DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Integrating Numerical Backend Modularity into Torchquad Using Autoray

Fritz Hofmeier

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Integrating Numerical Backend Modularity into Torchquad Using Autoray

Integrierung von Numerischer Backend Modularität in Torchquad mittels Autoray

Author:	Fritz Hofmeier
Supervisor:	Prof. Dr. rer. nat. habil. Hans-Joachim Bungartz
Advisors:	Dr.-Ing. Pablo Gómez, M.Sc. Fabio Gratl
Submission Date:	April 15, 2022

I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Munich, April 15, 2022 Fritz Hofmeier

Abstract

GPU programming for the Python3 language has become increasingly popular, especially in the context of machine learning. However, the number of backends which provide an array programming interface for GPUs has increased as well and algorithms which are implemented with different interfaces are often incompatible. torchquad is an example of a Python3 module which implements multi-dimensional numerical integration on GPUs and it has employed the array programming interface provided by the backend PyTorch. No module providing equivalent features is available for other backends, such as JAX and TensorFlow.

The autoray module offers an interface for array programming that automatically works for numerical Python3 modules with an API which is similar to NumPy's API. Since it enables algorithm implementations which support multiple backends at once, autoray tackles the compatibility problem.

The topic of this thesis is the adaptation of torchquad's code so that it uses autoray instead of PyTorch and thus supports numerical integration with the backends NumPy, JAX and TensorFlow in addition to PyTorch. Furthermore, this thesis includes investigations about the limitations of modularity between backends, code compilation with PyTorch, JAX and TensorFlow, and performance differences between backends.

Execution time comparisons reveal that PyTorch, JAX and TensorFlow all parallelize well on the GPU, and without compilation, PyTorch is often more than twice as fast as JAX and TensorFlow. Furthermore, the use of autoray has not led to a decrease in code readability in torchquad and the limitations of modularity are mostly due to backend-specific differences, so we can conclude that autoray is in general well-suited to implement computations with support for multiple backends.

Zusammenfassung

GPU Programmierung in der Python3 Sprache wurde mit der Zeit immer gefragter, insbesondere im Zusammenhang mit maschinellem Lernen. Allerdings hat sich die Anzahl der Backends, die eine Schnittstelle für Array-Programmierung für GPUs zur Verfügung stellen, auch erhöht und Algorithmen, die unterschiedliche Schnittstellen verwenden, sind oft zueinander inkompatibel. torchquad ist ein Beispiel für ein Python3 Modul, das mehr-dimensionale numerische Integration auf GPUs implementiert und die Schnittstelle des PyTorch Backends verwendet. Es gibt kein Modul mit equivalenten Eigenschaften für andere Backends, wie z.B. JAX und TensorFlow.

Das autoray Modul implementiert eine Schnittstelle für Array-Programmierung, die automatisch für numerische Python3 Module funktioniert, welche in ihrer API NumPy ähneln. Da es Implementierungen von Algorithmen ermöglicht, die mehrere Backends gleichzeitig unterstützen, kann autoray das Kompatibilitätsproblem lösen.

Das Thema dieser Arbeit ist die Anpassung von torchquad's Code, damit es autoray statt PyTorch verwendet und dadurch numerische Integration mit den Backends NumPy, JAX und TensorFlow zusätzlich zu PyTorch unterstützt. Zudem enthält diese Arbeit Untersuchungen über die Grenzen der Modularität zwischen Backends, Code Kompilierungen mit PyTorch, JAX und TensorFlow, und Performanzunterschiede zwischen Backends.

Vergleiche von Ausführungszeiten zeigen, dass die Parallelisierung auf der GPU mit PyTorch, JAX und TensorFlow gut funktioniert und dass ohne Kompilierung PyTorch oft mehr als doppelt so schnell wie JAX und TensorFlow ist. Des Weiteren hat sich die Code-Lesbarkeit in torchquad durch die Verwendung von autoray nicht verschlechtert und die Grenzen der Modularität werden hauptsächlich von Backend-spezifischen Unterschieden verursacht. Dadurch kann man schlussfolgern, dass autoray im Allgemeinen gut geeignet ist, um Berechnungen zu implementieren, die mehrere Backends unterstützen.

Contents

1 Introduction

In recent years, data science and machine learning have become increasingly popular. These scientific disciplines typically involve expensive numerical calculations with a large amount of data and sometimes the calculations are executed for a long time on high-performance computing systems. Since the computations can often be parallelized with single-instruction, multiple-data (SIMD) operations, executing them on GPUs, TPUs, FPGAs or other dedicated hardware can lead to a significant reduction of execution time and overall power consumption. While it is possible to use a low-level language such as C to achieve high performance, the numerical calculations are often implemented via the interpreted Python3 language, which is considered easy to learn and has numerous modules for scientific computing.

The Python3 interpreter is slow in comparison to compiled languages; therefore, there are tools, for example NumPy and PyTorch, that have an array programming interface which enables a user to implement their calculations with high-level SIMD array operations [Har+20]. In the context of this thesis, these tools are denoted numerical backends. The exact implementation of the array operations in a backend is hidden from the user. Therefore, the developers of a backend can add support for dedicated hardware to increase performance. Furthermore, the execution of these operations defines a dynamically-generated computational graph, which makes it possible to implement a backend that records this graph for further computations. Examples for these computations are backpropagation for the calculation of a gradient and the compilation of the graph to optimised machine code for a further increase in performance.

There are numerous popular Python3 modules which implement a numerical backend. These backends often have a similar user-facing array programming interface although they differ in their implementation and support for dedicated hardware, distributed computing, numerical derivation techniques, lazy execution of numerical operations, and other functionality. If a developer desires to employ GPUs for fast computations, they may implement their algorithm with PyTorch, for example. However, this backend choice can hinder the application of the algorithm in Python3 programs which employ another backend, for example TensorFlow, and the algorithm can only be executed on hardware which is supported by PyTorch. Using the similarity in array programming interfaces between backends, the Python3 module autoray tackles this problem by offering an interface which wraps backend-specific APIs and automatically dispatches numerical operations to the currently selected backend.

The torchquad module is developed by the European Space Agency's Advanced Concepts Team. In version 0.2.4, torchquad is implemented with the PyTorch backend and it is the first module for numerical integration with PyTorch which supports multi-dimensional integrands, differentiability, and execution on the GPU [GTM21]. The main contribution described in this thesis is a rewrite of torchquad's code so that it uses autoray instead of PyTorch for numerical operations. Using autoray, most numerical integrators of torchquad now support NumPy, JAX and TensorFlow in addition to PyTorch. This enables users to employ the same quadrature algorithms in different Python3 projects which target different backends and hardware. In addition to support for multiple backends, the work conducted during this thesis includes an investigation of backend-specific code compilation to increase performance for repeated quadrature, and detailed performance analyses between backends and compilation options. Furthermore, code changes include the addition of special numerical operations, which have led to performance increases especially with the VEGAS+ integrator.

The next chapter in this thesis first gives background information about quadrature algorithms. Then it describes Python3 modules for numerical calculations such as autoray and the numerical backends, available hardware for SIMD operations, and work related to this thesis. After that, Chapter 3 explains the integration of autoray into torchquad and optional function compilation for increased performance, and then describes limitations of support for multiple backends. Then Chapter 4 presents an investigation of performance differences between JAX, TensorFlow and PyTorch with various configurations and shows a validation of the accuracy of torchquad's VEGAS+ implementation. Finally, Chapter 5 concludes this thesis.

2 Background

This chapter first gives an overview about the quadrature algorithms supported by torchquad. Then it explains how numerical computations are commonly implemented in the Python3 language. After that, it describes hardware for parallelized computing and other work related to this thesis.

2.1 Quadrature Algorithms

Numerical integration, also called quadrature, refers to the numerical approximation of an integral and typically involves the substitution of $\int_D f(x)dx$ with the sum $\sum_{x \in P \subset D} w_x f(x)$, where D is the integration domain, f is the integrand function, P is a finite set of sample points, and w_x is a weighting coefficient for the corresponding sample point x. The sample points and weighting coefficients differ between quadrature rules. At the time of writing, torchquad supports five multi-dimensional quadrature methods: the non-deterministic Monte Carlo and VEGAS+ [Lep21], and three composite Newton-Cotes methods, which are the composite Trapezoid, Simpson and Boole rules. In torchquad, the integration domain for all quadrature methods is a d-dimensional hypercuboid, which is a generalisation of a rectangle to multiple dimensions; however, in general, there are many possible domain types, for example the surface of a sphere and the two-dimensional unbounded space $\mathbb{R}^2$. With the Newton-Cotes rules, the sample points P are points in a regular grid which is d-dimensional, has n points per dimension and is translated and scaled into the integration domain. For example, for the domain $D = [0, 3]^d$, we have $P = \{0, \frac{3}{n-1}, \frac{6}{n-1}, \ldots, 3\}^d$. To calculate an integral result, the quadrature method evaluates the integrand at these sample points and then applies the multi-dimensional composite Trapezoid, Simpson or Boole rule. With the Monte Carlo method, P is a set of N points randomly chosen in the integration domain and the integral result is $\frac{V}{N} \sum_{x \in P} f(x)$, where f is the integrand and V is the integration domain volume.

The VEGAS+ quadrature method, also called VEGAS Enhanced, is adaptive, so it alternatingly evaluates the integrand and updates its strategy to select the next set of sample points. Similarly to Monte Carlo quadrature, VEGAS+ evaluates the integrand at randomly chosen sample points; however, it applies importance and stratified sampling, and evaluates the integrand at multiple sets of sample points

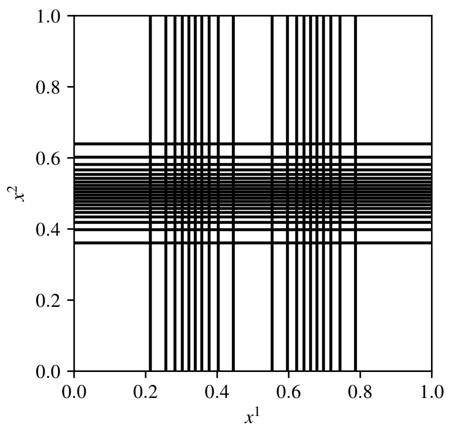


Figure 2.1: Visualisation of the VEGAS map transformation along the first two dimensions for a four-dimensional integrand. The integrand is a sum of two Gaussian peaks centered around $(0.33, 0.5, 0.5, 0.5)$ and $(0.67, 0.5, 0.5, 0.5)$, so the transformation along the first dimension moves points close to $x^1 = 0.33$ and $x^1 = 0.67$, and the transformation along the second dimension moves points close to $x^2 = 0.5$. The image is taken from [Lep21].

due to its adaptivity. It uses a so-called VEGAS map and stratification to transform random sample points so that the number of points is high in regions where the integrand has a major impact on the integral result or has a high variance and low in other regions. The VEGAS map maintains a monotonically increasing, piecewise linear function for each dimension of the integration domain. When an optimised map is applied to a set of sample points, it moves the points into regions where the absolute value of the integrand is high, which can have the effect of flattening peaks. Since there is a separate piecewise linear function for each dimension, the flattening of the map works well if the regions where the integrand is large are parallel to the integration domain boundary surfaces and has no significant effect if these regions are diagonally shaped. Figure 2.1 shows an example visualisation of a VEGAS map transformation. The stratification partitions a domain into hypercuboids and determines how many points to sample per hypercuboid. To calculate a set of sample points for an integrand evaluation in VEGAS+, the stratification generates random sample points, under the condition that each hypercuboid contains the configured number of points, and then the VEGAS map transforms these points. The number of points for a hypercuboid is set to a high number if the integrand has a higher variance within the region corresponding to this hypercuboid in comparison to other regions. In comparison to the map, the stratification works well for diagonal integrands;

however, the number of hypercuboids scales exponentially with the dimension, whereas the resolution of the map does not depend on the dimensionality. Since the optimal map and stratification are unknown, VEGAS+ estimates them from previous sample points and corresponding integrand evaluations, which constitutes the adaptivity of VEGAS+. There are numerous parameters which influence the adaptation of the map and stratification to the integrand, and some of these parameters depend on the exact implementation of the algorithm.

All of these quadrature rules can be parallelized well with SIMD operations. The grid points for the Newton-Cotes rules can be calculated with a Cartesian product, which is explained in Subsection 3.4.1, and the performance-critical part of the final Newton-Cotes integral calculation can be implemented with sums and products which involve array slices of dimension $d-1$, where d is the dimensionality of the integration domain. The Monte Carlo rule consists of random number generation and a sum of the integrand output, which can both be parallelized, and a parallelization of all performance-critical calculations in a VEGAS+ implementation is also possible, but it is more involved as explained later in this thesis. In comparison to the non-deterministic rules, the Trapezoid, Simpson and Boole rules can exactly integrate polynomials of degree one, three and five, respectively, but in general, for a high accuracy with composite Newton-Cotes integration, the required number of sample points grows exponentially with the dimension. This exponential growth is an example of the so-called curse of dimensionality, which denotes the phenomenon that for certain algorithms the computational complexity increases exponentially with the dimension [BBC61]. The next section describes Python3 modules with which it is possible to implement efficient numerical integration.

2.2 Numerical Python3 Modules

Python3 has modules which enable fast numerical calculations. This section first explains commonalities between the numerical backend modules and autoray's interface to abstract away differences between these backends. After that it describes the numerical backends NumPy, Torch, JAX and TensorFlow, which are currently supported by torchquad, and features of these backends such as support for code compilation.

2.2.1 Common Numerical Operations

This section explains common operations of Python3 modules which implement a NumPy-like API. While it is possible to store a large vector of floating-point numbers as a list or a similar Python3 container type and use Python3 loops to perform component-wise operations on them, the flexibility of the data types and data structures makes it

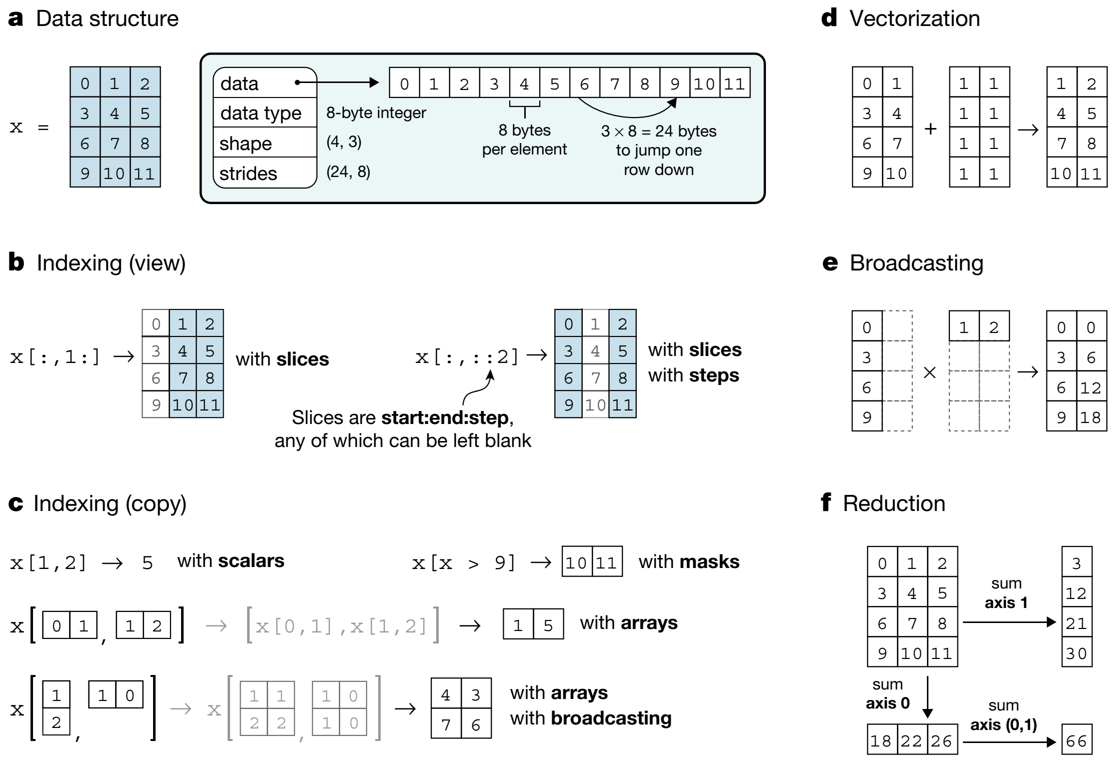


Figure 2.2: NumPy's fundamental array concepts, which include the array data structure, array indexing, component-wise operations, broadcasting, and reduction operations such as a sum along a dimension. The visualisation is taken from [Har+20].

difficult for the Python3 implementation to optimise the executed machine code for a low memory usage and running time. To solve this problem, it is possible to use the tensor type of a numerical backend, which implements a NumPy-like API.

Array programming with NumPy [Har+20] describes the fundamental array concepts which apply to numerical backends implementing the NumPy-like API or an API similar to it. Figure 2.2 shows a visualisation of these concepts.

From a user perspective, a tensor, also called array, is an object which contains a hidden pointer to raw data in the memory and information about this data such as the element data type and tensor shape, and the object has methods to perform common numerical operations. The raw data corresponds to an array in low-level languages such as C or to a similar data buffer on specialised hardware, e.g. GPUs. The element data type defines how many bytes one element in the raw data occupies and how to interpret the elements; a data type can be, for example, an IEEE 754 32 bit floating-point number. The shape information defines the number of elements and the dimensionality for numerical operations on the tensor. Figure 2.2 additionally shows tensor views, which are tensors where some of the raw data is skipped according to the information in an additional strides entry in the tensor data structure. Numerical backends may employ views to reduce the memory usage in some situations, which we can consider to be an implementation detail that can differ between backends.

