
Small Kernel 2D-Conv.	1018 fps	923 fps	93 fps	105 fps
Big Kernel 2D-Conv.	365 fps	598 fps	95 fps	88 fps
Few Filter 2D-Conv.	1855 fps	1334 fps	180 fps	149 fps
Many Filter 2D-Conv.	10 fps	13 fps	5 fps	4 fps

Tabelle XIII
DURCHSATZRATEN FÜR INFERENZ-BATCHES BEI 2D-CONVOLUTIONAL LAYERN

Zusammenfassend lässt sich für Inferenz Batches sagen, dass für alle Geräte außer der TPU eine deutlich höhere Latenz und eine geringere Durchsatzrate bei der ersten verglichen zu allen folgenden Inferenzen vorzufinden ist. Bei den Single-Operation Layern sind die Latenzen der CPU etwa 4 mal höher bei der ersten Inferenz verglichen zu den folgenden 63 Inferenzen. Bei der integrierten GPU sind die Latenzen in etwa 3 mal höher und bei Intel's Neural Compute Stick 2 haben wir Latenzunterschiede zwischen 100 und 300 Mikrosekunden. Die Durchsatzraten sind ebenso stark betroffen. Hier ist die CPU in den letzten 63 Inferenzen einer Batch fast 5 mal schneller als bei der ersten Inferenz, die GPU ist mehr als doppelt so schnell in den letzten Inferenzen als in der ersten Inferenz in einer Batch. Intel's Neural Compute Stick 2 wies neben der TPU die geringsten Unterschiede auf. Der Stick war zwischen 50 und 100 fps schneller auf den letzten 63 Inferenzen verglichen mit der ersten Inferenz. Bei den Fully-Connected Layern zeichnet sich ein ähnliches Bild ab. Allerdings fallen die Veränderungen in den Durchsatzraten der ersten zu den letzten Inferenzen in der Batch hier ein wenig moderater aus. Bei der CPU sprechen wir von dem Faktor 4, bei der GPU von einem Faktor größer 2 und bei Intel's Neural Compute Stick 2 sogar ein paar wenigen fps. Bei den Convolutional Layer schrumpfen diese Faktoren noch weiter. Intel's Neural Compute Stick 2 und die TPU weisen so gut wie keine Veränderungen und die CPU und GPU nur wenige 100 fps Unterschied zwischen der ersten und den darauf folgenden Inferenzen auf. Erwähnenswert ist hier nur die Convolution mit vielen Filtern. Diese zeigte sowohl für die erste Inferenz als auch die darauf folgenden die selben schlechten Werte für den Durchsatz.

C. Energieverbrauchsmessungen

Wir haben mit einem Strommessgerät den Stromverbrauch über die USB Verbindung gemessen und hatten leider keine Möglichkeit den Stromverbrauch der fest verbauten Hardware zu messen. Der Energieverbrauch wird hier in Picowatt pro Sekunde für eine einzelne Inferenz über ein Netz aus einem Layer angegeben. Hier sind die Ergebnisse aus Abbildung 194 für unserer Strommessung bei Inferenzen über Single-Operation Layer zusammengefasst:

Layer Network	TPU	MYRIAD
ReLU	1.41 pW/sec	137.25 pW/sec
Leaky ReLU	-	134.02 pW/sec
tanh	3.30 pW/sec	135.10 pW/sec
sigmoid	7.06 pW/sec	137.25 pW/sec
Scalar Multiplication	1.88 pW/sec	137.79 pW/sec

Tabelle XIV
ENERGIEVERBRAUCH FÜR EINE EINZELNE INFERENZ BEI SINGLE-OPERATION LAYERN

Wie man in Tabelle 14 sehen kann, verbraucht Intel's Neural Compute Stick 2 deutlich mehr Energie als Coral's TPU. Wir haben hier einen Unterschied von mehr 130 pW/sec. Die TPU bietet keinen Support für die "Leaky ReLU"-Operation, deshalb können wir hier keine Aussage über den Energieverbrauch der TPU treffen, da diese gar nicht bei der Inferenz benutzt wird.

Layer Network	TPU	MYRIAD
Small Dense	6.12 pW/sec	138.32 pW/sec
Big Dense	213.62 pW/sec	162.01 pW/sec

Tabelle XV
ENERGIEVERBRAUCH FÜR EINE EINZELNE INFERENZ BEI FULLY-CONNECTED LAYERN

Die Ergebnissen aus Abbildung 194 für die Fully-Connected Layer werden in Tabelle 15 wiedergespiegelt. Dort sehen wir, dass die TPU wieder deutlich weniger Energie als Intel's Neural Compute Stick 2 für kleine Fully-Connected Layer verbraucht. Hier verbraucht Intel's Neural Compute Stick 2 mehr als 130 pw/sec mehr als die TPU. Das Ergebniss für große Fully- Connected Layer ist jedoch weitaus interessanter. Es scheint, dass die TPU für Layer mit wenigen Parametern optimiert ist, allerdings dafür bei Layern mit großer Parameterzahl deutlich weniger energieeffizient ist. Für große Fully-Connected Layer ist Intel's Neural Compute Stick 2 um mehr als 50 pW/sec energieeffizienter als die TPU.
Zu guter Letzt betrachten wir die Auswertung der Resultate aus Abbildung 195, die den Energieverbrauch bei Convolutional Layern aufzeigt:

Layer Network	TPU	MYRIAD
Simple 2D-Conv.	219.46 pW/sec	2199.20 pW/sec
Dilated 2D-Conv.	220.12 pW/sec	2600.72 pW/sec
Strided 2D-Conv.	52.28 pW/sec	651.53 pW/sec
Small Kernel 2D-Conv.	209.10 pW/sec	1863.89 pW/sec
Big Kernel 2D-Conv.	302.07 pW/sec	1940.41 pW/sec
Few Filter 2D-Conv.	124.61 pW/sec	1170.11 pW/sec
Many Filter 2D-Conv.	9098.96 pW/sec	74017.84 pW/sec

Tabelle XVI
ENERGIEVERBRAUCH FÜR EINE EINZELNE INFERENZ BEI 2D-CONVOLUTIONAL LAYERN

In Tabelle 16 sehen wir, dass die TPU auch bei Convolutional Layern in den meisten Fällen deutlich energieeffizienter als Intel's Neural Compute Stick 2 ist. Für simple, dilated und strided Convolutions ist die TPU mehr als 10 mal energieeffizienter als Intel's Neural Compute Stick 2. Für Convolutions mit kleinen Kernels und wenigen Filtern ist die TPU nur noch etwa 9 mal energieeffizienter und für Convolutions mit einem großen Kernel nur noch etwas mehr als 6 mal energieeffizienter als Intel's Neural Compute Stick 2. Für Convolutions mit vielen Filtern ist Intel's Neural Compute Stick 2 sogar um mehr als 1600 pW/sec energieeffizienter als die TPU.

V. FAZIT

Insgesamt lässt sich zusammenfassend über die Hardware Beschleuniger sagen, dass die TPU von Coral in den meisten Fällen besser performt. Mit anderen Worten Coral's TPU ist in so gut wie allen Fällen schneller und auch energieeffizienter als Intel's Neural Compute Stick 2. Allerdings verringert sich der Performanceunterschied zwischen Coral's TPU und Intel's Neural Compute Stick 2 mit steigender Parameterzahl pro Layer. Intel's Neural Compute Stick 2 performt für Convolutions mit vielen Filtern sogar besser als die TPU und bietet Support für ein breiteres Spektrum an Operationen verglichen zu Coral's TPU. Darüber hinaus bietet Intel mit OpenVINO ein Framework, dass autonom bestmöglich die Gesamtheit der verfügbaren Intel Hardware für ein Heterogenous Inferencing Setup nutzt [5]. Wie wir durch unsere Messungen aber auch sehen konnten, sind "Low-Power" USB Hardware Beschleuniger von vergleichsweise hohen Latenzen betroffen. Bei Inferenzen weist die CPU immer die geringsten Latenzen auf, da sie am engsten in das System integriert ist. In unseren Messungen zeigt die TPU von Coral in den meisten Fällen ähnlich hohe Latenzen und Durchsatzraten wie die integrierte GPU. Bei "Low-Power" Hardware Beschleunigern geht es in erster Linie darum, dass diese möglichst energieeffizient Inferenzen ausführen können. Bei modernen leistungsstarken PCs, egal ob es sich um einen Laptop oder einen Desktop-Rechner handelt, erledigt die CPU die Inferenzen in den meisten Fällen schneller als die "Low-Power" Hardware Beschleuniger. Allerdings gilt es hier zu beachten, dass GPUs und CPUs oft noch von anderen laufenden und systemrelevanten Prozessen zeitgleich zu den Inferenzen beansprucht werden. Dies führt dazu, dass einige Inferenzen deutlich länger brauchen als andere, da für die Inferenz erst noch auf einen anderen Prozess gewartet werden muss. Hier liegt der Vorteil in speziell für Inferenzen genutzter Hardware. Zum einen nehmen sie dem System die Last der Inferenzausführung zum größten Teil ab und zum anderen können diese Geräte die Infernzoperationen energieeffizienter durchführen. Wir können uns ein Heterogeneous Inferencing Setup mit "Low-Power AI Accelerators" vorstellen, wenn es das Ziel ist eine Netzwerkinferenz möglichst energieeffizent zu vollziehen. Wir würden hierfür vor allem auf Coral's TPU setzen und nur für die nicht unterstützten Operationen und Layer mit vielen Parametern auf andere Hardware-Beschleuniger wie Intel's Neural Compute Stick 2 zurückgreifen. Andernfalls ist der Einsatz von Hardware-Beschleunigern auch durchaus sinnvoll bei leistungsschwachen Edge Devices, denn Geräte mit einem "Low-Power" SoC haben oft nicht genügend Leistung um Inferenzen performant neben den anderen Systemprozessen auszuführen. Wenn man ein Heterogeneous Inferencing Setup erstellen will, das vor allem auf überragende Performance zugeschnitten ist, so muss man auf leistungsstärkere Hardware-Beschleuniger setzen, die enger in das System integriert sind.

VI. Verwandte Arbeiten

In [10] haben Mattson et al. die Design Entscheidungen hinter der Entwicklung von dem häufig genutzten und weit verbreiteten MLPerf Benchmark erklärt. In dem Artikel beschreiben Sie detailliert den Aufbau und die korrekte Nutzung der MLPerf Benchmarksuite. Mattson et al. gehen hierbei auf den MLPerf Training und MLPerf Inference Benchmark ein, die anhand einer Auswahl künstlicher neuronalen Netzen zur Evaluation der Machine Learning Performance von Geräten genutzt werden können. Im Kontext unserer Arbeit ist vor allem letzterer Benchmark von Interesse, da MLPerf Inference oft zur Performance Analyse von Hardware bei AI Applikationen herangezogen wird - so auch von Reuther et al. in [14], [15]. Reuther et al. hat zahlreiche Hardware Beschleuniger unterschiedlicher Klassen für Deep Learning Anwendungen in seinen jährlichen Erhebung vorgestellt und verglichen. Er stellte 2021 in [15] eine Effizienz Verbesserung von mehr als 20% für viele der vorgestellten Hardware Beschleuniger im Vergleich zu den Vorgängermodellen fest. Die Vorgängermodelle analysierten Reuther et al. 2020 in [14]. Im Vergleich zu unserer Arbeit wurden in [14], [15] nur bereits bekannte Benchmark Evaluationsdaten für einen Vergleich herangezogen, während wir in unserer Arbeit für ausgewählte "Low-Power AI Accelerators" auf der Layer-Ebene eigene Benchmarks ausgeführt haben und diese mit Ergebnissen aus unseren CPU und integrierten GPU Benchmarks verglichen haben. Elordi et al. erstellten in [3] ein Infrastrukturdesign für Heterogeneous IoT Platforms, allerdings für den spezifischen Anwendungsfall einer Gesichtserkennungs-Applikation in der Altenpflege. In ihrer Arbeit übermitteln IoT-Geräte Daten an ein zentrales System, welches anhand dieser Daten passende Machine Learning Modelle aus einem Modell-Pool auswählt, die Inferenzausführung vorbereitet und auf verschiedene Systeme entsprechend deren Hardware-Komponenten verteilt. Im Rahmen der Entwicklung des Systems namens Automated Heterogeneous Deployment Optimizer (AHDO) haben Elordi et al. eine Performance Evaluation von verschiedenen Heterogeneous Inference Setups durchgeführt. Ihr Fokus lag aber auf dem Entwurf und der Implementation des automatisierten Deployment Systems und nicht auf der Performance Analyse. Darüber hinaus vollzogen Elordi et al. die Performance Auswertung auf Netzwerkebene und nicht auf Layer-Ebene, wie wir es in unser Arbeit taten. Kist liefert mit [6] eine Übersicht zu den verschiedenen Edge TPU Lösungen. Im Rahmen dessen geht er auf die Einsatzzwecke einer TPU ein und stellt Einordnungen zu Performance und Preisklassen bereit. Er fokussiert sich in seiner Arbeit aber eher auf die Anwendung von TPUs in Embedded Edge Systemen. Die von ihm verwendeten Performance Analysen sind Netzwerk basiert und nutzen "Operationen pro Sekunde" (OPS) als Vergleichswert, anstatt sich auf Laufzeiten und Latenzen zu fokussieren. Li et al. untersuchten bereits 2021 in [9] die Hardware spezifischen Defizite in der, durch Neural Architecture Search (NAS) durchgeführten, Optimierung zur Effizienzmaximierung und entwickelten ein latency-aware compound scaling (LACS) Konzept, um eine Optimierung sowohl maximale Genauigkeit als auch minimale Latenzen zu erzielen. Mit Hilfe des neuen Konzepts konnten häufig eingesetzte CNN-Modelle gegenüber bisheriger Methoden bedeutend weiter optimiert werden. Ihre Performance Analysen auf Netzwerkebene zeigten, dass die Optimierungen mittels LACS zu einer bis zu mehr als doppelt so schnellen Inferenz führten. Li und Jouppi haben anschließend 2022 in [8] das gerade vorgestellte Neural Architecture Search (NAS) Tool weiter entwickelt, um mittels Pareto Optimierung ein Machine Learning Modell dahingehend anzupassen, dass es spezifische Hardware optimal nutzt, um das gewählte Verhältnis zwischen Performance- und Qualitätszielen effizient zu erreichen. Der Fokus ihrer Arbeit lag dabei weiterhin auf der automatisierten Software Optimierung zur Effizienzmaximierung bei einer verteilter Systemarchitektur bestehend aus TPU und GPU. Im Rahmen dessen wurden bestehender Netze zum Zwecke der Optimierung angepasst und anschließend in einer Performance Evaluationen mit den ursprünglichen Netzen verglichen. Im Gegensatz zu Li et al. Arbeiten haben wir keinerlei Netzwerkadaptionen vorgenommen. Wir analysierten die Auswirkung einer Verteilung der Inferenz Operationen auf die Hardware Beschleuniger auf der Layer-Ebene, während Li et al. sich auf die Performance Evaluation auf Netzwerkebene fokussierten. Darüber hinaus nutzten wir in unseren Benchmark die gesamten Hardware, die wir zur Verfügung hatten, nicht nur spezielle Beschleuniger.

VII. Ausblick auf Zukünftige Arbeit

Gegeben des Fazits aus unserer Auswertung empfehlen wir die Implementation eines layerbasierten Heterogenous Inferencing Setups, welches an Hand der aktuellen Hardware Auslastung und einer Kompatibilitätsmatrix Inferenzoperationen auf unterschiedliche Hardwarekomponenten des Systems verteilt. Dieses Setup kann für verschiedene Direktiven optimiert werden - entweder mittels energiesparenden Hardware Beschleuniger in Richtung möglichst energieeffizienter Inferenz oder mittels leistungsstarker Hardware Beschleuniger in Richtung höchstem Durchsatz bei Inferenzen. Für ersteres Setup empfehlen wir auf Basis dieser Arbeit ein verteiltes Inferenz Setup bei dem eine Google TPU vorrangig für alle Layer mit einem geringen Operationsumfang eingesetzt wird, während wir für Layer mit einer größeren Parameteranzahl den Einsatz von Intel's Neural Compute Stick 2 empfehlen. Um höhere Durchsatzraten zu erreichen kann die Nutzung einer leistungsstärkeren CPU vor allem für Convolutional Layer sinnvoll sein. Darüber hinaus lassen sich Durchsatzraten weiter erhöhen, wenn man statt den einfachen Single-Module Beschleunigern die in [3], [6] erwähnten VPU und TPU Multi-Module Karten verwendet. Für ein Heterogenous Inferencing Setup, dass nicht zwangsweise energieeffizent aber möglichst hohe Durchsatzraten erreichen soll, empfehlen wir hingegen

den Einsatz von leistungsstarken Hardware Beschleunigern. Hier bieten sich die von Reuther et al. gelisteten Beschleuniger wie Intel's Arria FPGA oder NVIDIA's DGX GPU an [14]. Vor der Implementation eines solchen Setups raten wir aber dringendst ähnliche Benchmarks durchzuführen, um eine effiziente Verteilung der Inferenz auf die Beschleuniger garantieren zu können. Unsere Benchmarks können in Zukunft weiterentwickelt werden um noch granularer verschiedene Layer-Konfigurationen, die in aktuell genutzten Netzen vorkommen, zu testen. Der Ausbau der Benchmarks kann helfen die Grenzen für die Einsatzgebiete der verschiedenen Hardware Beschleuniger noch genauer zu bestimmen. Es wäre sicherlich auch interessant neben den Messungen für einzelne Operationen Performance-Benchmarks für häufig genutzten Netzwerkbausteine und Layer-Pattern durchzuführen.

LITERATUR

[1] coral: Get started with the USB Accelerator. https://coral.ai/docs/accelerator/get-started/#3-run-a-model-on-the-edge-tpu, besucht 25.05.2022

[2] coral: TensorFlow models on the Edge TPU. https://coral.ai/docs/edgetpu/models-intro/, besucht 24.05.2022

[3] Elordi, U., Lunerti, C., Unzueta, L., Goenetxea, J., Aranjuelo, N., Bertelsen, A., Arganda-Carreras, I.: Designing Automated Deployment Strategies of Face Recognition Solutions in Heterogeneous IoT Platforms. Information 12(12), 532 (2021)

[4] Goodfellow, I., Bengio, Y., Courville, A.: Deep Learning. MIT Press (2016), http://www.deeplearningbook.org

[5] Intel: OpenVINO Docs: Heterogeneous execution. https://docs.openvino.ai/latest/openvino_docs_OV_UG_Hetero_execution.html, besucht 30.05.2022

[6] Kist, A.M.: Deep Learning on Edge TPUs. arXiv preprint arXiv:2108.13732 (2021)

[7] Krenker, A., Bešter, J., Kos, A.: Introduction to the artificial neural networks. Artificial Neural Networks: Methodological Advances and Biomedical Applications. InTech pp. 1–18 (2011)

[8] Li, S., Jouppi, N.P.: Unlocking the Full Potential of Datacenter ML Accelerators with Platform-Aware Neural Architecture Search. Google AI Blog (Feb 2022), https://ai.googleblog.com/2022/02/unlocking-full-potential-of-datacenter.html

[9] Li, S., Tan, M., Pang, R., Li, A., Cheng, L., Le, Q.V., Jouppi, N.P.: Searching for fast model families on datacenter accelerators. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. pp. 8085–8095 (2021)

[10] Mattson, P., Reddi, V.J., Cheng, C., Coleman, C., Diamos, G., Kanter, D., Micikevicius, P., Patterson, D., Schmuelling, G., Tang, H., Wei, G.Y., Wu, C.J.: MLPerf: An Industry Standard Benchmark Suite for Machine Learning Performance. IEEE Micro 40(2), 8–16 (2020)

[11] openvino: Converting a TensorFlow* Model. https://docs.openvino.ai/2021.4/openvino_docs_MO_DG_prepare_model_convert_model_Convert_Model_From_TensorFlow.html, besucht 28.05.2022

[12] openvino: Runtime Optimization Guide. https://docs.openvino.ai/2021.4/openvino_docs_IE_DG_Intro_to_Performance.html, besucht 28.05.2022

[13] openvino: Supported Devices. https://docs.openvino.ai/2021.4/openvino_docs_IE_DG_supported_plugins_Supported_Devices.html, besucht 25.05.2022

[14] Reuther, A., Michaleas, P., Jones, M., Gadepally, V., Samsi, S., Kepner, J.: Survey of machine learning accelerators. In: 2020 IEEE high performance extreme computing conference (HPEC). pp. 1–12. IEEE (2020)

[15] Reuther, A., Michaleas, P., Jones, M., Gadepally, V., Samsi, S., Kepner, J.: AI Accelerator Survey and Trends. In: 2021 IEEE High Performance Extreme Computing Conference (HPEC). pp. 1–9 (2021)

[16] Sato, K., Young, C., Patterson, D.: An in-depth look at Googleś first Tensor Processing Unit (TPU). GOOGLE CLOUD - AI & MACHINE LEARNING (May 2017)

[17] Treuheit, N., Barthel, M.: Heterogenous-Inferencing. https://github.com/Nils-Treuheit/Heterogenous-Inferencing, besucht 04.05.2022

[18] Zhang, D., Mishra, S., Brynjolfsson, E., Etchemendy, J., Ganguli, D., Grosz, B., Lyons, T., Manyika, J., Niebles, J.C., Sellitto, M., et al.: The ai index 2021 annual report. arXiv preprint arXiv:2103.06312 (2021)

ANHANG

A. Energieverbrauchsmessungen

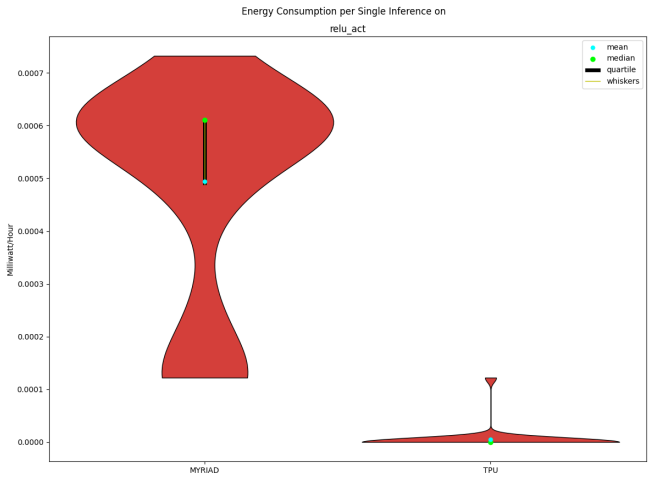

Abbildung 4. Energieverbrauch für eine Inferenz über einen ReLU-Layer

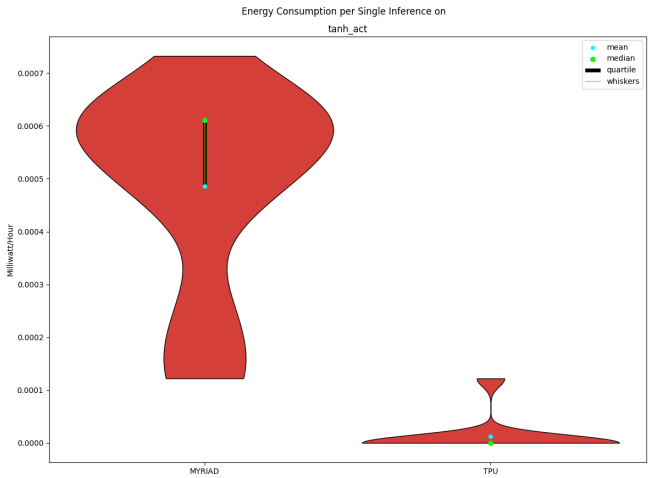

Abbildung 5. Energieverbrauch für eine Inferenz über einen tanh-Layer

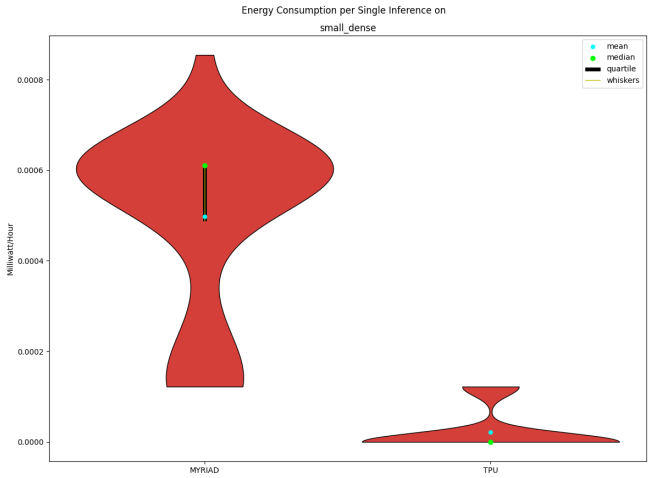

Abbildung 6. Energieverbrauch für eine Inferenz über einen kleinen Dense-Layer

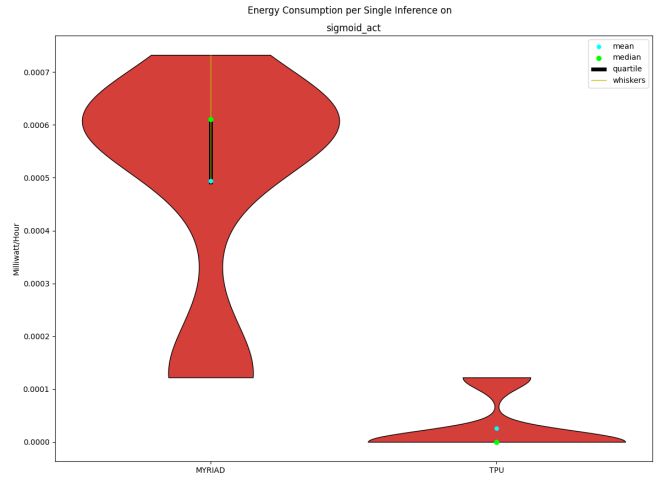

Abbildung 7. Energieverbrauch für eine Inferenz über einen sigmoid-Layer

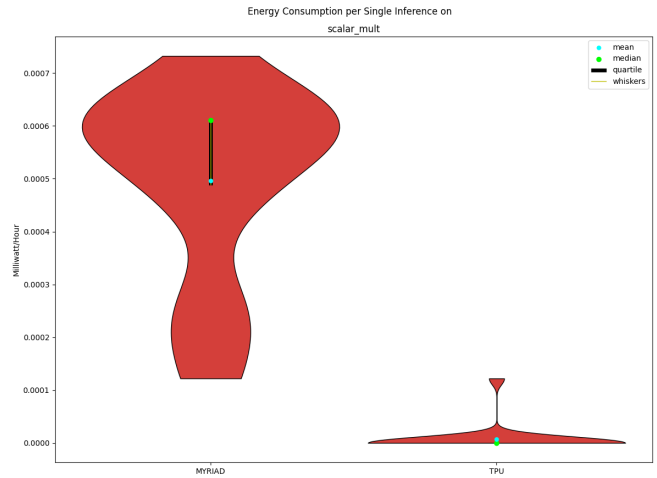

Abbildung 8. Energieverbrauch für eine Inferenz über einen Skalarprodukt-Layer

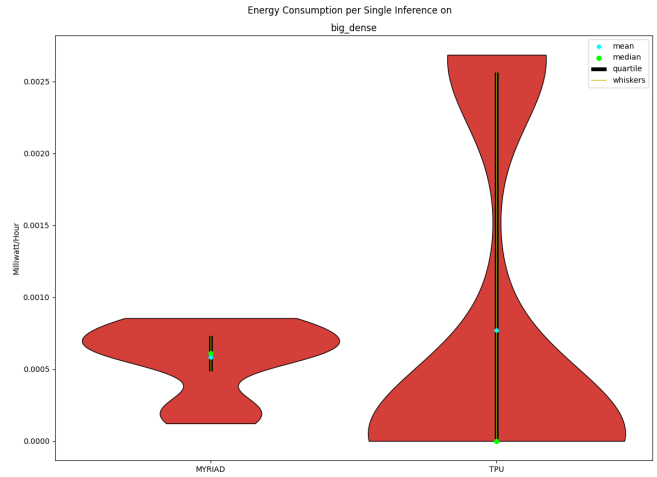

Abbildung 9. Energieverbrauch für eine Inferenz über einen großen Dense-Layer

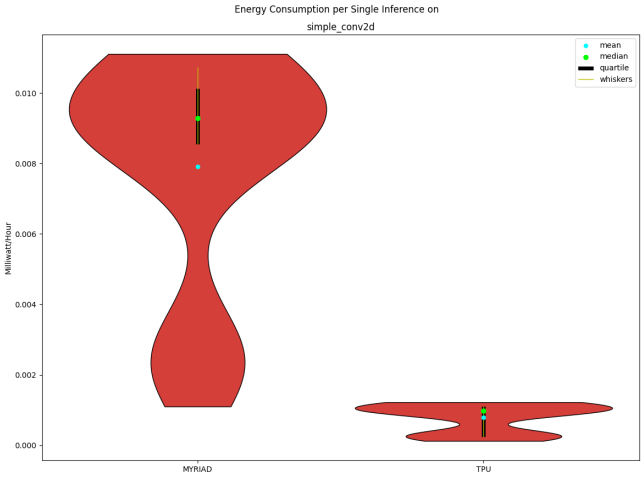

Abbildung 10. Energieverbrauch für eine Inferenz über einen einfachen Convolutional-Layer

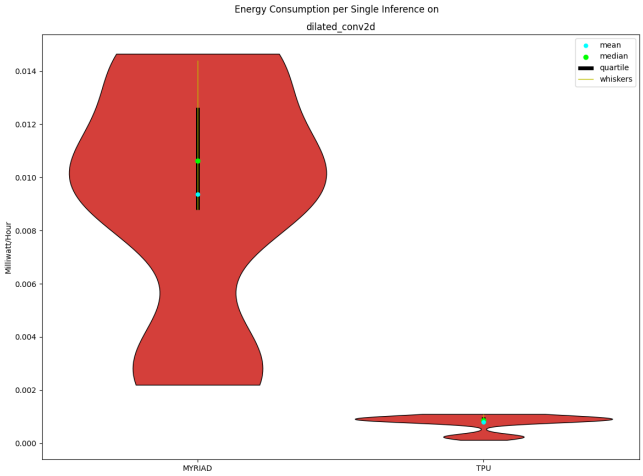

Abbildung 11. Energieverbrauch für eine Inferenz über einen dialeted Convolutional-Layer

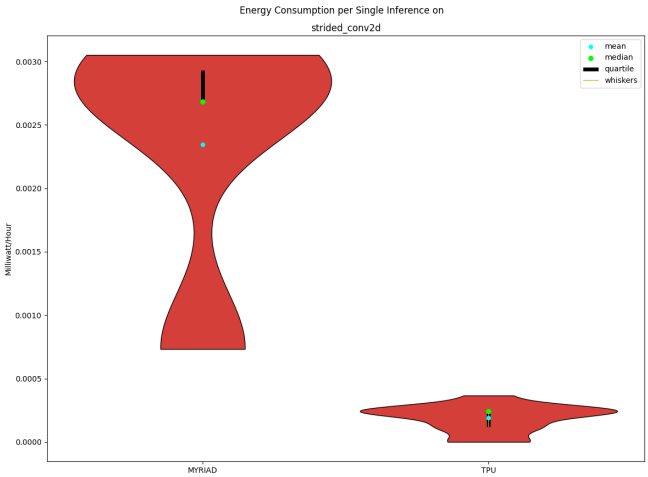

Abbildung 12. Energieverbrauch für eine Inferenz über einen strided Convolutional-Layer

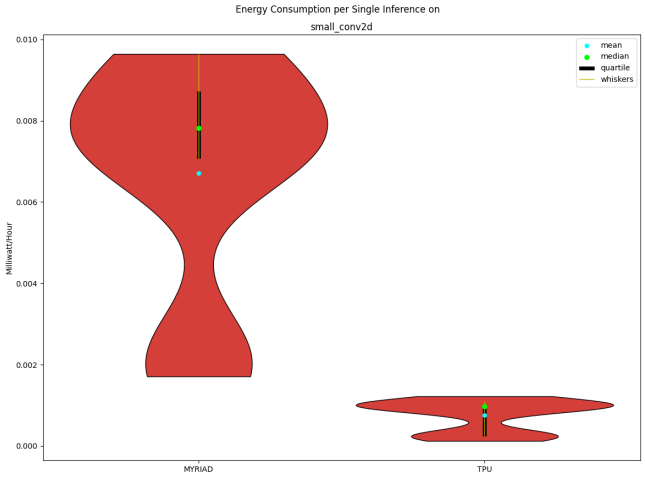

Abbildung 13. Energieverbrauch für eine Inferenz über einen kleinen Convolutional-Layer

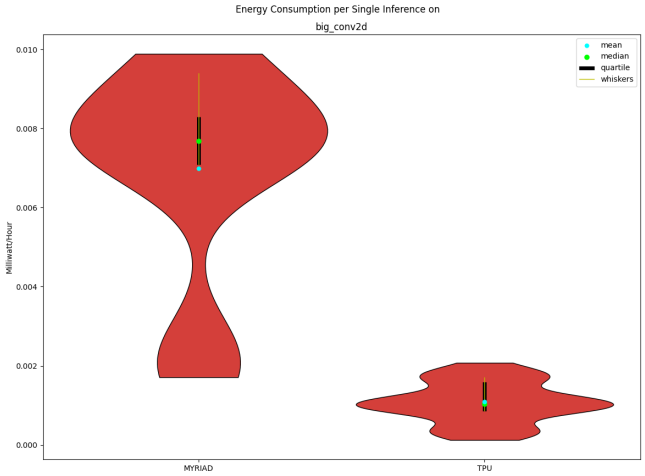

Abbildung 14. Energieverbrauch für eine Inferenz über einen großen Convolutional-Layer

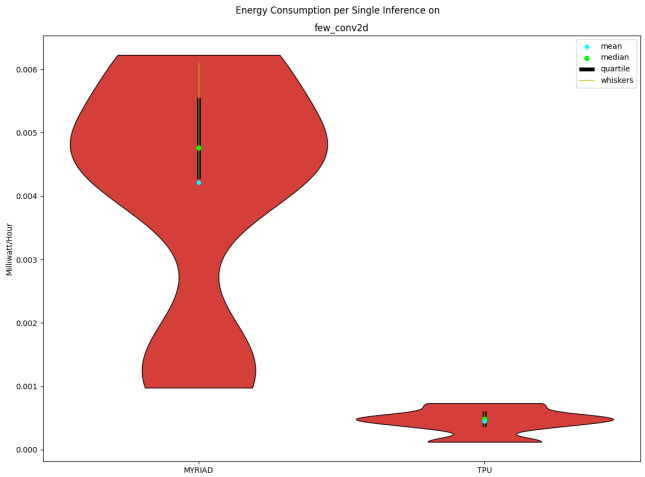

Abbildung 15. Energieverbrauch für eine Inferenz über einen Convolutional-Layer mit wenig Filtern

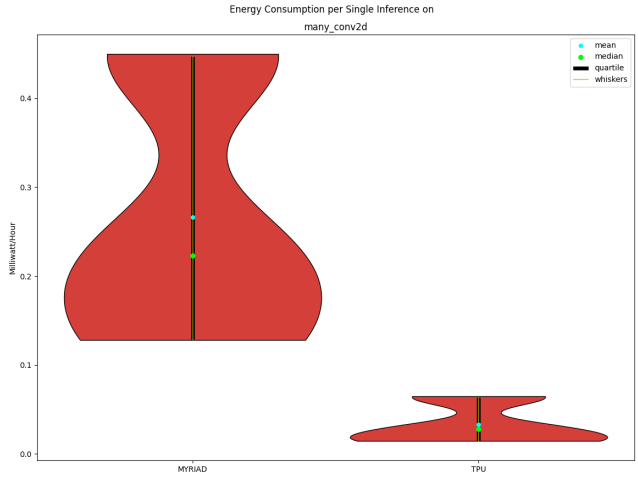

Abbildung 16. Energieverbrauch für eine Inferenz über einen Convolutional-Layer mit vielen Filtern

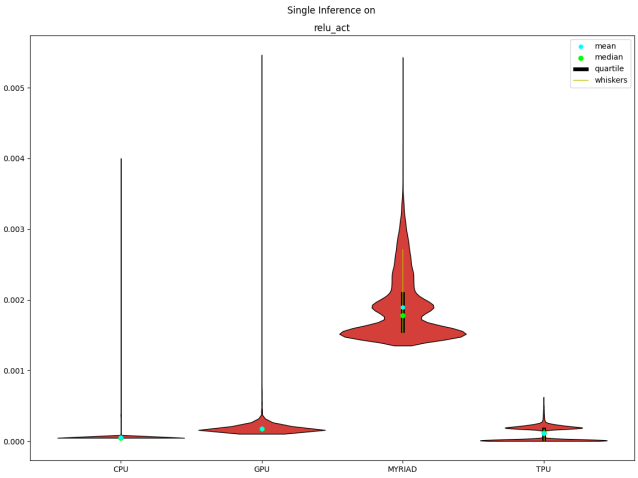

Abbildung 17. Laufzeit für eine Inferenz über einen ReLU-Layer

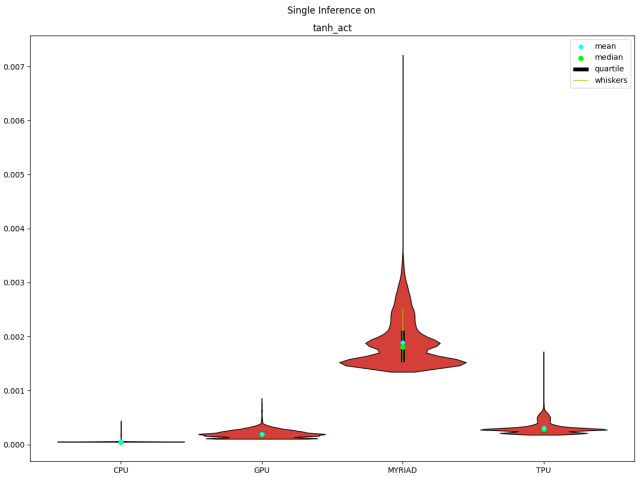

Abbildung 18. Laufzeit für eine Inferenz über einen tanh-Layer

Abbildung 19. Laufzeit für eine Inferenz über einen kleinen Dense-Layer

B. Einzelne Inferenz-Laufzeit Messungen

Abbildung 20. Laufzeit für eine Inferenz über einen sigmoid-Layer

Abbildung 21. Laufzeit für eine Inferenz über einen Skalarprodukt-Layer

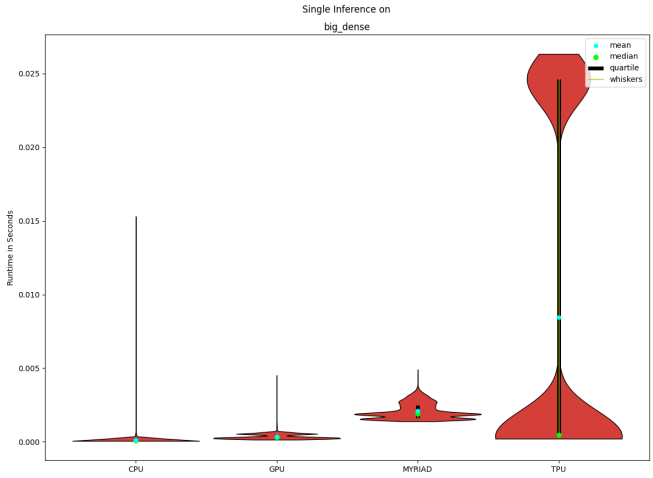

Abbildung 22. Laufzeit für eine Inferenz über einen großen Dense-Layer

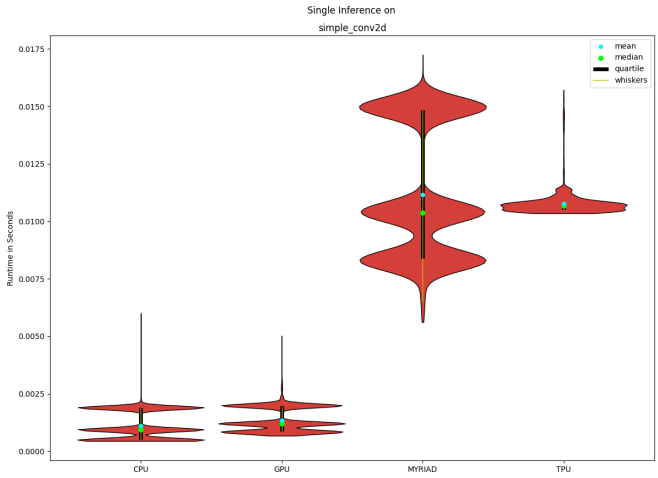

Abbildung 23. Laufzeit für eine Inferenz über einen einfachen Convolutional-Layer

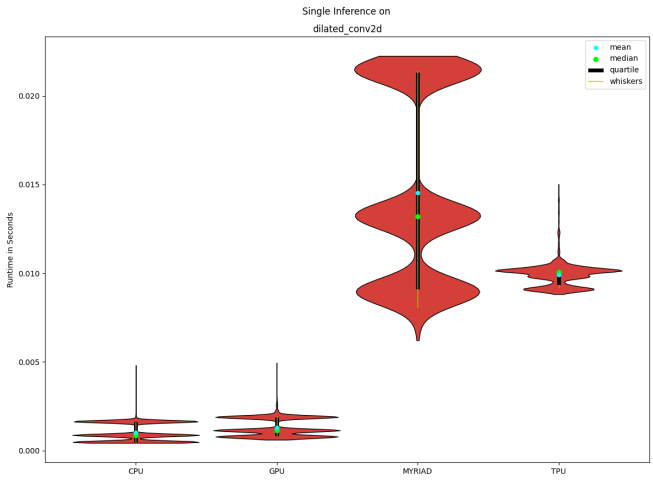

Abbildung 24. Laufzeit für eine Inferenz über einen dialeted Convolutional-Layer

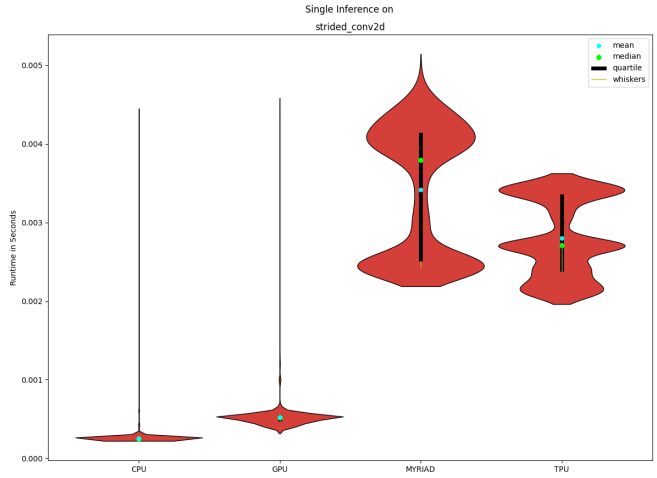

Abbildung 25. Laufzeit für eine Inferenz über einen strided Convolutional-Layer

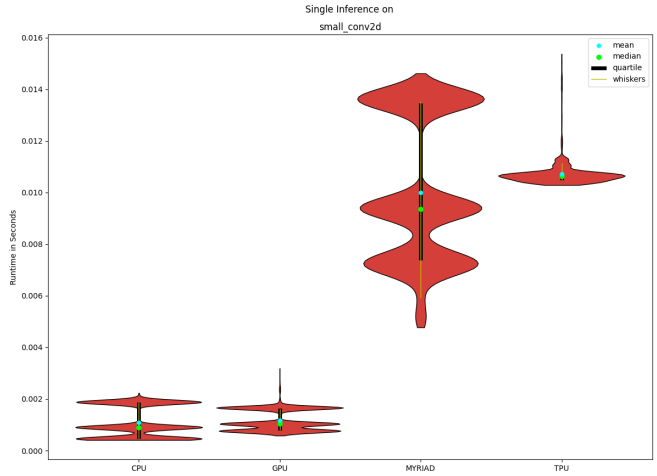

Abbildung 26. Laufzeit für eine Inferenz über einen kleinen Convolutional-Layer

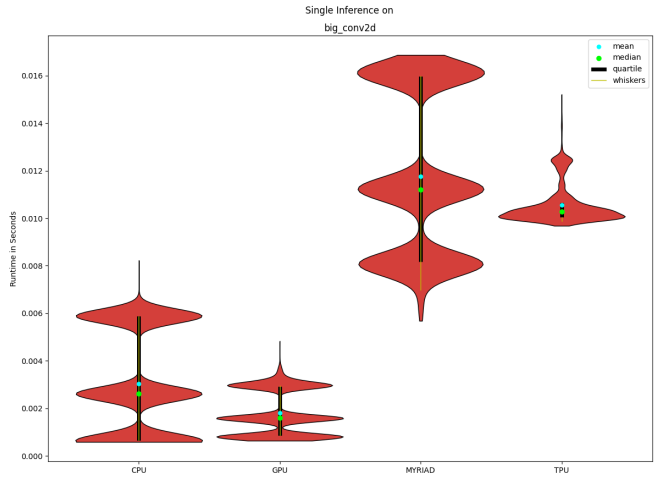

Abbildung 27. Laufzeit für eine Inferenz über einen großen Convolutional-Layer

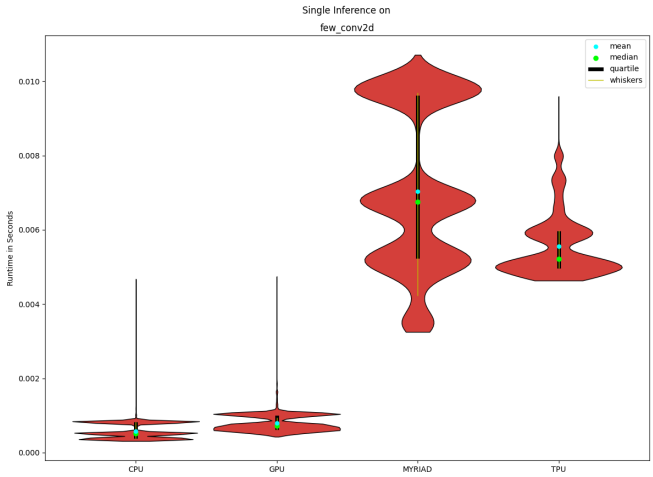

Abbildung 28. Laufzeit für eine Inferenz über einen Convolutional-Layer mit wenig Filtern

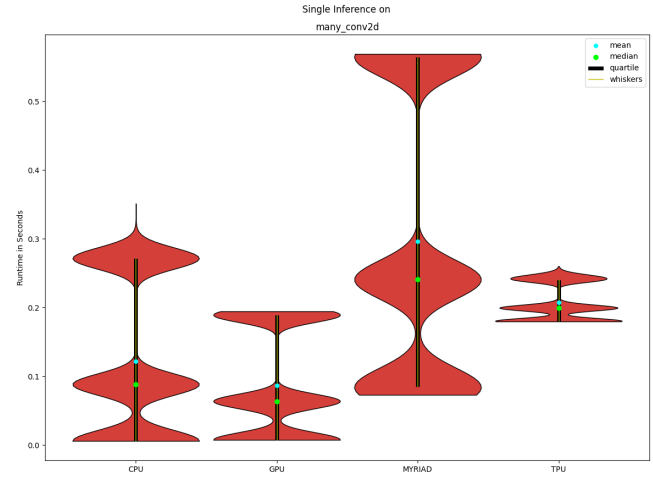

Abbildung 29. Laufzeit für eine Inferenz über einen Convolutional-Layer mit vielen Filtern

C. Batch Inferenz-Laufzeit Messungen

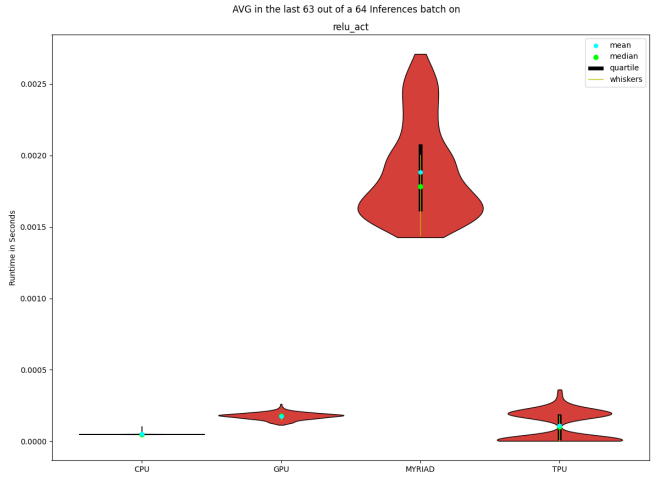

Abbildung 30. AVG Laufzeit für 63 Inferenzen über einen ReLU-Layer

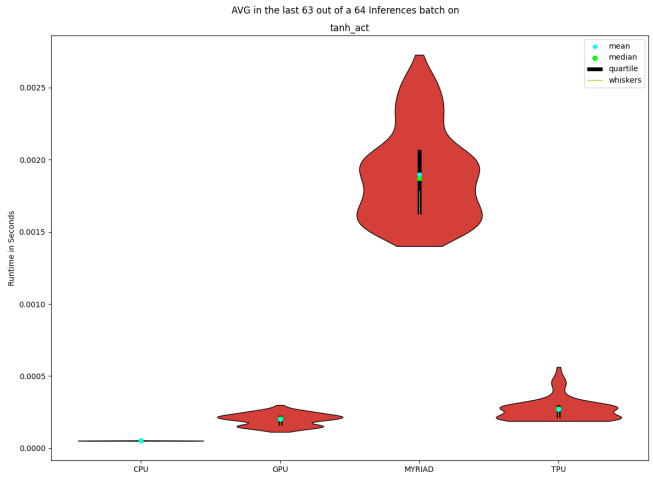

Abbildung 31. AVG Laufzeit für 63 Inferenzen über einen tanh-Layer

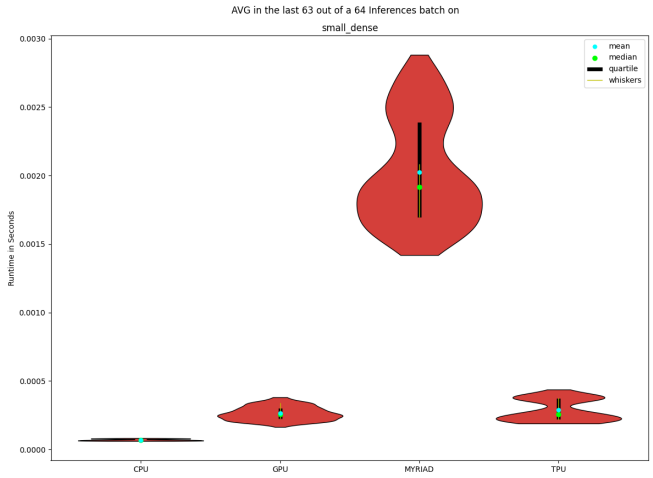

Abbildung 32. AVG Laufzeit für 63 Inferenzen über einen kleinen Dense-Layer

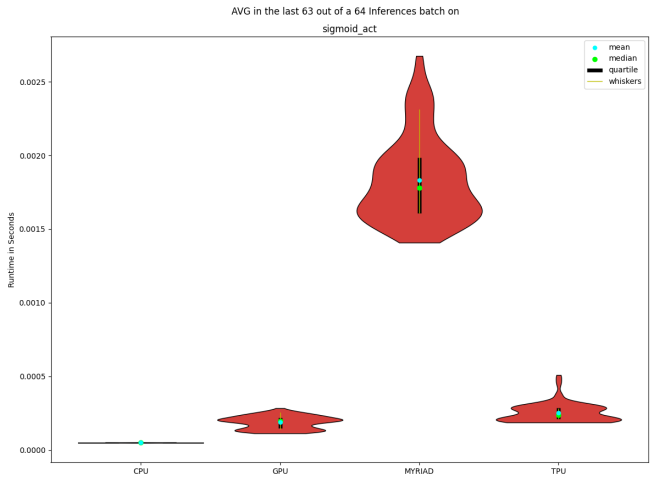

Abbildung 33. AVG Laufzeit für 63 Inferenzen über einen sigmoid-Layer

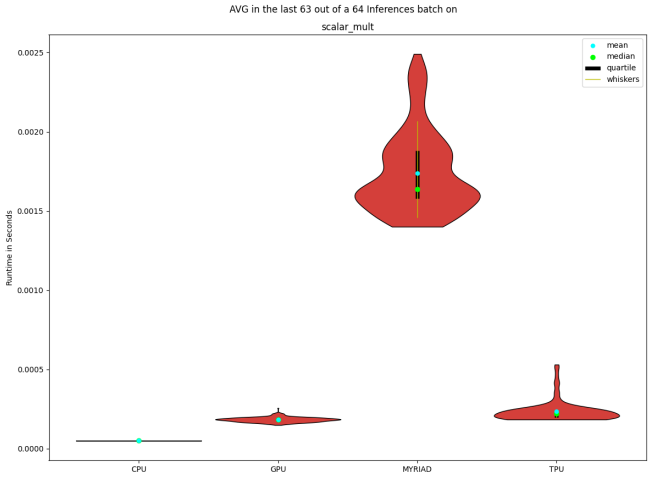

Abbildung 34. AVG Laufzeit für 63 Inferenzen über einen Skalarprodukt-Layer

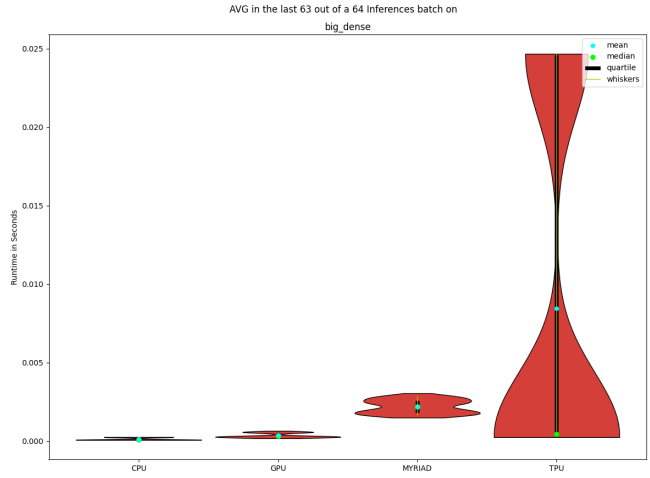

Abbildung 35. AVG Laufzeit für 63 Inferenzen über einen großen Dense-Layer

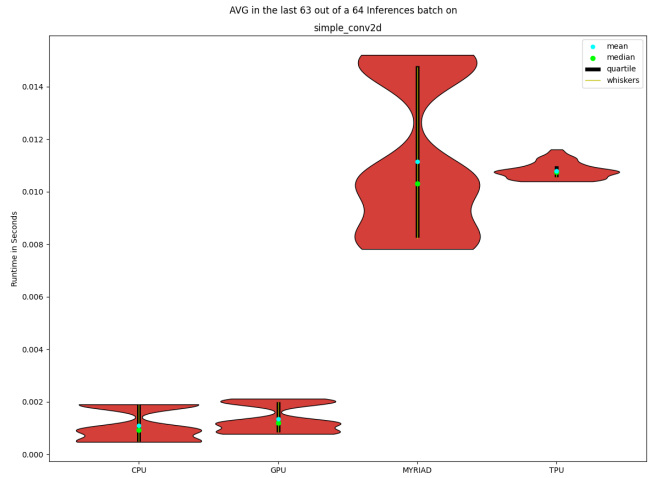

Abbildung 36. AVG Laufzeit für 63 Inferenzen über einen einfachen Convolutional-Layer

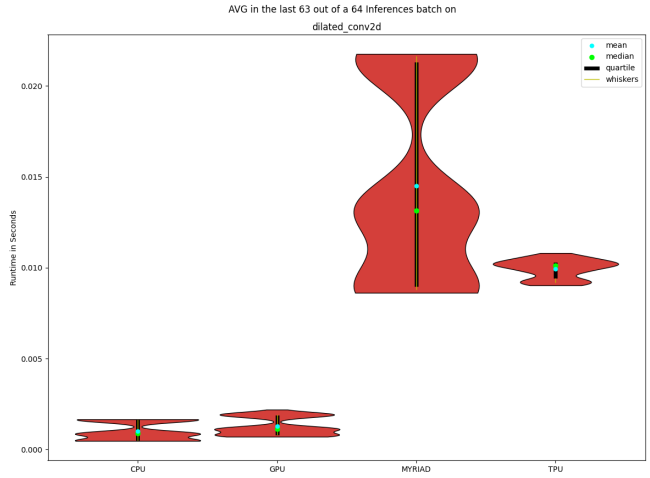

Abbildung 37. AVG Laufzeit für 63 Inferenzen über einen dialeted Convolutional-Layer

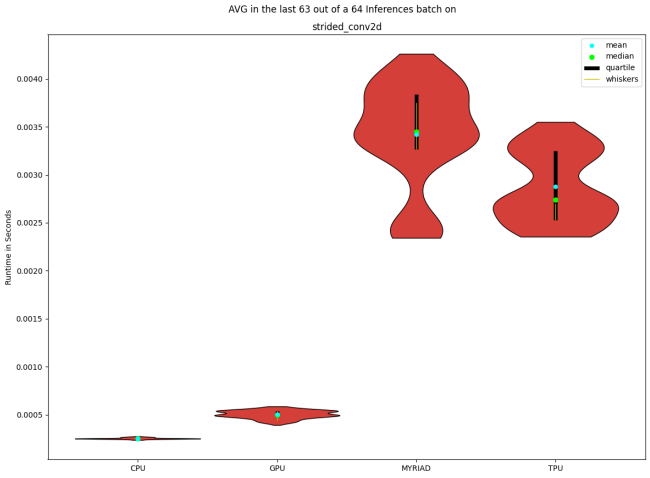

Abbildung 38. AVG Laufzeit für 63 Inferenzen über einen strided Convolutional-Layer

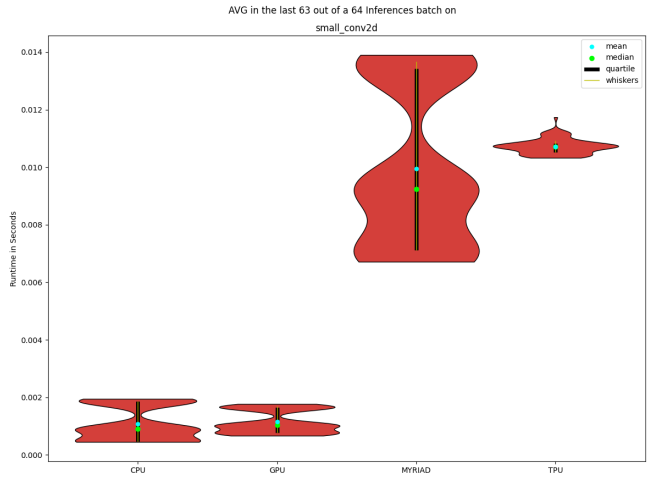

Abbildung 39. AVG Laufzeit für 63 Inferenzen über einen kleinen Convolutional-Layer

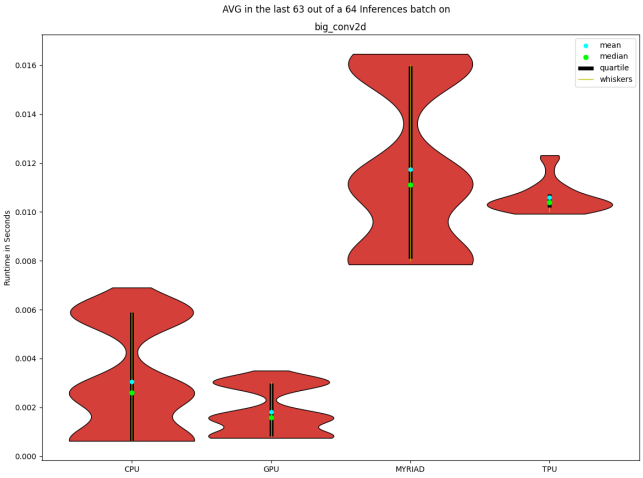

Abbildung 40. AVG Laufzeit für 63 Inferenzen über einen großen Convolutional-Layer

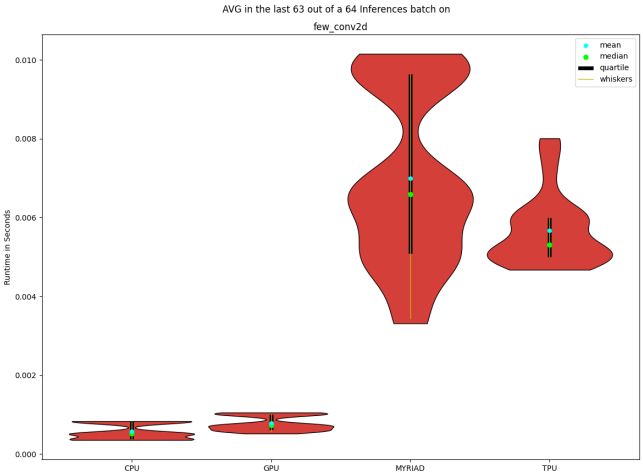

Abbildung 41. AVG Laufzeit für 63 Inferenzen über einen Convolutional-Layer mit wenig Filtern

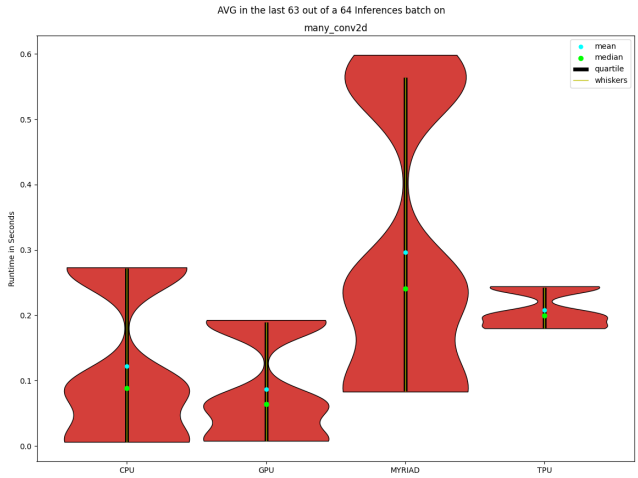

Abbildung 42. AVG Laufzeit für 63 Inferenzen über einen Convolutional-Layer mit vielen Filtern

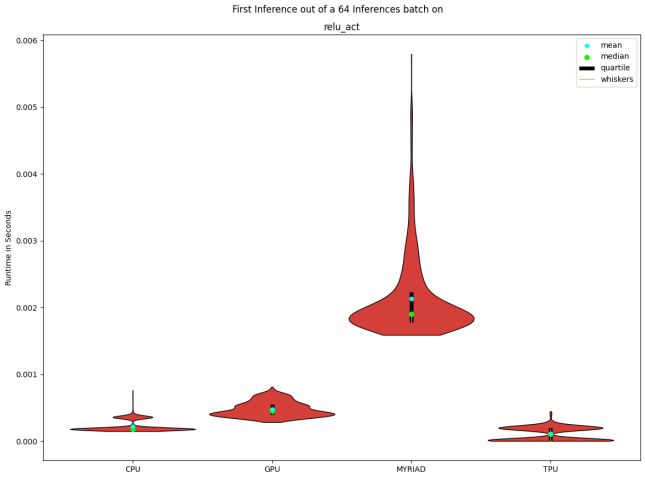

Abbildung 43. Laufzeit der ersten Inferenz über einen ReLU-Layer in einem Batchdurchlauf

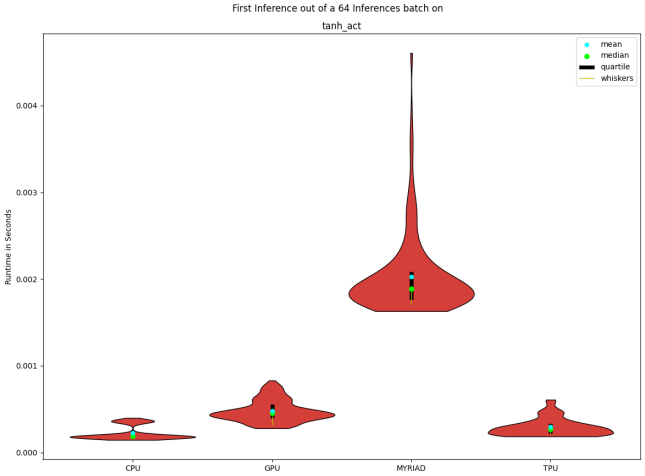

Abbildung 44. Laufzeit der ersten Inferenz über einen tanh-Layer in einem Batchdurchlauf

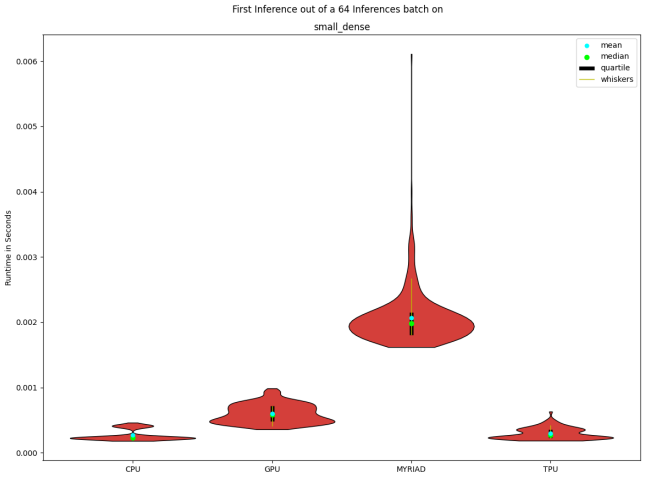

Abbildung 45. Laufzeit der ersten Inferenz über einen kleinen Dense-Layer in einem Batchdurchlauf

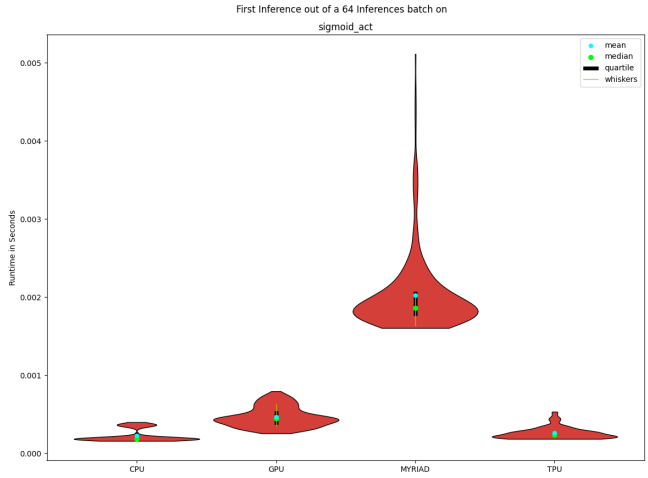

Abbildung 46. Laufzeit der ersten Inferenz über einen sigmoid-Layer in einem Batchdurchlauf

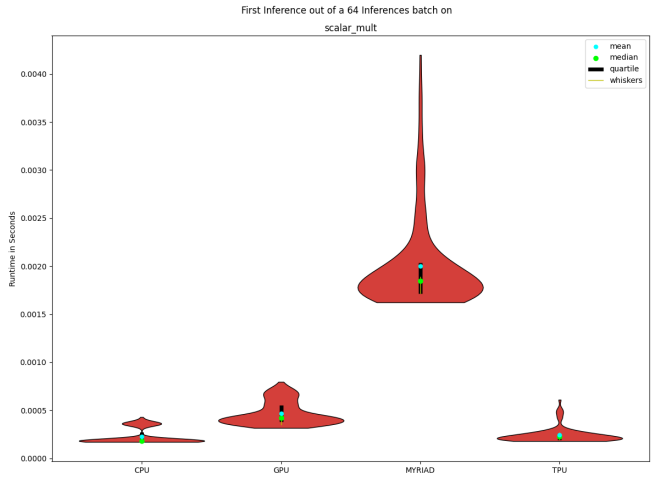

Abbildung 47. Laufzeit der ersten Inferenz über einen Skalarprodukt-Layer in einem Batchdurchlauf

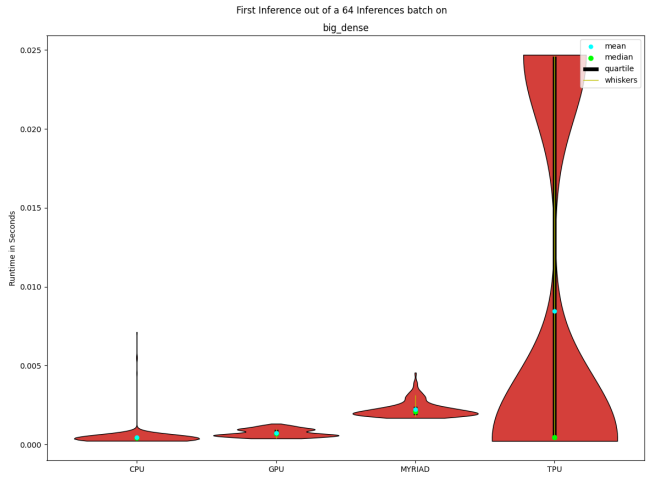

Abbildung 48. Laufzeit der ersten Inferenz über einen großen Dense-Layer in einem Batchdurchlauf