There are dedicated functions to create tensors with user-specified shapes and to convert a Python3 container, e.g. a list, to a tensor. In comparison to a vector or matrix interpretation of an array, on a tensor multiplication, exponentiation, and other Python3 operations are all executed component-wise. If a binary operation is applied on tensors with different shapes, broadcasting is used for the component-wise operation unless the shapes are incompatible. Conceptually, broadcasting repeats each of the operand tensors along some of their dimensions until they have the same shape and the following binary operation is executed component-wise. For example, adding a tensor x of shape $(3, 4)$ to a tensor y of shape $(1, 4)$ first repeats y three times along its first dimension and then performs component-wise addition with x. Implementation-wise, the repetition and component-wise operation can be merged for a higher performance and lower memory requirement.

Since tensors in torchquad are often very large and tensor operations can be executed lazily or be traced for a JIT compilation, we can think of Python3 as a language for metaprogramming which enables us to define computationally intensive quadrature algorithms with tensor operations.

2.2.2 The autoray Module

With the Python3 library autoray (AUTOmatic-arRAY) [Gra+], we can write Python3 code for numerical calculations with operations that are automatically dispatched to a chosen numerical backend, which makes it possible to reuse the same code for multiple backends. These backends implement a NumPy-like API.

The so-called `do` function is autoray's main component for backend-agnostic numerical operations. This function receives as arguments a numerical operation, the arguments for this operation, and optionally the numerical backend. The numerical operation argument is a string and can be, for example, `"sum"`, which maps to the `numpy.sum` function if NumPy is the backend. The arguments for the numerical operation in this example include the tensor on which to perform the sum and optional arguments such as `axis`. autoray has multiple ways to define the backend for the numerical operation. The backend argument of `do` can be omitted, a tensor, or a string, e.g. `"numpy"`. If it is a tensor, autoray infers the backend string from it. If it is omitted,

```python
c = [0.3, 2.0, 5.0]

import numpy as np
def f_np(x):
    y = np.array(c, dtype=x.dtype)
    return np.sum(y * x[:3])

import torch
def f_torch(x):
    y = torch.tensor(c, dtype=x.dtype)
    return torch.sum(y * x[:3])

import tensorflow as tf
def f_tf(x):
    y = tf.constant(c, dtype=x.dtype)
    return tf.math.reduce_sum(y * x[:3])

from jax import numpy as jnp
def f_jax(x):
    y = jnp.array(c, dtype=x.dtype)
    return jnp.sum(y * x[:3])
```

```python
c = [0.3, 2.0, 5.0]

from autoray import numpy as anp
def f(x):
    y = anp.array(c, dtype=x.dtype, like=x)
    return anp.sum(y * x[:3])
```

Figure 2.3: Simple example code to showcase an application of autoray. All five functions get as input a one-dimensional tensor x and calculate $f(x) = 0.3x_1 + 2x_2 + 5x_3$ with vectorized tensor operations. In the function on the right, autoray determines the backend for the tensor initialisation of `y` from the tensor `x` passed to the `like` argument, the current backend automatically handles the slicing of `x` and multiplication by `y`, and for the sum operation, autoray determines the backend from the tensor passed as input argument to `anp.sum`. The code images were generated with Geany.
Left: Separate implementations of f for NumPy, PyTorch, TensorFlow and JAX which all support only the respective backend.
Right: A single function which supports all backends simultaneously using autoray.

autoray either uses a backend defined by a context manager or it infers the backend string from a tensor input argument of the numerical operation, for example the first argument in case of the sum operation.

When autoray executes the `do` function, it first determines the numerical backend. Then it uses the numerical operation string and backend string as key to get a cached function which maps NumPy-like to backend-specific arguments and then applies a backend-specific function on the mapped arguments. For example, if the operation and backend strings are `"sum"` and `"torch"`, it may replace an `axis` argument with `dim` and then executes `torch.sum`. If the function is not cached, autoray creates it, which typically involves the import of the numerical backend and the assembling of a function which replaces, for example, argument names. From a user perspective, two application examples of `do` are the replacements of `x = torch.ones((3,1))` by `do("ones", (3,1), like="torch")` to initialise a PyTorch tensor with ones and `torch.sum(x, dim=0)` by `do("sum", x, axis=0)` to execute the sum operation. Since autoray caches the functions in a dictionary, its time overhead is mostly negligible compared to the time required for the execution of the numerical operations.

In addition to the `do` function, for a different code style autoray offers a NumPy mimic object which internally executes `do`. torchquad commonly imports this object and denotes it `anp`. To use `anp` instead of `do`, we can replace, for example, `do("sum", x, axis=0)` by `anp.sum(x, axis=0)` when performing the sum operation. Figure 2.3 shows an example where a function supports multiple backends using autoray. Furthermore, autoray has helper functions for backend-agnostic type conversions, common naming of data types and backend name inference from tensors.

2.2.3 Numerical Backend Implementation Techniques

This section explains terms and implementation techniques which are common to numerical backends and may be helpful to distinguish backend-specific compromises between flexibility and performance. [Suh+21] gives an overview about implementation techniques for fast numerical computations in Python3. It is possible to differentiate between eager execution mode and execution of a static graph. With eager execution, the Python3 code is executed and performs numerical operations or calls functions which do not belong to a numerical backend, for example message printing. For higher performance, the numerical backend may execute operations on the GPU asynchronously to the Python3 interpreter as long as it is transparent to the user; for example, a `print` statement may execute concurrently to a numerical computation on the GPU unless the to-be-printed message contains the output of this numerical computation. On the other hand, a backend can support the compilation of a Python3 function to a static graph of numerical operations. This compilation can be tracing, where Python3 code is executed once to collect executed numerical operations, direct compilation, where a subset of the Python3 language is compiled by the backend instead of being executed by the CPython interpreter, or a compromise between tracing and direct compilation [Suh+21]. Later the operations in the graph can be executed without interpreting the Python3 code again. Since the graph can be strongly optimised and compiled to hardware-specific machine instructions, its execution is typically faster than eager execution of the Python3 code, but it is less flexible because complicated Python3 control flow may not be preserved during tracing or cannot be compiled.

Accelerated Linear Algebra (XLA) [Goo] is a compiler for a graph of numerical computations developed by Google. It abstracts away the conversion of the graph to machine code optimised for a given hardware. Its most important optimisation is the fusion of nodes in the graph, for example to merge a multiplication and addition into a single CUDA kernel [Goo]. The abstraction is helpful to add support for new hardware to libraries which use XLA [NSW18]. XLA has some limitations which restrict its applicability; for example, input tensors of the computation graph must have a fixed size [SVK20] and operations which change data in-place are unsupported [Suh+21].

2.2.4 Supported Numerical Backends

The numerical backends and their versions currently supported by torchquad are PyTorch 1.10.0, TensorFlow 2.7.0, JAX 0.2.25 and NumPy 1.19.5. All of these backends are free open-source software. In comparison to NumPy, the other backends are all frameworks for machine learning and support automatic differentiation and CUDA.

PyTorch is the numerical backend which torchquad supported before the code changes associated with this thesis. In comparison to other deep learning frameworks such as TensorFlow, PyTorch has a great focus on high performance in eager execution mode [Pas+19]. The performance-critical parts of PyTorch are implemented in the C++ libtorch library, which simplifies implementations of Torch in multiple programming languages and allows fast vectorized tensor operations [Pas+19]. Furthermore, when using CUDA, PyTorch often executes operations on the GPU asynchronously from the Python3 interpreter, which can lead to a performance improvement [Pas+19].

TensorFlow is a numerical backend developed by Google. Its initial version was released in 2015 [LP19] and it is the successor of DistBelief from 2011, which is part of the Google Brain project [Mar+15]. Since version 2, which was released in 2019 [He19], it uses eager execution mode by default. It also supports trace compilation of Python3 code to a static graph, which is optimised with TensorFlow's Grappler[1] graph optimisation system, and additionally with XLA if it is enabled. XLA

[1]TensorFlow graph optimization with Grappler: `https://www.tensorflow.org/guide/graph_optimization` (Accessed: 2022-03-16)

performs hardware-specific optimisation [Goo] but TensorFlow disables it by default for backwards compatibility since some of TensorFlow's functionality is unsupported by XLA.

JAX (JAX is Autograd and XLA) [Bra+18] is another numerical backend developed by Google. It is based on XLA for fast compiled numerical operations [SVK20] and Autograd [MDA15] for automatic differentiation of functions. It supports execution of a static graph compiled with XLA and eager execution mode, where it performs numerical operations asynchronously. Since it is built to use XLA from the ground up [SVK20], functions which execute numerical operations in JAX should be pure and avoid indexed assignments. In comparison to PyTorch and TensorFlow, the Python3 module for JAX itself does not implement a large number of common machine learning algorithms; these algorithms are available in separate modules, for example Flax [Hee+20], Haiku [Hen+20] and Elegy [Poe21].

NumPy is the main array programming module for Python3 and is used by many other Python3 modules, for example Matplotlib and pandas [Har+20]. It was first published in 2005 and is based on two predecessors: Numeric and Numarray, which is a reimplementation of Numeric [Har+20]. NumPy does not support execution on the GPU and uses only a single CPU core. However, it is optimised for a fast synchronous eager mode execution, thus calling array operations has less overhead compared to other backends, and it uses SIMD instructions of the CPU.

2.2.5 Available Compilation Methods

PyTorch, TensorFlow and JAX offer functions to compile Python3 code, which can increase the performance significantly. With JAX we can wrap a Python3 function with `jax.jit` so that it is compiled just in time (JIT) into code optimised for selected hardware. During the first execution of the wrapped function, JAX executes the Python3 code while tracing which numerical operations are performed, which gives a computational graph encoded in the jaxpr intermediate language; then it applies the XLA compiler on the jaxpr code to generate optimised machine code, and finally, it evaluates the function on the user-provided arguments and returns the result. In subsequent executions of the wrapped function, JAX executes only the compiled code.[2] This means the Python3 interpreter executes the function's Python3 code only once and side effects, such as printing messages to stdout, no longer happen when the compiled function runs. Even without `jax.jit`, JAX internally compiles some of its functions just in time during the first execution, which leads to a noticeable time difference between

[2]JAX's JIT compilation documentation: `https://jax.readthedocs.io/en/latest/jax-101/02-jitting.html` (Accessed: 2022-03-17)

the first and later function executions; with NumPy, PyTorch and TensorFlow this time difference is much smaller if the function is not explicitly compiled.

In comparison to JAX's JIT compilation, TensorFlow's `tf.function` uses Autograph by default, which, in addition to tracing, analyses Python3 code to convert conditions and loops to compiled code if the conditions depend on user-provided input [Suh+21]. `tf.function` with Autograph calls the function with a special abstract argument and can execute all branches of a condition at once to generate a compiled function which contains those conditions.[3] If one of the branches throws an exception, the compiler executes it and thus fails. JAX's compilation on the other hand does not permit such conditions which depend on the input at all. It is also possible to explicitly disable autograph in `tf.function` with an argument. Furthermore, `tf.function` has the `jit_compile=True` argument, which enables JIT compilation to XLA and thus further speeds up the compiled program. This argument is disabled by default because some TensorFlow operations cannot be compiled with XLA. Analogously to `jax.jit`, we can wrap a function with `tf.function` and it is compiled during its first execution.

The trace-compilation for the PyTorch backend, `torch.jit.trace`, behaves more like Ahead-Of-Time compilation because it executes the function with example inputs to produce a compiled version. `torch.jit.trace` only executes the Python3 code once and remembers the executed numerical operations, which is similar to JAX's compilation, whereas another function, `torch.jit.script`, reads and compiles Python3 source code to TorchScript[4]. The use of `torch.jit.trace` requires more code than the compilations with TensorFlow and JAX. It needs a function which has only positional arguments, which we need to define explicitly in the code, and it needs example arguments, which we can define with additional code. With `jax.jit` and `tf.function` the function is compiled automatically on its first execution and always returns a result whereas `torch.jit.trace` compiles the function explicitly and we need separate code to execute the compiled function. Additional ways to compile code for PyTorch are CUDA Graphs and PyTorch/XLA. CUDA Graphs[5] were recently added in PyTorch 1.10.0 during the time of this thesis and their API is not yet stable. For their application in torchquad, they have too many constraints on the Python3 code: All input tensors must have the `requires_grad` attribute enabled, i.e. the function should be used for backpropagation, and an error showed that autoray uses operations which are not permitted when

[3]Effects of Autograph's tracing process: `https://github.com/tensorflow/tensorflow/blob/170529f4df7aaeebe1c8c2ae6c1256c44becb4fa/tensorflow/python/autograph/g3doc/reference/control_flow.md#effects-of-the-tracing-process` (Accessed: 2022-03-17)

[4]PyTorch's TorchScript documentation: `https://pytorch.org/docs/stable/jit.html` (Accessed: 2022-03-25)

[5]CUDA Graphs introduction: `https://pytorch.org/blog/accelerating-pytorch-with-cuda-graphs/` (Accessed: 2022-03-18)

PyTorch captures a CUDA stream. PyTorch/XLA [Suh+21] enables compilation to XLA for PyTorch and is available in Google Colab to support TPUs. At the time of writing, there is apparently no prebuilt version of PyTorch/XLA which works with Python 3.9 on the Python Package Index (PyPi) or Anaconda, and PyTorch/XLA does not yet support CUDA according to its documentation.

The trace compilations of all backends have the following in common: The Python3 function is executed only once to generate a compiled function and thus side effects such as printing messages or global variable modification are lost when executing the compiled code. Therefore, the to-be-compiled function should be pure. The applications of compilation in torchquad are explained in Section 3.5.

2.3 Hardware Acceleration

There exists a broad range of hardware which is specialised to accelerate numerical calculations which involve linear algebra, and numerical backends often focus on high-performance with a subset of this hardware; for example, we can assume that NumPy targets CPUs whereas PyTorch mainly targets GPUs, and Google's JAX and TensorFlow mainly target both GPUs and TPUs. While it is possible to implement numerical operations with *For* loops provided by the programming language, the language may use the CPU with only a single thread and no vector instructions. We can consider the use of CPU vector instructions, multiple cores and the use of multiple CPUs as hardware acceleration which is implemented in common numerical backends.

Graphics Processing Units (GPUs) are hardware accelerators which are mainly designed for graphics processing tasks. These tasks typically involve SIMD operations, so GPUs have a significantly higher number of concurrently executing threads than CPUs and higher memory bandwidth while each thread is slower than a thread on the CPU and has less cache memory [Mem+17]. In comparison to a CPU, the use of a GPU often requires driver software since most GPU architectures differ between vendors. To implement numerical operations, it is possible to use APIs designed for graphics processing, e.g. Vulkan, APIs for general-purpose processing with common GPUs, e.g. Open Computing Language (OpenCL), and vendor-specific proprietary general-purpose processing APIs, e.g. NVIDIA's Compute Unified Device Architecture (CUDA) [Mem+17]. Due to the GPU architecture differences, achieving high performance is often easier with vendor-specific APIs than APIs which support many devices; for example, a performance comparison between OpenCL and CUDA with near-identical kernels shows smaller execution times with CUDA [KDH11]. torchquad currently focuses on fast quadrature with CUDA but has plans to investigate support for other hardware as well in the future.

In addition to CPUs and GPUs, there exists hardware specifically designed for numerical operations, which are common in the field of machine learning. Tensor Processing Units (TPUs) were developed by Google in 2015 and are employed in datacentres which perform inference for deep learning [Jou+17]. In comparison to GPUs, the main purpose of the TPU is the acceleration of machine learning algorithms to reduce the average energy consumption per numerical operation. According to [Jou+17], which compares the TPU with a CPU and GPU for deep learning inference using TensorFlow in datacentres, the TPU focuses on low latency instead of average throughput, which is different with CPUs and GPUs, and the TPU reached a significant speedup over a CPU and GPU for the selected tasks while requiring less power for the same computations. Field-Programmable Gate Arrays (FPGAs) are another type of hardware which can be used for a concurrent execution of numerical operations. With this hardware, applications related to machine learning often execute faster than on the CPU and require less power than a GPU [NSW18]. Software support for the TPU and FPGAs [NSW18] is enabled by XLA, which is explained in Subsection 2.2.3. Testing on Google Colab has shown that it is possible to use torchquad with a TPU; however, the results chapter of this thesis excludes measurements for a TPU because of difficulties with the parallel installation of PyTorch, JAX and TensorFlow all with TPU support on Google Colab.

2.4 Related Work

This section describes two alternatives to autoray, other work about performance comparisons between backends, and other Python3 modules for numerical integration and VEGAS+. An alternative way to implement Python3 code which works with multiple backends at once is NumPy's `__array_function__` protocol.[6] Similarly to autoray's NumPy mimic, with this protocol NumPy can dispatch a numerical operation to a function of the backend which corresponds to the input arguments. In comparison to autoray, at the time of writing the protocol is still very experimental, it only works with numerical backends which implement this protocol, thus it is not compatible with old versions of backends or backends which are no longer in active development, and it may not target complex functionality such as lazy execution. Another Python3 module for numerical backend modularity is unumpy [Qua21] by Quansight Labs. While it may provide functionality similar to autoray, its documentation mentions a very incomplete

[6] `__array_function__` protocol: `https://numpy.org/neps/nep-0018-array-function-protocol.html` (Accessed: 2022-03-17)

coverage and no support for JAX and TensorFlow.[7]

Most of the work which compares performances between PyTorch, JAX or TensorFlow seems to focus on high-level machine learning functionality and not on the low-level numerical operations which we are interested in; for example, [CS17] investigates the benefit of compilation with XLA in TensorFlow for machine learning algorithms such as Convolutional Neural Networks and [LP19] compares CPU and GPU times for well-known Artificial Neural Networks with TensorFlow 1.13. In [SVK20], Pranav Subramani et al. investigate existing and new Differentially Private Stochastic Gradient Descent (DPSGD) implementations for JAX, TensorFlow and PyTorch, and compare execution times with and without function compilation. Since implementing DPSGD involves low-level numerical calculations, their performance comparisons have similarities to the benchmarking of torchquad in this thesis; similar to our results, compilation with XLA has led to significant performance improvements and they have reached a higher performance with XLA-compiled code for JAX than for TensorFlow 2. They also compare generated XLA assembly code and its optimisation between TensorFlow and JAX.

There are numerous numerical integration modules for Python3. For example, SciPy and quadpy [Sch+21] support well-known multi-dimensional quadrature rules, and sparseSpACE [Obe+21] implements adaptive Sparse Grid quadrature rules, which are well suited to tackle the curse of dimensionality. In comparison to these modules, torchquad focuses mainly on the parallelization of numerical integration on the GPU and on support for full differentiability, which, for example, enables its application in the context of gradient descent optimisation techniques. Furthermore, related modules which implement VEGAS+ integration are VegasFlow [CC20a][CC20b], which supports TensorFlow on the GPU, G. P. Lepage's reference implementation [Lep21], and Yongcheng Wu's CIGAR [Wu20], which is an implementation in C++ and the basis for torchquad's `VEGAS`.

[7]Quansight Labs' unumpy documentation at the time of writing: `https://web.archive.org/web/20220310132356/https://unumpy.uarray.org/en/latest/generated/unumpy.html`

3 Implementation

The main change in torchquad's code is the addition of support for other numerical backends than PyTorch. In addition to the replacement of functions from PyTorch with those of autoray, the code has to fulfil certain requirements so that it works with all backends. After a summary of torchquad's classes and methods, the next sections describe these requirements and the implemented code changes, the abstraction of backend-specific random number generation in torchquad, and numerical operations which torchquad requires but which are not implemented directly by all backends. After that, this chapter explains backend-specific function compilation to increase the performance and properties of the backends which limit the possibilities of code modularisation with autoray.

3.1 torchquad's Code Structure

This section gives an overview about torchquad's implementation of the quadrature rules at the time of writing. Important classes and their methods can be summarized as follows:

- The `Trapezoid`, `Simpson`, `Boole`, `MonteCarlo`, and `VEGAS` classes implement numerical integration rules and have the following methods in common:
  - `.integrate(fn, dim, N, integration_domain, backend="torch", [...])` numerically integrates the `dim`-dimensional integrand `fn` on the given integration domain with the integration rule corresponding to the class. It evaluates the integrand up to `N` times. If the integration domain is a tensor, it infers the backend from it; otherwise, it uses the `backend` string argument. With the non-deterministic `MonteCarlo` and `VEGAS`, the method has additional arguments for random number generation, and for `VEGAS` it has further arguments to configure the adaption to the integrand.
  - `.evaluate_integrand(fn, points)` evaluates the integrand `fn` on the given sample points and counts the number of evaluations.
- `Trapezoid`, `Simpson`, and `Boole` are subclasses of `NewtonCotes` and have the following methods:

  - `.calculate_grid(N, integration_domain)` calculates sample points on a regular grid for the integrand evaluation.
  - `.calculate_result(function_values, dim, n_per_dim, hs)` applies the composite Newton-Cotes rule, which differs between the three integrator subclasses. `function_values` is the integrand output, which is an output of `.evaluate_integrand(fn, points)`, and the other arguments are used to define the grid structure on which the function was evaluated.
  - `.get_jit_compiled_integrate(dim, N, integration_domain, backend)` compiles the previous two methods and assembles a function which performs numerical integration for a given integrand and domain, as explained in Section 3.5.

- `MonteCarlo` is structured similarly to the composite Newton-Cotes integrators and it has the following methods:
  - `.calculate_sample_points(N, integration_domain, seed=None, rng=None)` returns `N` points randomly chosen within the integration domain. `seed` and `rng` are arguments which can influence the random number generation.
  - `.calculate_result(function_values, integration_domain)` returns an integral result for the given integrand output and domain, analogously to the method of the `NewtonCotes` classes.
  - `.get_jit_compiled_integrate([...])` compiles the previous two methods and is analogous to the method with the same name of the `NewtonCotes` classes.

- The `RNG` class implements backend-agnostic random number generation and is explained in Section 3.3.

- `VEGAS` implements the VEGAS+ [Lep21] rule. In comparison to the other integrators, its integration cannot be structured into the execution of three methods where one of them evaluates the integrand, and `VEGAS` does not support JAX and TensorFlow, as explained in Section 3.6. `VEGAS` uses `VEGASMap` and `VEGASStratification` for sample point calculations and adaption to the integrand.

- `VEGASMap` implements the map component of VEGAS+ and has the following methods:
  - `.get_X(y)` moves the points `y` into regions where the absolute value of the integrand is high. This transformation works well as soon as the `VEGASMap` has adapted to the integrand. As explained in Section 2.1, the method applies

a monotonically increasing, piecewise linear mapping on each component of y, where the mapping is different for each dimension.

  - `.get_Jac(y)` returns the Jacobian of `VEGASMap`'s transformation.
  - `.accumulate_weight(y, jf_vec2)` collects information about an integrand output which is needed to update the mapping transformation later. `y` are sample points which were previously passed to `.get_X(y)`, and `jf_vec2` is the squared product of the integrand output and the Jacobian entries from `.get_Jac(y)`.
  - `.update_map()` updates the mapping transformation, which contributes to the adaptation to the integrand function.

- `VEGASStratification` implements the stratification component of VEGAS+ and has the following methods:
  - `.get_NH(nevals_exp)` gets as input an approximate overall number of evaluations `nevals_exp` and returns a tensor which defines the number of points per hypercuboid. As explained in Section 2.1, hypercuboids covering parts of the integration domain where the integrand has a high variance receive more points as soon as the VEGAS+ stratification has adapted to the integrand.
  - `.get_Y(nevals)` returns points which are sampled randomly from the integration domain and fulfil the condition that the number of points that are in each hypercuboid is defined by `nevals`, which is an output of the `.get_NH(nevals_exp)` method.
  - `.accumulate_weight(nevals, weight_all_cubes)` collects information about an integrand output which is needed to update the stratification later. `nevals` is the output of the `.get_NH(nevals_exp)` method and `weight_all_cubes` is the product of the integrand output with the Jacobian entries from the `VEGASMap`.
  - `.update_DH()` recalculates values which `.get_NH(nevals_exp)` uses to determine the number of points per hypercuboid, which contributes to the adaptation to the integrand function.

The actual implementation of torchquad has more details which are hidden here for brevity and can be found in torchquad's documentation. For example, some methods have more default values for their arguments, such as default numbers of evaluations `N`, there are additional helper classes which torchquad uses internally, e.g. `BaseIntegrator` and `IntegrationGrid`, and torchquad has helper functions to configure logging and the floating-point precision. The next section describes general code changes for the implementation of numerical backend modularity in torchquad.

3.2 Common Code Rewriting Steps

The three main steps which have been followed to support other numerical backends than PyTorch with autoray can be summarized as follows. The first step is the replacement of numerical operations which use PyTorch with corresponding operations from autoray's NumPy mimic. This includes the following changes:

- Replace the import of the `torch` module with the import of autoray's NumPy mimic, which is denoted `anp` in torchquad's code.
- Replace functions from the `torch` module with corresponding functions from `anp`
- For numerical operations where autoray cannot infer the backend from input arguments, add the `like` argument to specify the backend explicitly as a string or via a tensor from the same backend. These operations are mostly tensor initialisations.
- Replace torch-specific argument names with the corresponding NumPy argument names; for example, replace `dim` with `axis`

After this step, the code executes the same PyTorch operations as before, but in an indirect way since these operations are wrapped by autoray. This step involves the largest code changes and is similar to a code-style-only change. Since these changes are separate from the other steps, it can be easier to find a bug if a regression happens later.

The next step is the replacement of numerical operations or sequences of operations to support more numerical backends. The changes involve, for example, the following replacements:

- Replace `x.long()` with `astype(x, "int64")` because the `.long()` method is defined for PyTorch's tensor but not the NumPy array type. This method converts the tensor entries to 64-bit integers and `astype` is a backend-agnostic type conversion function which we can import from autoray.
- Replace `.unsqueeze()` tensor method calls with `anp.reshape` function calls. `.unsqueeze()` is another example of a tensor method which is supported by PyTorch but not other backends.
- Replace sequences of numerical operations which use in-place operations. In-place operations are not or only indirectly supported by JAX and TensorFlow, which is

a limitation as explained in Section 3.6. The possibilities for the code replacement are situation-dependent; for example, indexed assignments can sometimes be replaced with `anp.concatenate`, `anp.where` and broadcasting operations.

In comparison to the first step, the code changes are usually smaller, the new code may execute different PyTorch operations than before, and a bug which causes a regression is more likely to be introduced in this step.

The final step is the addition of tests and support for user-configurable floating-point precision. The tests are implemented using the pytest [Kre+21] module and check, for example, if integrators do not crash, the integrals are as accurate as expected, and if the data type of the result integral or individual operations correspond to the user-configured type. To support different floating-point precisions, this step includes the addition of `dtype` arguments to numerical operations. Without these arguments, the numerical integration would work with all backends except that the user would be unable to specify the precision which the NumPy and TensorFlow backend use for the operations, as explained in Section 3.6.

The general code rewriting steps presented in this section are not sufficient to implement all quadrature algorithms. torchquad also requires a backend-independent way for random number generation and special numerical operations to execute the quadrature algorithms, which is explained in the following sections.

3.3 Random Number Generation

The non-deterministic integrators `MonteCarlo` and `VEGAS` require a function to generate random or pseudo-random numbers which are uniformly distributed in $[0, 1)$.

It is possible to distinguish between pseudo-random number generators (PRNGs) with global state and PRNGs with local state. If the state is global, a single function call is sufficient to generate numbers and there is a single global number generator. If the state is local, the user has to initialise the number generator before they can sample numbers from it and it is possible to initialise multiple generators, where the state of each one can be changed individually, e.g. by reseeding them. Random number generation with a global state can be problematic if a user passes an integrand function which changes the PRNG state when evaluated. If, for example, an integrand passed to a `VEGAS` integrator resets the global seed to a fixed value when evaluated and `VEGAS` uses the global PRNG state, `VEGAS` would generate the same sample points in each iteration after the adaptation to the integrand.

While it is possible to use autoray's `anp.random.uniform` to generate backend-specific random numbers, this function maintains a global state, so instead the author has decided to implement a RNG class in torchquad which abstracts away the differences between the numerical backends. This class supports a seed argument so that it is possible to reproduce randomly generated numbers. With a fixed seed, generated numbers are consistent if the numerical backend, its version, and the chosen hardware do not change. If the seed is unset, the backend-specific PRNG is initialised to a non-deterministic state. The following parts in this section explain the differences between random number generation in the supported backends.

To generate numbers with NumPy, we can initialise a PRNG object with `rng = numpy.random.default_rng(seed)` once and then sample pseudo-random numbers from it with `rng.random(size, dtype)`. The `seed` argument determines the generator's initial state. If it is `None`, the generator is initialised with a truly random state; otherwise, the initial state is configured with the provided seed. To generate random bits, NumPy uses a Permuted Congruential Generator[1] by default at the time of writing. An alternative way for random number generation with NumPy is to call `np.random.seed(seed)` to configure a seed and `numpy.random.random(size)` to generate numbers; however, these functions are deprecated, use a global state and always use the Mersenne Twister[2]. In comparison to other numerical backends, NumPy offers an additional function to generate uniform random numbers in $[0, 1]$ instead of $[0, 1)$; we do not use it so that the behaviour between backends is consistent.

The PRNG of PyTorch maintains a global state for the currently selected device. The state can be seeded with `torch.random.manual_seed(seed)` and set to a non-deterministic value with `torch.random.seed()`, and random numbers can be sampled with `torch.rand(size, dtype)`. Furthermore, there are functions to get and set the global state. By default torchquad's RNG does not use these functions because they can slow down the random number generation slightly, PyTorch does not discourage the use of a global state, which is different with other numerical backends, and we consider integrands which change a PRNG seed to be uncommon. Nonetheless, users can explicitly enable state saving and restoring if needed.

With TensorFlow, we can initialise a randomly seeded PRNG object with `rng = tf.random.Generator.from_non_deterministic_state()`, and use `rng = tf.random.Generator.from_seed(seed)` if the user provides a seed, and we can generate random numbers with `rng.uniform(shape, dtype)`. Similar to NumPy, TensorFlow additionally has a PRNG with a global state whose use is deprecated.

With JAX the state for number generation is directly exposed to us and can be initialized with `PRNGKey(seed)`. For a non-deterministic initial state, we have to provide a truly-random seed value. To generate random numbers, we need to split

[1]NumPy's Permuted Congruential Generator documentation: `https://numpy.org/doc/stable/reference/random/bit_generators/pcg64.html` (Accessed: 2022-03-16)

[2]NumPy's Mersenne Twister documentation: `https://numpy.org/doc/stable/reference/random/bit_generators/mt19937.html` (Accessed: 2022-03-16)

the `PRNGKey` object with `jax.random.split` into two new `PRNGKey` objects, remember one of them for later number generation and pass the other one, denoted `key`, to `jax.random.uniform(key, shape, dtype)`. In comparison to PyTorch and TensorFlow, when JIT-compiling a function which generates random numbers, the old and new states have to be an input and output so that JAX generates different random numbers in each invocation of the compiled function.

3.4 Special Numerical Operations

In addition to random number generation, torchquad requires numerical operations which have short mathematical definitions but are not directly implemented by all backends. This section describes two of them: the Cartesian product of one-dimensional tensors, which we need to calculate grid points for Newton-Cotes integrators, and an operation that we denote addition at indices, which enables a fast VEGAS+ implementation.

3.4.1 Cartesian Tensor Product

In comparison to the Cartesian product of sets, the output of this Cartesian tensor product operation must be ordered for an application of a Newton-Cotes rule in later calculations. In torchquad, the operation is implemented with `anp.meshgrid`, the `.ravel()` method of the backend-specific tensor, and `anp.stack`:

1. The `anp.meshgrid` function gets as input d one-dimensional tensors, repeats each input tensor along $d-1$ dimensions, and returns the d repeated tensors as a list. Each input tensor is repeated along different dimensions. With this function, torchquad repeats tensor entries of the Cartesian product operands.
2. The `.ravel()` method flattens a tensor to a one-dimensional one with the same number of elements.
3. `anp.stack` concatenates a list of tensors along a specified dimension. torchquad applies it on flattened tensors from the `anp.meshgrid` output to generate the Cartesian product result.

These functions are documented precisely at NumPy's API reference[3]. At the time of writing, the indexing order of the `anp.meshgrid` operation differs between backends. This order defines which dimensions are chosen to repeat each input tensor. We can

[3]NumPy Routines documentation: `https://numpy.org/doc/stable/reference/routines.html` (Accessed: 2022-03-16)

ignore the differences because the composite Newton-Cotes rules produce the same result with all orders up to floating-point inaccuracies.

3.4.2 Addition at Indices

Another special numerical operation is addition at indices. Given the indices $p_0, p_1, \ldots, p_{n-1} \in \mathbb{Z}_k^n$ and numbers $x_0, x_1, \ldots, x_{n-1}$, we want to calculate $y_i = \sum_{\{j \in \mathbb{Z}_n | p_j = i\}} x_j$ efficiently for all $i \in \mathbb{Z}_k$. This operation appears in many places in the VEGAS+ integration. For example, for each dimension torchquad's VEGAS map weight accumulation, which is implemented in `VEGASMap.accumulate_weight`, gets as input n numbers $\vec{x}$ and n corresponding point locations, maps the points to interval indices $\vec{p}$, and then adds each x_j to a value y_i, where $i = p_i$ is the interval index.

PyTorch implements this numerical operation with the `torch.scatter_add_` function, but NumPy does not have an equivalent function to the author's knowledge. Therefore, torchquad has a replacement for this function using multiple indicator matrices. There are two solutions for this replacement which require few lines of code:

1. The first solution uses a Python3 `for` loop which iterates over all integers $j \in \mathbb{Z}_n$ and adds x_j to y_{p_j}.
2. The second solution uses an indicator matrix and is implemented in VegasFlow.[4] It follows these steps:
    a) Create an integer tensor with all indices in $\mathbb{Z}_k$ using `anp.arange`
    b) Reshape this tensor to the shape $(k, 1)$
    c) Compare the reshaped tensor with p using `anp.equal`, which broadcasts both of its input tensors and therefore creates a large indicator matrix M. In this matrix $m_{i,j}$ is true if and only if x_j should be added to y_i according to p_i.
    d) Execute `anp.where` to replace all $m_{i,j}$ by

$$\hat{m}_{i,j} = \begin{cases} x_j, & m_{i,j} \text{ is } \texttt{True} \\ 0.0, & \text{otherwise} \end{cases} \tag{3.1}$$

    e) Sum up the replaced values with `anp.sum` along the second matrix dimension, which outputs $\vec{y}$.

[4]VegasFlow's `consume_array_into_indices`: `https://github.com/N3PDF/vegasflow/blob/21209c928d07c00ae4f789d03b83e518621f174a/src/vegasflow/utils.py#L16` (Accessed: 2022-03-16)

Both of these solutions have performance limitations. The disadvantage of the first solution is the overhead of the Python3 interpreter and the high number of single-element array index accesses. The second solution uses vectorized operations, but it scales poorly because the indicator matrix has $n \cdot k$ entries, thus we need $\Theta(nk)$ memory and time in comparison to the $\Theta(n)$ time in the first solution. torchquad implements a compromise between the two solutions, which can be summarized as follows:

1. Sort $\vec{x}$ and $\vec{p}$ along $\vec{p}$ unless they are already sorted
2. Partition the sorted tensors with a strided Python3 `for` loop. In each iteration we consider the indices $j_1, j_1 + 1, \ldots, j_2$ into the sorted $\vec{x}$ and $\vec{p}$.
3. In the loop body, use the previously explained second solution with an indicator matrix and add the result to $\vec{y}$ at the corresponding indices. The indicator matrix has $j_2 - j_1 + 1$ columns and, since the elements in $\vec{p}$ are sorted, it requires only $p_{j_2} - p_{j_1} + 1$ instead of k rows.