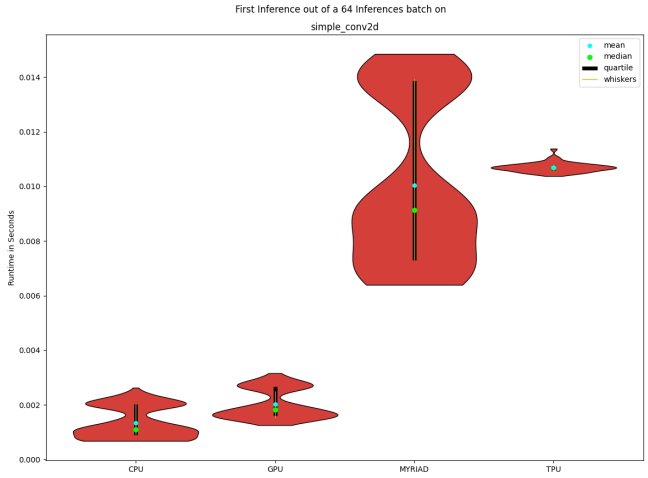

Abbildung 49. Laufzeit der ersten Inferenz über einen einfachen Convolutional-Layer in einem Batchdurchlauf

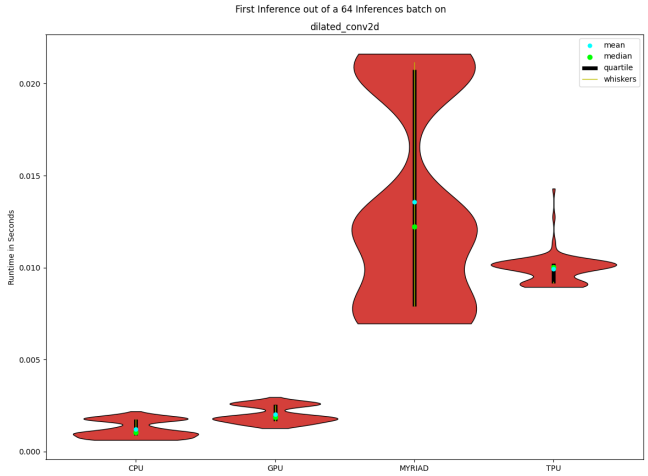

Abbildung 50. Laufzeit der ersten Inferenz über einen dialeted Convolutional-Layer in einem Batchdurchlauf

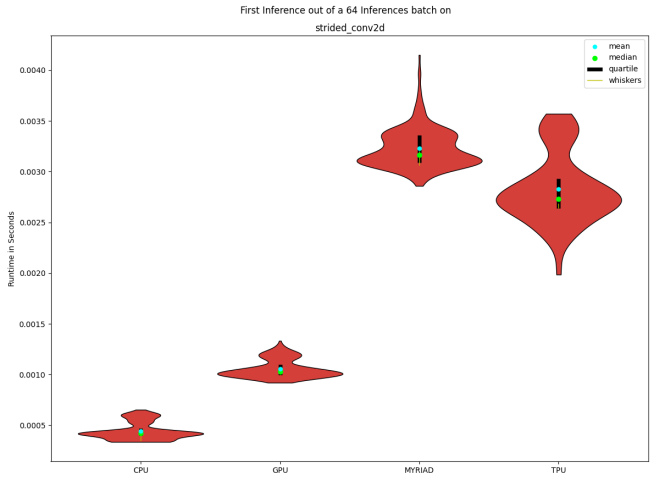

Abbildung 51. Laufzeit der ersten Inferenz über einen strided Convolutional-Layer in einem Batchdurchlauf

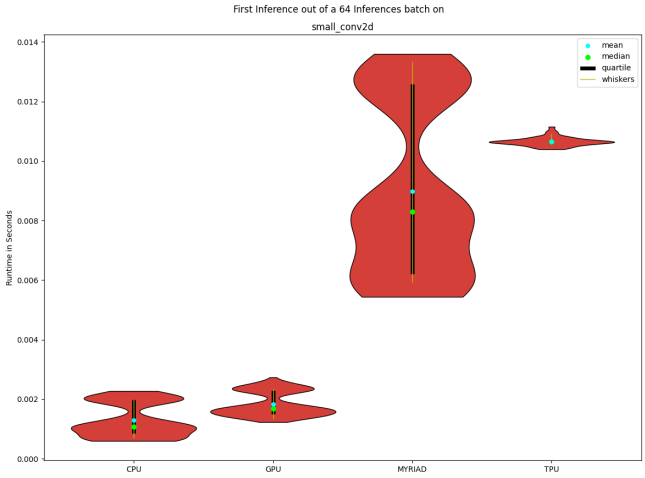

Abbildung 52. Laufzeit der ersten Inferenz über einen kleinen Convolutional-Layer in einem Batchdurchlauf

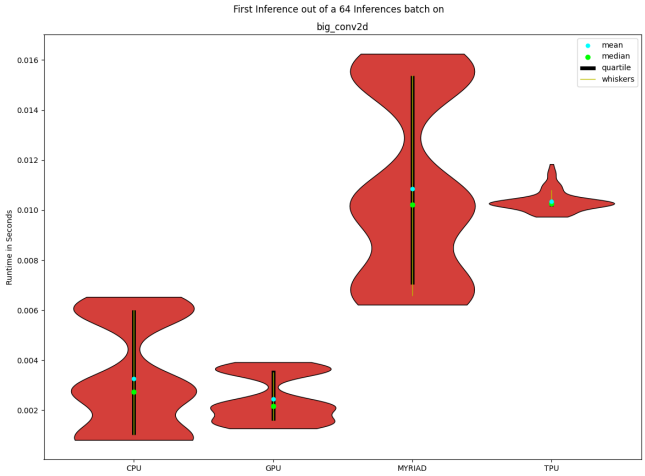

Abbildung 53. Laufzeit der ersten Inferenz über einen großen Convolutional-Layer in einem Batchdurchlauf

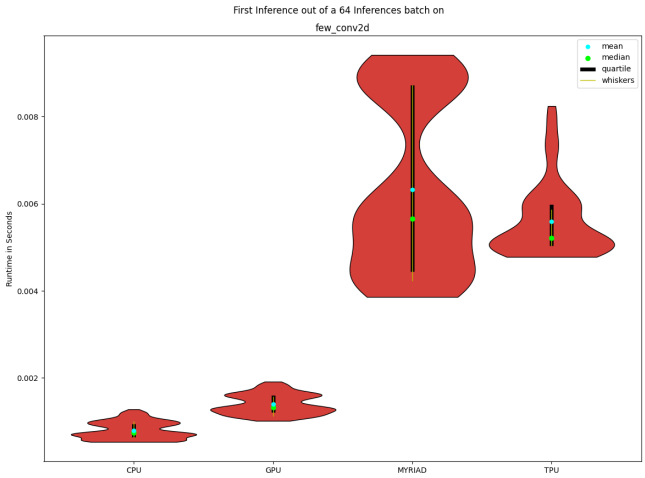

Abbildung 54. Laufzeit der ersten Inferenz über einen Convolutional-Layer mit wenig Filtern in einem Batchdurchlauf

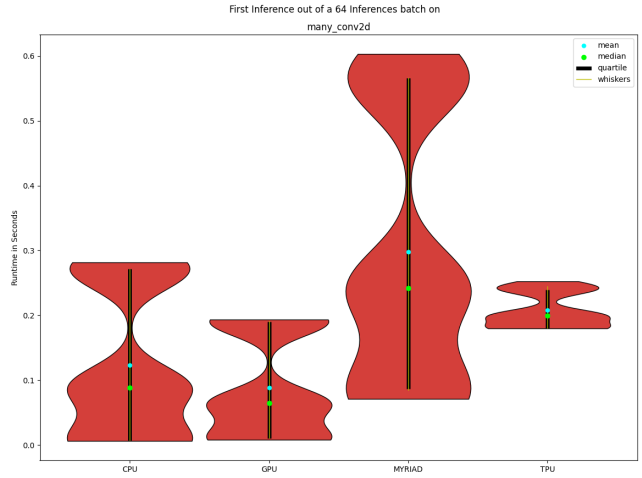

Abbildung 55. Laufzeit der ersten Inferenz über einen Convolutional-Layer mit vielen Filtern in einem Batchdurchlauf

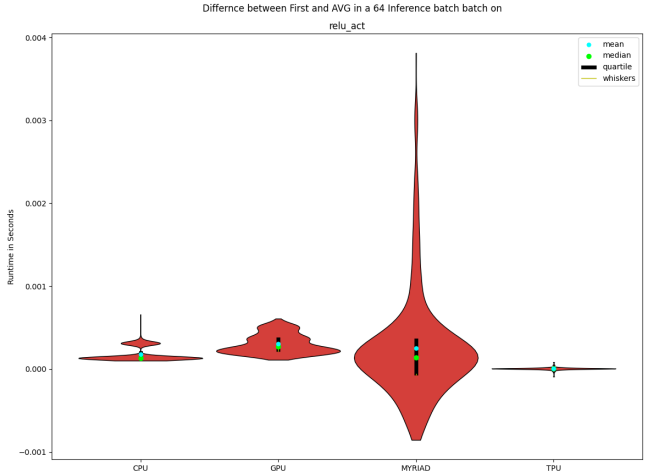

Abbildung 56. Differenz Laufzeit der ersten Inferenz und des AVG über einen ReLU-Layer in einem Batchdurchlauf

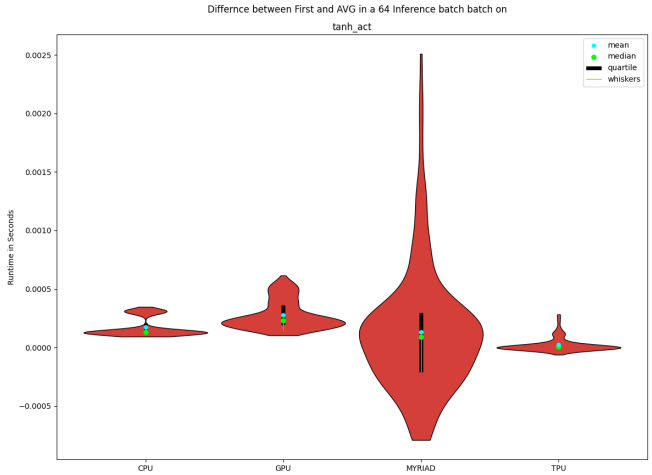

Abbildung 57. Differenz Laufzeit der ersten Inferenz und des AVG über einen tanh-Layer in einem Batchdurchlauf

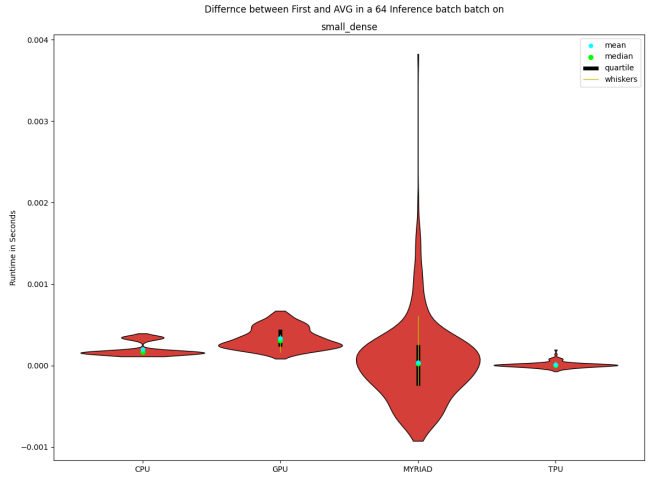

Abbildung 58. Differenz Laufzeit der ersten Inferenz und des AVG über einen kleinen Dense-Layer in einem Batchdurchlauf

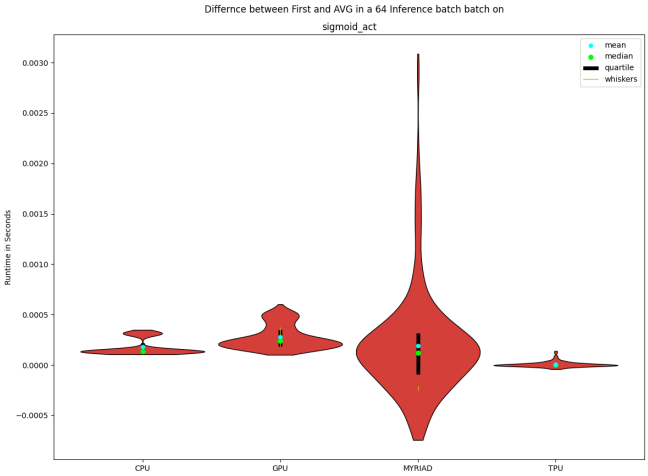

Abbildung 59. Differenz Laufzeit der ersten Inferenz und des AVG über einen sigmoid-Layer in einem Batchdurchlauf

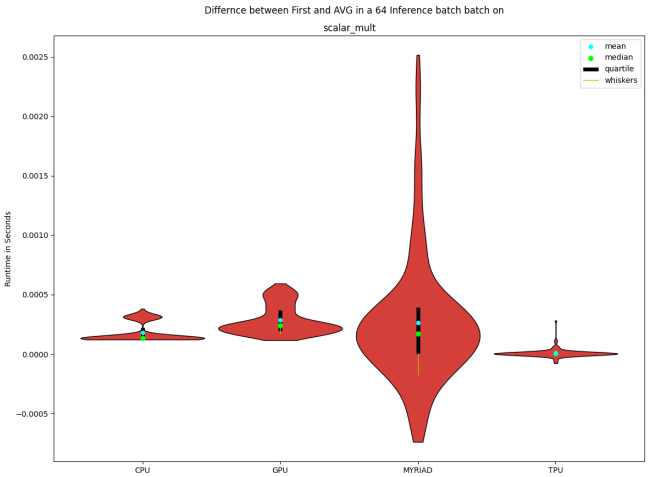

Abbildung 60. Differenz Laufzeit der ersten Inferenz und des AVG über einen Skalarprodukt-Layer in einem Batchdurchlauf

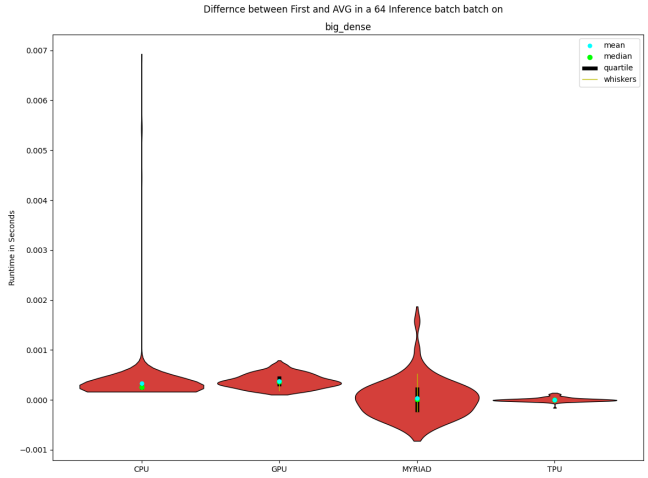

Abbildung 61. Differenz Laufzeit der ersten Inferenz und des AVG über einen großen Dense-Layer in einem Batchdurchlauf

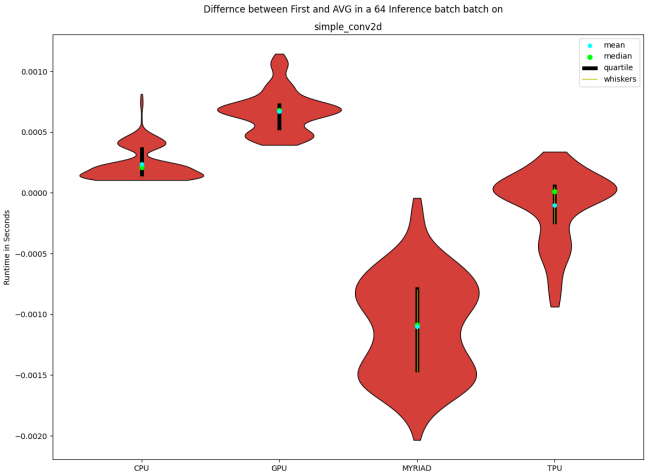

Abbildung 62. Differenz Laufzeit der ersten Inferenz und des AVG über einen einfachen Convolutional-Layer in einem Batchdurchlauf

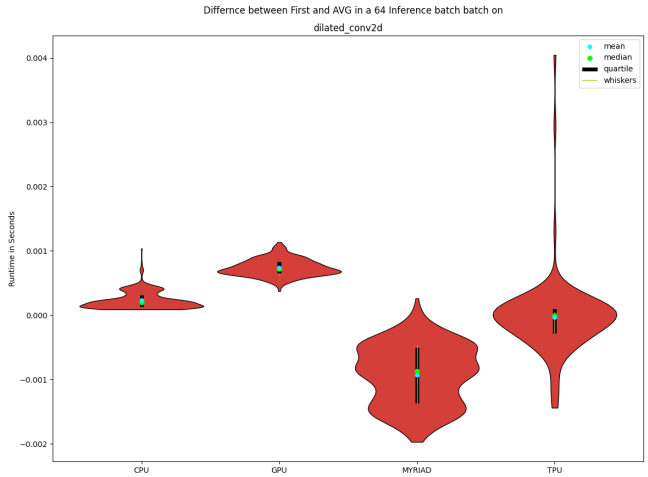

Abbildung 63. Differenz Laufzeit der ersten Inferenz und des AVG über einen dialeted Convolutional-Layer in einem Batchdurchlauf

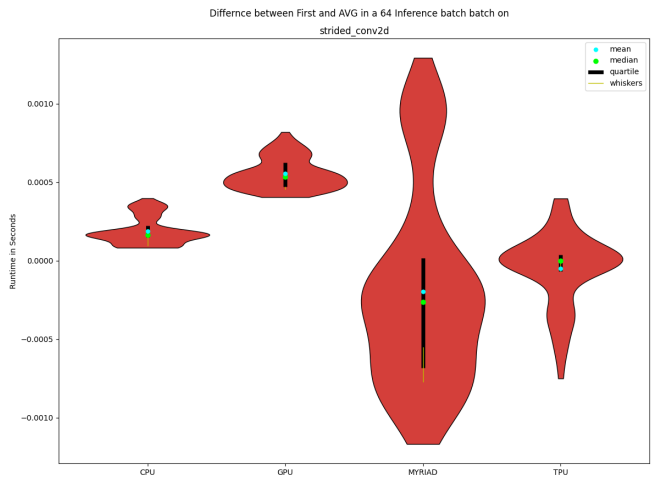

Abbildung 64. Differenz Laufzeit der ersten Inferenz und des AVG über einen strided Convolutional-Layer in einem Batchdurchlauf

Abbildung 65. Differenz Laufzeit der ersten Inferenz und des AVG über einen kleinen Convolutional-Layer in einem Batchdurchlauf

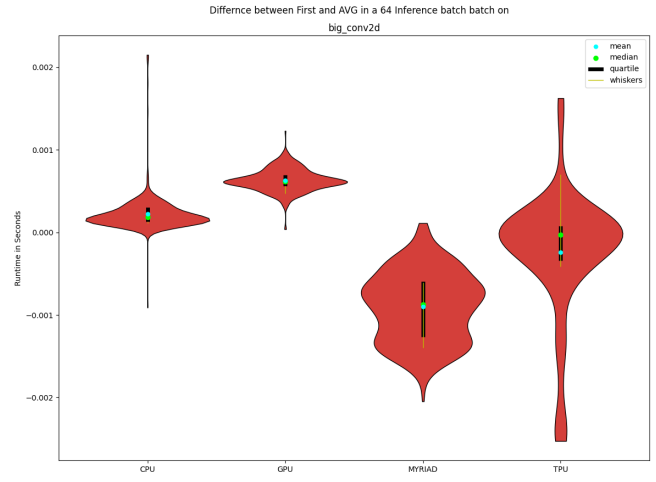

Abbildung 66. Differenz Laufzeit der ersten Inferenz und des AVG über einen großen Convolutional-Layer in einem Batchdurchlauf

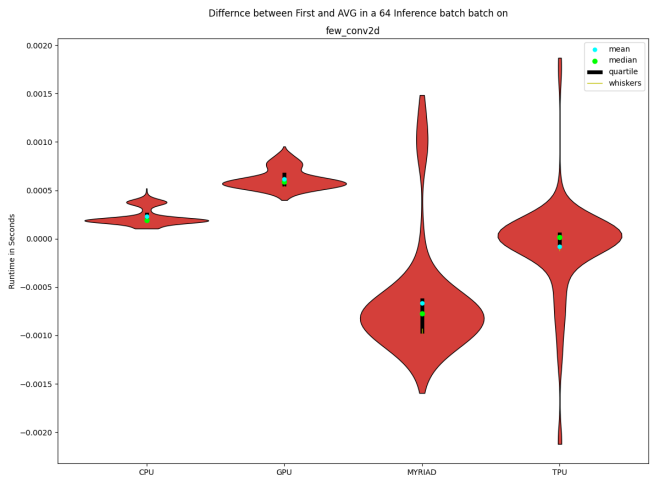

Abbildung 67. Differenz Laufzeit der ersten Inferenz und des AVG über einen Convolutional-Layer mit wenig Filtern in einem Batchdurchlauf

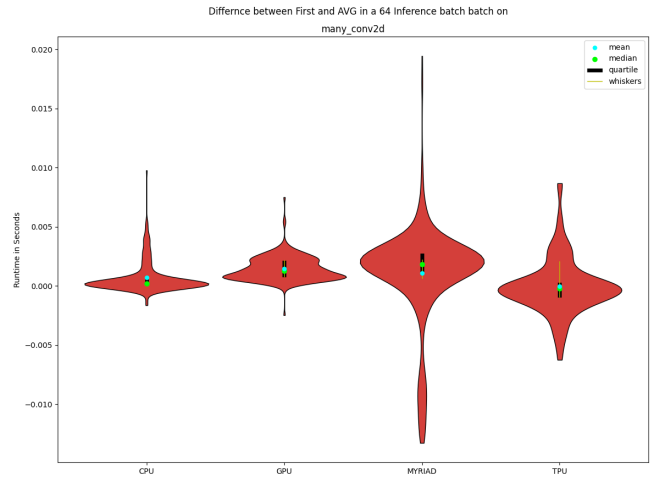

Abbildung 68. Differenz Laufzeit der ersten Inferenz und des AVG über einen Convolutional-Layer mit vielen Filtern in einem Batchdurchlauf

D. Benchmarks mit gestackten Layern

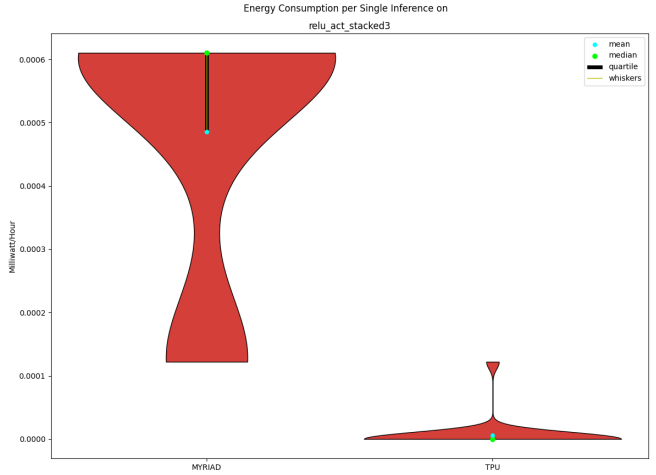

Abbildung 69. Energieverbrauch für eine Inferenz über ein Netz aus 3 ReLU-Layern

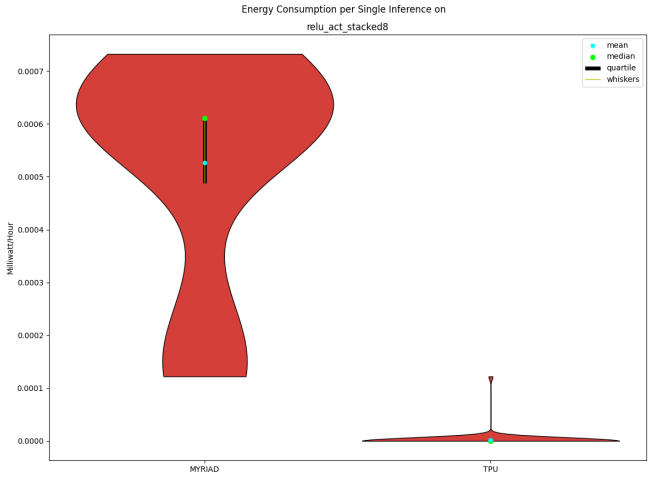

Abbildung 70. Energieverbrauch für eine Inferenz über ein Netz aus 8 ReLU-Layern

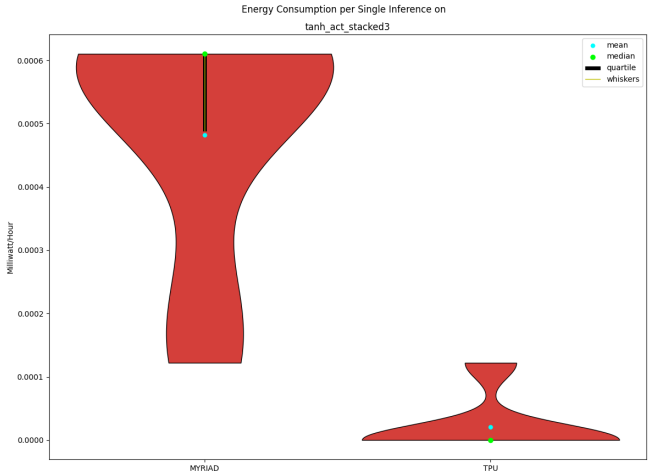

Abbildung 71. Energieverbrauch für eine Inferenz über ein Netz aus 3 tanh-Layern

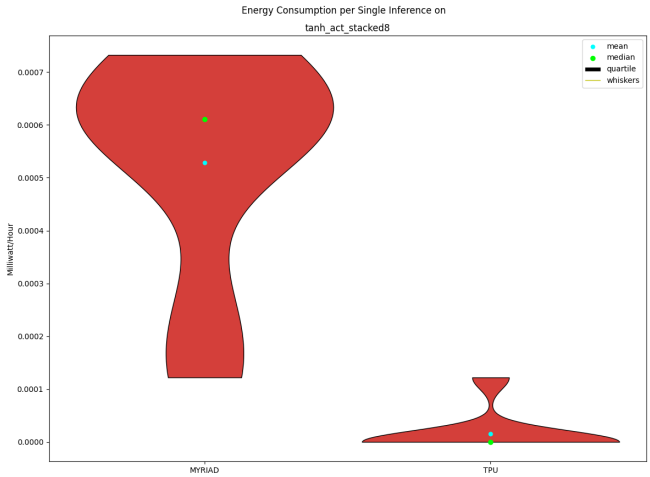

Abbildung 72. Energieverbrauch für eine Inferenz über ein Netz aus 8 tanh-Layern

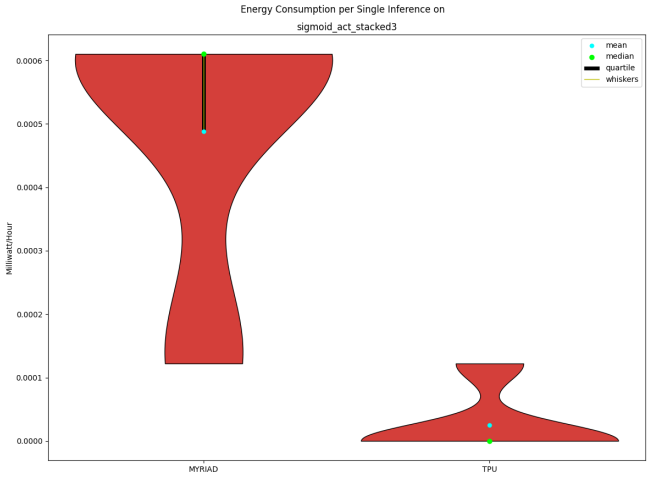

Abbildung 73. Energieverbrauch für eine Inferenz über ein Netz aus 3 sigmoid-Layern

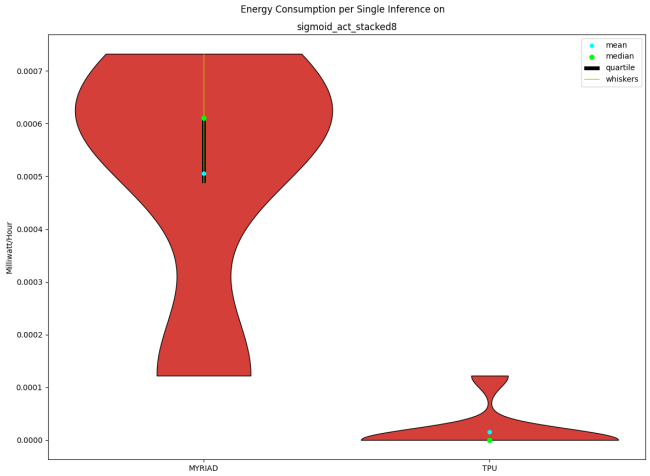

Abbildung 74. Energieverbrauch für eine Inferenz über ein Netz aus 8 sigmoid-Layern

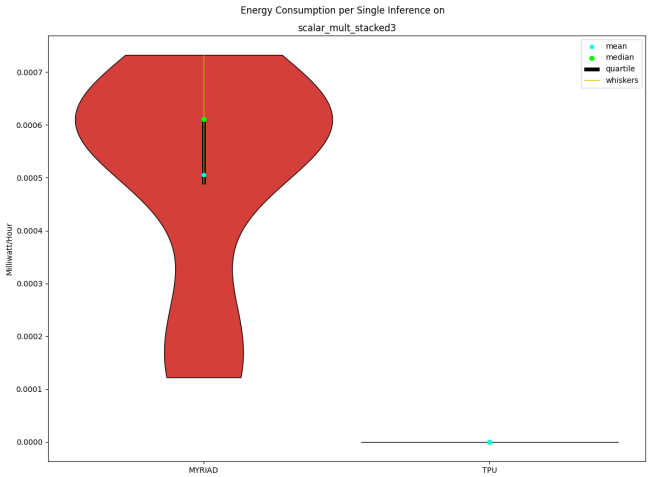

Abbildung 75. Energieverbrauch für eine Inferenz über ein Netz aus 3 Skalarprodukt-Layern

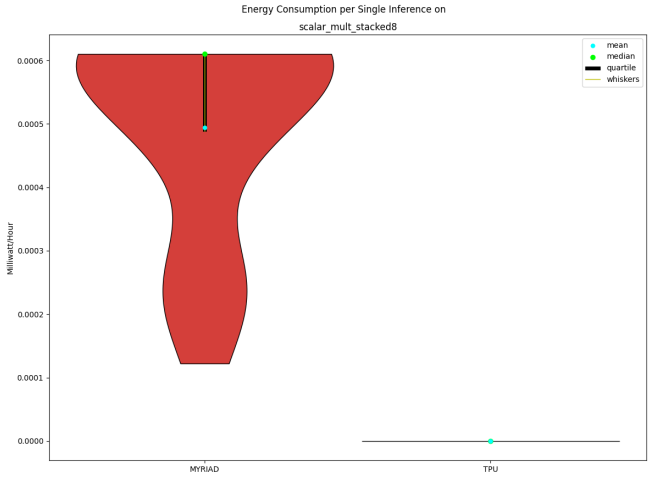

Abbildung 76. Energieverbrauch für eine Inferenz über ein Netz aus 8 Skalarprodukt-Layern

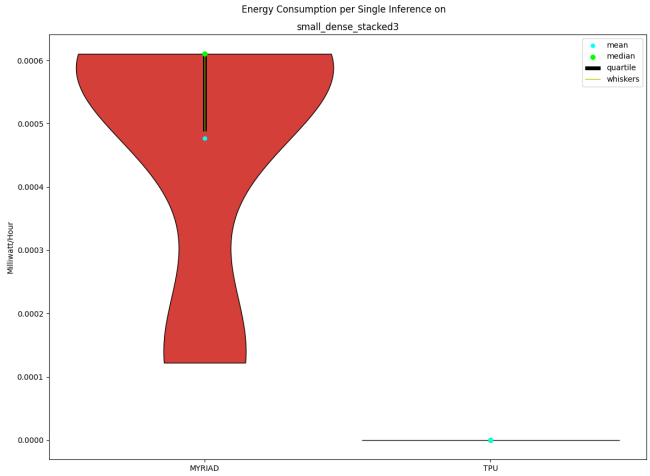

Abbildung 77. Energieverbrauch für eine Inferenz über ein Netz aus 3 kleinen Dense-Layern

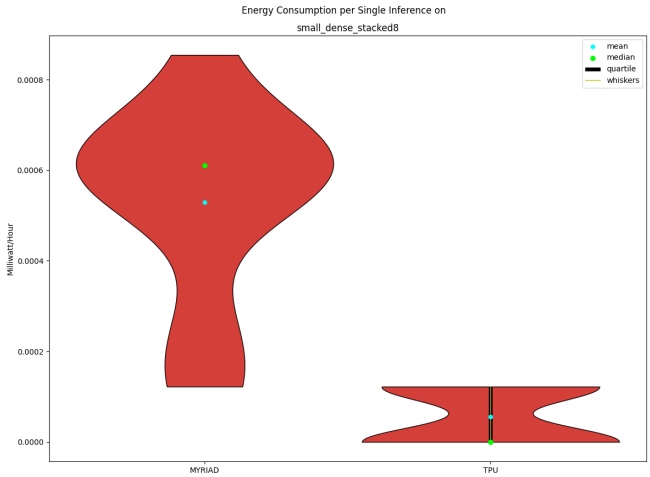

Abbildung 78. Energieverbrauch für eine Inferenz über ein Netz aus 8 kleinen Dense-Layern

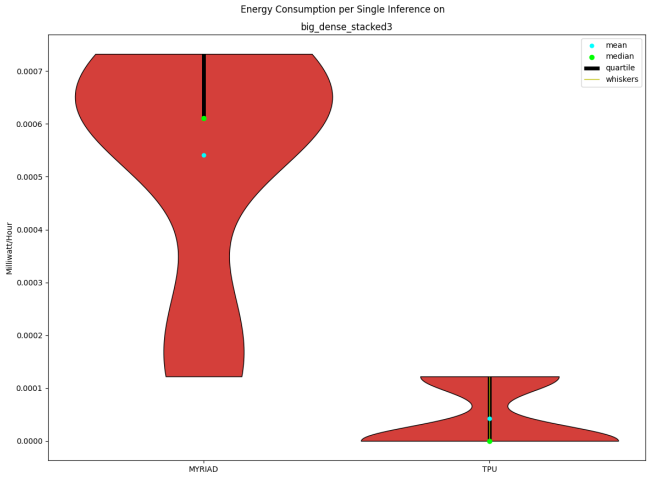

Abbildung 79. Energieverbrauch für eine Inferenz über ein Netz aus 3 großen Dense-Layern

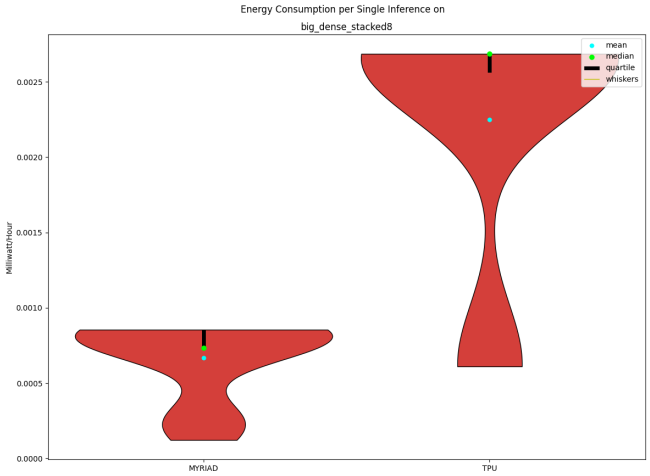

Abbildung 80. Energieverbrauch für eine Inferenz über ein Netz aus 8 großen Dense-Layern

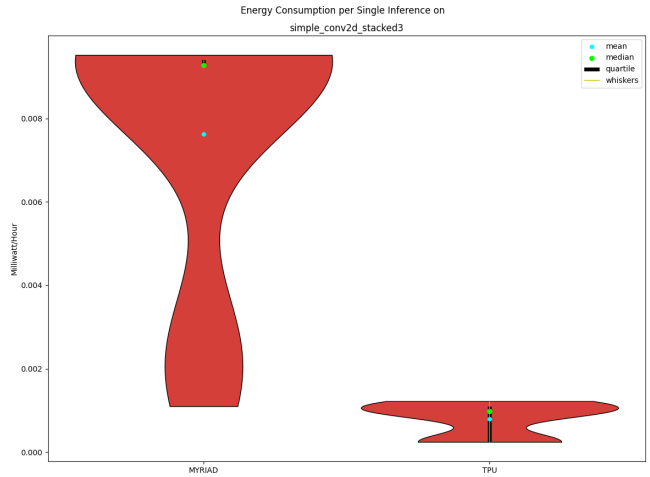

Abbildung 81. Energieverbrauch für eine Inferenz über ein Netz aus 3 einfachen Convolutional-Layern

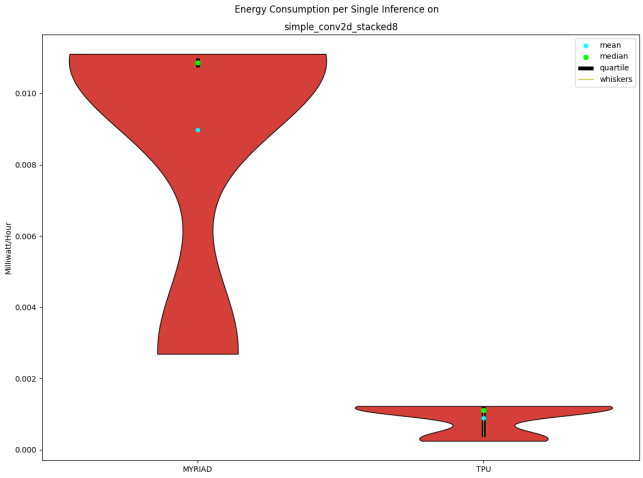

Abbildung 82. Energieverbrauch für eine Inferenz über ein Netz aus 8 einfachen Convolutional-Layern

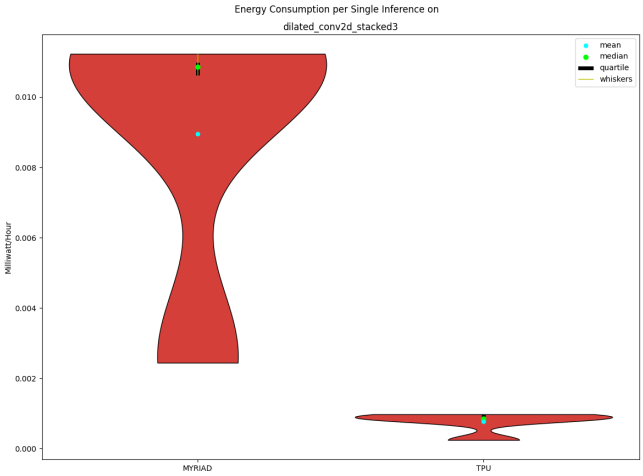

Abbildung 83. Energieverbrauch für eine Inferenz über ein Netz aus 3 dialeted Convolutional-Layern

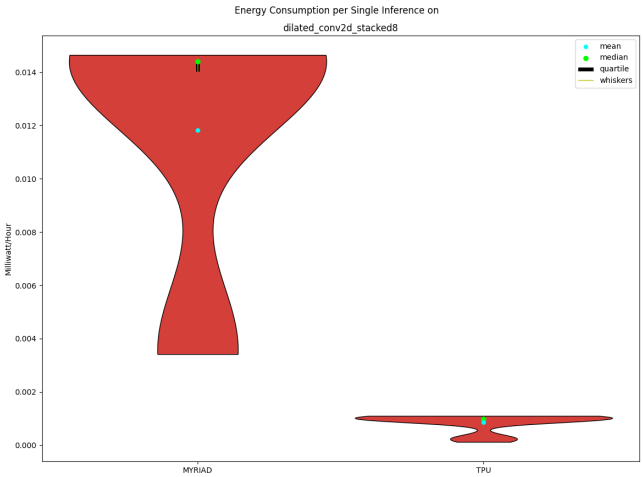

Abbildung 84. Energieverbrauch für eine Inferenz über ein Netz aus 8 dialeted Convolutional-Layern

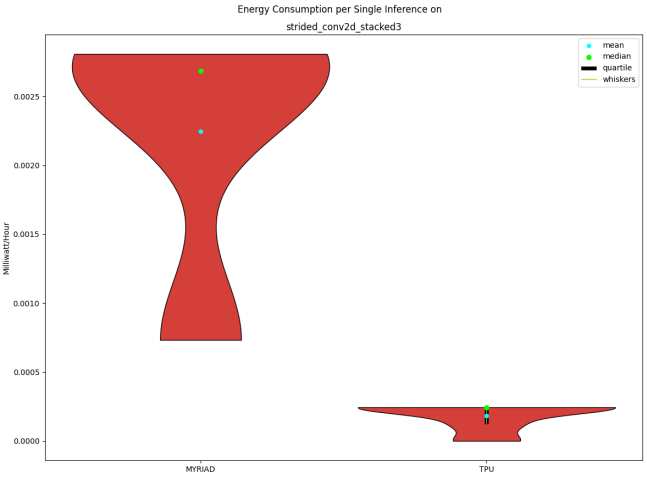

Abbildung 85. Energieverbrauch für eine Inferenz über ein Netz aus 3 strided Convolutional-Layern

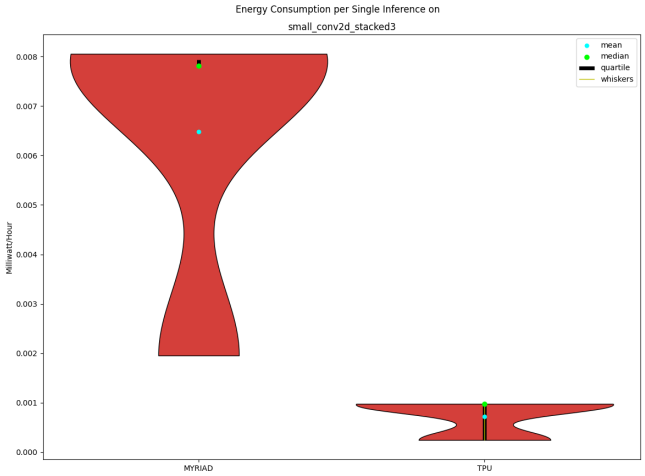

Abbildung 86. Energieverbrauch für eine Inferenz über ein Netz aus 3 kleinen Convolutional-Layern

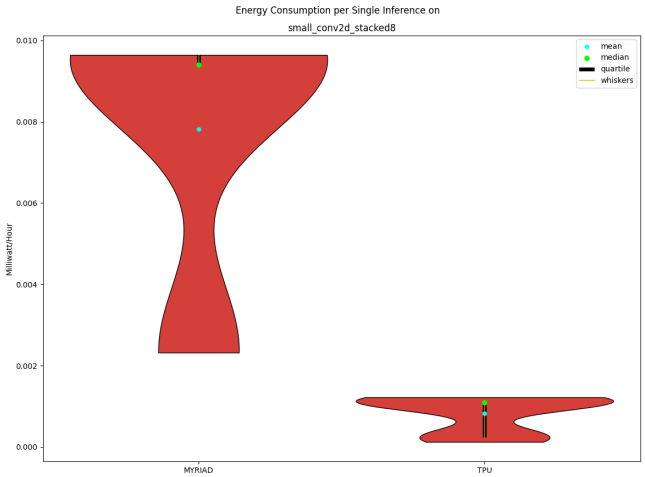

Abbildung 87. Energieverbrauch für eine Inferenz über ein Netz aus 8 kleinen Convolutional-Layern

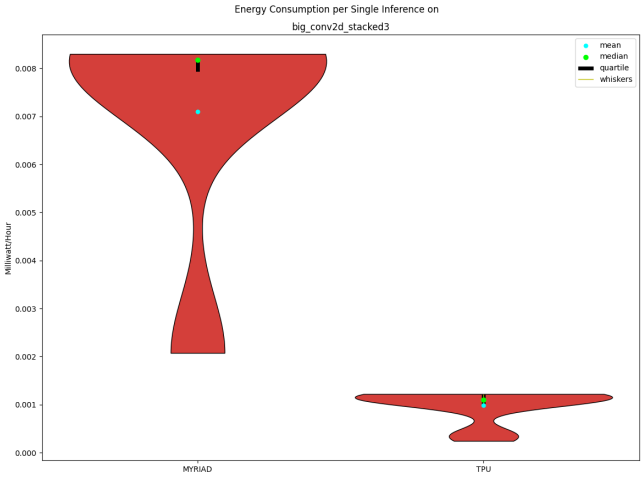

Abbildung 88. Energieverbrauch für eine Inferenz über ein Netz aus 3 großen Convolutional-Layern

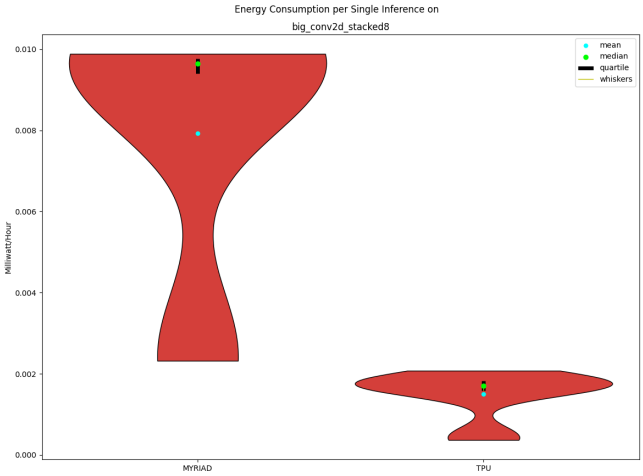

Abbildung 89. Energieverbrauch für eine Inferenz über ein Netz aus 8 großen Convolutional-Layern

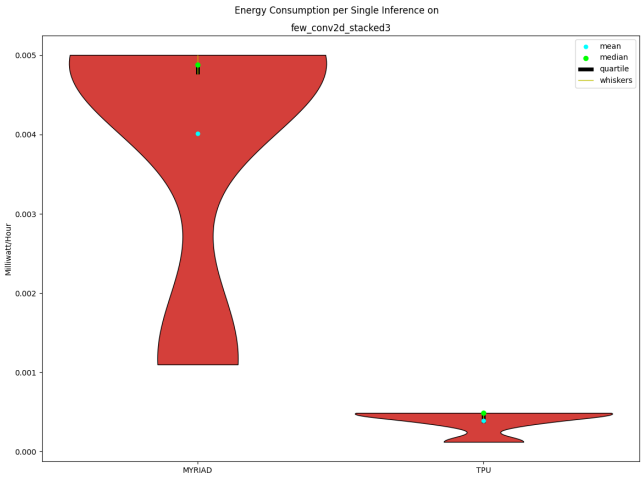

Abbildung 90. Energieverbrauch für eine Inferenz über ein Netz aus 3 Convolutional-Layern mit wenig Filtern

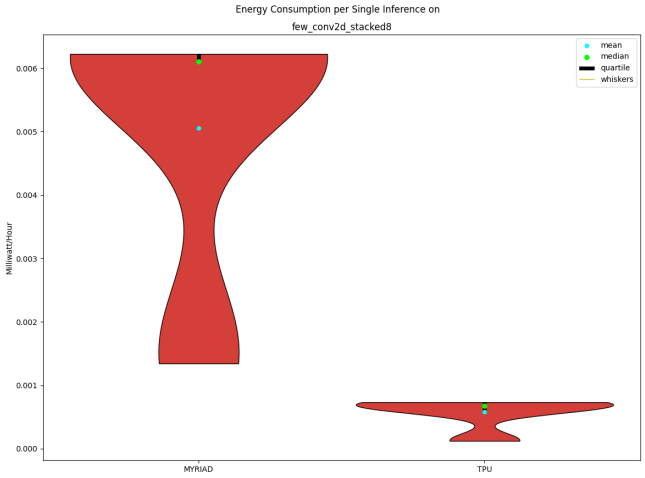

Abbildung 91. Energieverbrauch für eine Inferenz über ein Netz aus 8 Convolutional-Layern mit wenig Filtern

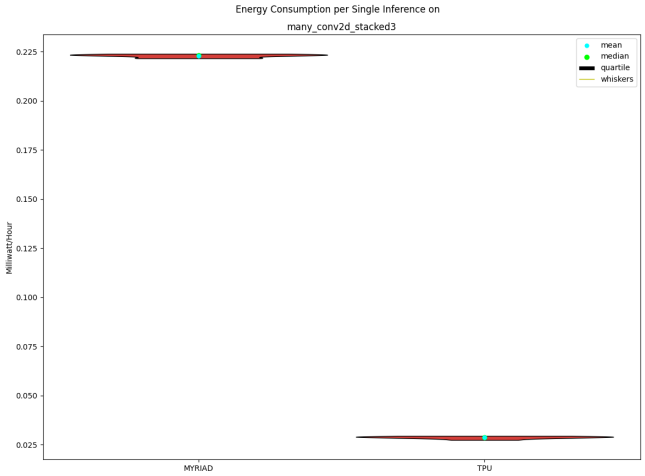

Abbildung 92. Energieverbrauch für eine Inferenz über ein Netz aus 3 Convolutional-Layern mit vielen Filtern

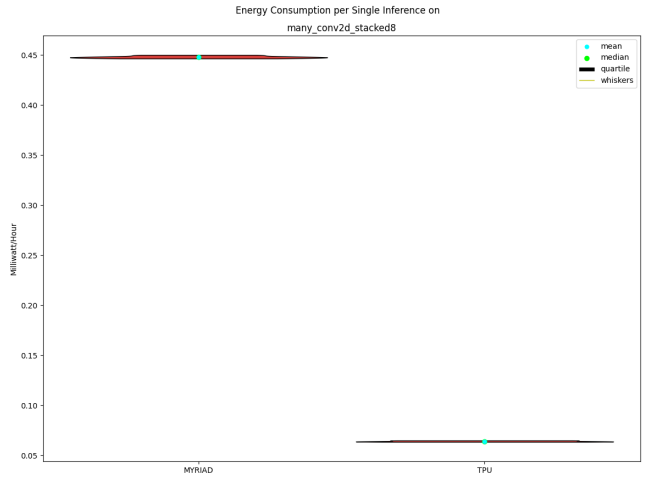

Abbildung 93. Energieverbrauch für eine Inferenz über ein Netz aus 8 Convolutional-Layern mit vielen Filtern

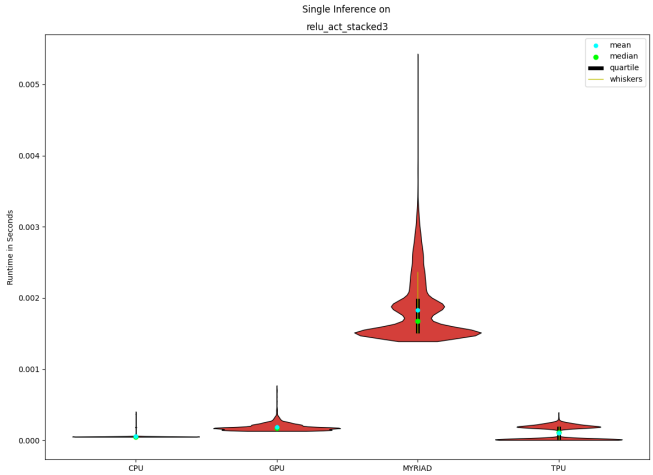

Abbildung 94. Laufzeit für eine Inferenz über ein Netz aus 3 ReLU-Layern

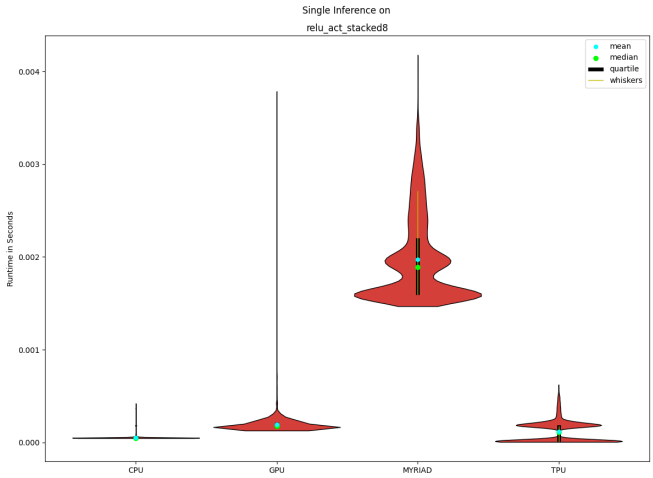

Abbildung 95. Laufzeit für eine Inferenz über ein Netz aus 8 ReLU-Layern

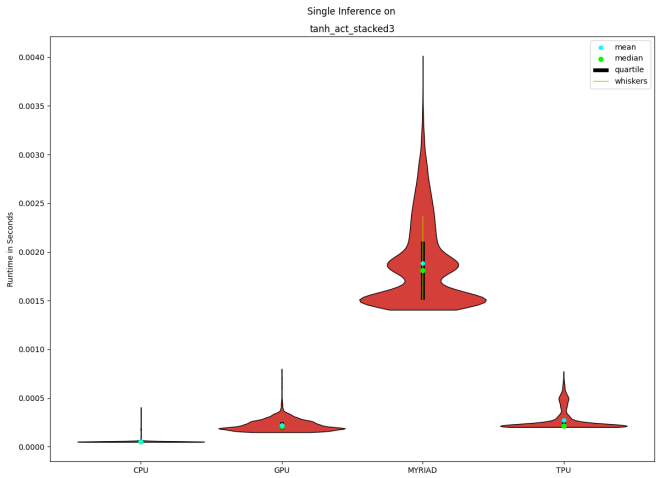

Abbildung 96. Laufzeit für eine Inferenz über ein Netz aus 3 tanh-Layern

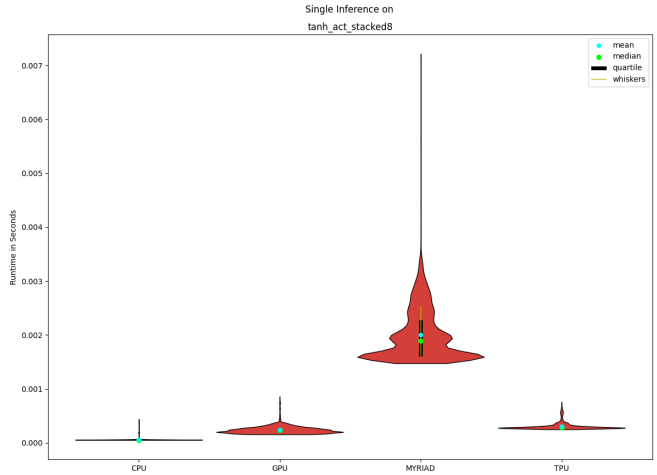

Abbildung 97. Laufzeit für eine Inferenz über ein Netz aus 8 tanh-Layern

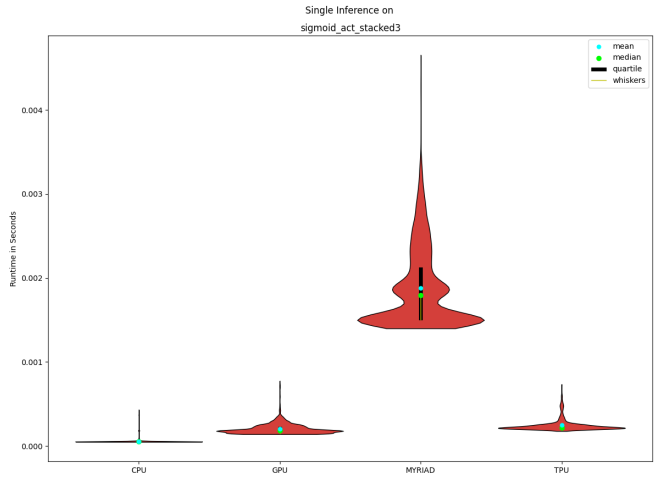

Abbildung 98. Laufzeit für eine Inferenz über ein Netz aus 3 sigmoid-Layern

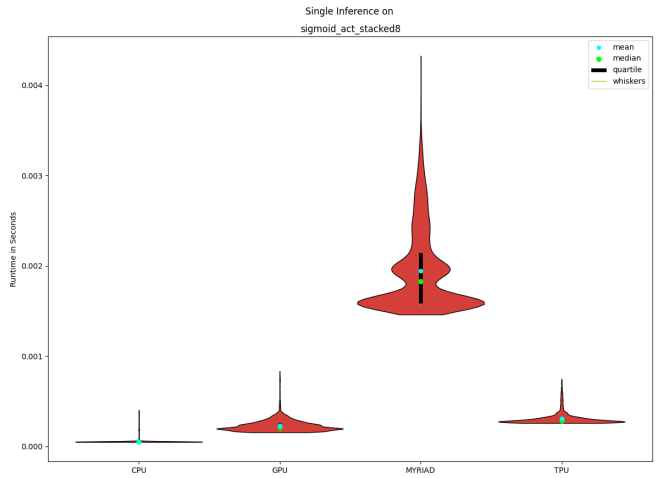

Abbildung 99. Laufzeit für eine Inferenz über ein Netz aus 8 sigmoid-Layern

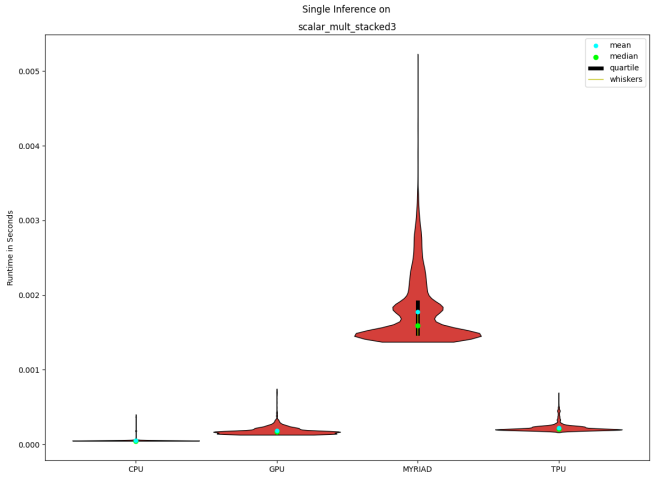

Abbildung 100. Laufzeit für eine Inferenz über ein Netz aus 3 Skalarprodukt-Layern

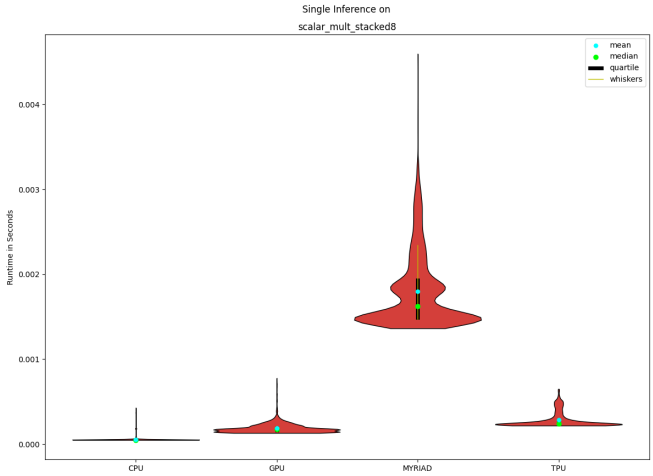

Abbildung 101. Laufzeit für eine Inferenz über ein Netz aus 8 Skalarprodukt-Layern

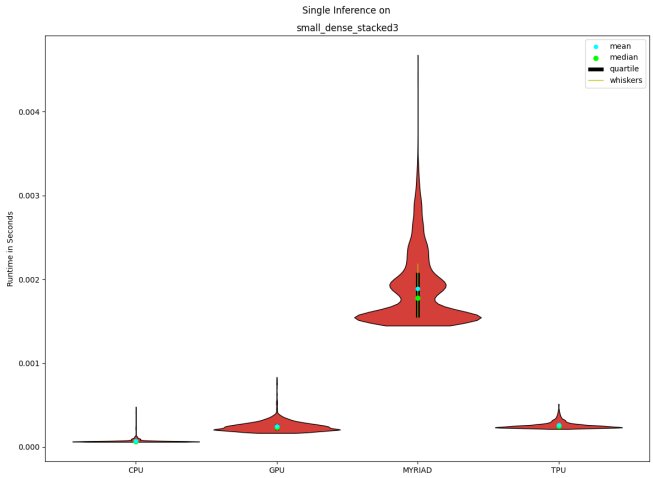

Abbildung 102. Laufzeit für eine Inferenz über ein Netz aus 3 kleinen Dense-Layern

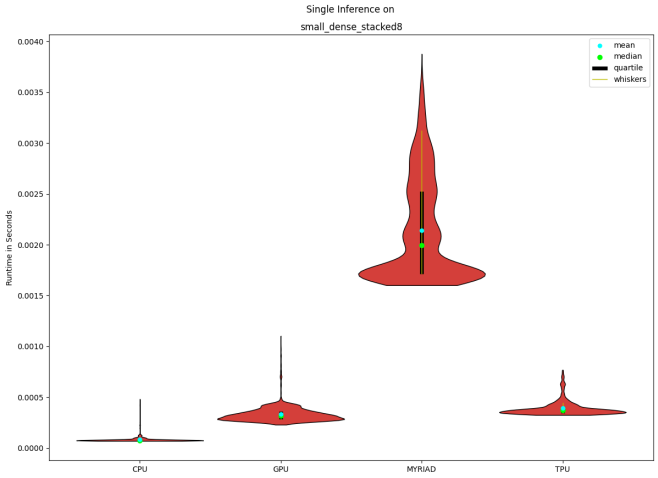

Abbildung 103. Laufzeit für eine Inferenz über ein Netz aus 8 kleinen Dense-Layern

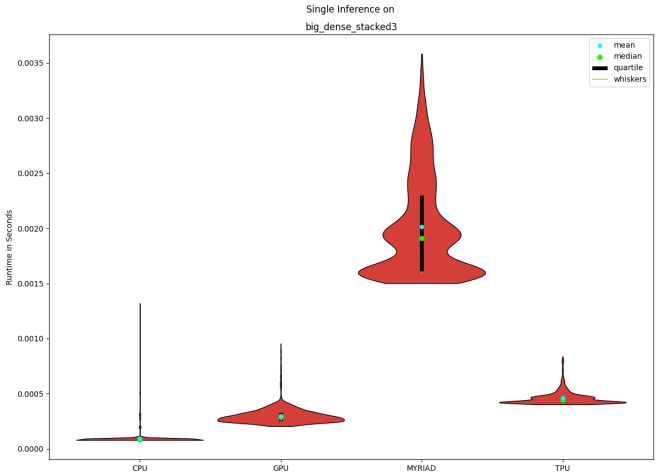

Abbildung 104. Laufzeit für eine Inferenz über ein Netz aus 3 großen Dense-Layern

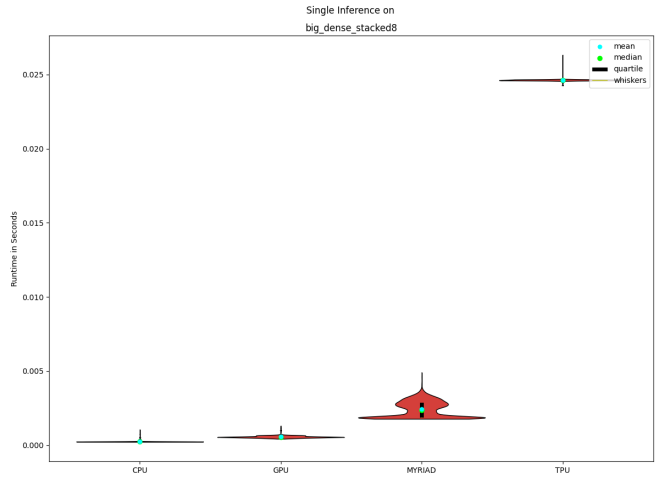

Abbildung 105. Laufzeit für eine Inferenz über ein Netz aus 8 großen Dense-Layern

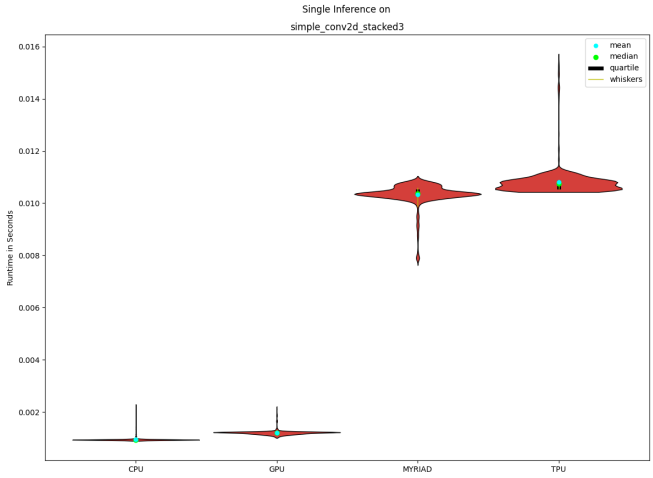

Abbildung 106. Laufzeit für eine Inferenz über ein Netz aus 3 einfachen Convolutional-Layern

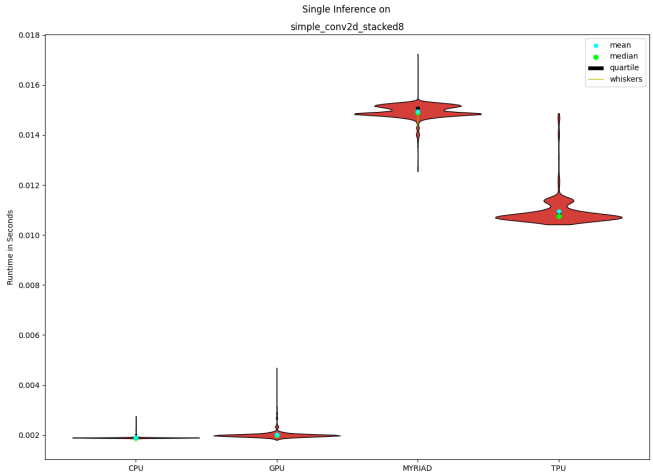

Abbildung 107. Laufzeit für eine Inferenz über ein Netz aus 8 einfachen Convolutional-Layern

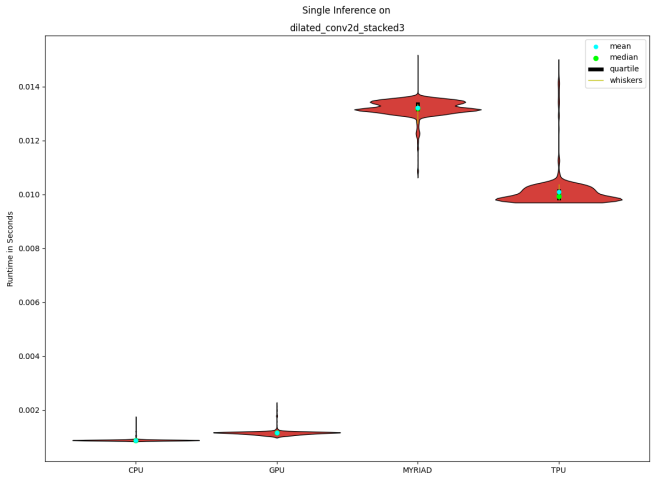

Abbildung 108. Laufzeit für eine Inferenz über ein Netz aus 3 dialeted Convolutional-Layern

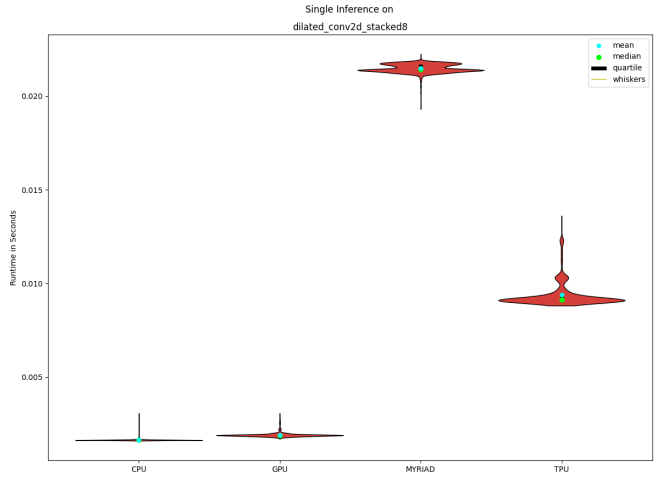

Abbildung 109. Laufzeit für eine Inferenz über ein Netz aus 8 dialeted Convolutional-Layern