This approach has the performance advantage of vectorized operations while the indicator matrices are sufficiently small even for large input tensors. In the current implementation, the indicator matrices have 500 columns for all except the last `for` loop iteration, and it may be possible to replace the `for` loop with recursive subdivisions to reduce the sum of the number of indicator matrix entries, which may further increase the performance with NumPy. torchquad currently partitions with the `for` loop since code with recursive subdivisions would probably be more difficult to understand and fast integration with NumPy is not one of torchquad's main goals.

The addition at indices numerical operation is only needed for NumPy and PyTorch at the time of writing because torchquad's `VEGAS` does not support JAX and TensorFlow. Together with `anp.repeat`, the operation enables a fast vectorized VEGAS+ implementation.

3.5 Compiled Functions in torchquad

Depending on the application, torchquad's numerical integration may be executed repeatedly on the same domain, integration grid or with the same integrand. To increase the performance in this situation, it is possible to compile functions as explained in Subsection 2.2.5. torchquad's composite Newton-Cotes and Monte Carlo integrators execute three main steps: the calculation of sample points, the integrand evaluation on these points, and the integral result calculation given the integrand output. On the one hand, it is possible to compile the whole integration, which includes all three steps, by applying a backend-specific compilation function such as `jax.jit`. The user has to apply this compilation themselves because it requires comparatively few code changes, it is possible to include more code in addition to numerical integration in a to-be-compiled function, and the integrand function may require certain settings for the compilation, such as disabling XLA with TensorFlow. On the other hand, torchquad offers a method which compiles only the first and third steps. With this method, we can obtain a performance benefit even with integrands which are not compilable. It uses `tf.function` with the `jit_compile=True` argument for TensorFlow, `jax.jit` for JAX, and `torch.jit.trace` for PyTorch. torchquad does not use `torch.jit.script` because it fails with undefined attribute lookup errors when using autoray, probably because TorchScript only works with a subset of the Python3 language. Compilation for the VEGAS+ integration is not yet implemented in torchquad since it would require extensive code changes as explained in the next section.

The benefits of compilation are investigated in Chapter 4. A disadvantage of function compilation is the additional time required to compile or recompile the code, so if we execute numerical integration only a few times or certain arguments, e.g. the number of points, change frequently, the program may be slower overall.

3.6 Limitations

Some differences between numerical backends make it difficult to change Python3 code so that it works fast with all backends while the code is still easy to read and understand. One of these differences is the support for in-place operations. With NumPy and PyTorch, it is common to index a tensor on the left side of an equation, for example with a slice, to set selected parts of the tensor to new values or to reduce the memory usage in certain situations. On the other hand, with JAX an in-place tensor change requires a special syntax and it is executed out-of-place unless the code is compiled,[5] and TensorFlow supports in-place changes only with the `tf.Variable` tensor type while `tf.Tensor`, which torchquad commonly uses, does not support in-place changes at all. In some situations, it is possible to rewrite the code with `anp.concatenate` and `anp.where` without significantly deteriorating the code quality or performance. Another difference between backends is the performance of uncompiled code, which is investigated in Chapter 4.

torchquad's VEGAS+ implementation uses many in-place tensor operations. Furthermore, the code for `VEGAS` is written in an object-oriented way and the numbers of sample points, thus tensor sizes, vary between iterations, which hinders trace compilation of functions because methods adjust object members and are thus not

[5]In-place operations in JAX: `https://jax.readthedocs.io/en/latest/_autosummary/jax.numpy.ndarray.at.html` (Accessed: 2022-03-17)

pure, and a variation in tensor sizes leads to time-consuming recompilations. An experimental rewritten code for the VEGAS+ stratification implementation to support uncompiled JAX and TensorFlow was very slow compared to NumPy and PyTorch. The author has decided to not add support for JAX and TensorFlow to the VEGAS+ implementation because function compilation would probably require extensive code changes to group performance-relevant code not related to integrand evaluation into pure functions, replacing in-place operations may lead to performance degradation with PyTorch and NumPy or to code which is difficult to understand, and for TensorFlow, there already exists the performance-optimized VEGAS+ implementation VegasFlow.

Furthermore, backends differ in their setup and their data type configuration. TensorFlow and JAX use CUDA by default, whereas for PyTorch it needs to be enabled explicitly, and for TensorFlow, we need to enable its NumPy behaviour[6]. The backends support different ways to set the floating-point precision to 32 or 64 bit. With NumPy and TensorFlow we cannot set a global default precision but we can pass a `dtype` argument to functions which initialise a tensor, with JAX the behaviour is the other way round by default, i.e. it is possible to change the precision globally and a `dtype` argument is ignored unless JAX is configured with `"jax_enable_x64"`, and PyTorch supports both a global default precision and use of the `dtype` argument. There also are differences in automatic type conversions; for example, when NumPy divides an array of empty shape and `float32` data type by a Python3 `float`, it outputs an array of empty shape and `float64` data type, while other backends and the division of arrays with non-empty shape preserve the 32-bit precision. torchquad uses pytest tests to validate if its code uses the expected data types.

[6]TensorFlow's NumPy behaviour: `https://www.tensorflow.org/guide/tf_numpy` (Accessed: 2022-03-17)

4 Results

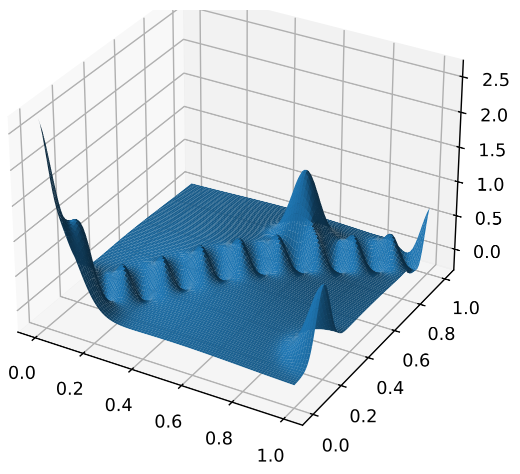
This chapter investigates the performance of torchquad's numerical integration and its bottlenecks with different backends. All measurements are collected on the same computer, which has the Ubuntu 20.04 operating system, a 24-core AMD EPYC 7402 CPU, 251.65 GiB memory, and four NVIDIA GeForce RTX 3080 GPUs, from which all except one are blacklisted with the `CUDA_VISIBLE_DEVICES` environment variable. The Python3 interpreter and its modules were installed in an Anaconda [Ana21] environment. The versions of the interpreter and modules are Python 3.9.7, the numerical backend versions listed in Subsection 2.2.4, and autoray 0.2.5. The next sections explain integrand functions commonly used for the measurements, benchmarking results for time comparisons, code profiling to investigate bottlenecks and reasons for time differences between backends, and a validation of torchquad's VEGAS+ accuracy.

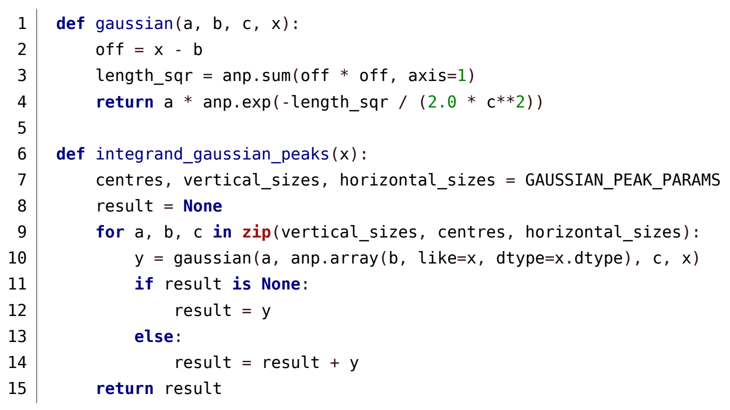
4.1 Example Integrands

To compare the execution times and memory requirements of a quadrature algorithm between numerical backends, the integrand should be fast to evaluate but not too simple. It should be fast so that we can compare the times needed to calculate the sample points, and the final integral value from the integrand's output while ignoring the time required to evaluate the integrand function. However, the integrand should not be too simple because a compilation of the whole numerical integration or lazy execution should not optimize away interesting parts of the quadrature algorithm. For these reasons, the following `sin_prod` integrand is used for most benchmarking and profiling measurements:

$$\text{sin\_prod}(\vec{x}) = \prod_{i=1}^{dim} \sin(x_i) \tag{4.1}$$

Another example integrand, which is denoted `gaussian_peaks`, consists of the sum of 25 Gaussian functions and is therefore more expensive to evaluate. It is visualized in Figure 4.1. This integrand uses vector operations which are common to all supported backends, so we can use it to roughly compare how well different backends scale on the GPU and what impact their JIT compilations have on the performance.





```python
def gaussian(a, b, c, x):
    off = x - b
    length_sqr = anp.sum(off * off, axis=1)
    return a * anp.exp(-length_sqr / (2.0 * c**2))

def integrand_gaussian_peaks(x):
    centres, vertical_sizes, horizontal_sizes = GAUSSIAN_PEAK_PARAMS
    result = None
    for a, b, c in zip(vertical_sizes, centres, horizontal_sizes):
        y = gaussian(a, anp.array(b, like=x, dtype=x.dtype), c, x)
        if result is None:
            result = y
        else:
            result = result + y
    return result
```

Figure 4.1: The 2D integrand `gaussian_peaks`.
Left: Integrand plot.
Right: Implementation of the integrand. `GAUSSIAN_PEAK_PARAMS` contains parameters for 25 Gaussian functions. The code image was generated with Geany.

Furthermore, the `vegas_peak` integrand, which is defined and explained later, has been implemented for the VEGAS+ accuracy comparison in Section 4.4.

4.2 Benchmarking

For a comparison of required quadrature times between different numerical backends, benchmarking scripts have been implemented to measure median times with different numbers of sample points. For PyTorch, JAX and TensorFlow the scripts have three cases for the compilation of the quadrature code:

- `uncompiled`: Execute code in eager mode
- `parts compiled`: Separately compile the three functions for sample point calculation, integrand evaluation, and integral calculation from the integrand output
- `all compiled`: Compile the whole to-be-measured code at once

While compiled functions execute faster, a long time may be required for compilation or recompilation; in the benchmarking scripts this time belongs to the warm-ups, so it is excluded from the measurement results. It is possible to force a CUDA synchronisation before and after the integrand evaluation as explained later in the profiling section; testing has shown that the results for the `parts compiled` case with `sin_prod` integrand look very similar with and without synchronisation, which indicates that this example integrand is complicated enough for the benchmarking measurements. More information about function compilation can be found in Subsection 2.2.5 and Section 3.5. Since torchquad's composite Trapezoid and Simpson implementations, and benchmarking results for them are very similar to `Boole`, the next sections exclude measurements for these integrators. Furthermore, the `VEGAS` integrator is excluded because it works only with NumPy and PyTorch, and its performance depends on how it adapts to the integrand and user-configurable arguments for this adaptation.

The next section explains how the script measures the median time and the following sections describe measurement results for example integrations, parts of the quadrature computations, different floating-point precisions, and gradient calculations.

4.2.1 Time Measurement

To measure the median time required to execute a function, the benchmarking script uses `torch.utils.benchmark` for PyTorch, and for all other backends it executes warm-ups and then `timeit.Timer`. The script contains code to ensure that an asynchronous computation must be finished before measuring the time. It converts the calculated integral result to a NumPy array when comparing integration times; this enforces a synchronisation with all backends. When comparing only the sample point calculation time, it uses backend-specific synchronisation. With JAX it executes the `.block_until_ready()` method on tensors and with PyTorch the `torch.utils.benchmark` function executes `torch.cuda.synchronize()` to finish asynchronous computations. For TensorFlow, the author could not find a function for an explicit synchronization; nonetheless, the sample point calculation time measurements coincide with the integration time measurements. NumPy does not use the GPU and a synchronisation is not needed. The benchmarking script is configured to successively increase the number of evaluations and to abort as soon as the median time is above a chosen threshold or an out of memory exception happens.

4.2.2 Example Quadrature Time Comparisons

Figure 4.2 shows quadrature time comparisons for the composite Boole and Monte Carlo implementations. For the Boole integrator, it depicts measurement results for a 1D and 3D version of the `sin_prod` integrand and the 2D `gaussian_peaks` integrand. For `MonteCarlo`, it depicts results for a high-dimensional version of the `sin_prod` integrand.

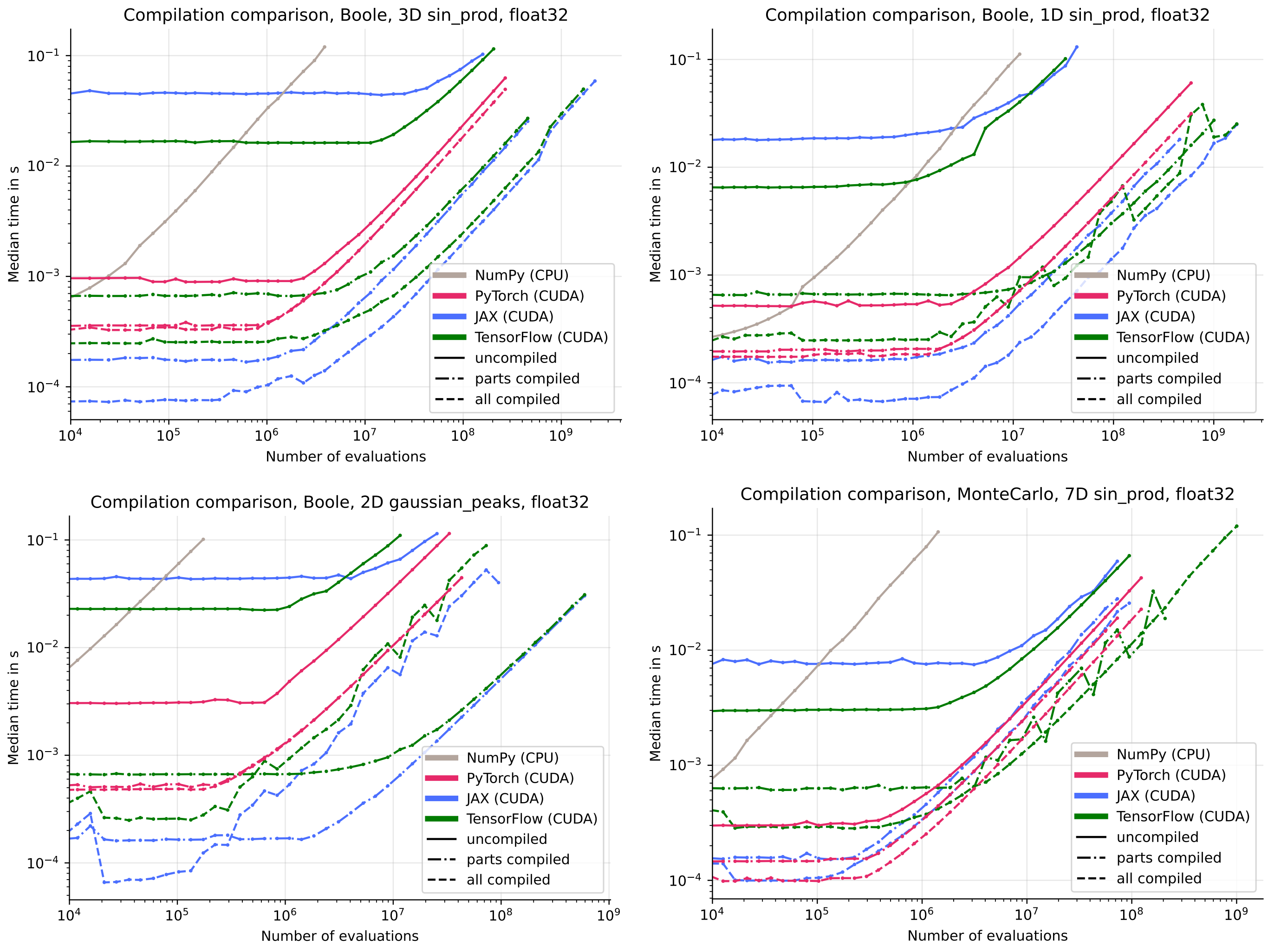


Figure 4.2: Time comparisons between backends and compilation modes.
Top left: Boole quadrature, 3D `sin_prod` integrand.
Top right: Boole quadrature, 1D `sin_prod` integrand.
Bottom left: Boole quadrature, 2D `gaussian_peaks` integrand.
Bottom right: Monte Carlo quadrature, 7D `sin_prod` integrand

The plots do not look smooth, especially for compiled JAX and TensorFlow. The reasons for this could be the optimisation decisions made by XLA for different numbers of evaluations, and different caching behaviour of the GPU depending on how much memory is needed. Running the benchmarking scripts multiple times always leads to the same small spikes in the plot. For all four plots, we can make the following common observations:

- PyTorch, JAX and TensorFlow all parallelize well on CUDA except in the 1D uncompiled JAX and TensorFlow cases. The times start to rise after approximately a million sample points. The reason why 1D uncompiled JAX and TensorFlow are comparatively slow is explained in Subsection 4.3.4.
- Uncompiled JAX and TensorFlow are the slowest cases where the GPU is utilized. We can consider JAX's eager mode to be especially slow since it is slower than TensorFlow for a small number of points. Furthermore, in comparison to eager execution with the other backends, JAX's eager mode internally compiles individual operations during the warm-up, which is not visible in the plots.
- The benefit of compilation often decreases for JAX, TensorFlow and PyTorch in this order.
- For a sufficiently small number of points, the `all compiled` JAX case is often the fastest.
- The benefit of compiling all at once instead of three parts separately is significantly lower than the benefit of compiling parts instead of no compiling. This observation depends on the integrand complexity and optimisation decisions; the profiling section shows the performance impact between the three compilation configurations.

The `gaussian_peaks` integrand is expensive to evaluate, so we may use it to generally compare the performance of numerical backends. We may conclude that trace-compiling of expensive functions improves the performance in general and that compiling small parts of a program separately works as well as compiling the whole program at once with PyTorch, while with JAX and TensorFlow it makes a significant difference if we compile all at once or the parts separately, perhaps because these two backends are slower in eager mode than PyTorch. In comparison to the other plots, the plot with the `gaussian_peaks` integrand in Figure 4.2 shows that TensorFlow and JAX are faster in the `parts compiled` case than in the `all compiled` case. The benchmarking script trace-compiles the whole `integrate` method of the Boole class, which includes the sample point calculation and integrand evaluation, in the `all compiled` case while it compiles the integrand evaluation separately in the `parts compiled` case, so the time difference could be due to XLA making suboptimal optimisation decisions.