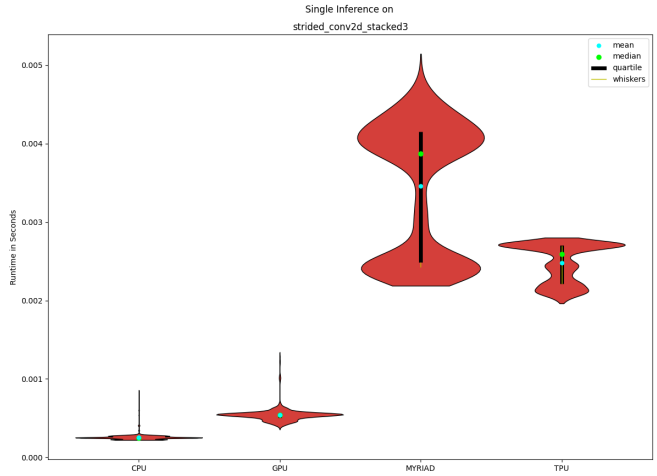

Abbildung 110. Laufzeit für eine Inferenz über ein Netz aus 3 strided Convolutional-Layern

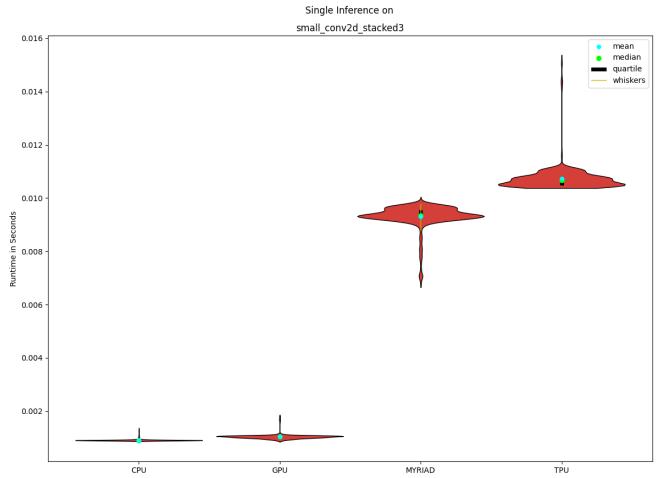

Abbildung 111. Laufzeit für eine Inferenz über ein Netz aus 3 kleinen Convolutional-Layern

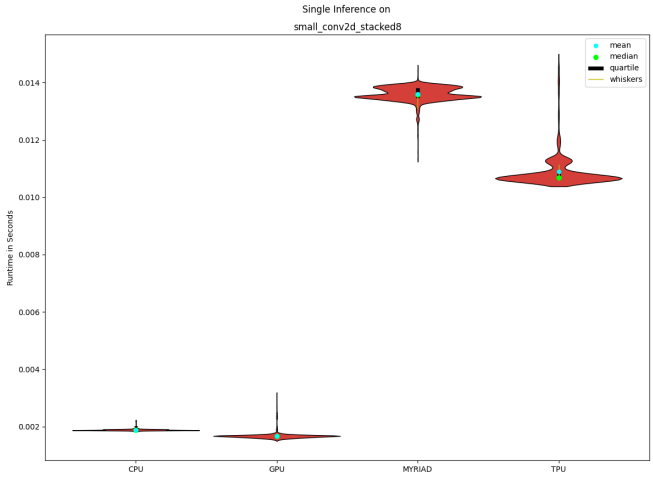

Abbildung 112. Laufzeit für eine Inferenz über ein Netz aus 8 kleinen Convolutional-Layern

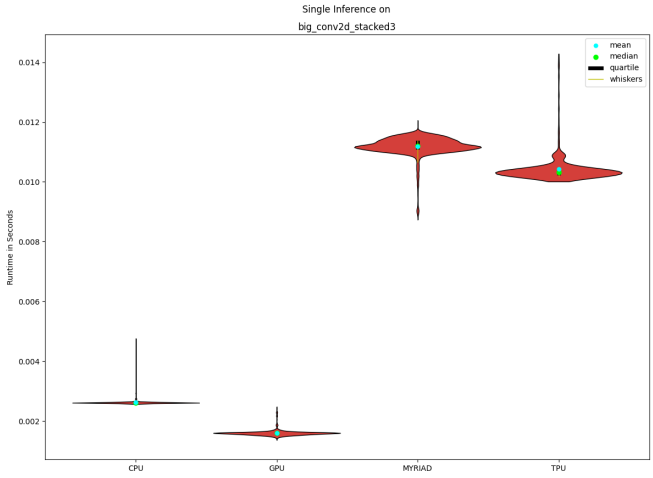

Abbildung 113. Laufzeit für eine Inferenz über ein Netz aus 3 großen Convolutional-Layern

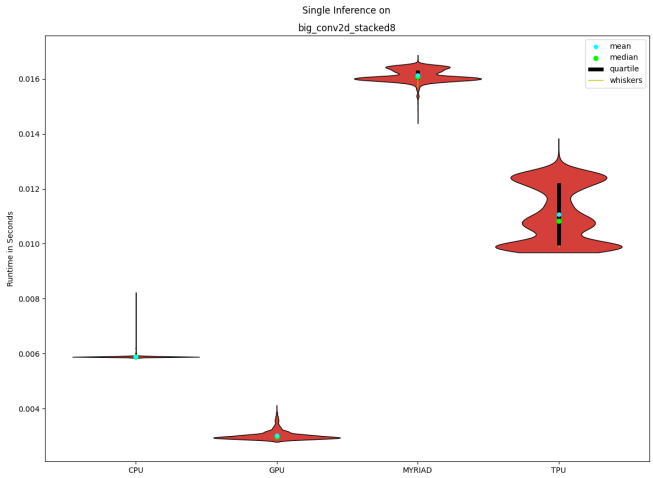

Abbildung 114. Laufzeit für eine Inferenz über ein Netz aus 8 großen Convolutional-Layern

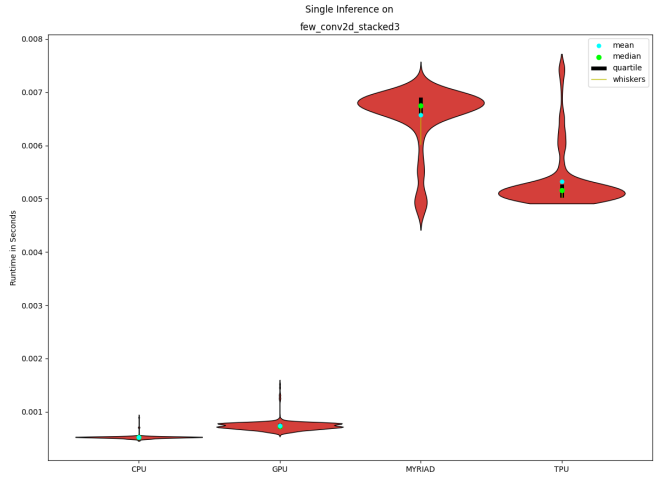

Abbildung 115. Laufzeit für eine Inferenz über ein Netz aus 3 Convolutional-Layern mit wenig Filtern

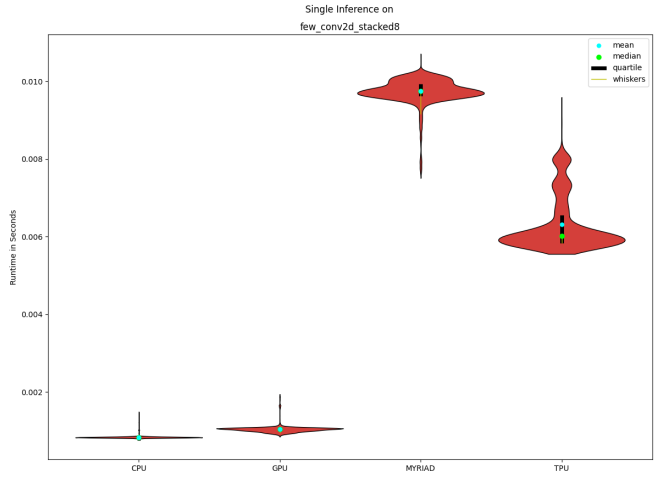

Abbildung 116. Laufzeit für eine Inferenz über ein Netz aus 8 Convolutional-Layern mit wenig Filtern

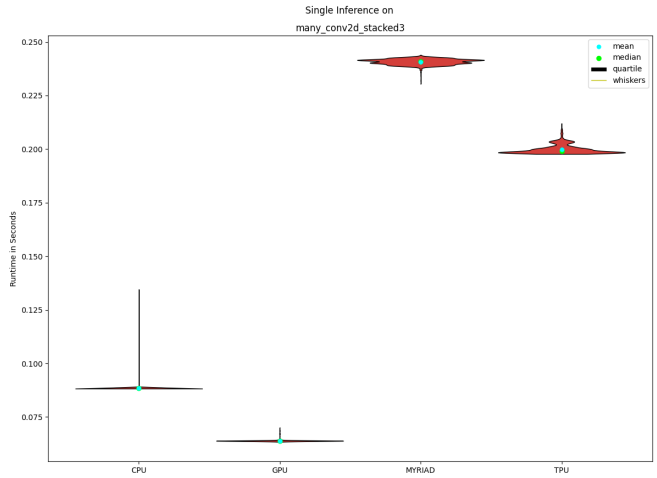

Abbildung 117. Laufzeit für eine Inferenz über ein Netz aus 3 Convolutional-Layern mit vielen Filtern

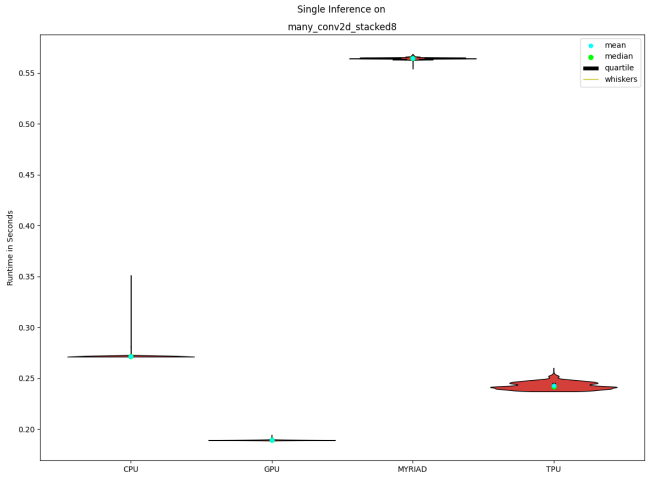

Abbildung 118. Laufzeit für eine Inferenz über ein Netz aus 8 Convolutional-Layern mit vielen Filtern

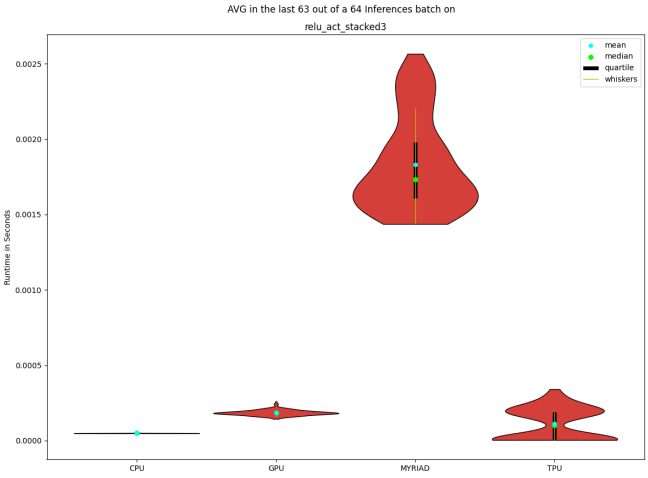

Abbildung 119. AVG Laufzeit für 63 Inferenzen über ein Netz aus 3 ReLU-Layern

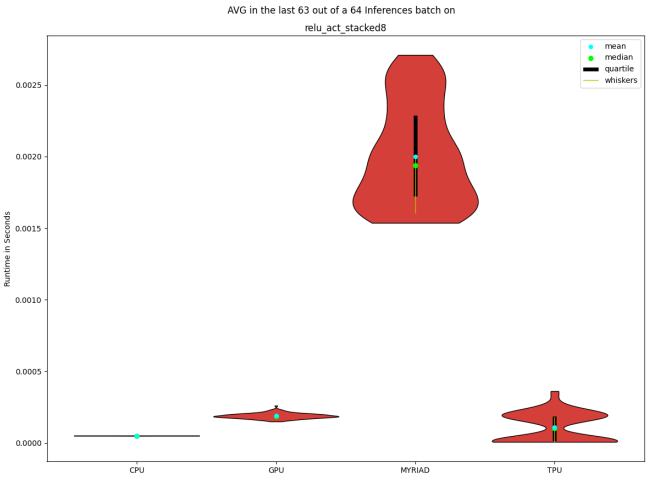

Abbildung 120. AVG Laufzeit für 63 Inferenzen über ein Netz aus 8 ReLU-Layern

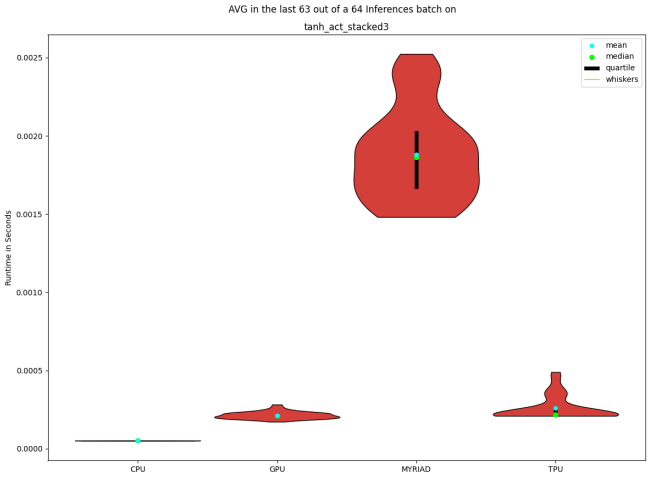

Abbildung 121. AVG Laufzeit für 63 Inferenzen über ein Netz aus 3 tanh-Layern

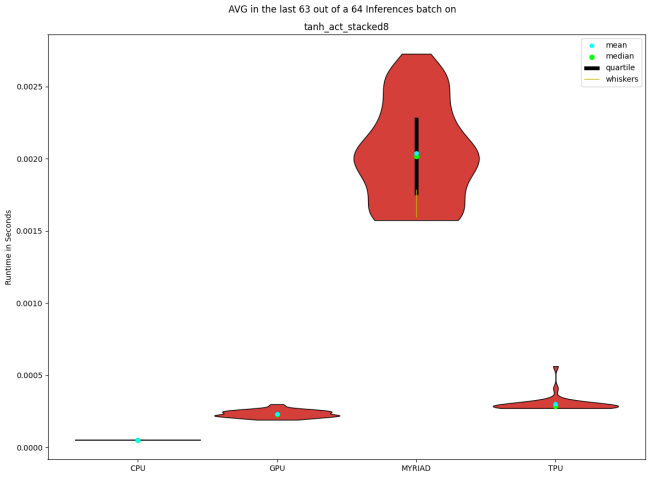

Abbildung 122. AVG Laufzeit für 63 Inferenzen über ein Netz aus 8 tanh-Layern

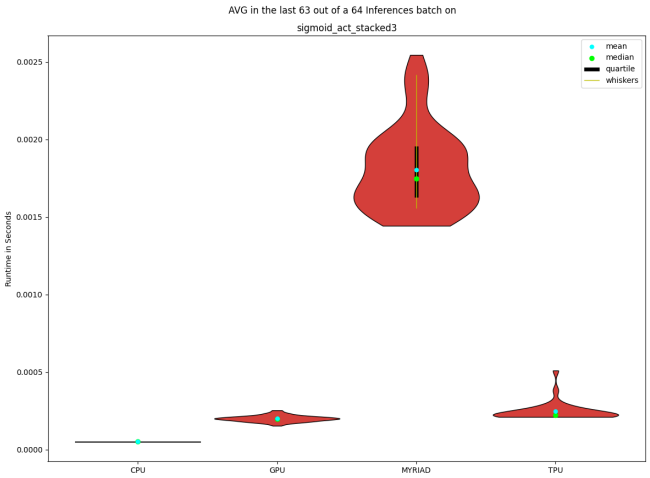

Abbildung 123. AVG Laufzeit für 63 Inferenzen über ein Netz aus 3 sigmoid-Layern

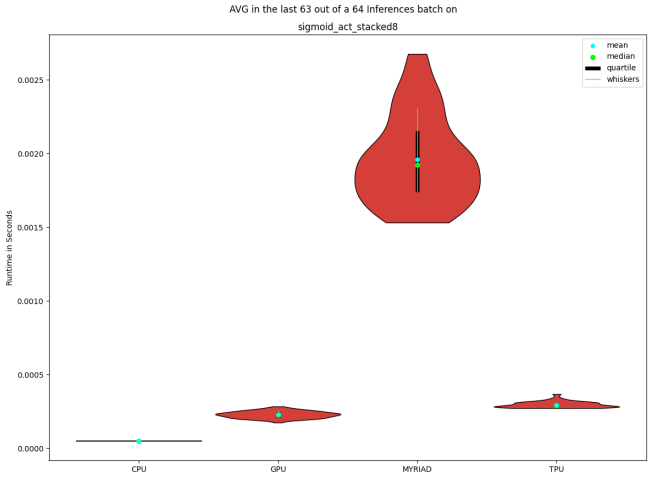

Abbildung 124. AVG Laufzeit für 63 Inferenzen über ein Netz aus 8 sigmoid-Layern

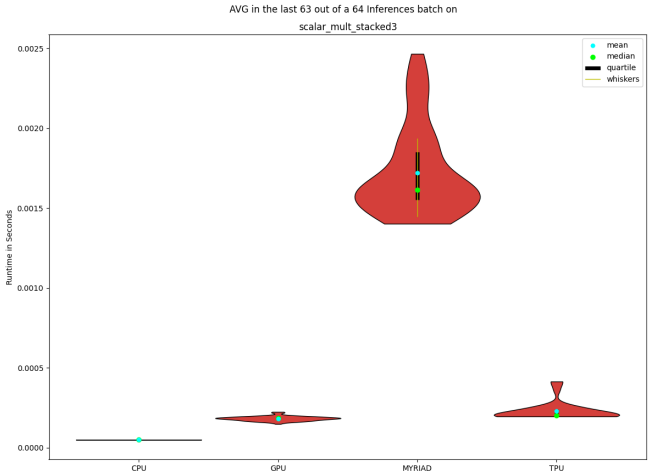

Abbildung 125. AVG Laufzeit für 63 Inferenzen über ein Netz aus 3 Skalarprodukt-Layern

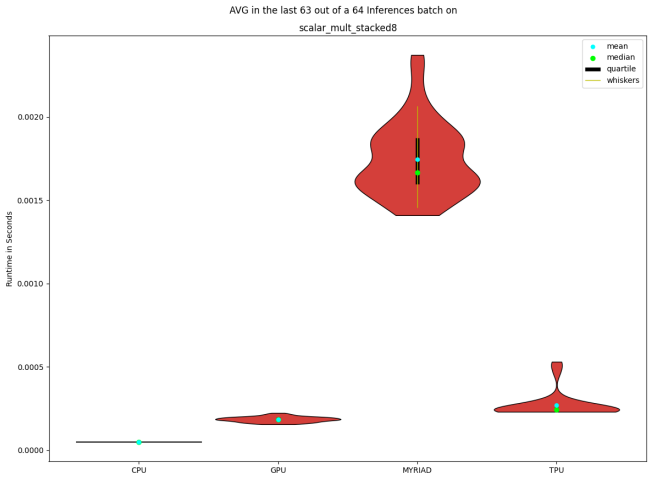

Abbildung 126. AVG Laufzeit für 63 Inferenzen über ein Netz aus 8 Skalarprodukt-Layern

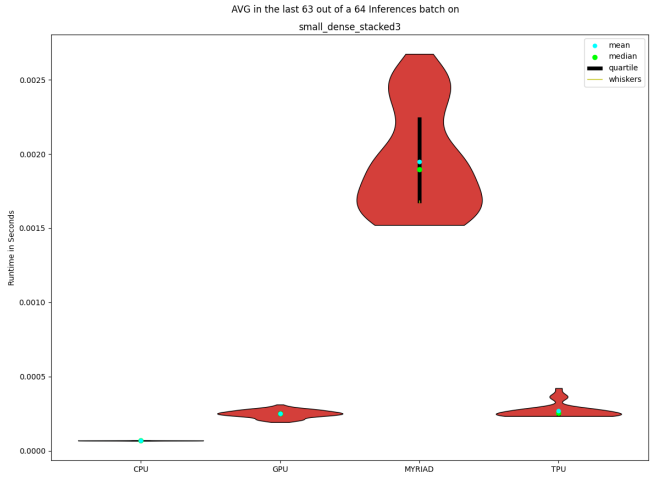

Abbildung 127. AVG Laufzeit für 63 Inferenzen über ein Netz aus 3 kleinen Dense-Layern

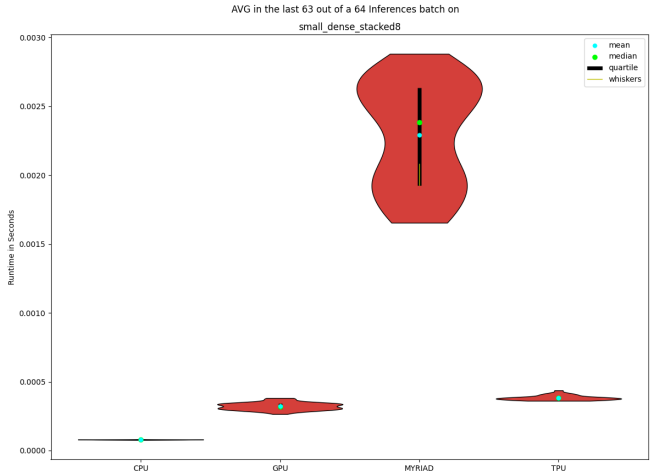

Abbildung 128. AVG Laufzeit für 63 Inferenzen über ein Netz aus 8 kleinen Dense-Layern

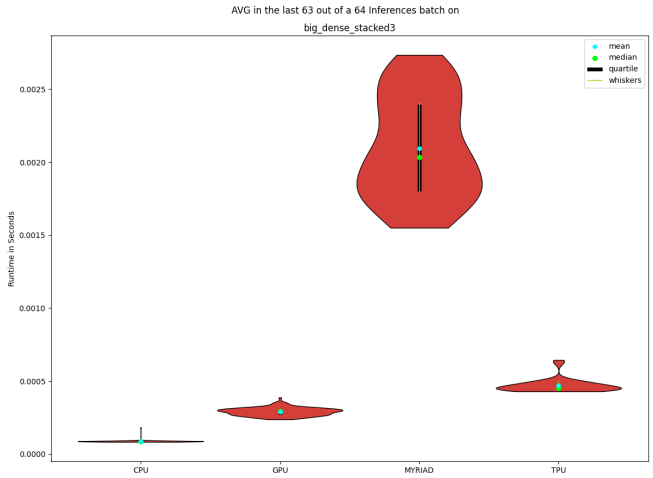

Abbildung 129. AVG Laufzeit für 63 Inferenzen über ein Netz aus 3 großen Dense-Layern

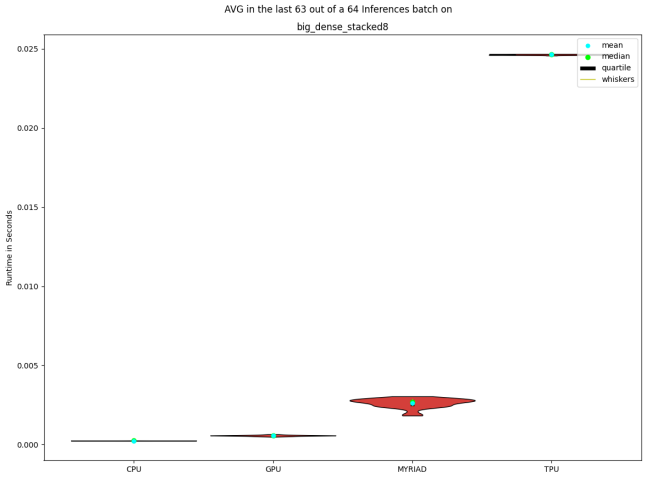

Abbildung 130. AVG Laufzeit für 63 Inferenzen über ein Netz aus 8 großen Dense-Layern

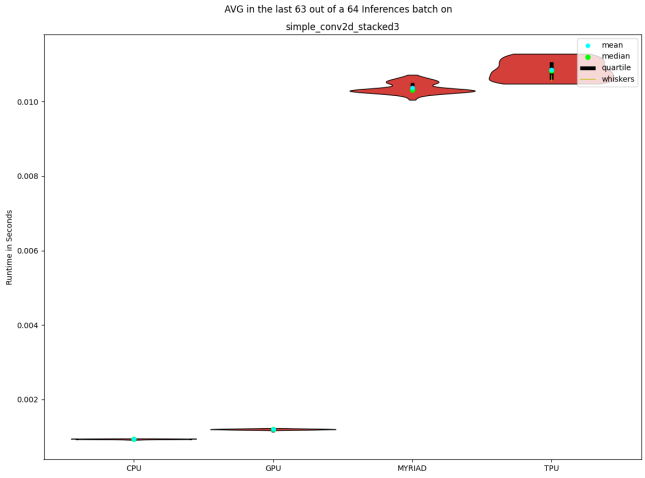

Abbildung 131. AVG Laufzeit für 63 Inferenzen über ein Netz aus 3 einfachen Convolutional-Layern

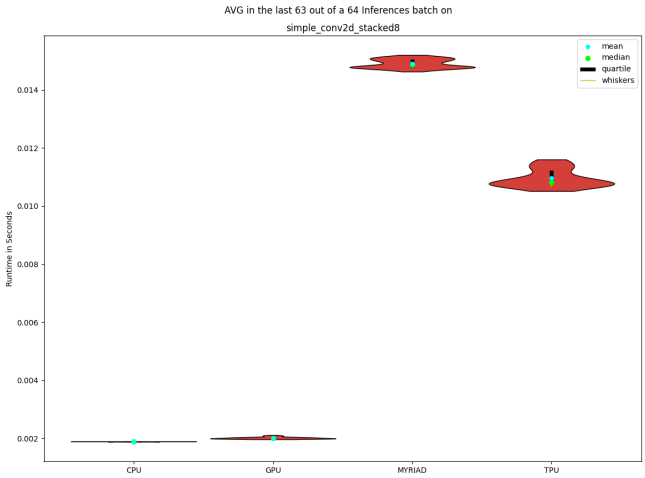

Abbildung 132. AVG Laufzeit für 63 Inferenzen über ein Netz aus 8 einfachen Convolutional-Layern

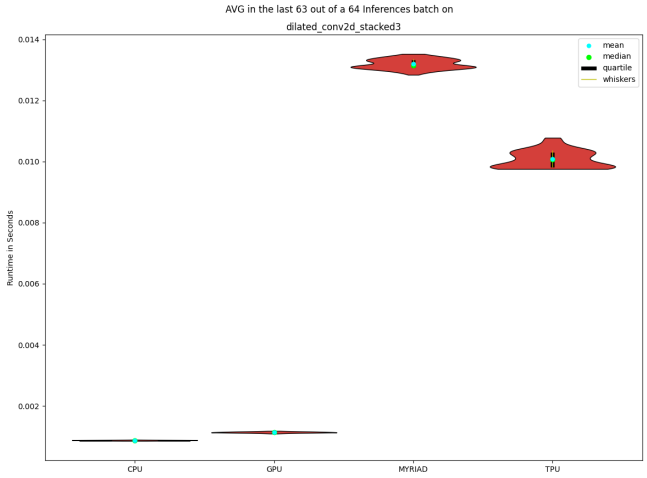

Abbildung 133. AVG Laufzeit für 63 Inferenzen über ein Netz aus 3 dialeted Convolutional-Layern

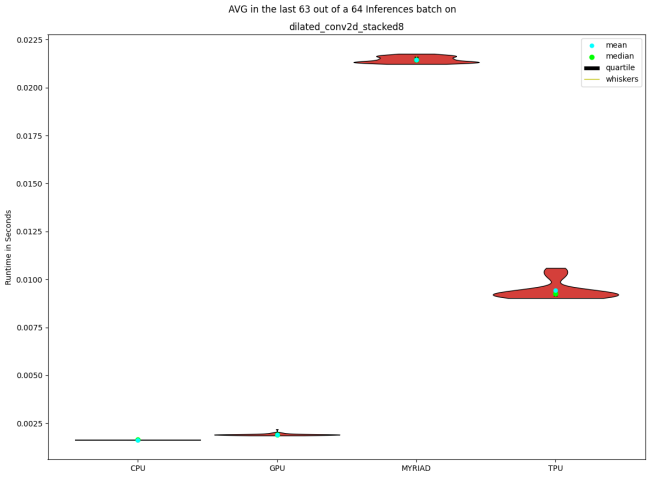

Abbildung 134. AVG Laufzeit für 63 Inferenzen über ein Netz aus 8 dialeted Convolutional-Layern

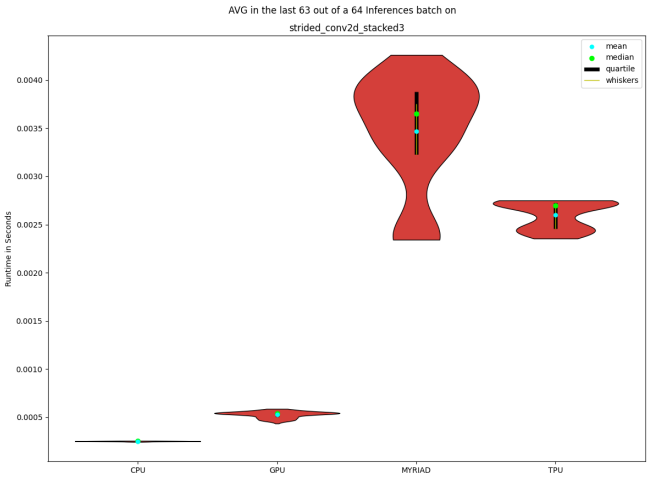

Abbildung 135. AVG Laufzeit für 63 Inferenzen über ein Netz aus 3 strided Convolutional-Layern

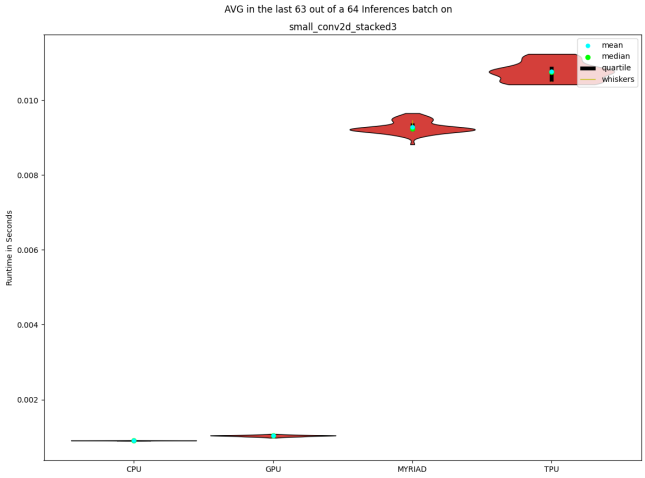

Abbildung 136. AVG Laufzeit für 63 Inferenzen über ein Netz aus 3 kleinen Convolutional-Layern

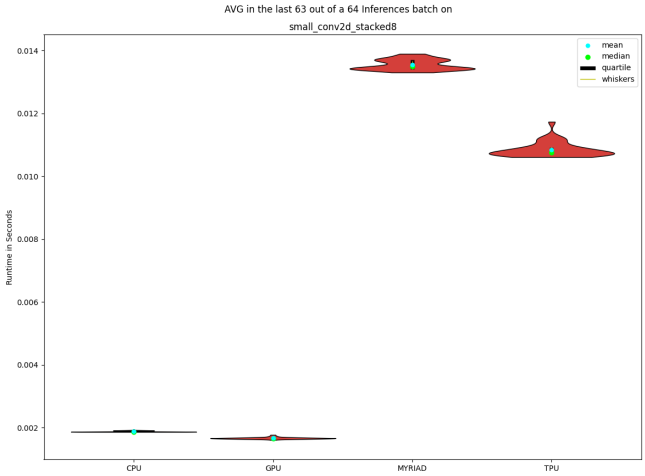

Abbildung 137. AVG Laufzeit für 63 Inferenzen über ein Netz aus 8 kleinen Convolutional-Layern

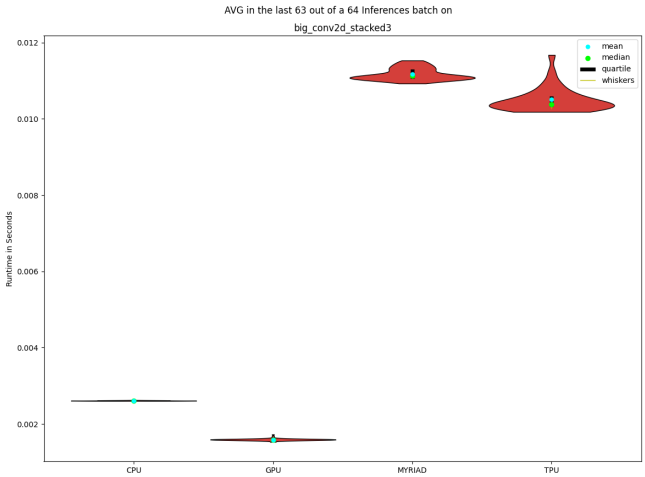

Abbildung 138. AVG Laufzeit für 63 Inferenzen über ein Netz aus 3 großen Convolutional-Layern

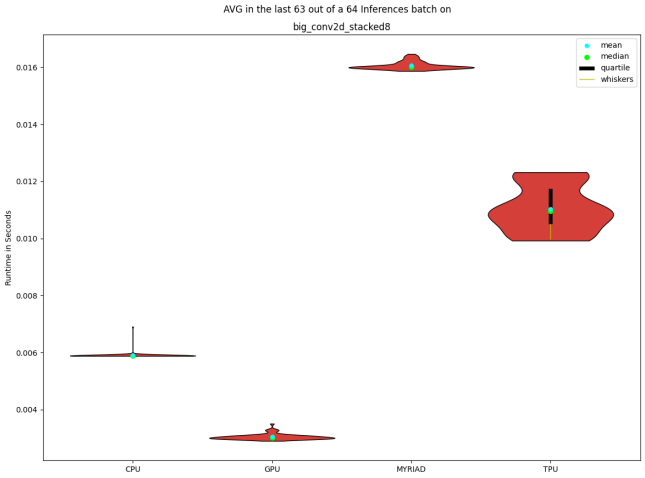

Abbildung 139. AVG Laufzeit für 63 Inferenzen über ein Netz aus 8 großen Convolutional-Layern

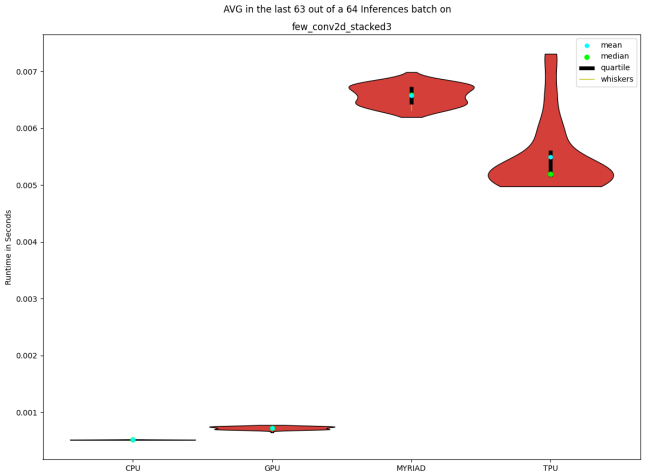

Abbildung 140. AVG Laufzeit für 63 Inferenzen über ein Netz aus 3 Convolutional-Layern mit wenig Filtern

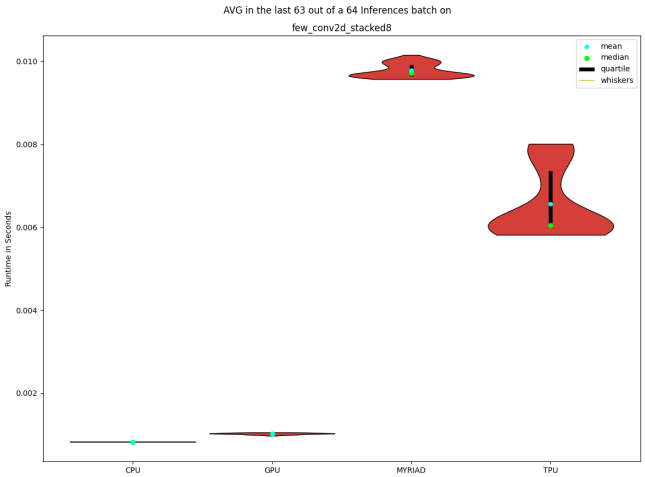

Abbildung 141. AVG Laufzeit für 63 Inferenzen über ein Netz aus 8 Convolutional-Layern mit wenig Filtern

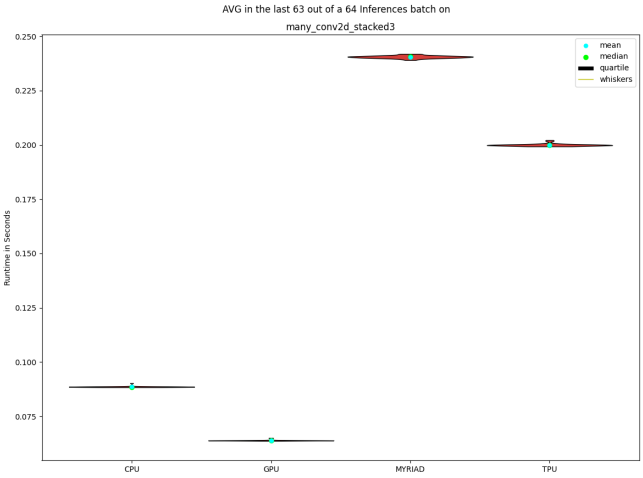

Abbildung 142. AVG Laufzeit für 63 Inferenzen über ein Netz aus 3 Convolutional-Layern mit vielen Filtern

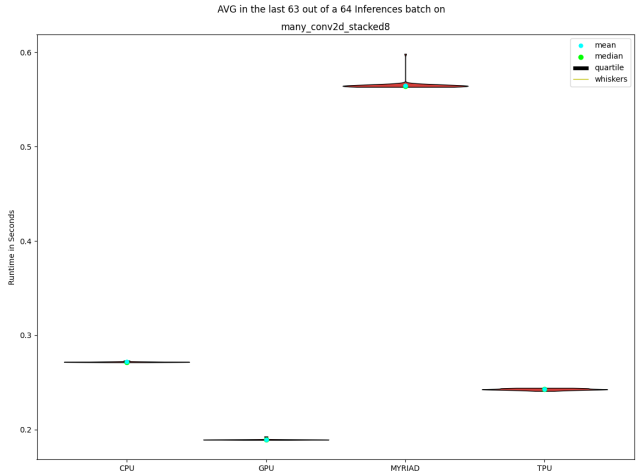

Abbildung 143. AVG Laufzeit für 63 Inferenzen über ein Netz aus 8 Convolutional-Layern mit vielen Filtern

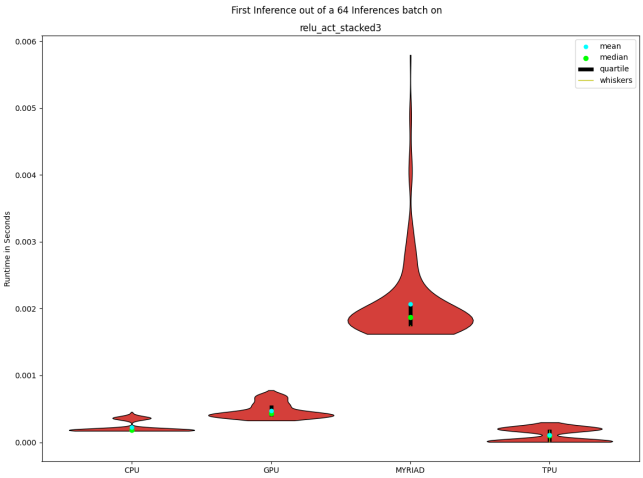

Abbildung 144. Laufzeit der ersten Inferenz über ein Netz aus 3 ReLU-Layern in einem Batchdurchlauf

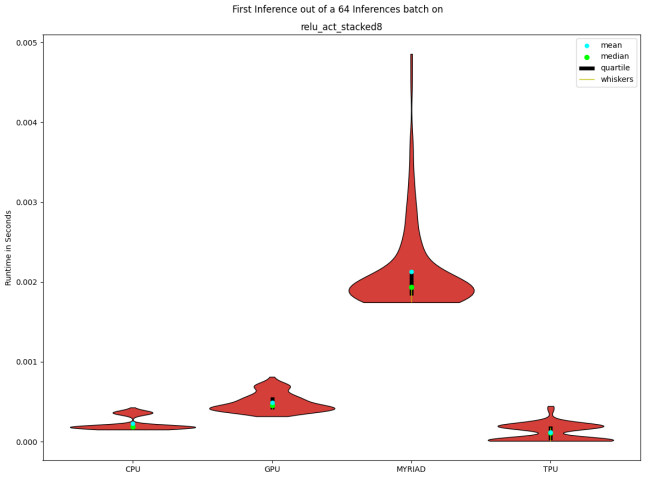

Abbildung 145. Laufzeit der ersten Inferenz über ein Netz aus 8 ReLU-Layern in einem Batchdurchlauf

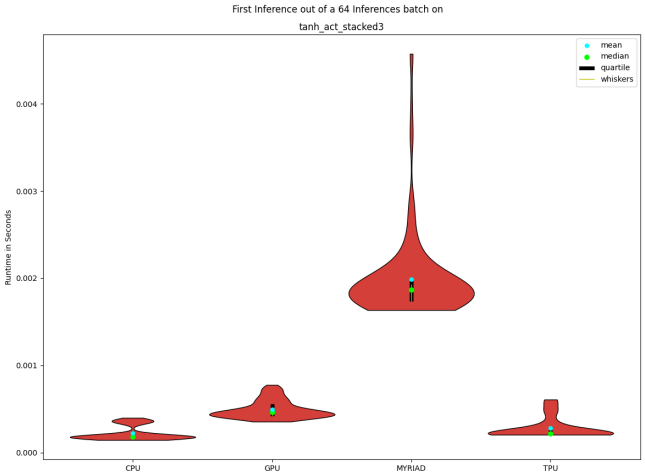

Abbildung 146. Laufzeit der ersten Inferenz über ein Netz aus 3 tanh-Layern in einem Batchdurchlauf

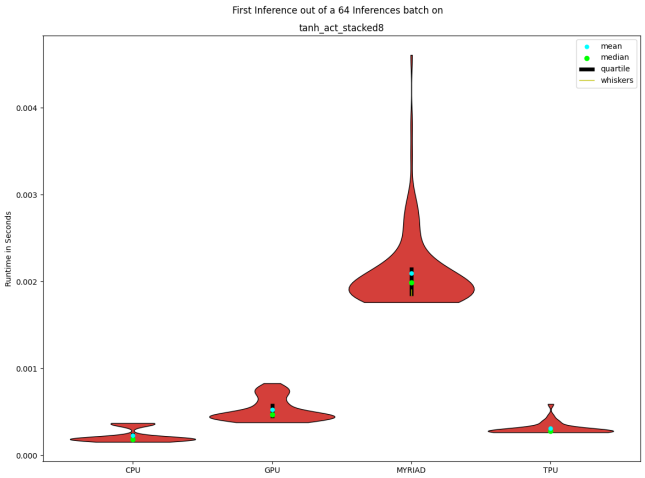

Abbildung 147. Laufzeit der ersten Inferenz über ein Netz aus 8 tanh-Layern in einem Batchdurchlauf

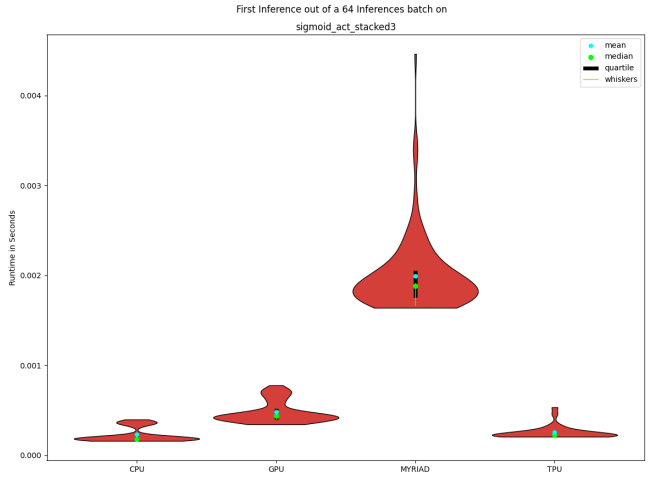

Abbildung 148. Laufzeit der ersten Inferenz über ein Netz aus 3 sigmoid-Layern in einem Batchdurchlauf

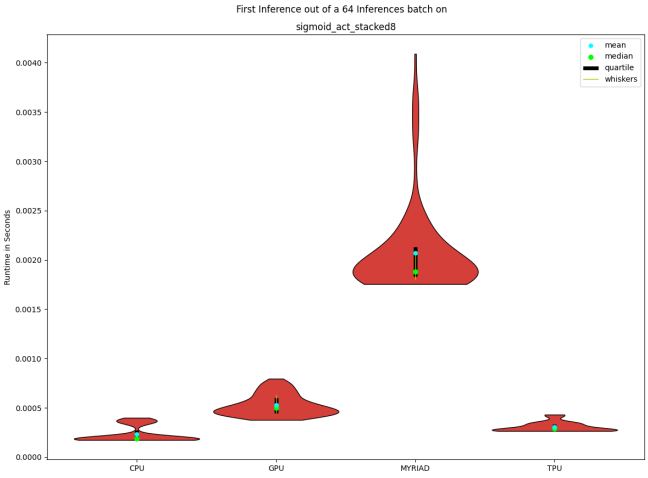

Abbildung 149. Laufzeit der ersten Inferenz über ein Netz aus 8 sigmoid-Layern in einem Batchdurchlauf

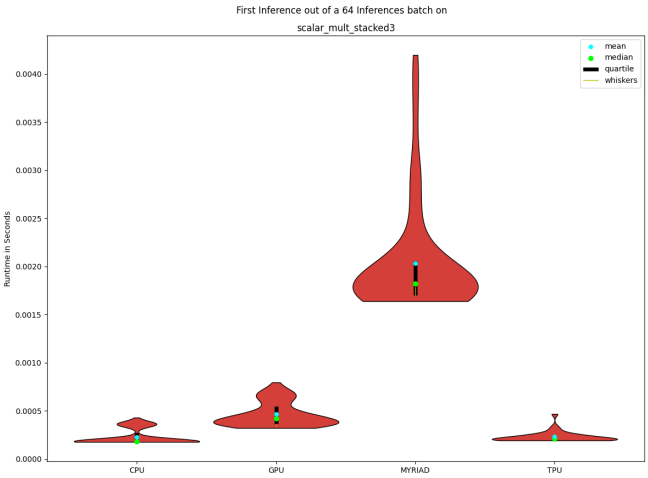

Abbildung 150. Laufzeit der ersten Inferenz über ein Netz aus 3 Skalarprodukt-Layern in einem Batchdurchlauf

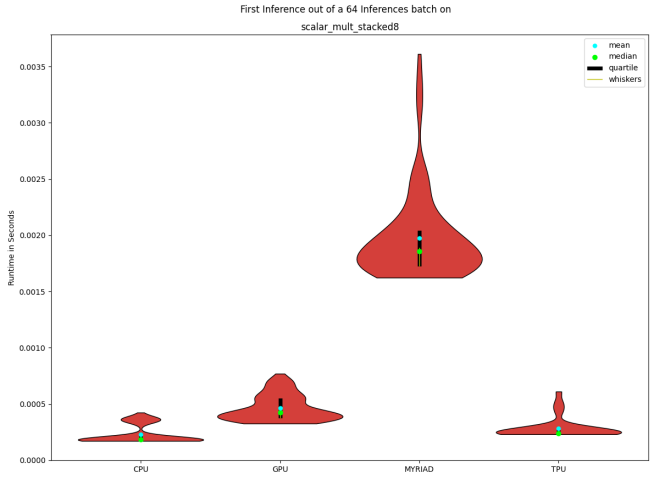

Abbildung 151. Laufzeit der ersten Inferenz über ein Netz aus 8 Skalarprodukt-Layern in einem Batchdurchlauf

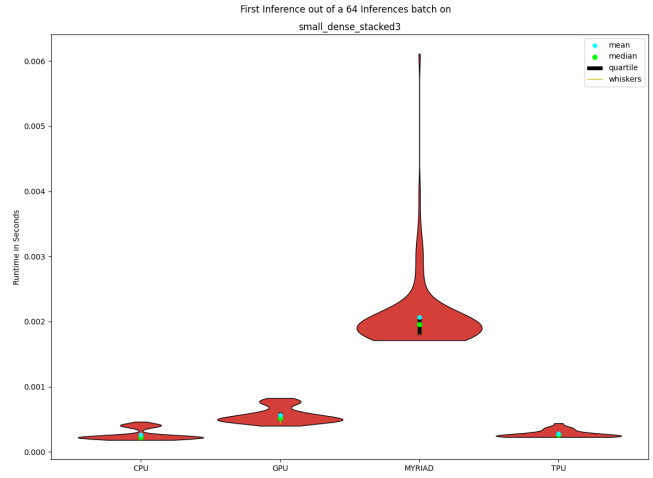

Abbildung 152. Laufzeit der ersten Inferenz über ein Netz aus 3 kleinen Dense-Layern in einem Batchdurchlauf

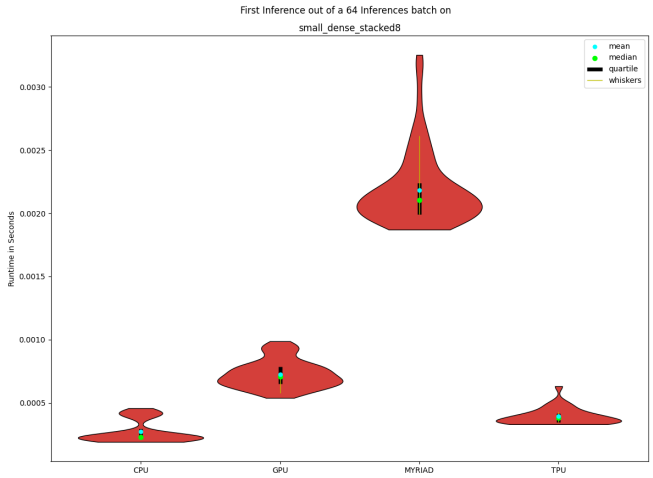

Abbildung 153. Laufzeit der ersten Inferenz über ein Netz aus 8 kleinen Dense-Layern in einem Batchdurchlauf

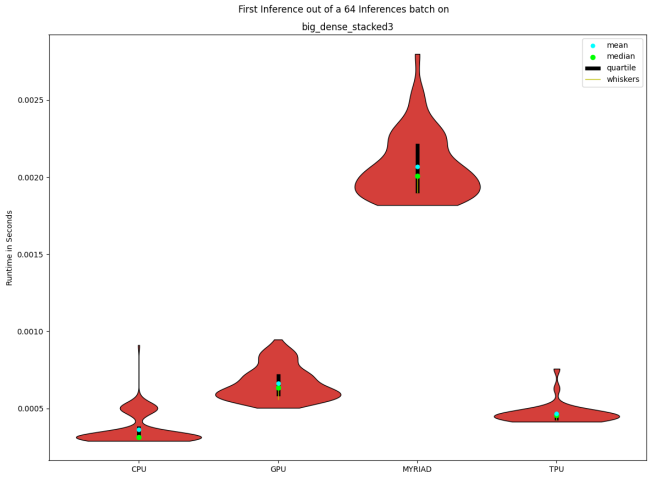

Abbildung 154. Laufzeit der ersten Inferenz über ein Netz aus 3 großen Dense-Layern in einem Batchdurchlauf

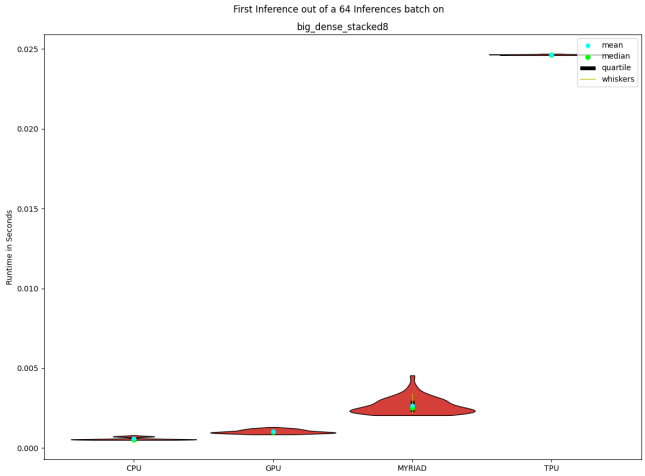

Abbildung 155. Laufzeit der ersten Inferenz über ein Netz aus 8 großen Dense-Layern in einem Batchdurchlauf

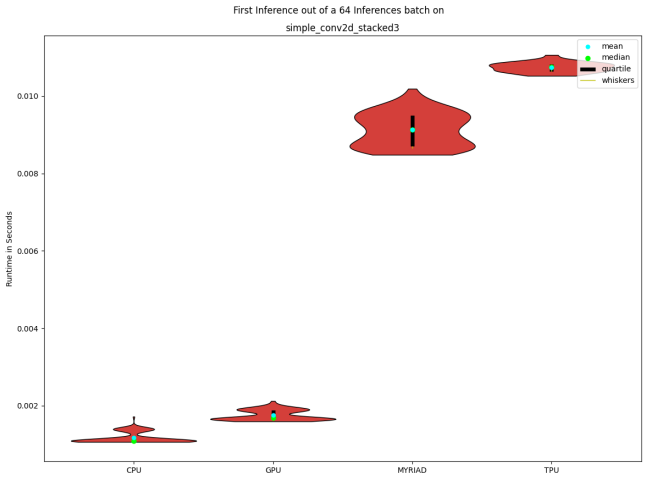

Abbildung 156. Laufzeit der ersten Inferenz über ein Netz aus 3 einfachen Convolutional-Layern in einem Batchdurchlauf

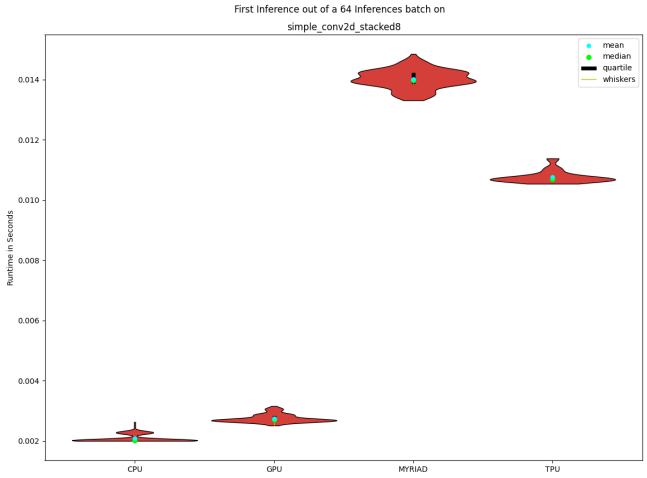

Abbildung 157. Laufzeit der ersten Inferenz über ein Netz aus 8 einfachen Convolutional-Layern in einem Batchdurchlauf

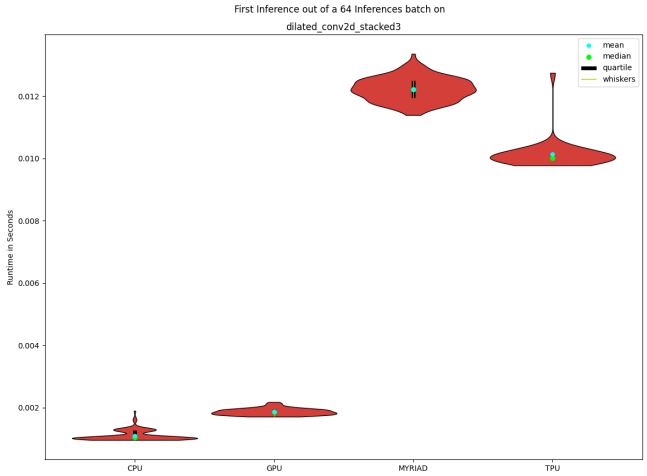

Abbildung 158. Laufzeit der ersten Inferenz über ein Netz aus 3 dialeted Convolutional-Layern in einem Batchdurchlauf

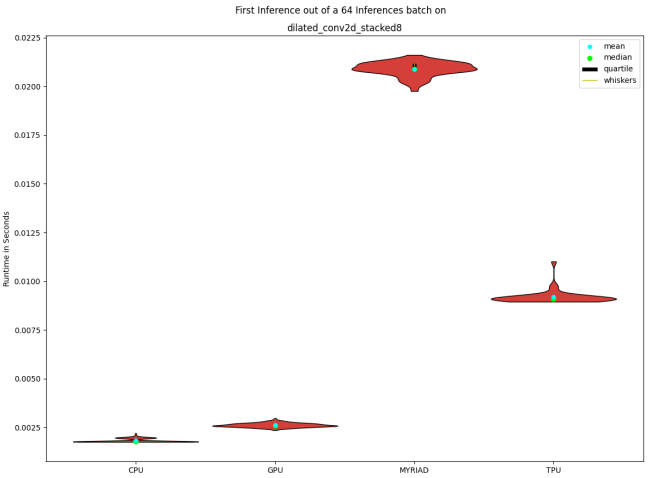

Abbildung 159. Laufzeit der ersten Inferenz über ein Netz aus 8 dialeted Convolutional-Layern in einem Batchdurchlauf

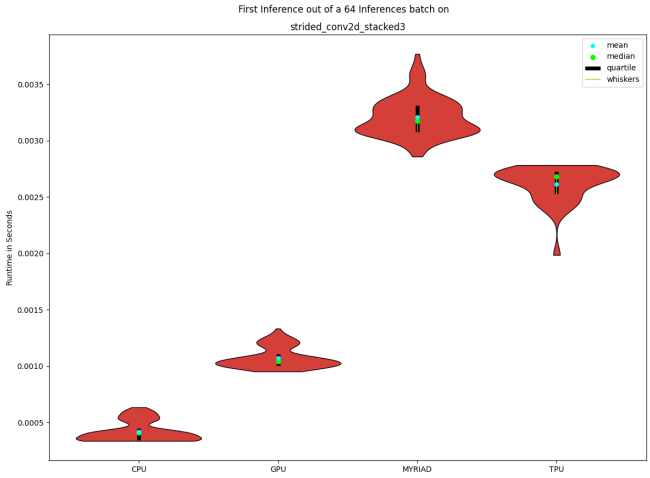

Abbildung 160. Laufzeit der ersten Inferenz über ein Netz aus 3 strided Convolutional-Layern in einem Batchdurchlauf

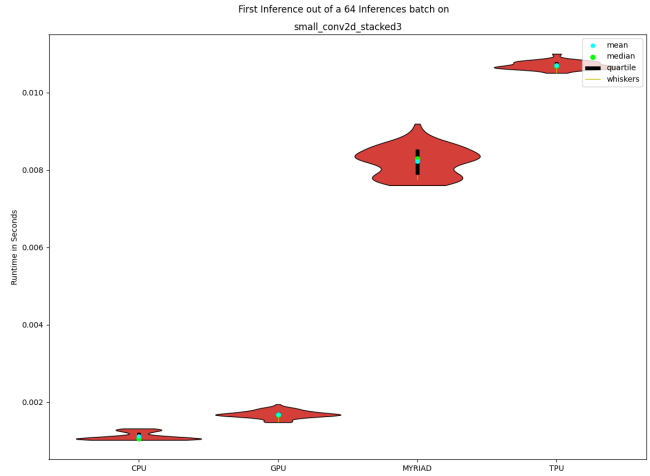

Abbildung 161. Laufzeit der ersten Inferenz über ein Netz aus 3 kleinen Convolutional-Layern in einem Batchdurchlauf

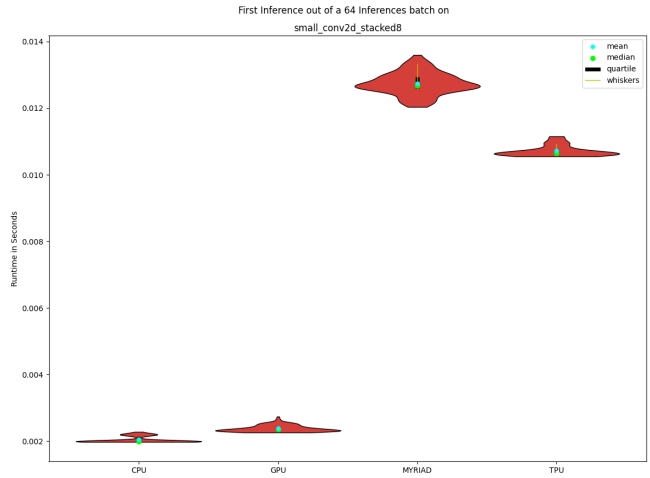

Abbildung 162. Laufzeit der ersten Inferenz über ein Netz aus 8 kleinen Convolutional-Layern in einem Batchdurchlauf

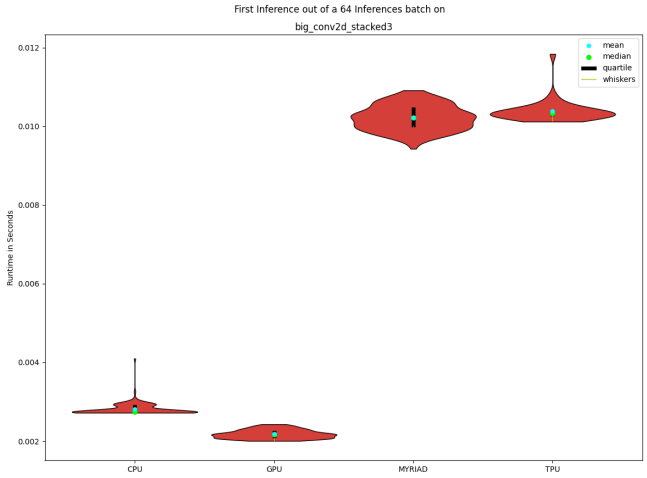

Abbildung 163. Laufzeit der ersten Inferenz über ein Netz aus 3 großen Convolutional-Layern in einem Batchdurchlauf

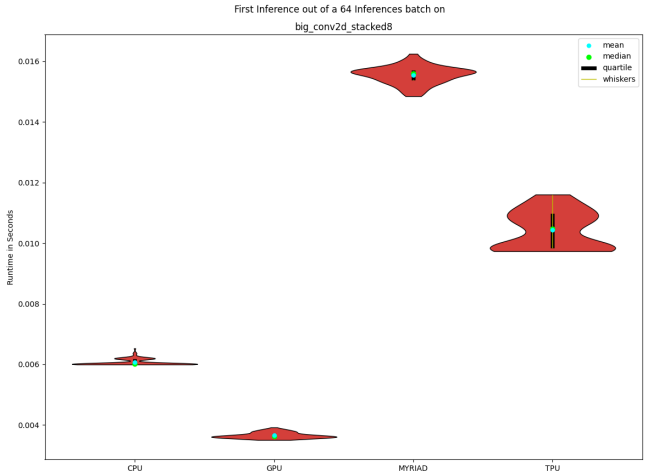

Abbildung 164. Laufzeit der ersten Inferenz über ein Netz aus 8 großen Convolutional-Layern in einem Batchdurchlauf

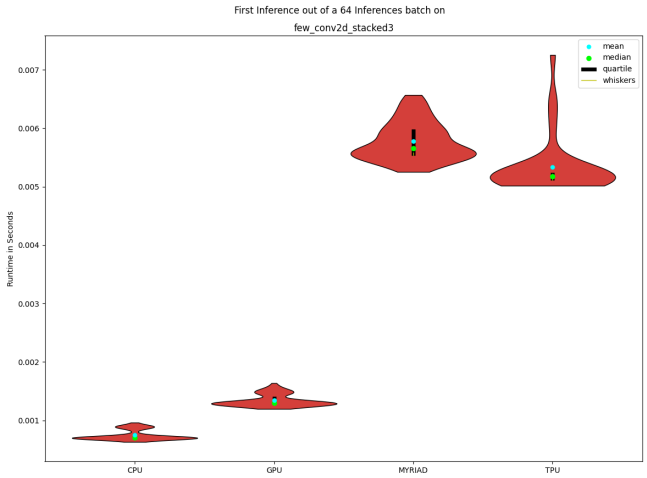

Abbildung 165. Laufzeit der ersten Inferenz über ein Netz aus 3 Convolutional-Layern mit wenig Filtern in einem Batchdurchlauf

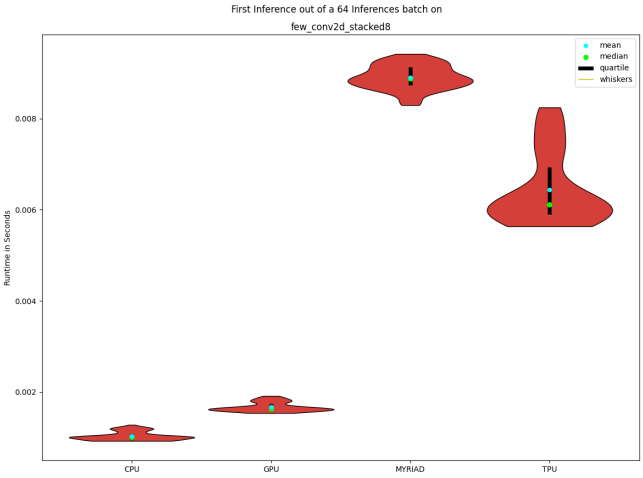

Abbildung 166. Laufzeit der ersten Inferenz über ein Netz aus 8 Convolutional-Layern mit wenig Filtern in einem Batchdurchlauf

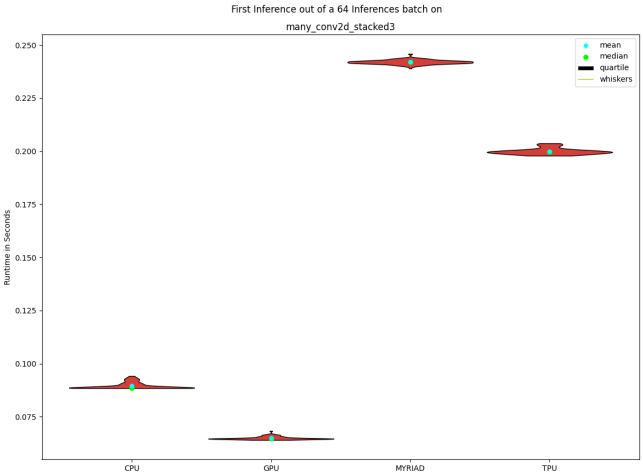

Abbildung 167. Laufzeit der ersten Inferenz über ein Netz aus 3 Convolutional-Layern mit vielen Filtern in einem Batchdurchlauf

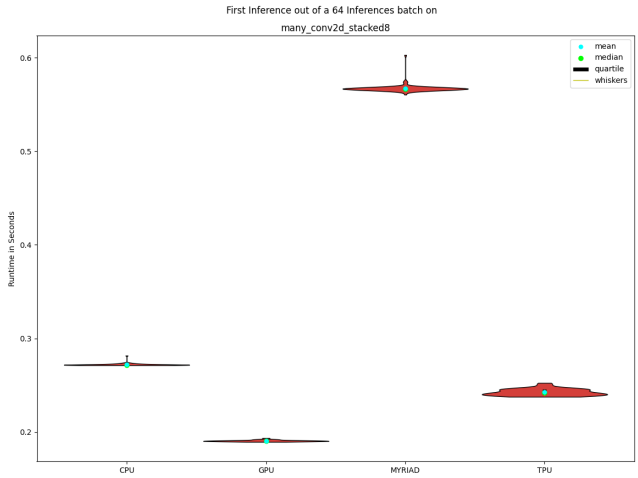

Abbildung 168. Laufzeit der ersten Inferenz über ein Netz aus 8 Convolutional-Layern mit vielen Filtern in einem Batchdurchlauf

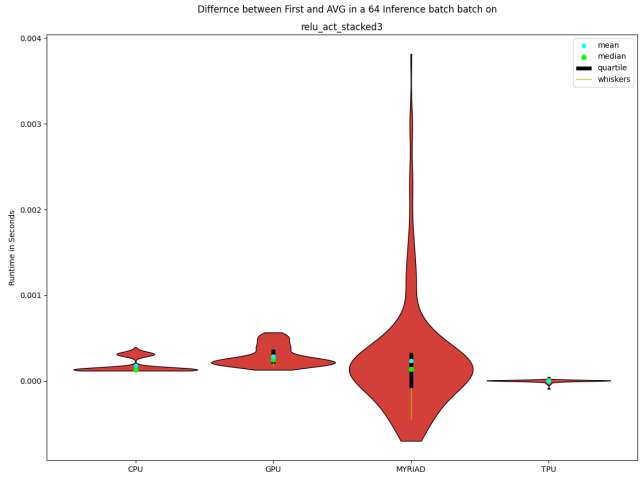

Abbildung 169. Differenz Laufzeit der ersten Inferenz und des AVG über ein Netz aus 3 ReLU-Layern in einem Batchdurchlauf