The performance differences with the Monte Carlo integrator may be caused by the pseudo-random number generator implementations. Nonetheless, the Monte Carlo implementation parallelizes on the GPU as well as Boole.

Table 4.1 shows concrete time values from the 3D `sin_prod` integrand plot for a high number of evaluations. With eager execution, PyTorch is faster than TensorFlow and JAX by a factor of ca. 2.6 and 3.4, and in the `all compiled` case TensorFlow and JAX are faster than PyTorch by a factor of ca. 7.4 and 9.0. Other integrand functions, dimensionality and integration rules lead to different factors as can be seen in the plot for `gaussian_peaks`, for example. A comparison of concrete values for these cases is difficult since the plot has small peaks with JAX and TensorFlow.

	`uncompiled`	`parts compiled`	`all compiled`
PyTorch	**21.9**	-	17.1
JAX	74.8	**5.3**	**1.9**
TensorFlow	58.0	5.9	2.3

Table 4.1: Median times in ms for the 3D `sin_prod` integrand from Figure 4.2 for $92959677 \approx 93 \cdot 10^6$ sample points. The smallest times are highlighted.

4.2.3 Time Comparisons of Quadrature Steps

torchquad's Newton-Cotes integrators execute three steps to perform the quadrature: grid point calculation (step1), integrand evaluation, and application of the composite Newton-Cotes rule (step3). The first two steps are the same for Trapezoid, Simpson and Boole. The first step calculates grid points, so its performance may be mostly impacted by memory allocation and movement times, and the third step applies a quadrature rule, which involves a high number of additions and multiplications, and thus may involve more arithmetic calculations than memory manipulations. This section shows benchmarking results for the individual execution of the first and last step. Measuring these instead of a whole quadrature as in the previous section has the advantage that we exclude the time to evaluate the user-provided integrand and the disadvantage that enforcing CUDA synchronisation before and after the calculations is more difficult to implement reliably. Figure 4.3 and Figure 4.4. depict the measurement results for step1 and step3, respectively. We can make the following observations for these measurements:

- JAX and TensorFlow are slow with eager execution.

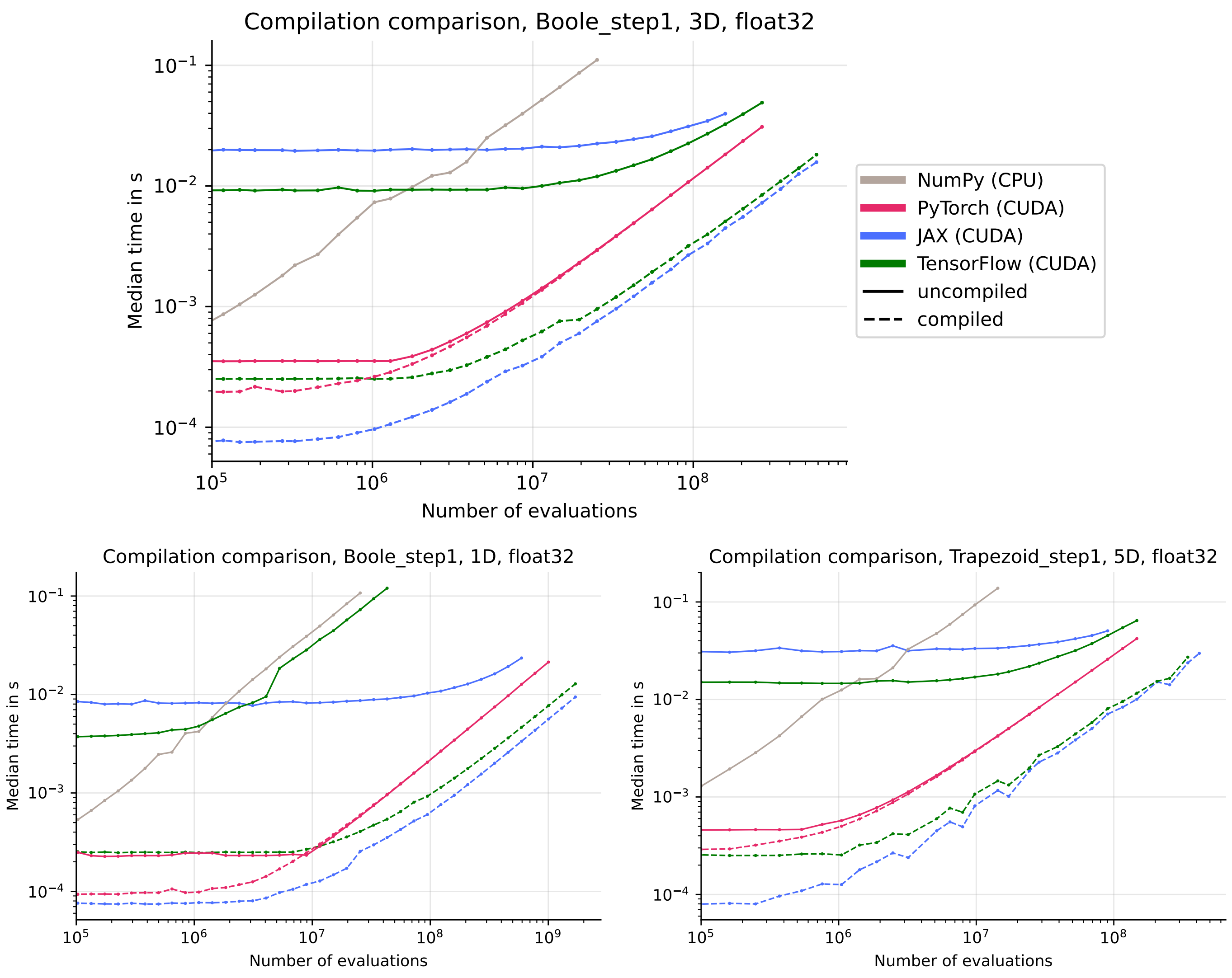


Figure 4.3: Runtime comparisons of `calculate_points`, the first step of Newton-Cotes integrators, for different dimensions. The 5D case uses the composite Trapezoid instead of Boole rule since it has fewer restrictions on the valid number of evaluations; nonetheless, the step1 calculations are the same.

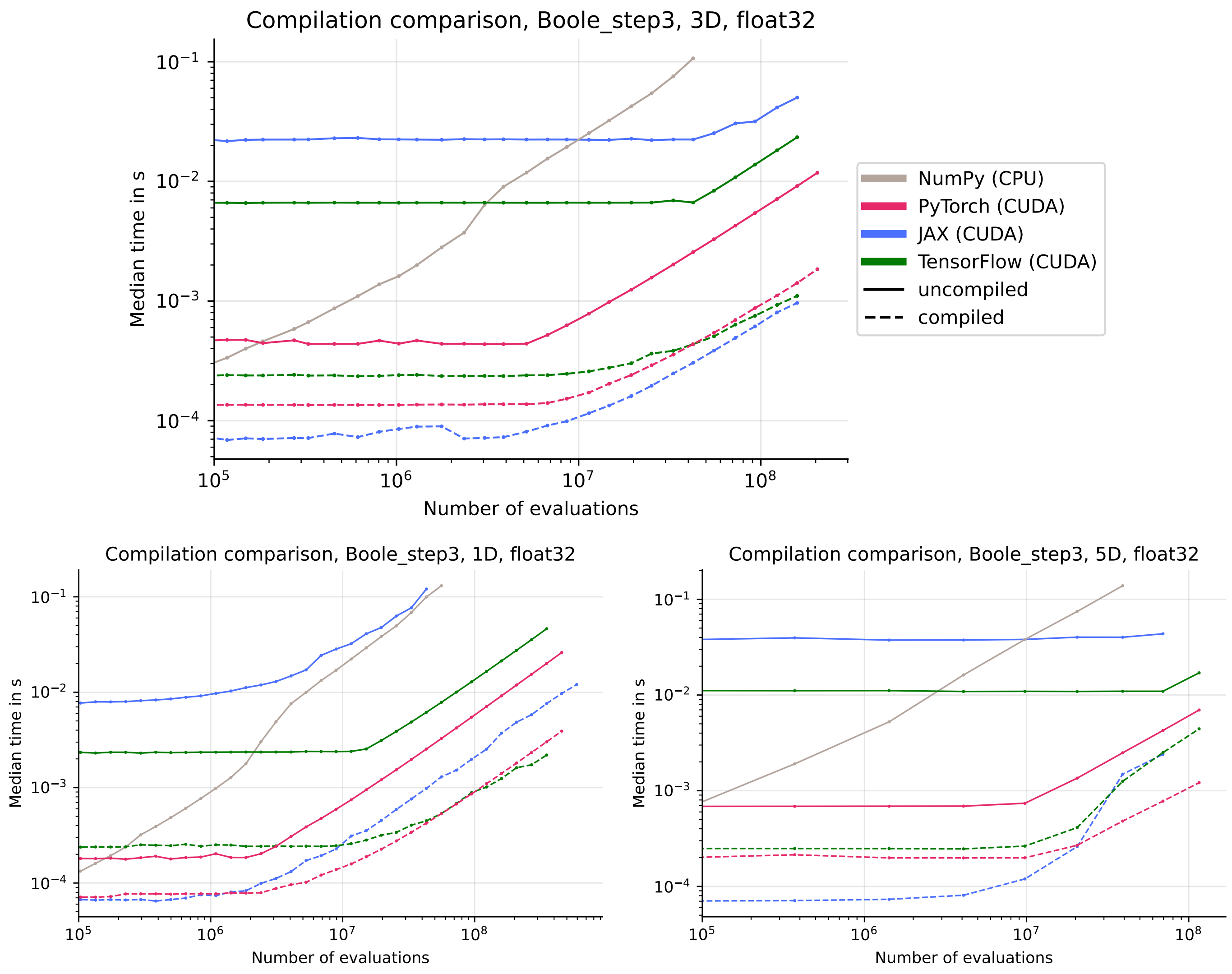


Figure 4.4: Runtime comparisons of `calculate_result`, the third step of Boole, for different dimensions

- For a low number of points, compiled JAX and TensorFlow have the same overhead for all dimensions whereas PyTorch's overhead grows with the dimension.
- Uncompiled TensorFlow in the step1 1D case rises similarly to NumPy, which indicates that CUDA is not fully employed.
- Uncompiled JAX in the step3 1D case rises similarly to NumPy.
- In the step1 case with many points, PyTorch has no performance gain from compilation and is slower than compiled JAX and TensorFlow. Furthermore, compiled JAX is slightly faster than compiled TensorFlow.
- In the step3 case with many points, PyTorch benefits from compilation and depending on the dimensionality different backends are the fastest.
- In the step3 case with few points, PyTorch is often faster than TensorFlow.

4.2.4 Impact of Floating-Point Precisions

With different floating-point precisions, plots over the number of evaluations have shown that for a sufficiently small number of points the precision has a negligible overhead on the performance and for a high number of points the differences between precisions converge to constant factors, which was visible as nearly-parallel lines in the plots. Therefore a time comparison for a fixed high number of evaluations, which is shown in Table 4.2, suffices to compare performances. From these values, we can infer that JAX and TensorFlow have similar times with 16-bit and 32-bit numbers, especially if code is compiled, and the slight performance benefit with 16-bit numbers may be caused by smaller memory usage. On the other hand, PyTorch has a significant speedup with 16-bit numbers. Furthermore, with 64-bit numbers all backends show larger times; for example, in the `all compiled` JAX case, calculations with 64-bit precision are more than 16 times slower. NumPy uses the CPU instead of CUDA for calculations and therefore it does not have hardware support for 16-bit numbers. In comparison to 32-bit numbers, times with 16-bit and 64-bit numbers are approximately 2.9 and 2.0 times larger with NumPy.

	`uncompiled`	`parts compiled`	`all compiled`
`float16`, JAX	44.1	**1.7**	**0.9**
`float16`, TensorFlow	26.7	2.6	1.2
`float16`, PyTorch	**6.0**	4.7	4.7
`float32`, JAX	50.7	**2.4**	**0.9**
`float32`, TensorFlow	31.8	2.9	1.2
`float32`, PyTorch	**10.2**	7.9	7.9
`float64`, JAX	78.0	23.5	15.0
`float64`, TensorFlow	74.7	**18.9**	**13.3**
`float64`, PyTorch	**27.1**	22.6	22.6

	`float16`	`float32`	`float64`
NumPy (fewer points)	118.0	40.8	80.6

Table 4.2: Median times in ms for different floating-point precisions with the 3D `sin_prod` integrand and Boole quadrature.
Top: $42508549 \approx 42.5 \cdot 10^6$ sample points with backends which use CUDA; the times for the fastest backends for a given precision are highlighted.
Bottom: $1295029 \approx 1.3 \cdot 10^6$ sample points with NumPy

4.2.5 Time Comparisons with Gradient Calculation

Figure 4.5 shows time comparisons for quadrature with and without computations of the gradient over the integration domain for the 3D `sin_prod` integrand with the composite Boole rule. The benchmarking setup is as follows:

- For the `parts compiled` cases, the benchmarking script compiles only the sample point calculation, integrand evaluation and calculation of the integral from function values, while the gradient calculation is not compiled.
- For the `parts compiled` and the `uncompiled` case, the backend-specific gradient calculation is implemented as follows:
  - With PyTorch, the gradient is calculated with the `.backward()` method applied on the integral result in each measurement.
  - With TensorFlow, the gradient is calculated with the `tf.GradientTape()` context manager in each measurement.
  - With JAX, a gradient function is calculated once with `jax.grad(func)` in the first warm-up and then this returned gradient function is executed in each measurement.
- The gradient calculation in the `all compiled` case is different:

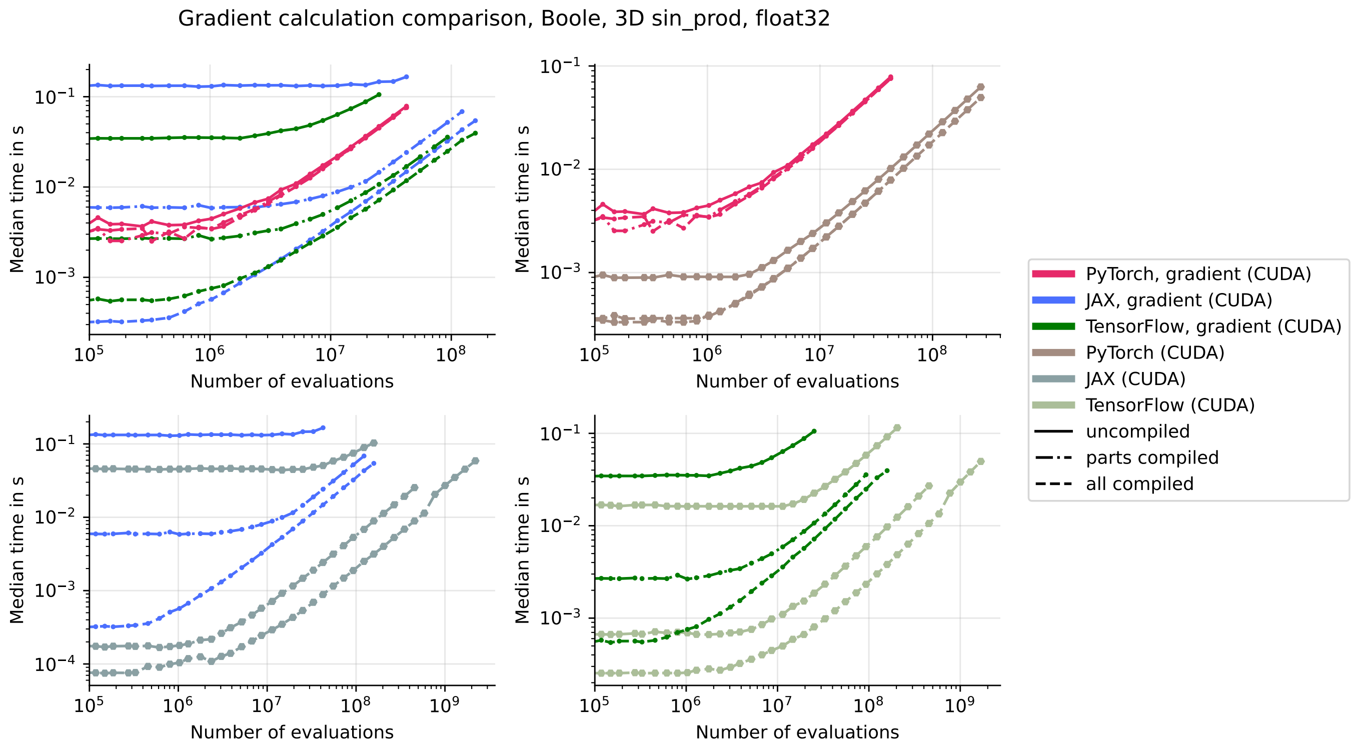


Figure 4.5: Runtime comparisons between backends with and without gradient calculation over the integration domain. The top left subplot compares times between backends when calculating gradients and the other subplots compare times with and without gradient calculation separately for each backend.

- With JAX, the returned gradient function is compiled.
- With TensorFlow, the context manager is included in a function which is compiled.
- With PyTorch, the code for numerical integration is compiled and the `.backward()` call is excluded from trace compilation because compiling it does not work.

The backend-specific benchmarking setup differences are probably the reason why the JAX and TensorFlow times are significantly larger in the parts compiled than in the all compiled case for small numbers of sample points. Ignoring these differences, the gradient calculation times have many similarities to the integral calculation times shown in Figure 4.2; for example, for sufficiently small numbers of evaluations, the slowest and fastest cases are uncompiled and fully compiled JAX and the second slowest and second fastest cases are uncompiled and fully compiled TensorFlow, for a high number of evaluations compiled JAX and TensorFlow overtake PyTorch, and all three backends parallelize well on the GPU. We can also see a difference when comparing the plots: for a high number of evaluations, TensorFlow is slightly faster than JAX when calculating gradients with the `sin_prod` integrand. Table 4.3 contains concrete measurement values for a high number of sample points. For the given setup with gradient calculation, in eager mode execution, PyTorch is faster than JAX and TensorFlow by a factor of ca. 3.1 and 2.3, whereas in the `all compiled` case, JAX and TensorFlow are faster than PyTorch by a factor of ca. 5.1 and 6.3.

	`uncompiled`	`parts compiled`	`all compiled`
PyTorch	**6.2**	4.7	4.7
JAX	45.1	**1.5**	**0.5**
TensorFlow	22.5	1.9	0.8
PyTorch (grad)	**46.9**	45.0	45.0
JAX (grad)	146.5	14.5	8.9
TensorFlow (grad)	105.6	**10.7**	**7.2**

Table 4.3: Median times in ms for the 3D `sin_prod` integrand from Figure 4.5 for $25153757 \approx 25 \cdot 10^6$ sample points. The lowest times for each backend with and without gradient calculation are highlighted.

With the `gaussian_peaks` integrand, compiled TensorFlow and JAX are approximately equally fast when calculating gradients and the `parts compiled` case is slower than the `all compiled` case, which was different when not calculating gradients in Figure 4.2. Furthermore, with this integrand, the gradient calculation overhead is much lower, probably because the integrand is expensive to evaluate. Otherwise, the performance differences between backends are similar to the differences with the `sin_prod` integrand. This section does not show concrete values or plots for the `gaussian_peaks` integrand because of the similarity with the measurement results for the `sin_prod` integrand.

4.3 Profiling

In addition to benchmarking, we can investigate how much time torchquad spends on individual numerical operations and the memory requirements for computations. The next sections first explain software tools to perform such investigations, and the common configuration and setup for profiling measurements, and then describe measurement results which show differences in execution and memory traces between backends.

4.3.1 Profiling Tools

Available profiling tools can be classified into backend-specific tools, which are part of the numerical backend modules, and general tools, which work for any Python3 code. PyTorch, JAX and TensorFlow have backend-specific profilers with context managers in Python3 which can trace selected code and save the collected data for a visualisation in TensorBoard[1]. The context managers are `torch.profiler.profile()`, `jax.profiler.trace()` and `tf.profiler.experimental.Profile()` for the respective backends. The interactive TensorBoard trace visualisation shows when and how long functions are executed in the context manager's scope over time. It additionally shows at which time the GPU is busy with CUDA operations and where these operations are executed asynchronously. Figure 4.6 and the following figures show cropped screenshots of TensorBoard trace visualisations. TensorBoard can also plot the memory usage over time, which is shown in Figure 4.10, for example. Since torchquad targets fast quadrature on the GPU, code is not profiled with the NumPy backend. In the next sections, these context managers are employed to investigate time and memory usage; for completeness, the rest of this section describes other tools which are not used for profiling in this thesis and the reasons for not choosing them.

JAX offers a function to save the current device memory profile to a file which can be decoded by Google's pprof tool.[2] This function can be helpful to detect memory

[1]TensorFlow's TensorBoard profiling guide: `https://www.tensorflow.org/guide/profiler?hl=en` (Accessed: 2022-03-24)

[2]JAX's Device Memory Profiling documentation: `https://jax.readthedocs.io/en/latest/device_memory_profiling.html` (Accessed: 2022-03-18)

leaks or to investigate which Python3 functions initialise large tensors, but it does not measure the memory usage over time or the peak memory usage, so it is not suited for our purposes. For example, executing the function after calculating the integral provides no useful information because deallocated tensors, e.g. the sample points, are not part of the profile.

cProfile[3] is a general profiling tool. It is part of the Python3 implementation and enables measuring how often and how long each function in Python3 code executes. It is possible to wrap the to-be-profiled code in a context manager, print summary statistics and visualise collected profiling information in a web browser with SnakeViz [Dav+21]. In comparison to the backend-specific profiling tools, cProfile does not generate a trace and it does not profile CUDA functions and other low-level operations. Furthermore, if code is fully compiled or executed asynchronously, the performance of Python3 functions does not represent the performance of parts of the quadrature algorithm.

Another general profiling tool is Memory Profiler [PG+20], which is developed by Fabian Pedregosa and others. It adds the `mprof` executable, with which it is possible to measure how much memory a program requires over time. To do this, it executes the program and at the same time it queries the operating system for the required host memory per thread in small fixed time intervals. The collected data can then be visualised in a plot. Another feature of Memory Profiler is line-by-line profiling of the memory usage in Python3 code. For our purposes this tool is unsuitable because `mprof` does not measure the GPU memory usage, the backend-specific profilers also measure and plot memory usage over time, and line-by-line profiling causes problems with JAX's asynchronous execution.

Other profiling tools include NVIDIA's developer tools[4]. With Nsight Systems, which offers the `nsys` command and uses Nsight Compute, it is possible to profile CUDA applications. For PyTorch, `nsys` can be used together with PyProf[5] to include additional information in the profiling output such as executed PyTorch operations and to limit the profiling to a certain region in the Python3 code. Since the backend-specific profilers also show CUDA information in the trace and other TensorBoard pages, the author has decided not to use nsys for the quadrature algorithm profiling.

[3]Python's profiler documentation: `https://docs.python.org/3/library/profile.html` (Accessed: 2022-03-18)

[4]NVIDIA's overview of its developer tools: `https://developer.nvidia.com/tools-overview` (Accessed: 2022-03-17)

[5]PyProf user guide: `https://docs.nvidia.com/deeplearning/frameworks/pyprof-user-guide/profile.html` (Accessed: 2022-03-17)

4.3.2 Profiling Setup

The profiling scripts use composite Boole quadrature with a 4D `sin_prod` integrand, which is defined in Section 4.1, $17850625 \approx 18 \cdot 10^6$ sample points, and 32-bit floating-point precision unless noted otherwise. To distinguish between the time needed for the integrand evaluation and remaining quadrature-related calculations, the scripts have explicit synchronisation between the three steps, which are sample point calculation, integrand evaluation and integral calculation. For PyTorch this means calling `torch.cuda.synchronize()` before and after integrand evaluation and with JAX the synchronisations are implemented with `.block_until_ready()` method calls on JAX tensors. TensorFlow does not have a function for explicit synchronisation to the author's knowledge, so the integrand evaluation cannot be separated in the corresponding traces. In the trace visualisations, the three steps of the Boole quadrature implementation are denoted `calculate_grid` (or `step1`), `integrand execution (synced)` and `calculate_result` (or `step3`), and the whole quadrature is implemented in Boole's `integrate` method. Separating the integrand in the trace with synchronisations does not work when the whole integrate method is compiled. The profiling scripts additionally convert the final quadrature result to a NumPy array to ensure that any asynchronous execution has finished before the measurement ends, which is similar to the synchronisation for benchmarking described in Subsection 4.2.1.

The next sections show profiling results which use the setup and tools described in this and the previous sections.

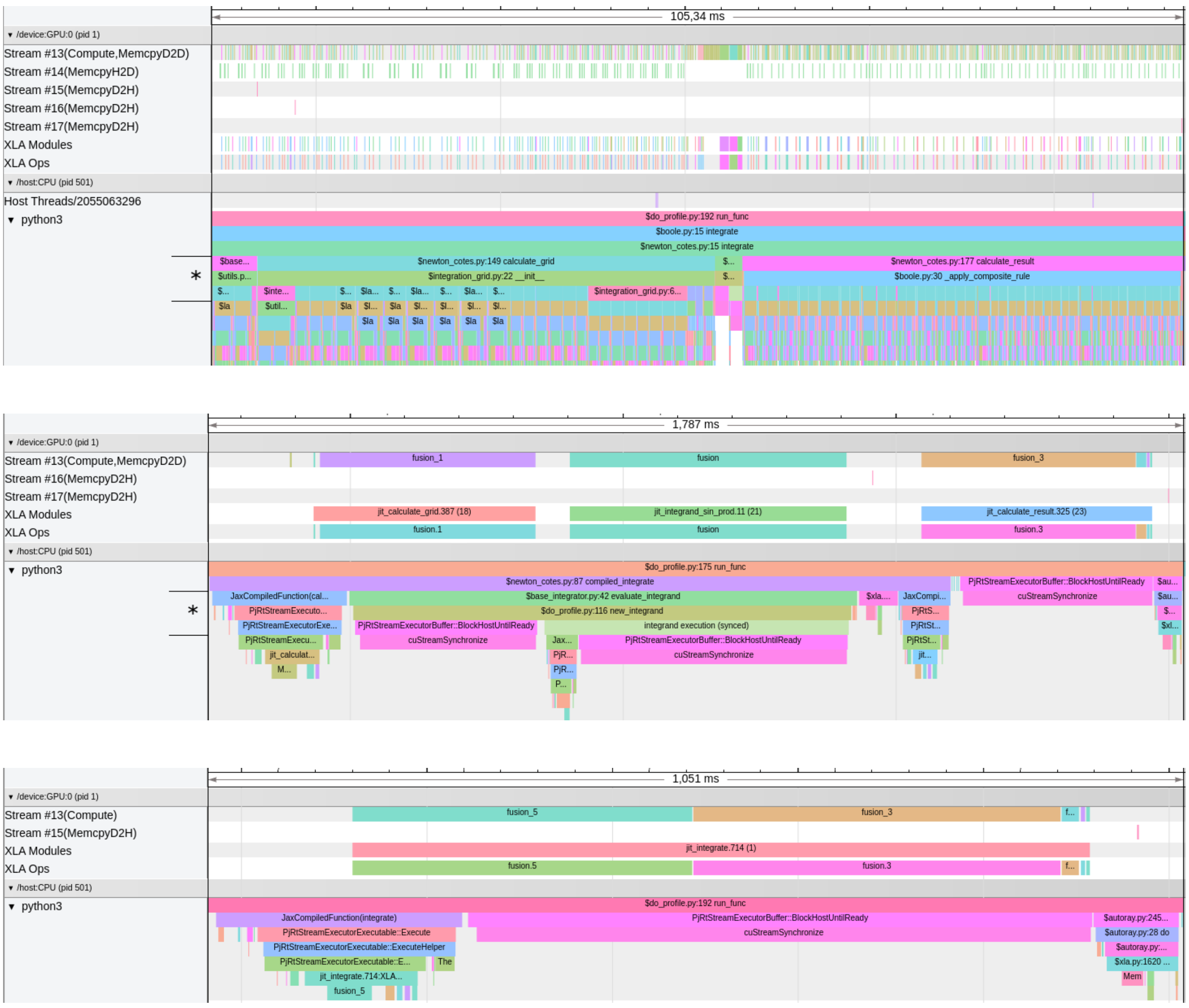

105,34 ms
/device:GPU:0 (pid 1)
Stream #13(Compute,MemcpyD2D)
Stream #14(MemcpyH2D)
Stream #15(MemcpyD2H)
Stream #16(MemcpyD2H)
Stream #17(MemcpyD2H)
XLA Modules
XLA Ops
/host:CPU (pid 501)
Host Threads/2055063296
python3
$do_profile.py:192 run_func
$boole.py:15 integrate
$newton_cotes.py:15 integrate
$newton_cotes.py:149 calculate_grid
$newton_cotes.py:177 calculate_result
$integration_grid.py:22 __init__
$boole.py:30 _apply_composite_rule
$integration_grid.py:6...
*
1,787 ms
/device:GPU:0 (pid 1)
Stream #13(Compute,MemcpyD2D)
Stream #16(MemcpyD2H)
Stream #17(MemcpyD2H)
XLA Modules
XLA Ops
/host:CPU (pid 501)
python3
fusion_1
fusion
fusion_3
jit_calculate_grid.387 (18)
jit_integrand_sin_prod.11 (21)
jit_calculate_result.325 (23)
fusion.1
fusion.3
$do_profile.py:175 run_func
$newton_cotes.py:87 compiled_integrate
PjRtStreamExecutorBuffer::BlockHostUntilReady
JaxCompiledFunction(cal...
$base_integrator.py:42 evaluate_integrand
$do_profile.py:116 new_integrand
integrand execution (synced)
cuStreamSynchronize
*
1,051 ms
/device:GPU:0 (pid 1)
Stream #13(Compute)
Stream #15(MemcpyD2H)
XLA Modules
XLA Ops
/host:CPU (pid 501)
python3
fusion_5
fusion_3
jit_integrate.714 (1)
fusion.5
fusion.3
$do_profile.py:192 run_func
JaxCompiledFunction(integrate)
PjRtStreamExecutorBuffer::BlockHostUntilReady
PjRtStreamExecutorExecutable::Execute
cuStreamSynchronize
PjRtStreamExecutorExecutable::ExecuteHelper
jit_integrate.714:XLA...
$autoray.py:28 do
$xla.py:1620 ...

Figure 4.6: TensorBoard visualisation of collected traces with JAX and different compilations for the previously described setup. Without compilation, there are numerous small operations in the CUDA stream, and `calculate_grid` and `calculate_result` are comparatively slow.
Top: Code executed in eager mode (`uncompiled`).
Middle: Sample point calculation, integrand evaluation, and result computation separately compiled (`parts compiled`).
Bottom: Whole integrate method compiled (`all compiled`)

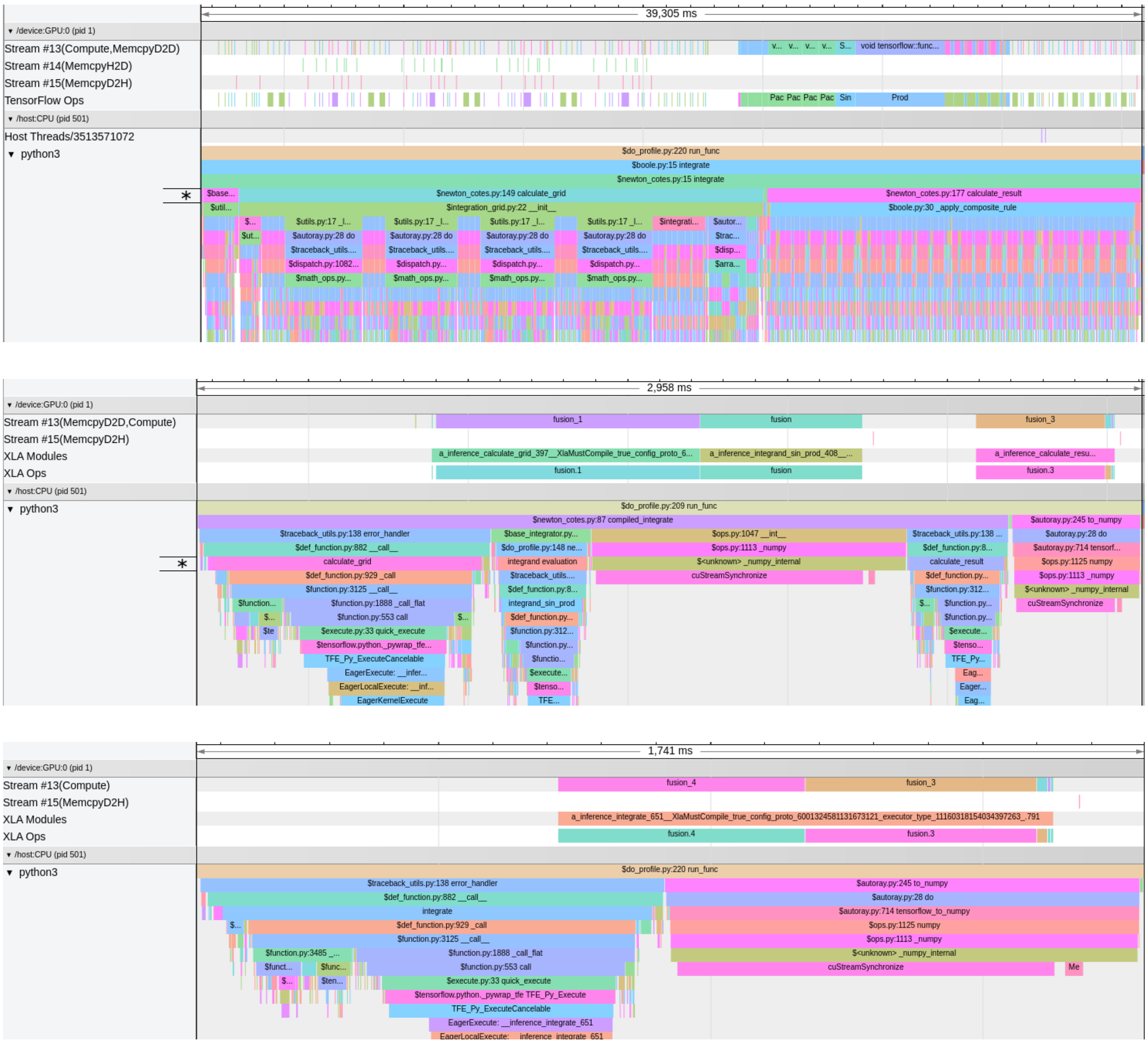

39,305 ms
/device:GPU:0 (pid 1)
Stream #13(Compute,MemcpyD2D)
Stream #14(MemcpyH2D)
Stream #15(MemcpyD2H)
TensorFlow Ops
/host:CPU (pid 501)
Host Threads/3513571072
python3
$do_profile.py:220 run_func
$boole.py:15 integrate
$newton_cotes.py:15 integrate
$newton_cotes.py:149 calculate_grid
$integration_grid.py:22 __init__
$newton_cotes.py:177 calculate_result
$boole.py:30 _apply_composite_rule
2,958 ms
Stream #13(MemcpyD2D,Compute)
XLA Modules
XLA Ops
fusion_1
fusion
fusion_3
$do_profile.py:209 run_func
$newton_cotes.py:87 compiled_integrate
$traceback_utils.py:138 error_handler
$def_function.py:882 __call__
calculate_grid
integrand evaluation
cuStreamSynchronize
calculate_result
$autoray.py:245 to_numpy
1,741 ms
Stream #13(Compute)
fusion_4
fusion.4
fusion.3
integrate
$function.py:3125 __call__
$execute.py:33 quick_execute
EagerExecute: __inference_integrate_651