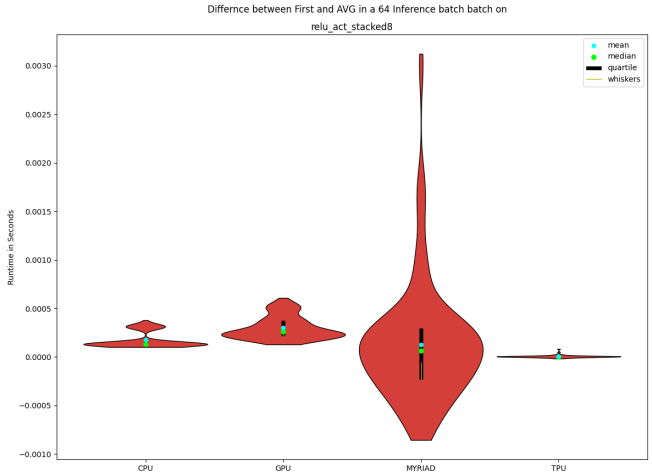

Abbildung 170. Differenz Laufzeit der ersten Inferenz und des AVG über ein Netz aus 8 ReLU-Layern in einem Batchdurchlauf

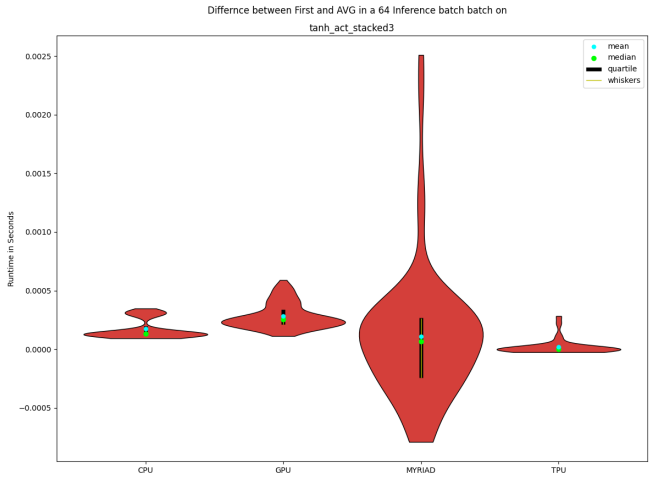

Abbildung 171. Differenz Laufzeit der ersten Inferenz und des AVG über ein Netz aus 3 tanh-Layern in einem Batchdurchlauf

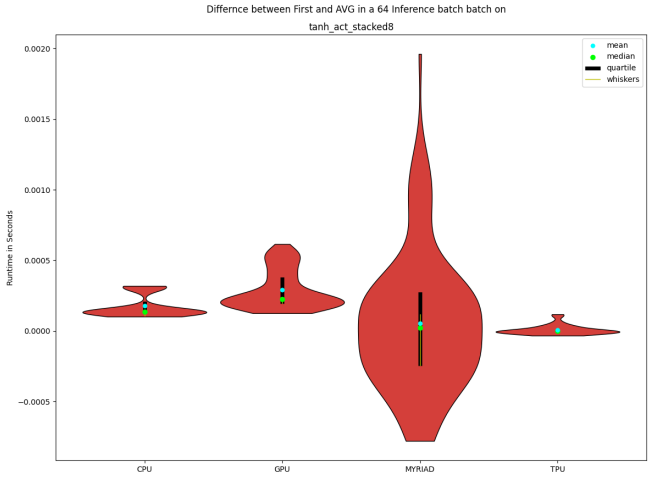

Abbildung 172. Differenz Laufzeit der ersten Inferenz und des AVG über ein Netz aus 8 tanh-Layern in einem Batchdurchlauf

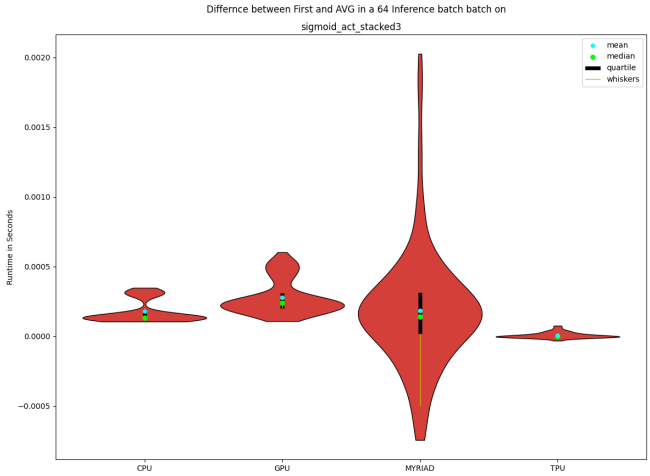

Abbildung 173. Differenz Laufzeit der ersten Inferenz und des AVG über ein Netz aus 3 sigmoid-Layern in einem Batchdurchlauf

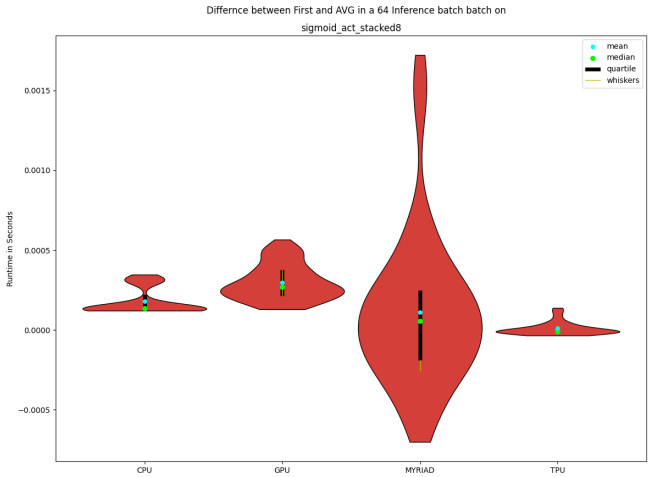

Abbildung 174. Differenz Laufzeit der ersten Inferenz und des AVG über ein Netz aus 8 sigmoid-Layern in einem Batchdurchlauf

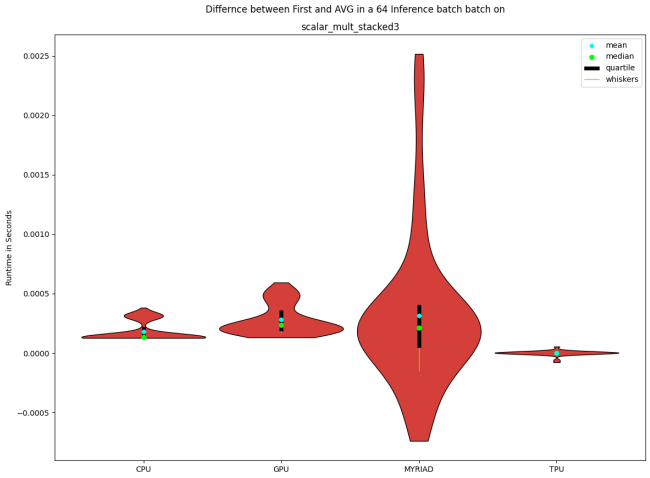

Abbildung 175. Differenz Laufzeit der ersten Inferenz und des AVG über ein Netz aus 3 Skalarprodukt-Layern in einem Batchdurchlauf

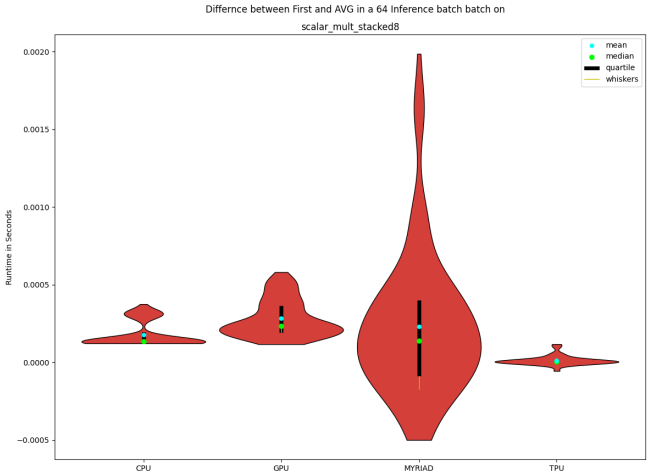

Abbildung 176. Differenz Laufzeit der ersten Inferenz und des AVG über ein Netz aus 8 Skalarprodukt-Layern in einem Batchdurchlauf

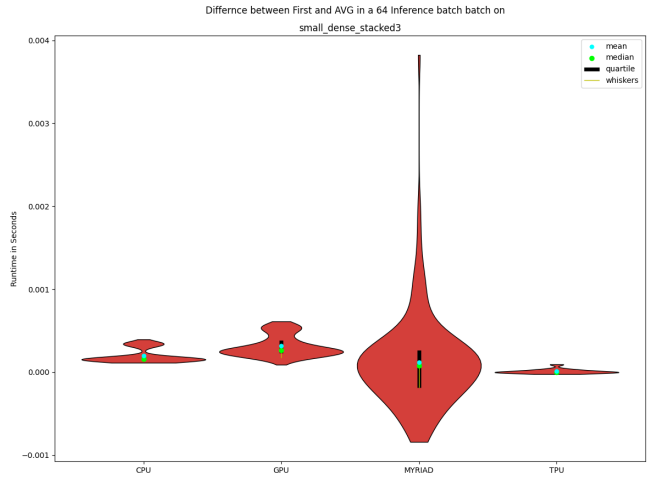

Abbildung 177. Differenz Laufzeit der ersten Inferenz und des AVG über ein Netz aus 3 kleinen Dense-Layern in einem Batchdurchlauf

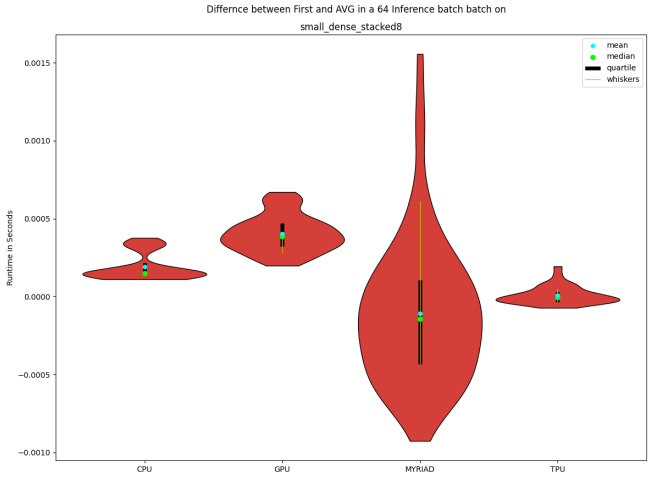

Abbildung 178. Differenz Laufzeit der ersten Inferenz und des AVG über ein Netz aus 8 kleinen Dense-Layern in einem Batchdurchlauf

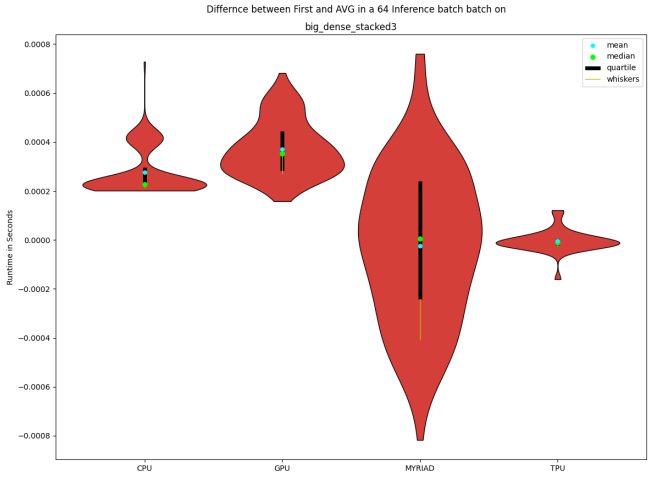

Abbildung 179. Differenz Laufzeit der ersten Inferenz und des AVG über ein Netz aus 3 großen Dense-Layern in einem Batchdurchlauf

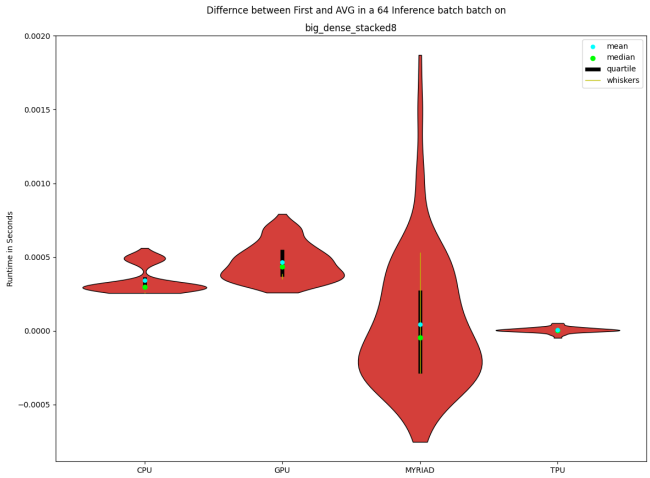

Abbildung 180. Differenz Laufzeit der ersten Inferenz und des AVG über ein Netz aus 8 großen Dense-Layern in einem Batchdurchlauf

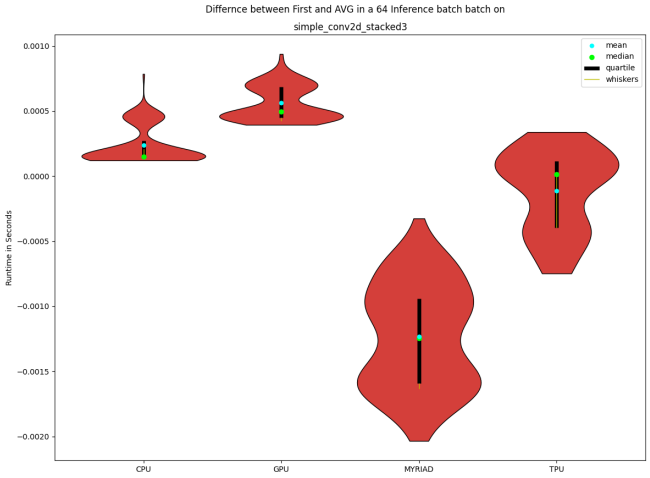

Abbildung 181. Differenz Laufzeit der ersten Inferenz und des AVG über ein Netz aus 3 einfachen Convolutional-Layern in einem Batchdurchlauf

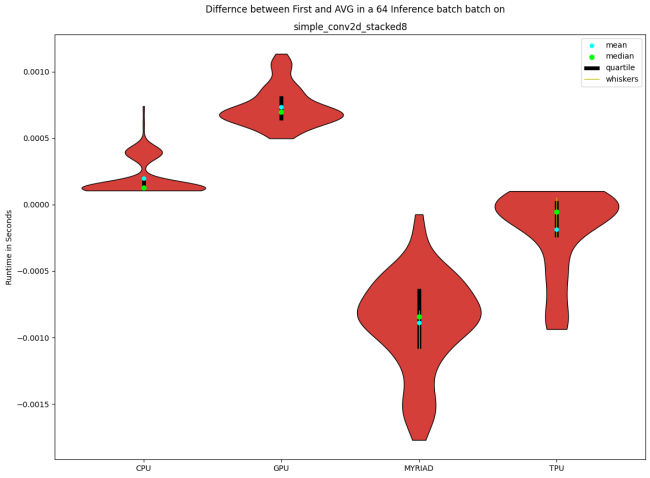

Abbildung 182. Differenz Laufzeit der ersten Inferenz und des AVG über ein Netz aus 8 einfachen Convolutional-Layern in einem Batchdurchlauf

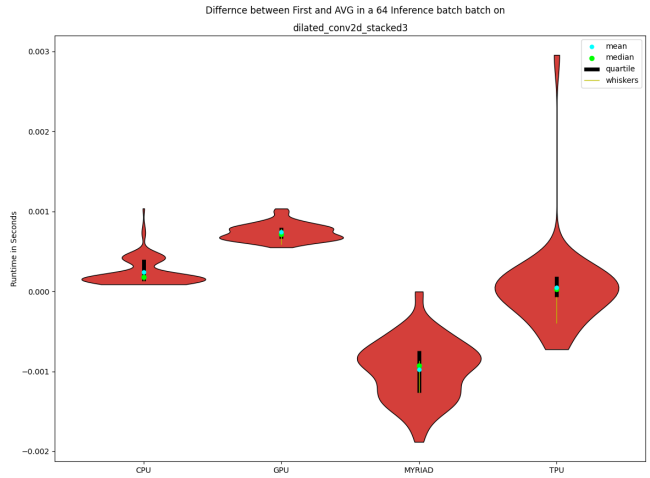

Abbildung 183. Differenz Laufzeit der ersten Inferenz und des AVG über ein Netz aus 3 dialeted Convolutional-Layern in einem Batchdurchlauf

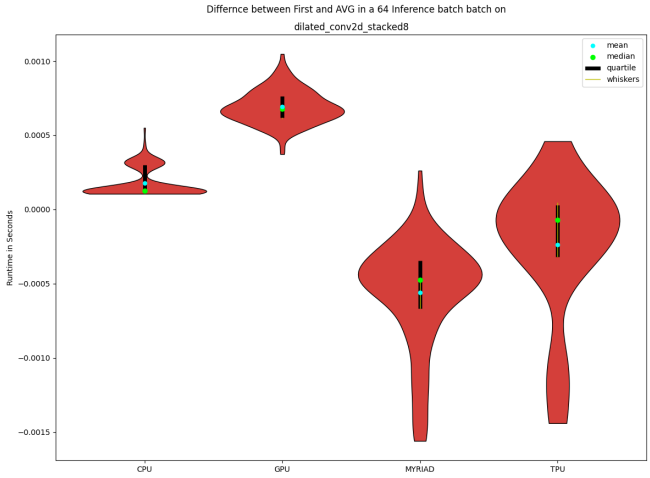

Abbildung 184. Differenz Laufzeit der ersten Inferenz und des AVG über ein Netz aus 8 dialeted Convolutional-Layern in einem Batchdurchlauf

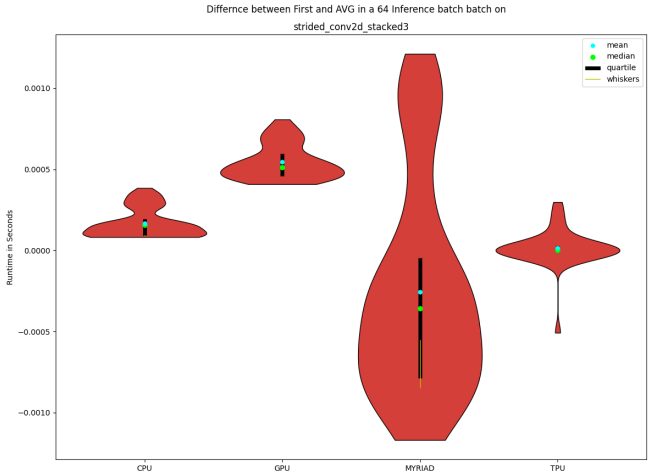

Abbildung 185. Differenz Laufzeit der ersten Inferenz und des AVG über ein Netz aus 3 strided Convolutional-Layern in einem Batchdurchlauf

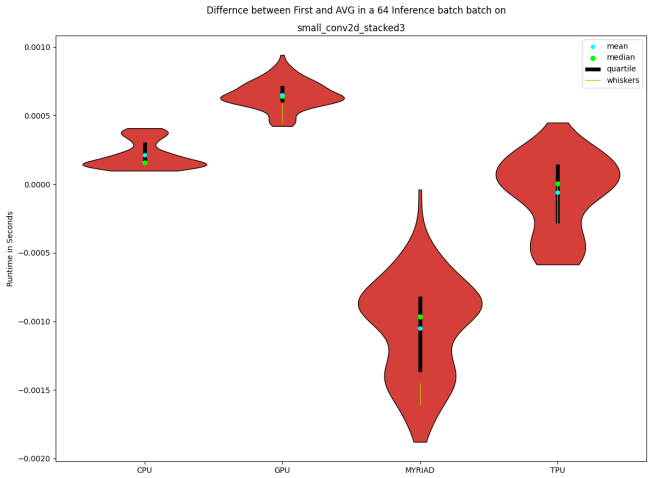

Abbildung 186. Differenz Laufzeit der ersten Inferenz und des AVG über ein Netz aus 3 kleinen Convolutional-Layern in einem Batchdurchlauf

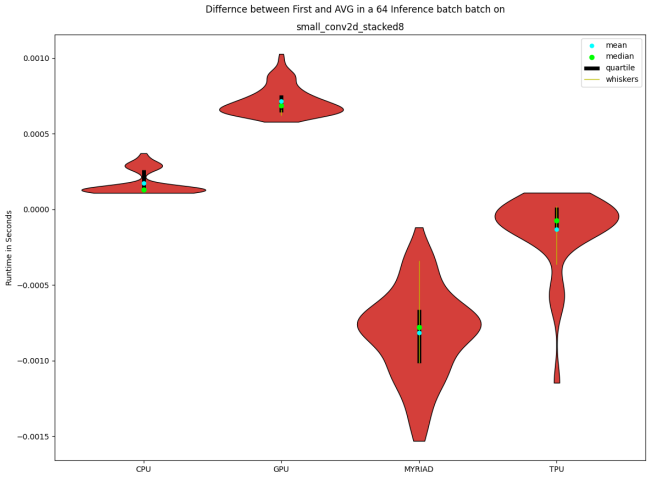

Abbildung 187. Differenz Laufzeit der ersten Inferenz und des AVG über ein Netz aus 8 kleinen Convolutional-Layern in einem Batchdurchlauf

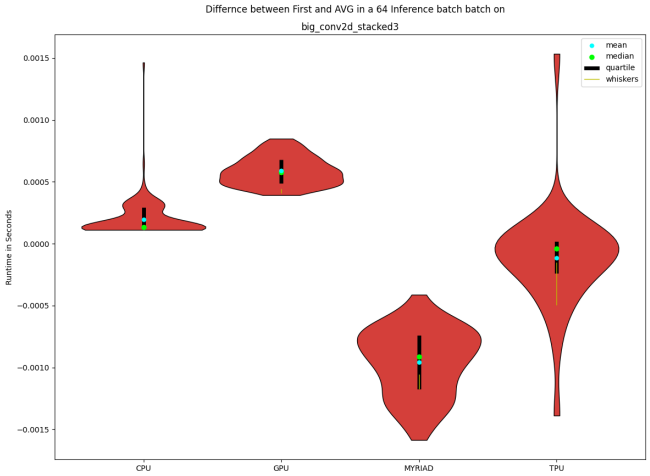

Abbildung 188. Differenz Laufzeit der ersten Inferenz und des AVG über ein Netz aus 3 großen Convolutional-Layern in einem Batchdurchlauf

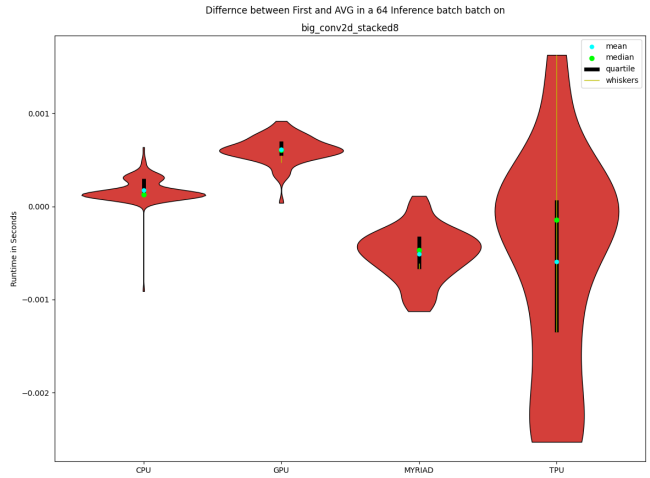

Abbildung 189. Differenz Laufzeit der ersten Inferenz und des AVG über ein Netz aus 8 großen Convolutional-Layern in einem Batchdurchlauf

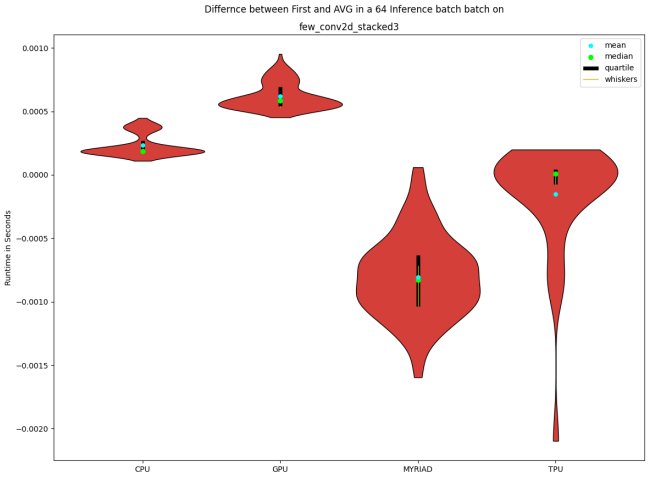

Abbildung 190. Differenz Laufzeit der ersten Inferenz und des AVG über ein Netz aus 3 Convolutional-Layern mit wenig Filtern in einem Batchdurchlauf

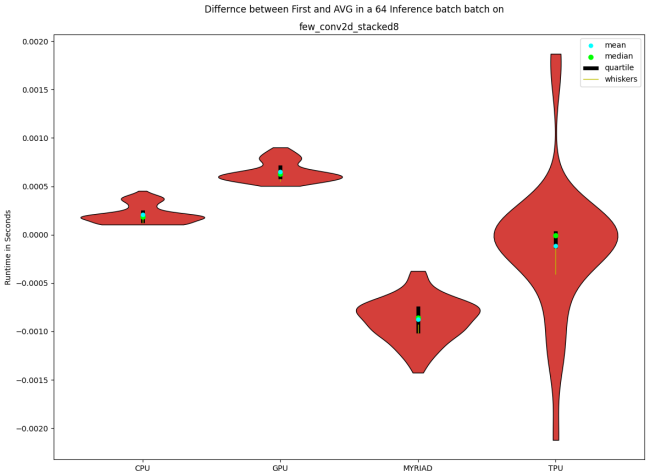

Abbildung 191. Differenz Laufzeit der ersten Inferenz und des AVG über ein Netz aus 8 Convolutional-Layern mit wenig Filtern in einem Batchdurchlauf

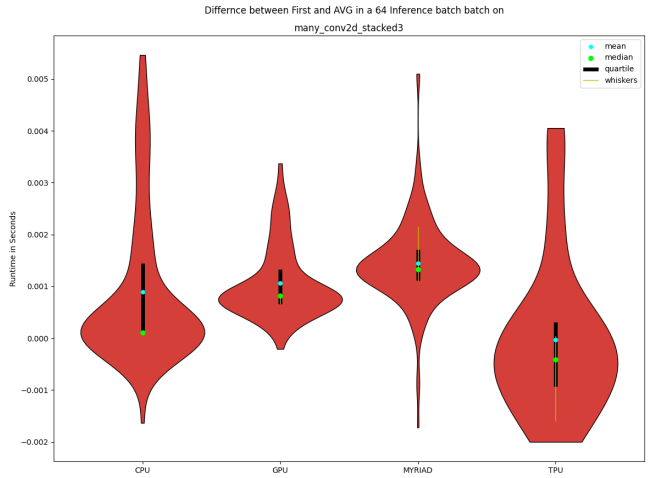

Abbildung 192. Differenz Laufzeit der ersten Inferenz und des AVG über ein Netz aus 3 Convolutional-Layern mit vielen Filtern in einem Batchdurchlauf

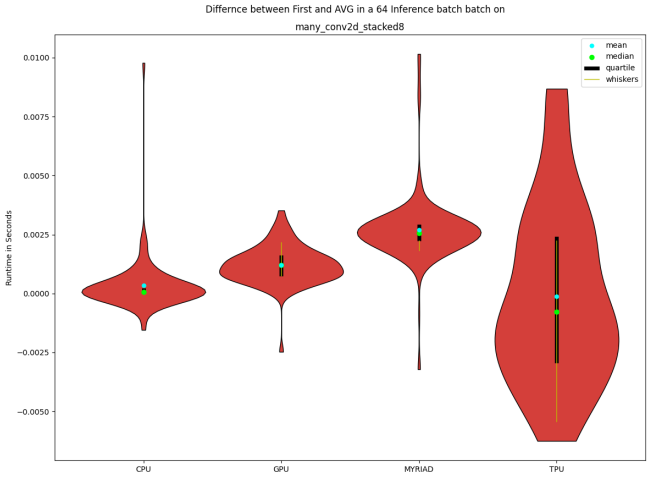

Abbildung 193. Differenz Laufzeit der ersten Inferenz und des AVG über ein Netz aus 8 Convolutional-Layern mit vielen Filtern in einem Batchdurchlauf