Figure 4.7: TensorBoard visualisation of collected traces with TensorFlow and different compilations for the previously described setup. Without compilation, there are numerous small operations in the CUDA stream, and `calculate_grid` and `calculate_result` are comparatively slow.
Top: Code executed in eager mode (`uncompiled`).
Middle: Sample point calculation, integrand evaluation, and result computation separately compiled (`parts compiled`).
Bottom: Whole integrate method compiled (`all compiled`)

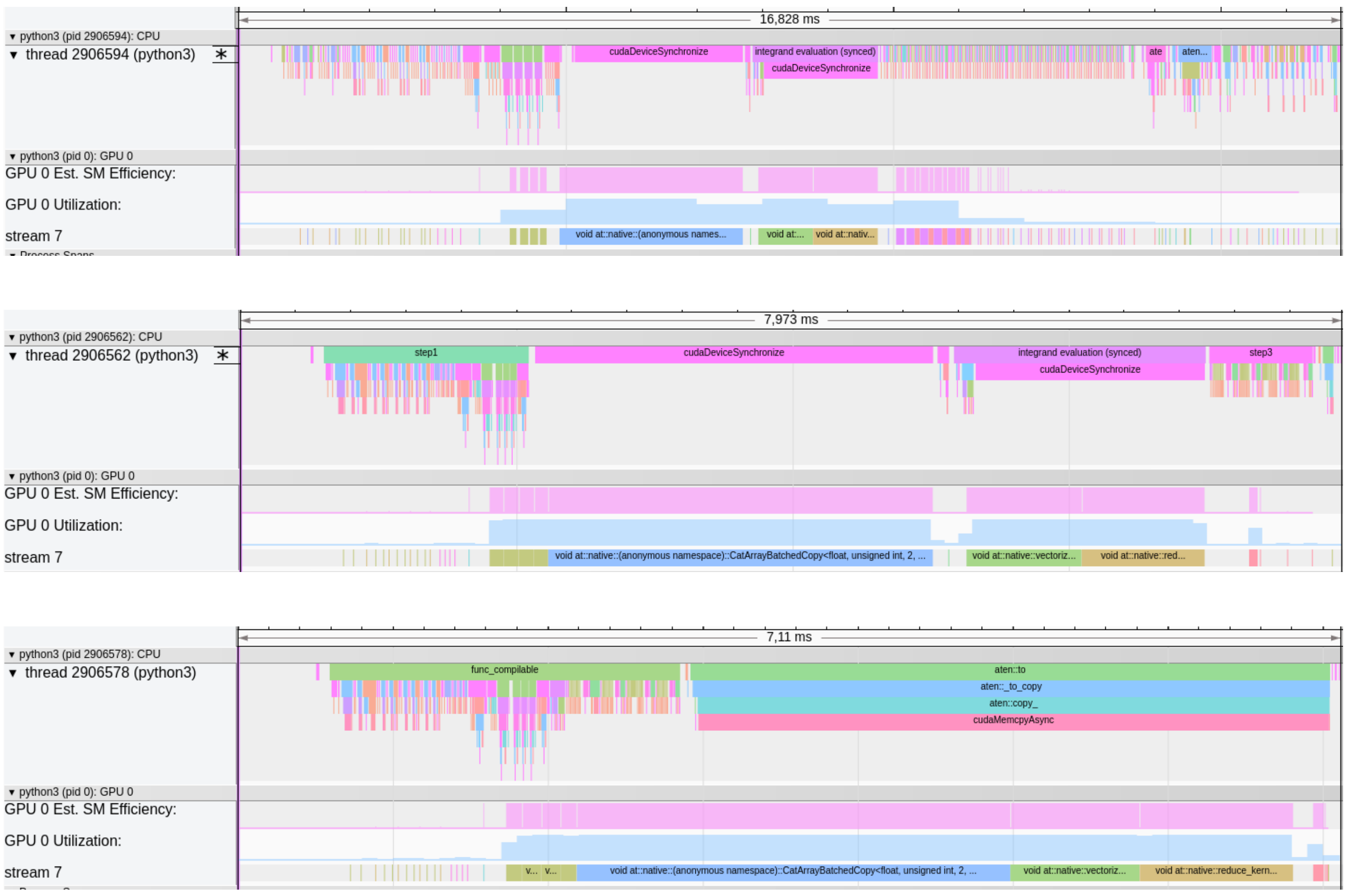

python3 (pid 2906594): CPU
thread 2906594 (python3)
16,828 ms
cudaDeviceSynchronize
integrand evaluation (synced)
cudaDeviceSynchronize
ate
aten...
python3 (pid 0): GPU 0
GPU 0 Est. SM Efficiency:
GPU 0 Utilization:
stream 7
void at::native::(anonymous names...
void at:...
void at::nativ...
python3 (pid 2906562): CPU
thread 2906562 (python3)
7,973 ms
step1
cudaDeviceSynchronize
integrand evaluation (synced)
cudaDeviceSynchronize
step3
python3 (pid 0): GPU 0
GPU 0 Est. SM Efficiency:
GPU 0 Utilization:
stream 7
void at::native::(anonymous namespace)::CatArrayBatchedCopy<float, unsigned int, 2, ...
void at::native::vectoriz...
void at::native::red...
python3 (pid 2906578): CPU
thread 2906578 (python3)
7,11 ms
func_compilable
aten::to
aten::_to_copy
aten::copy_
cudaMemcpyAsync
python3 (pid 0): GPU 0
GPU 0 Est. SM Efficiency:
GPU 0 Utilization:
stream 7
v... v...
void at::native::(anonymous namespace)::CatArrayBatchedCopy<float, unsigned int, 2, ...
void at::native::vectoriz...
void at::native::reduce_kern...

Figure 4.8: TensorBoard visualisation of collected traces with PyTorch and different compilations for the previously described setup. Without compilation, there are numerous small operations in the CUDA stream.
Top: Code executed in eager mode (`uncompiled`).
Middle: Sample point calculation, integrand evaluation, and result computation separately compiled (`parts compiled`).
Bottom: Whole integrate method compiled (`all compiled`)

4.3.3 Execution Trace Comparisons

Figure 4.6, Figure 4.7 and Figure 4.8 show screenshots of TensorBoard trace visualisations for JAX, TensorFlow and PyTorch. Each trace corresponds to a numerical integration. In these pictures, the horizontal axis corresponds to the time and on the vertical axis there are rows showing GPU utilisation information and rows for the Python3 interpreter stack. Since the stack is depicted over time, it has a hierarchical shape and it is visualised with an icicle plot. Where possible, asterisks were added to highlight the rows which show the execution of the three steps mentioned in the previous section. Except for TensorFlow, these rows show the explicit synchronisation before and after the integrand evaluation. The duration of each numercal integration is shown on the top of the pictures. The colours serve only as a means for a visual distinction between the rectangles. Analogously to the benchmarking measurements, we can distinguish between the `uncompiled`, `parts compiled` and `all compiled` cases. From the visualisations we can make the following observations:

- In the `uncompiled` case, the program executes a high number of small operations on the GPU, whereas in the other two cases, there are only a few operations executed on the GPU. This may indicate that the main benefit of compilation is due to the fusion of operations executed on the GPU. In the three trace figures this is visible in the rows which show CUDA stream operations: There are numerous small stripes in the `uncompiled` cases and big rectangles in the other cases.
- In the `all compiled` cases, the integrand is no longer visible in the trace.
- With JAX and TensorFlow in the `uncompiled` case, the integrand execution time is negligible in comparison to the grid point calculation and final integral calculation. This is visible as a small bright green rectangle between `evaluate_grid` and `calculate_result` rectangles in Figure 4.6 and Figure 4.7.
- With JAX in the `parts compiled` case, the explicit synchronisation before and after the integrand evaluation is visible as magenta rectangles in Figure 4.6 and overlaps with time where the GPU is busy.
- With PyTorch, the CUDA operations (stream 7) shown in Figure 4.8 look very similar in the `parts compiled` and `all compiled` cases although the operations in the Python3 CPU traces differ. This similarity may be the reason why the two compilation cases converge to similar median times in the benchmarking results.
- With PyTorch in the `parts compiled` case, the first step of the quadrature and the following CUDA synchronisation require a comparatively long time. This might indicate a performance bottleneck in the grid points calculation. In Figure 4.8,

this step is visible as a green rectangle labelled `step1` and a magenta one labelled `cudaDeviceSynchronize`.

4.3.4 Execution Traces for 1D Integrands

With the TensorBoard trace visualisation, we can find code segments which do not or only partly utilize the GPU for numerical operations. When an uncompiled `integrate` method of a Newton-Cotes integrator is executed with TensorFlow or JAX, parts of the calculations are executed on the CPU. Figure 4.9 shows trace visualisations with a single-dimensional integrand. These traces reveal that TensorFlow executes the linspace[6] operation on the CPU and then copies the result to the GPU. The input to this operation is the integration domain and its output is used for the grid point calculation. To investigate the missing GPU utilisation further, device placement logging has been enabled with `tf.debugging.set_log_device_placement(True)`. The generated logs mention that all operations have been executed on the GPU although the trace and benchmarking reveal that TensorFlow has used the CPU. TensorFlow has the context manager `with tf.device("/GPU:0"):`[7], which allows a user to specify that all numerical operations should be executed on the GPU. Testing has shown that with and without this context manager, and with both 32-bit and 64-bit floating-point precisions, TensorFlow executes the linspace operation on the CPU.

With JAX, the traces show that gather operations are executed on the CPU. These operations probably calculate a set of indices for slicing, copy them to the GPU and then these indices are used to apply the Boole composite quadrature rule on the integrand function's output. As an attempt to avoid operations on the CPU, the integration domain has been marked as committed with `jax.device_put(integration_domain, jax.devices()[0])`. If a tensor is committed to the GPU, JAX operations which have this tensor as input produce a committed tensor as output and fail if another input argument is committed to a different device.[8] JAX has executed the gather operations on the CPU even with data committed to the GPU and with both 64-bit and 32-bit floating-point precisions.

With a higher dimensionality, the operations on the CPU generate an output which is asymptotically smaller than the number of integrand evaluations, which makes these operations negligible. The incomplete utilisation of the GPU does not happen with

[6]TensorFlow's linspace operation documentation: `https://www.tensorflow.org/api_docs/python/tf/linspace` (Accessed: 2022-03-18)

[7]TensorFlow's `tf.device` context manager: `https://www.tensorflow.org/api_docs/python/tf/device` (Accessed: 2022-03-19)

[8]JAX documentation about committed devices: `https://jax.readthedocs.io/en/latest/faq.html?highlight=device_put#controlling-data-and-computation-placement-on-devices` (Accessed: 2022-03-17)

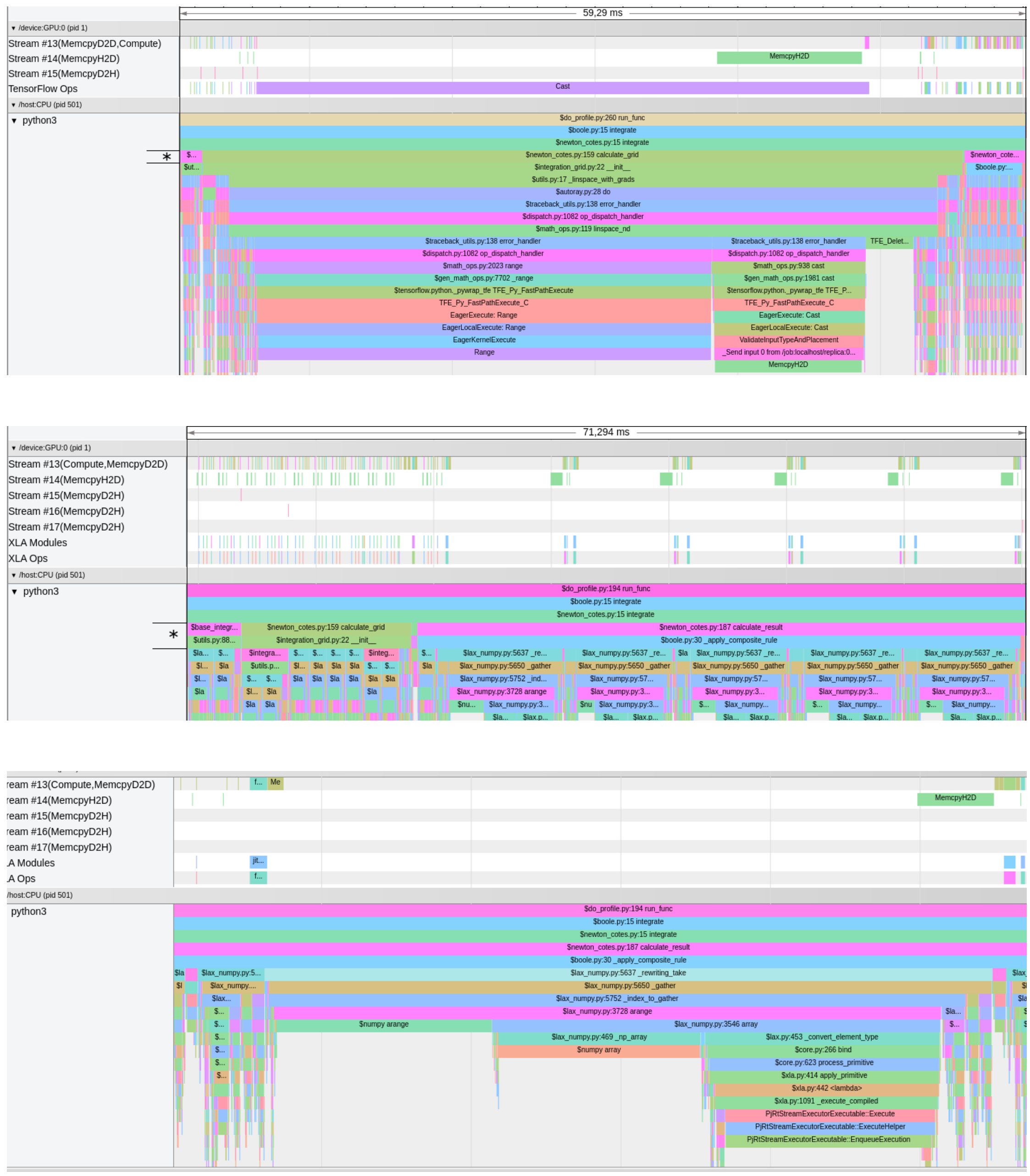

59,29 ms
/device:GPU:0 (pid 1)
Stream #13(MemcpyD2D,Compute)
Stream #14(MemcpyH2D)
Stream #15(MemcpyD2H)
TensorFlow Ops
MemcpyH2D
Cast
/host:CPU (pid 501)
python3
$do_profile.py:260 run_func
$boole.py:15 integrate
$newton_cotes.py:15 integrate
$newton_cotes.py:159 calculate_grid
$integration_grid.py:22 __init__
$utils.py:17 _linspace_with_grads
$autoray.py:28 do
$traceback_utils.py:138 error_handler
$dispatch.py:1082 op_dispatch_handler
$math_ops.py:119 linspace_nd
$math_ops.py:2023 range
$gen_math_ops.py:7702 _range
$tensorflow.python._pywrap_tfe TFE_Py_FastPathExecute
TFE_Py_FastPathExecute_C
EagerExecute: Range
EagerLocalExecute: Range
EagerKernelExecute
Range
$math_ops.py:938 cast
$gen_math_ops.py:1981 cast
EagerExecute: Cast
EagerLocalExecute: Cast
ValidateInputTypeAndPlacement
_Send input 0 from /job:localhost/replica:0...
TFE_Delet...
*
71,294 ms
Stream #13(Compute,MemcpyD2D)
Stream #16(MemcpyD2H)
Stream #17(MemcpyD2H)
XLA Modules
XLA Ops
$do_profile.py:194 run_func
$newton_cotes.py:187 calculate_result
$boole.py:30 _apply_composite_rule
$base_integr...
$utils.py:88...
$lax_numpy.py:5637 _re...
$lax_numpy.py:5650 _gather
$lax_numpy.py:5752 _ind...
$lax_numpy.py:3728 arange
$lax_numpy.py:5637 _rewriting_take
$lax_numpy.py:5752 _index_to_gather
$lax_numpy.py:3728 arange
$numpy arange
$lax_numpy.py:3546 array
$lax_numpy.py:469 _np_array
$numpy array
$lax.py:453 _convert_element_type
$core.py:266 bind
$core.py:623 process_primitive
$xla.py:414 apply_primitive
$xla.py:442 <lambda>
$xla.py:1091 _execute_compiled
PjRtStreamExecutorExecutable::Execute
PjRtStreamExecutorExecutable::ExecuteHelper
PjRtStreamExecutorExecutable::EnqueueExecution

Figure 4.9: TensorBoard visualisation of collected traces with TensorFlow and JAX for a one-dimensional integrand and Composite Boole quadrature. These traces show a low utilisation of the GPU.
Top: TensorFlow, `uncompiled`.
Middle: JAX, `uncompiled`.
Bottom: JAX, `uncompiled`, close-up view of the gather operation

the PyTorch backend or if the quadrature is compiled. In the benchmarking section the problem is visible in plots for one-dimensional integrands. Figure 4.3 shows that the curve for sample point calculation with `uncompiled` TensorFlow and NumPy rise similarly if the integrand is one-dimensional and Figure 4.4 analogously depicts this behaviour for the application of the composite Boole rule with JAX. In Figure 4.2, both `uncompiled` TensorFlow and `uncompiled` JAX rise early in the single-dimension case because the calculation of sample points and the quadrature rule application are both parts of the numerical integration.