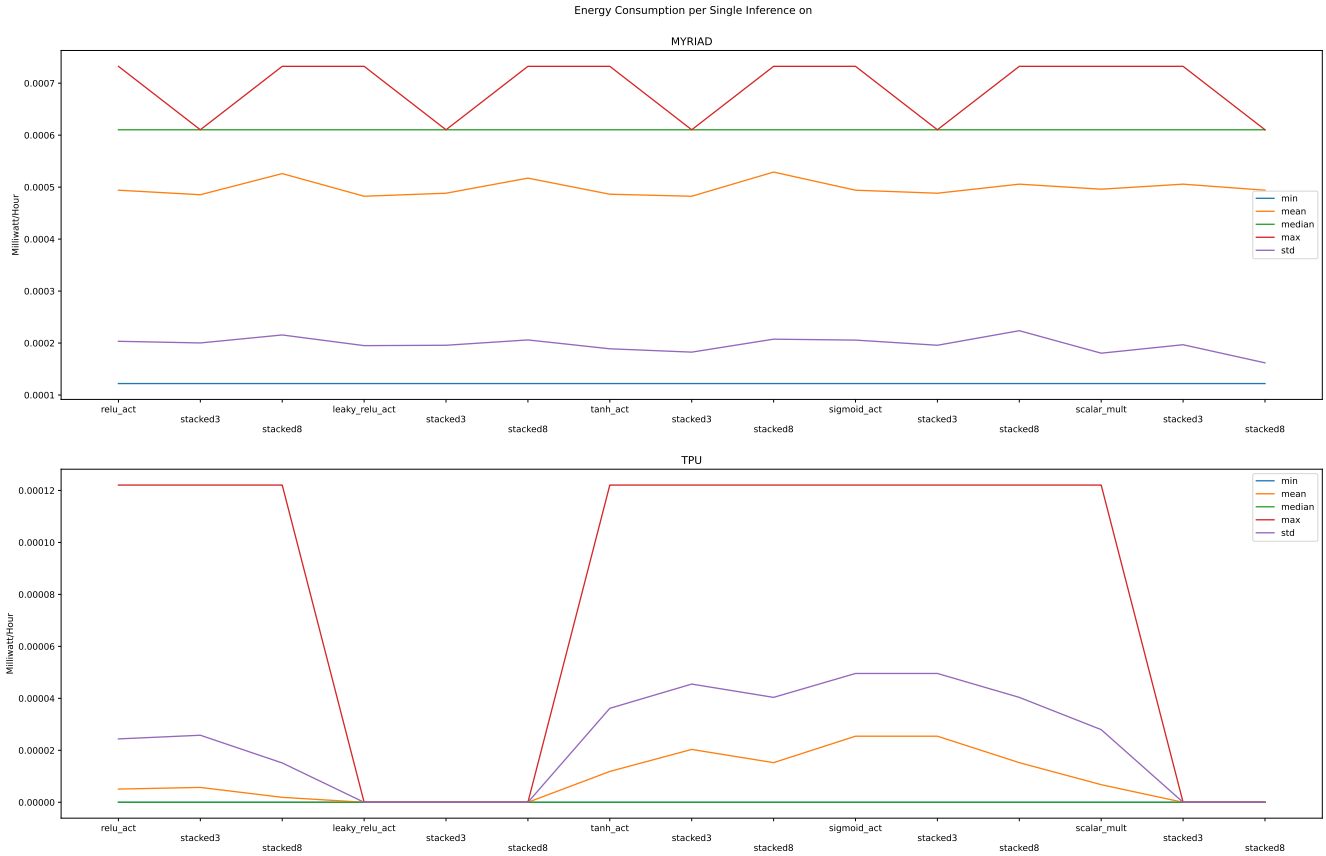

Abbildung 194. Energieverbrauchs-Statistik über Single-Operation Layer Netze

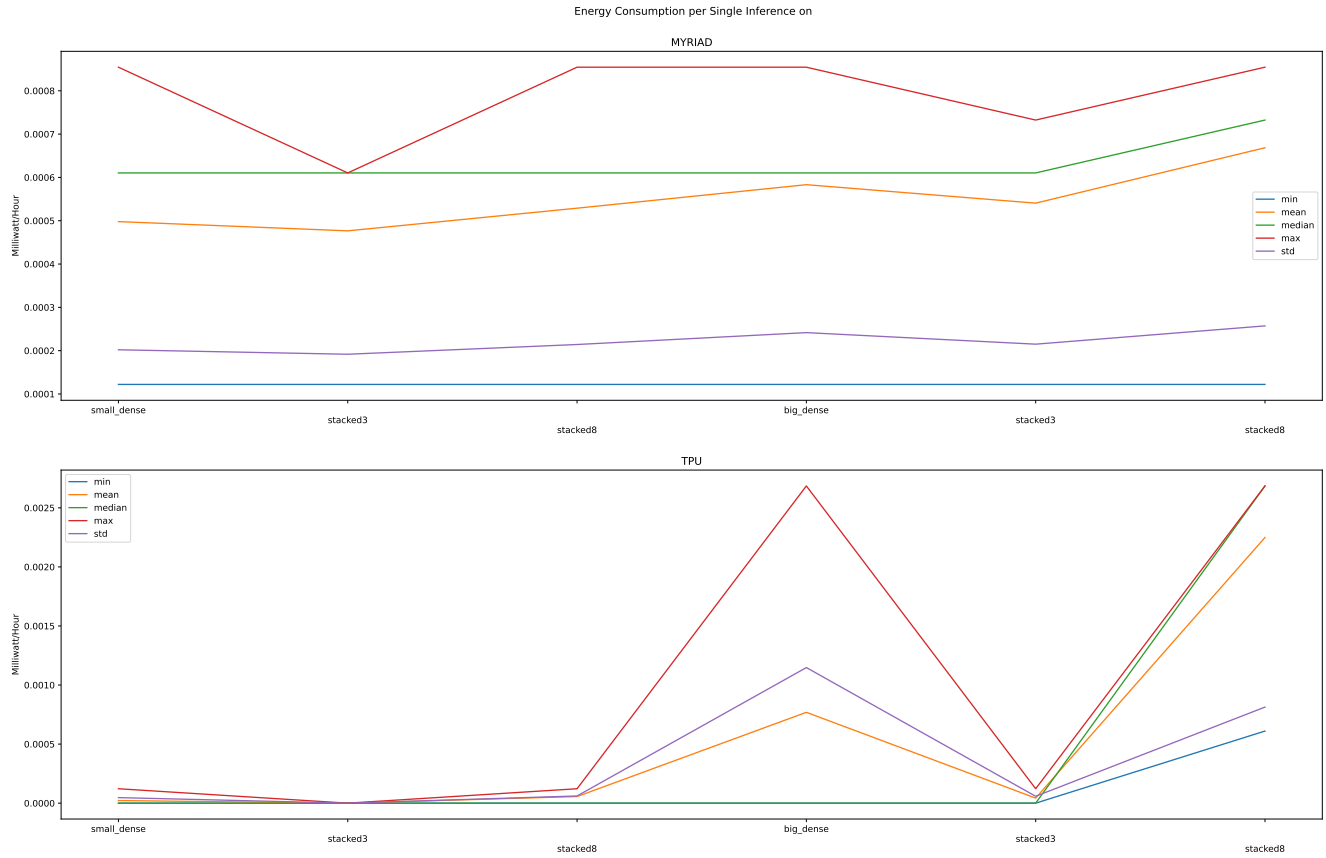

Abbildung 195. Energieverbrauchs-Statistik über Fully-Connected Layer Netze

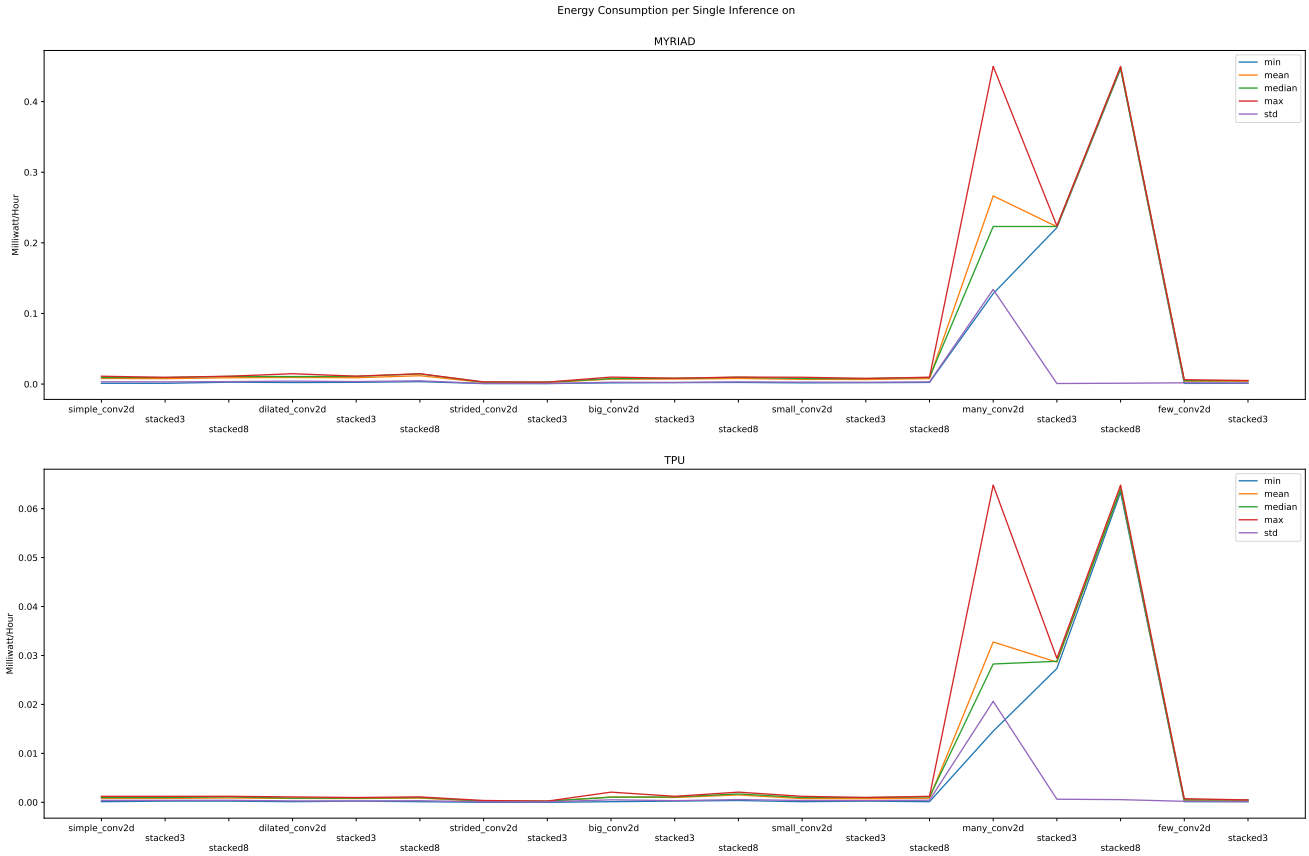

Abbildung 196. Energieverbrauchs-Statistik über Convolutional Layer Netze

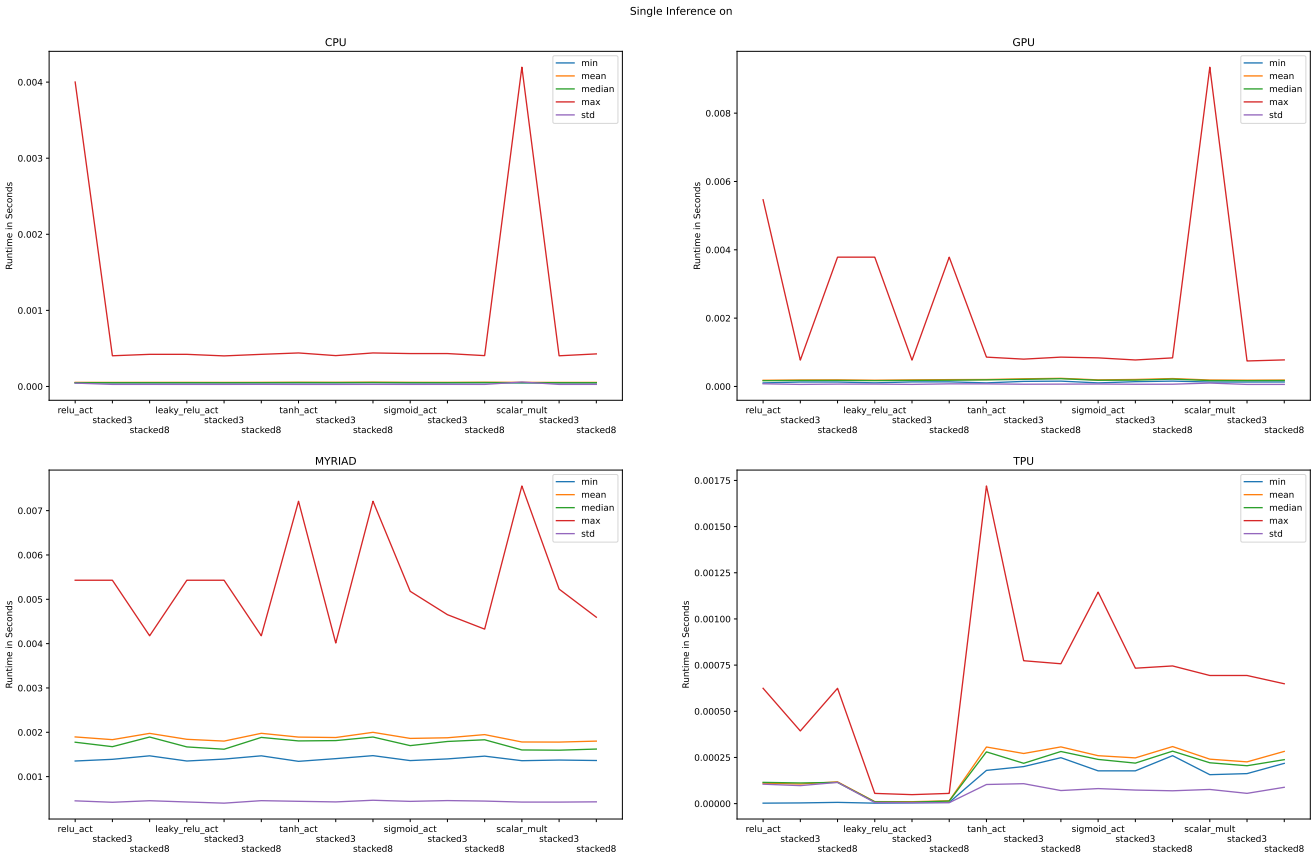

Abbildung 197. Laufzeit-Statistik für einzelne Inferenzen über Single-Operation Layer Netze

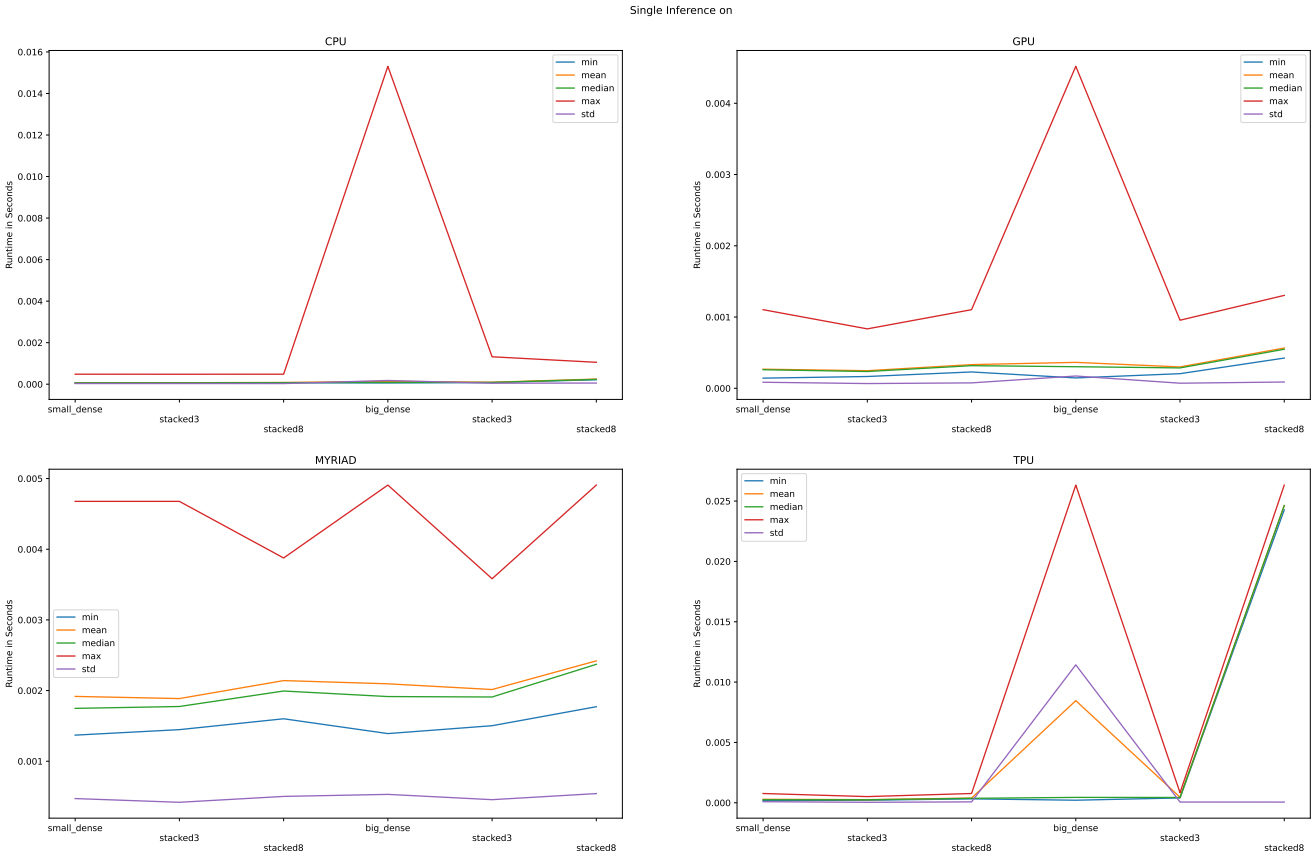

Abbildung 198. Laufzeit-Statistik für einzelne Inferenzen über Fully-Connected Layer Netze

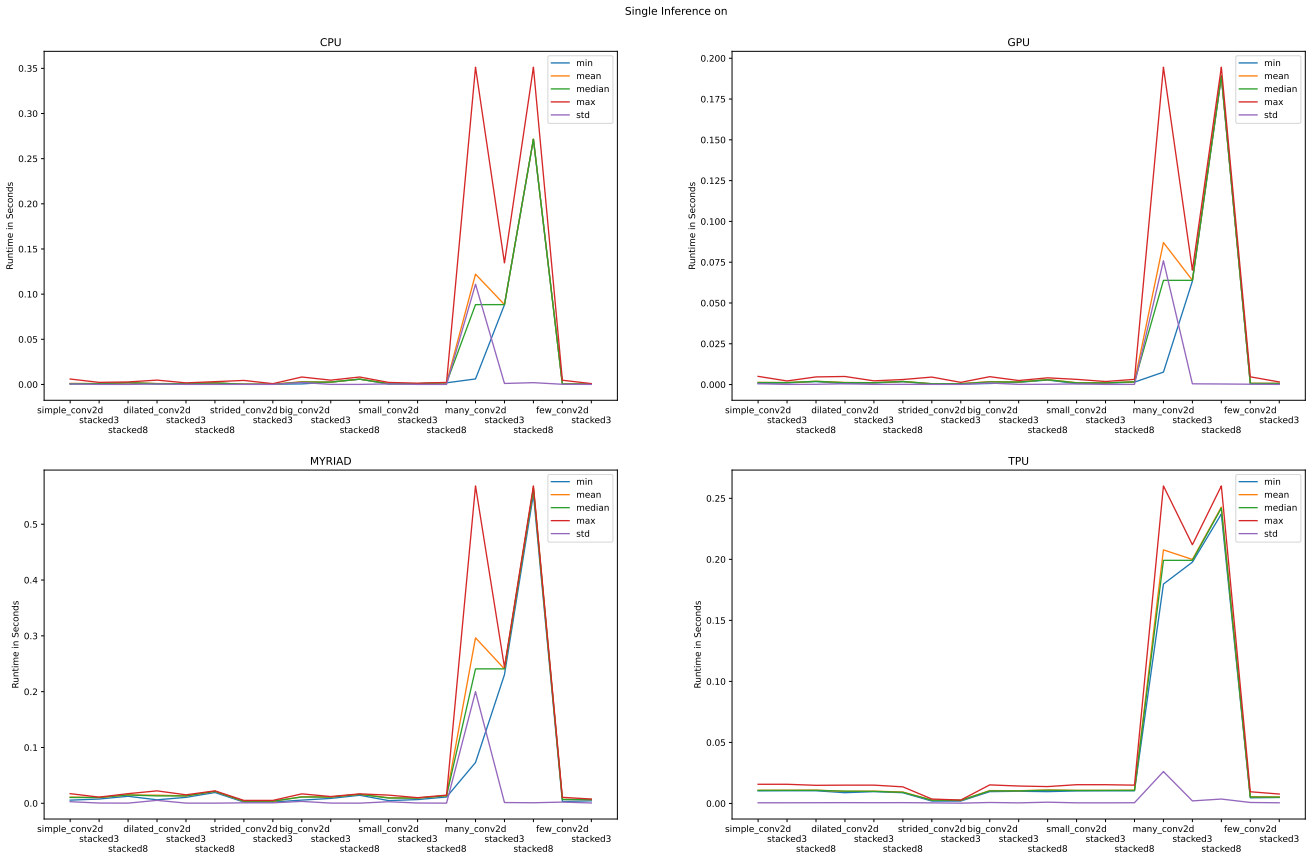

Abbildung 199. Laufzeit-Statistik für einzelne Inferenzen über Convolutional Layer Netze

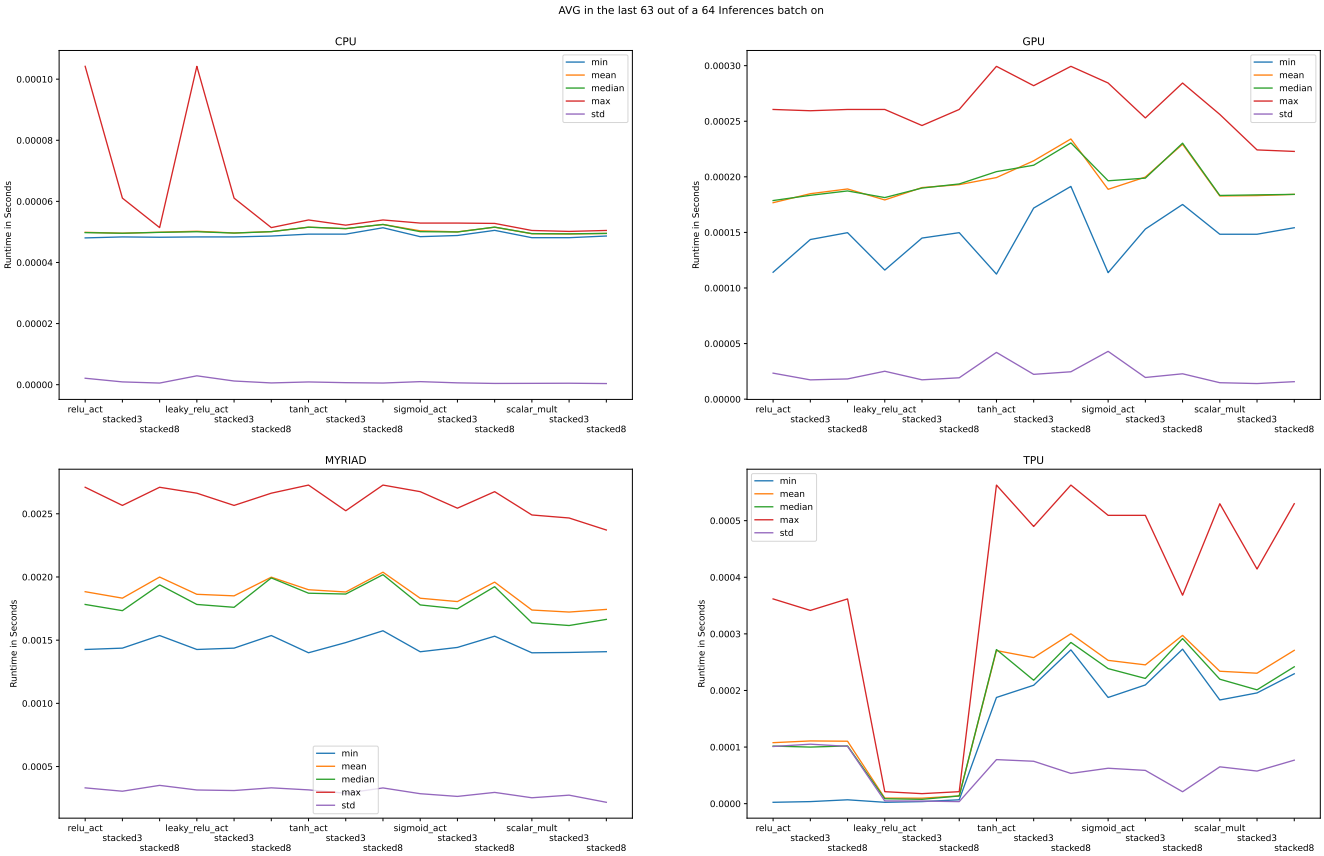

Abbildung 200. Laufzeit-Statistik für Batch Inferenzen über Single-Operation Layer Netze

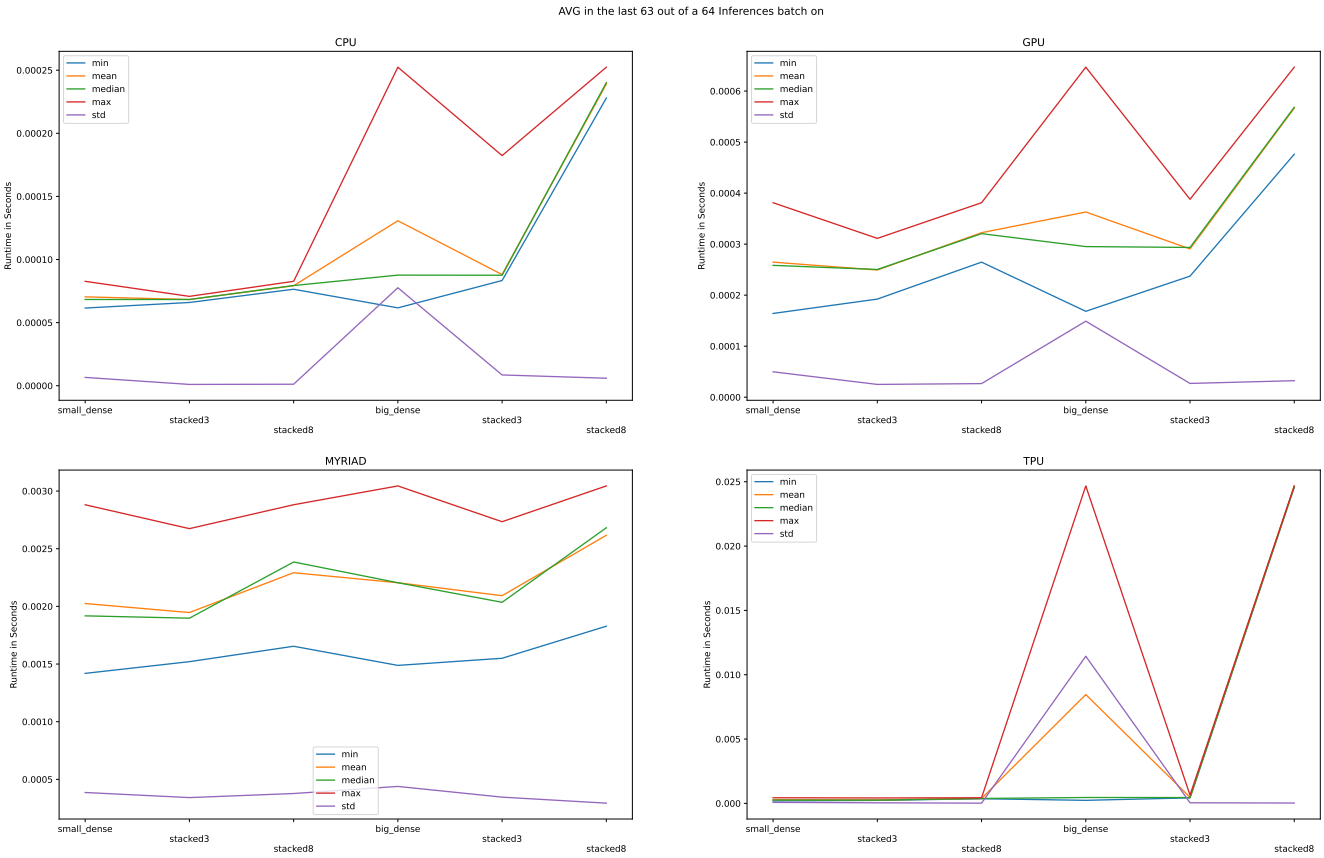

Abbildung 201. Laufzeit-Statistik für Batch Inferenzen über Fully-Connected Layer Netze

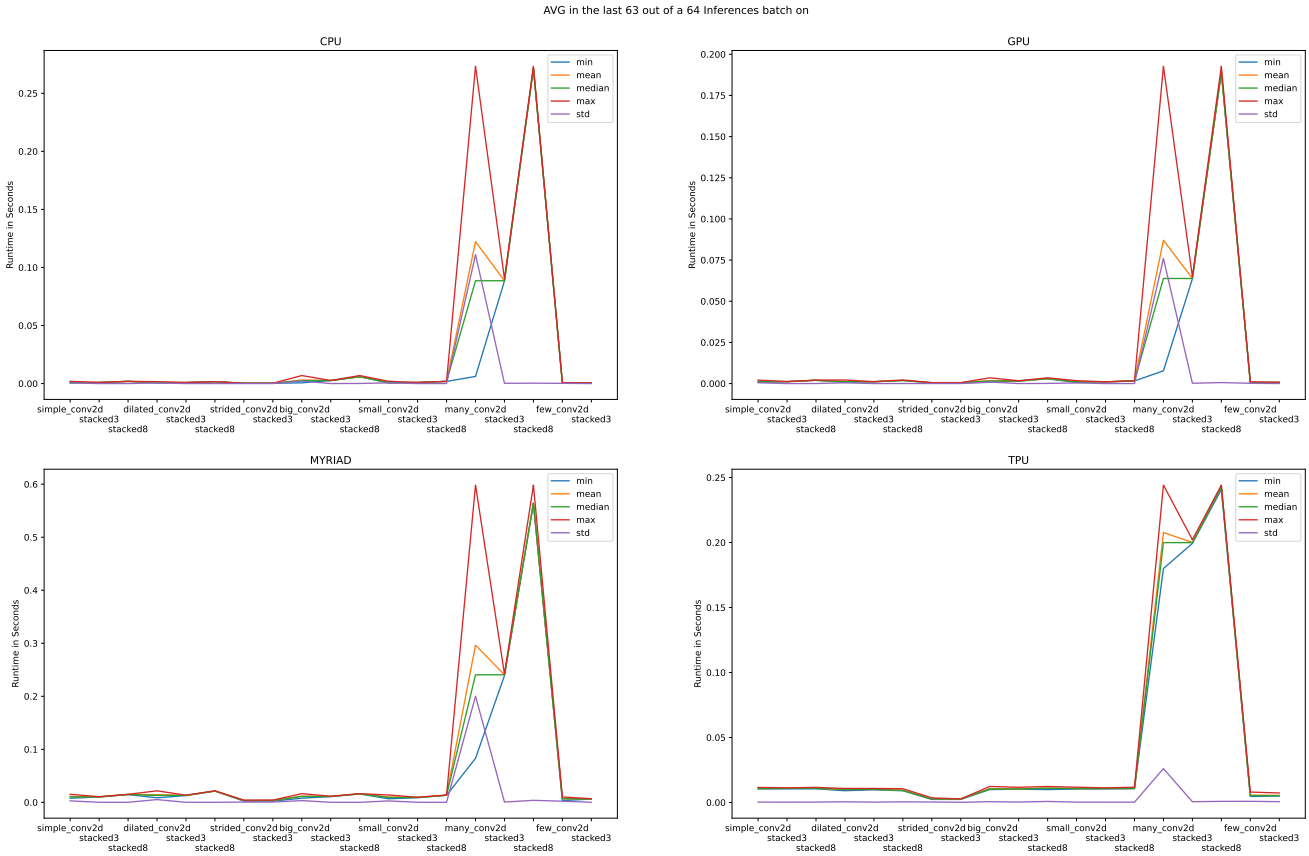

Abbildung 202. Laufzeit-Statistik für Batch Inferenzen über Convolutional Layer Netze

Abbildung 203. Laufzeit-Statistik für die Erste der Batch Inferenzen über Single-Operation Layer Netze

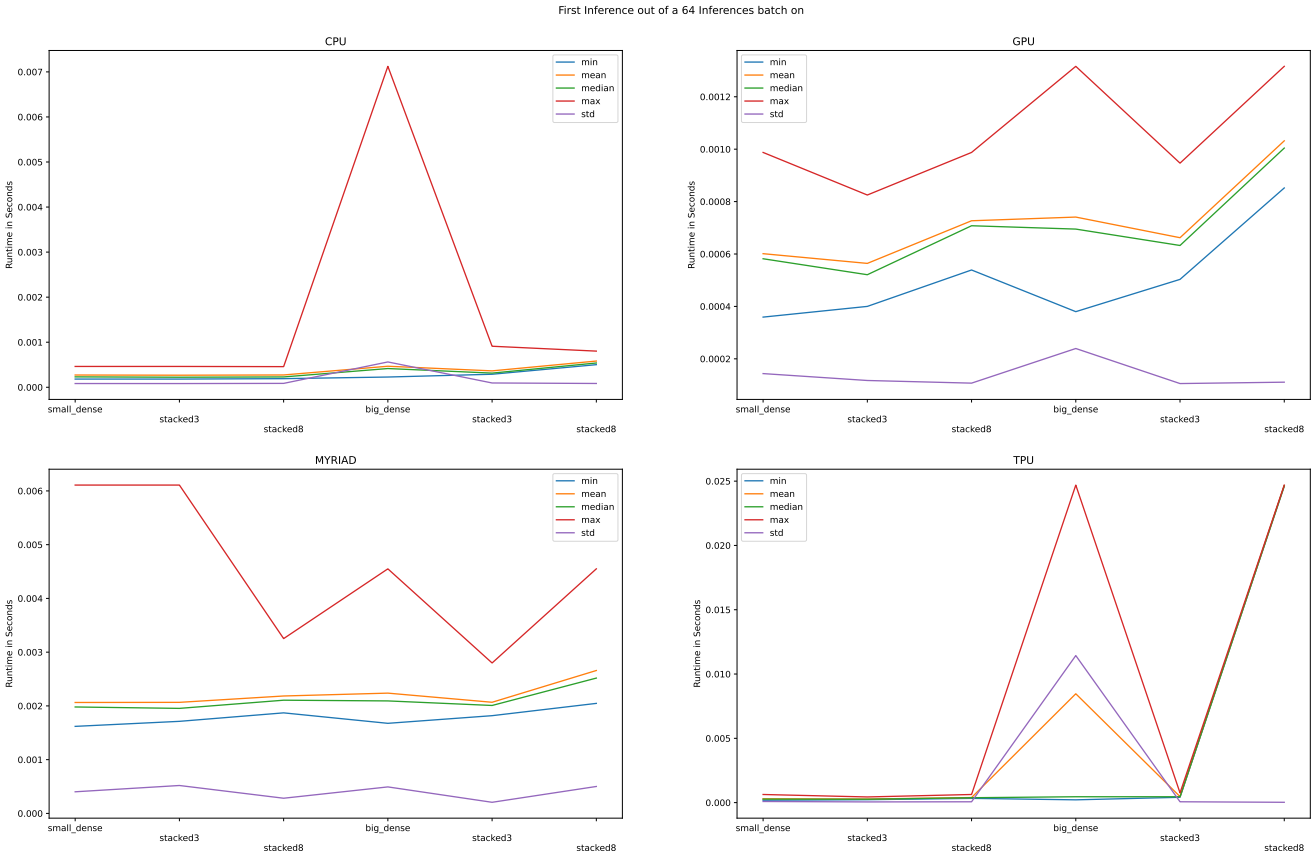

Abbildung 204. Laufzeit-Statistik für die Erste der Batch Inferenzen über Fully-Connected Layer Netze

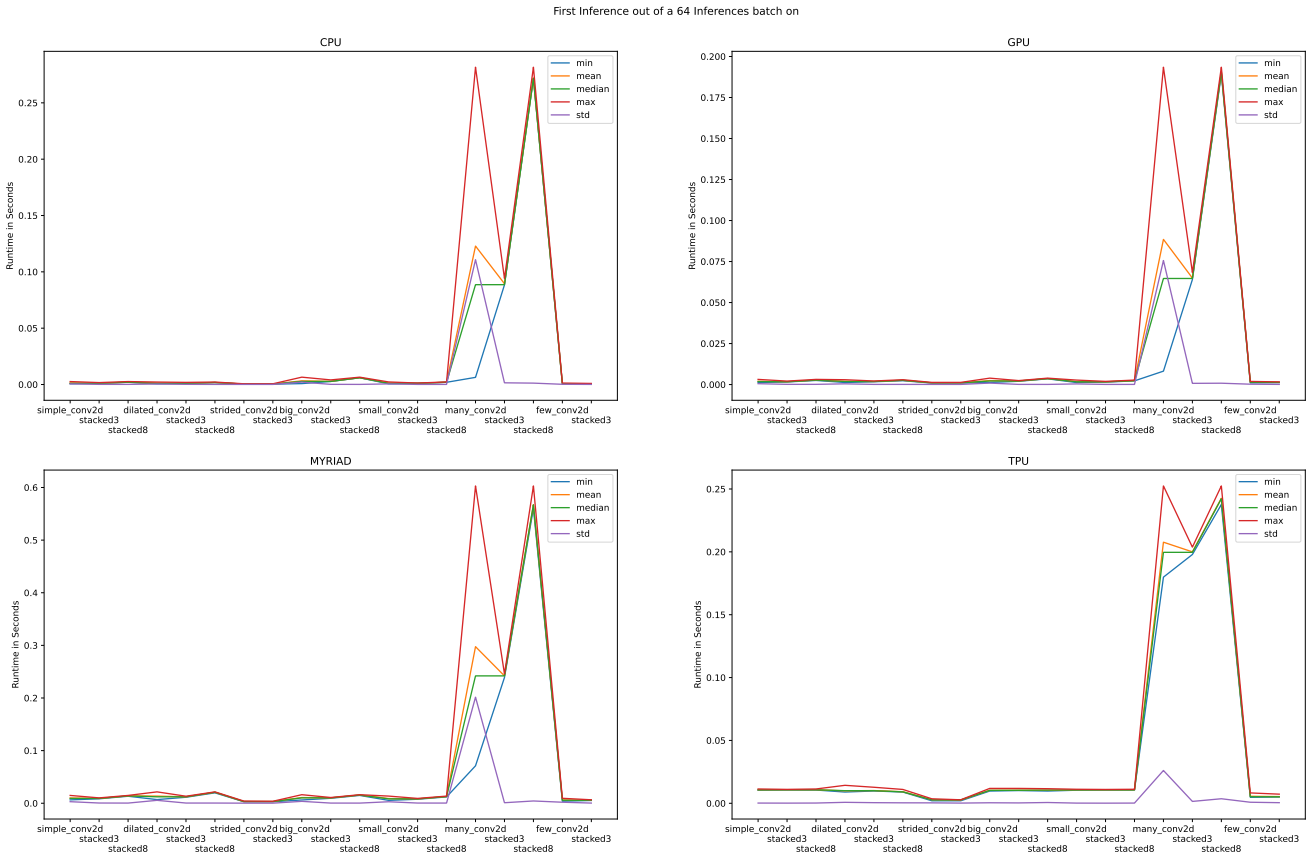

Abbildung 205. Laufzeit-Statistik für die Erste der Batch Inferenzen über Convolutional Layer Netze