4.3.5 Memory Requirement Comparisons

To investigate the memory requirement of the numerical backends, we can use the Memory View page of the TensorBoard profiler. The GPU memory usage over time for PyTorch is shown in Figure 4.10 for a numerical integration and Figure 4.11 for a gradient calculation of the integration with respect to the integration domain. In the PyTorch TensorBoard profiler plugin version used for these plots, units of the memory are shown as MB although they are MiB.[9] The implementation of the gradient calculation for profiling is mostly similar to the gradient calculation for benchmarking explained in Subsection 4.2.5; however, here in the `all compiled` case with TensorFlow, the `integrate` method is compiled but the gradient calculation with the `tf.GradientTape()` context manager is not, and the code has additional CUDA synchronisations as explained in Subsection 4.3.2. Since there are $N = 17850625$ four-dimensional sample points with 32-bit floating-point precision, at least $N \cdot 4 \cdot 4$ B ≈ 272.4 MiB memory is required to perform the quadrature. From the visualisation for PyTorch we can make three observations:

- When the whole integrate method is compiled, the peak memory usage is 544.8 MiB $= 2 \cdot 272.4$ MiB. This indicates that the compiled integrate function uses approximately two times the minimum required amount of memory for the calculation.
- Although the `parts compiled` case is faster than the `uncompiled` case, it has the same peak memory usage of 612.9 MiB $\approx$ 544.8 MiB $+ N \cdot 4$ MiB. The minimal amount of memory required to save the integrand output is $N \cdot 4$ MiB, so we may assume that the peak memory is caused by the simultaneous presence of two tensors which both contain the same sample points and one tensor with the integrand output.

[9]"MB" units in PyTorch's memory visualisations: `https://github.com/pytorch/kineto/blob/0467abc6739cf9811d6e0712253671927de44506/tb_plugin/torch_tb_profiler/utils.py#L57` (Accessed: 2022-03-17)

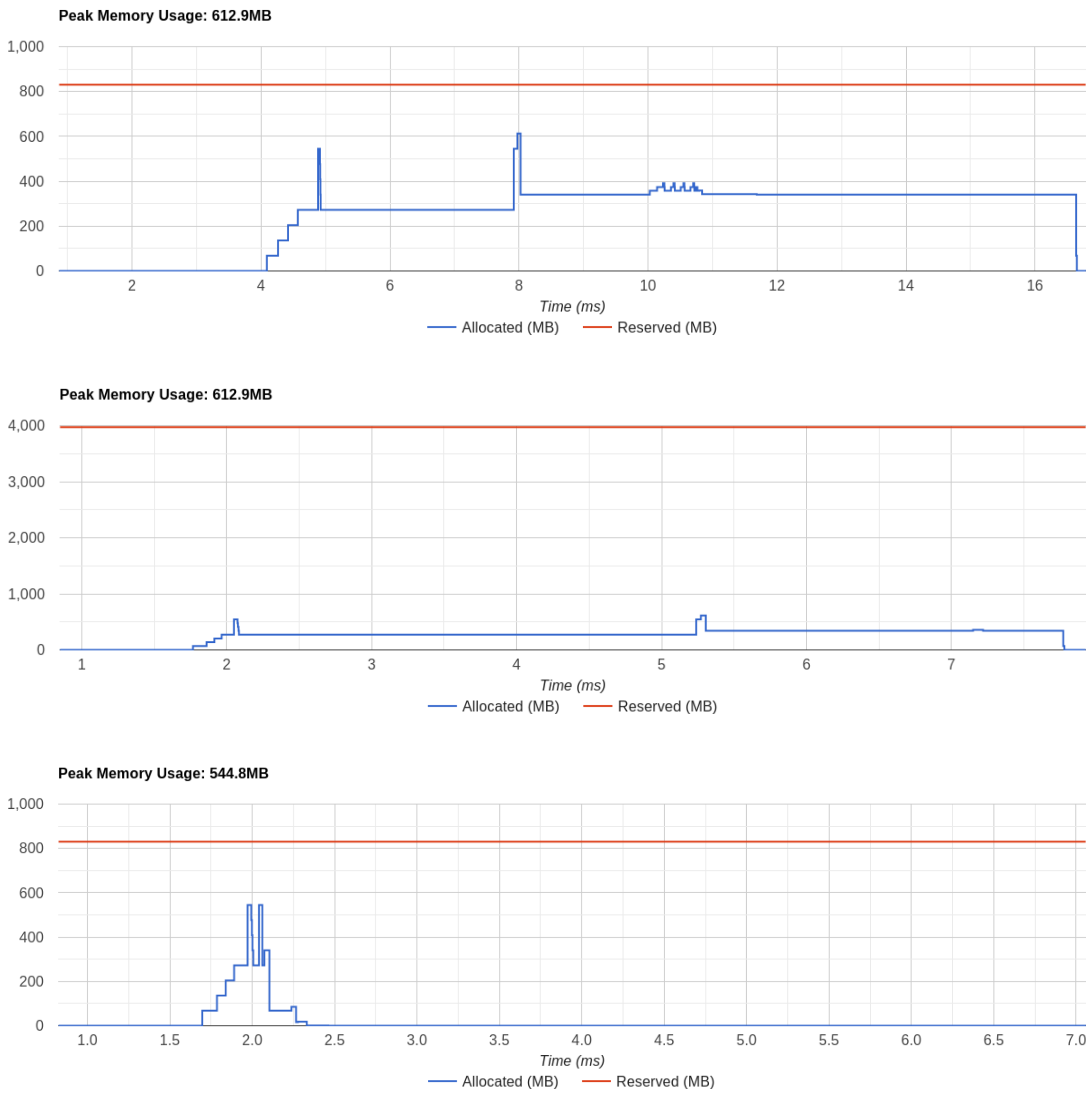


Figure 4.10: TensorBoard visualisation of GPU memory usage over time for an integration with PyTorch and different compilation configurations. Composite Boole quadrature, 4D `sin_prod` integrand, 17850625 sample points, 32-bit floating-point precision. Memory is shown in MiB, not MB. The blue and red lines correspond to the allocated and reserved GPU memory at each point in time.
Top: Code executed in eager mode (`uncompiled`).
Middle: Sample point calculation, integrand evaluation, and result computation separately compiled (`parts compiled`).
Bottom: Whole integrate method compiled (`all compiled`)

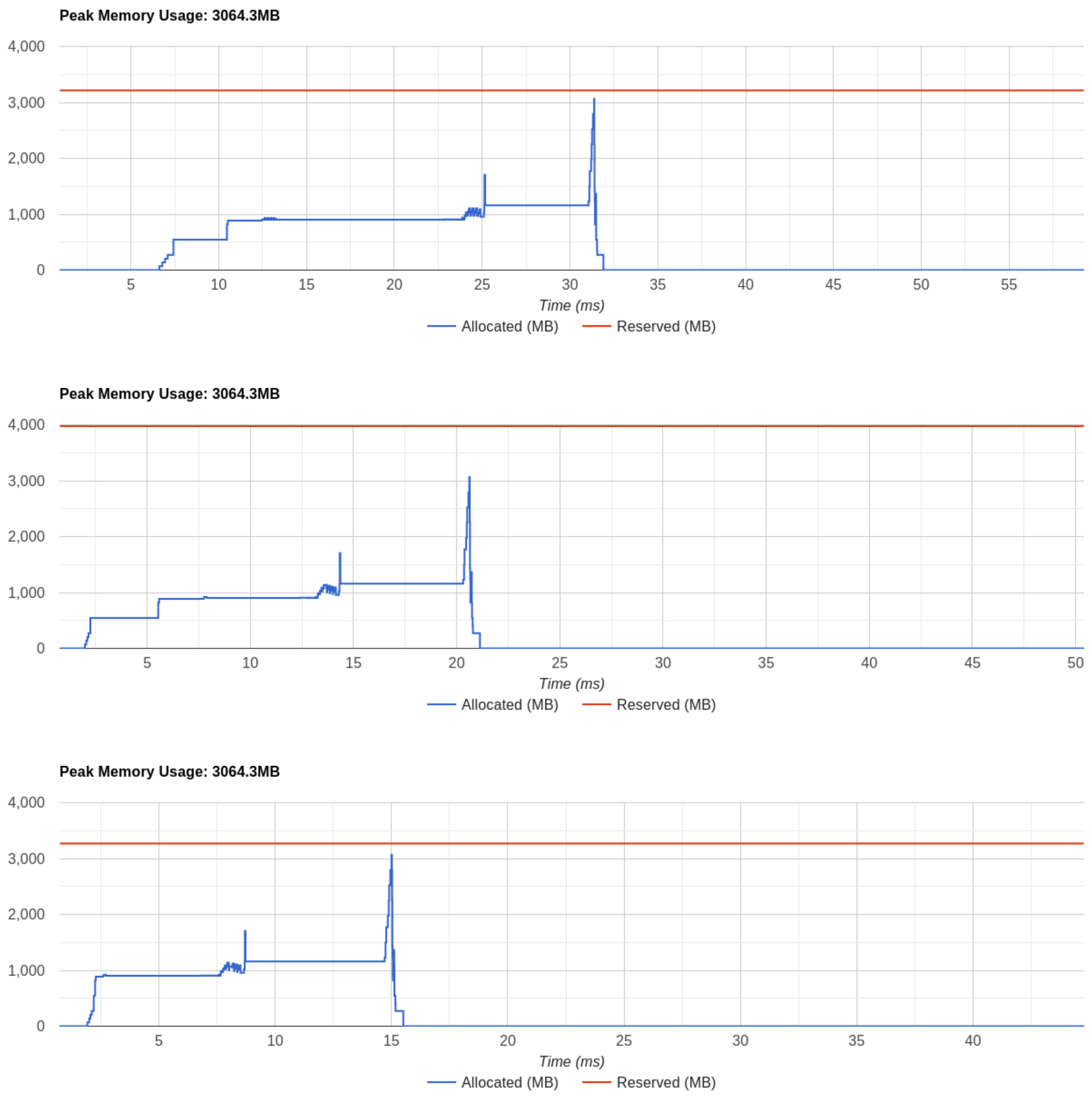


Figure 4.11: TensorBoard visualisation of GPU memory usage over time for the calculation of the gradient of an integration over the integration domain with PyTorch and different compilation configurations. Composite Boole quadrature, 4D `sin_prod` integrand, 17850625 sample points, 32-bit floating-point precision. Memory is shown in MiB, not MB. The blue and red lines correspond to the allocated and reserved GPU memory at each point in time.
Top: Code executed in eager mode (`uncompiled`).
Middle: Sample point calculation, integrand evaluation, and result computation separately compiled (`parts compiled`).
Bottom: Whole integrate method compiled (`all compiled`)

- With gradient calculation, all three cases use 3064.3 MiB $\approx 5 \cdot 612.9$ MiB memory, which is approximately five times the memory requirement compared to integration without gradient calculation.

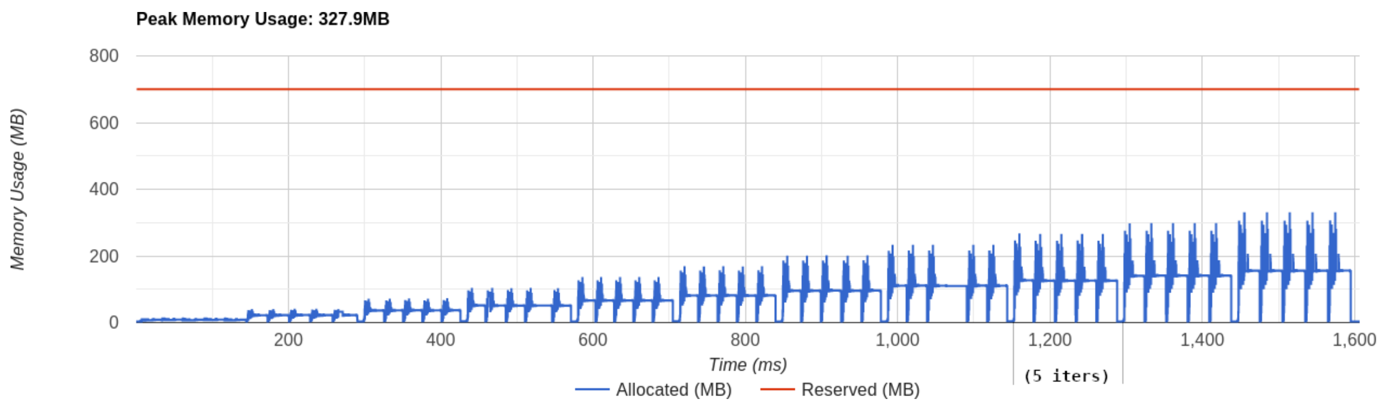
With JAX and TensorFlow, the TensorBoard Memory profiler has not worked as expected with compiled functions; for example, in the `all compiled` case with a complicated example integrand, it has reported only approximately 1 MiB peak memory usage. Furthermore, the interactive Memory profiler plots for these two backends show a lot of unused space, so screenshots of them are not included in this thesis. Nonetheless, for the `uncompiled` case, the TensorBoard output appears reasonable and is summarized in Table 4.4. By comparing the measurements for numerical integration to those which additionally include gradient calculation over the integration domain, we can see that TensorFlow has approximately twice the number of allocations and a memory requirement which is higher by a factor of ca. 4.3, whereas JAX has approximately the same number of allocations and the memory requirement is higher by a factor of ca. 1.3.

	Peak Memory Usage	Allocations	Deallocations
TensorFlow	0.67 GiB	219	219
TensorFlow (grad)	2.92 GiB	444	444
JAX	1.06 GiB	511	489
JAX (grad)	1.36 GiB	548	452

Table 4.4: Peak memory usage, and the number of allocations and deallocations reported by the TensorBoard Memory Viewer for JAX and TensorFlow in the `uncompiled` case with and without gradient computation. Composite Boole quadrature, 4D `sin_prod` integrand, 17850625 sample points, 32-bit floating-point precision.

In addition to profiling composite Boole quadrature, we can also investigate the memory usage of torchquad's VEGAS+ implementation. Figure 4.12 shows PyTorch's memory usage over time for a VEGAS+ quadrature with 50 iterations and ca. 10^8 function evaluations. For an understanding of the plots, we need some implementation details about torchquad's VEGAS+ implementation:

- Before the 50 main iterations, `VEGAS` executes five warm-up iterations with a very small number of points.
- Every iteration, `VEGAS` calculates sample points, evaluates the integrand on them, and saves the integral result of this iteration. Furthermore, between iterations, it updates the `VEGASMap` and `VEGASStratification` to adapt to the integrand.



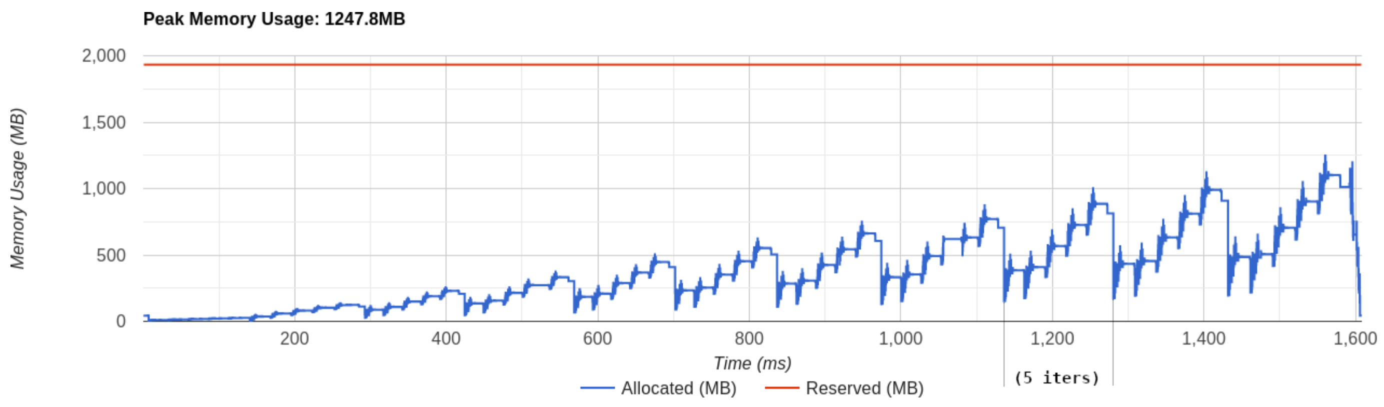
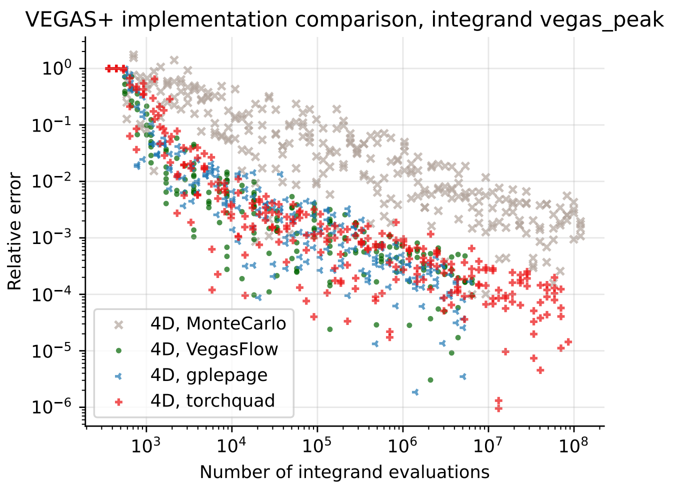


Figure 4.12: TensorBoard visualisation of GPU memory usage over time for VEGAS+ quadrature with PyTorch, 50 iterations, ca. $N = 10^8$ function evaluations and 32-bit floating-point precision. Memory is shown in MiB, not MB. The blue and red lines correspond to the allocated and reserved GPU memory at each point in time. The excerpt of the plot which corresponds to the iterations 35 to 39 is highlighed and denoted `(5 iters)`.
Top: VEGAS+ quadrature.
Bottom: VEGAS+ quadrature with gradient calculation

- Every fifth iteration, `VEGAS` resets previously calculated results and increases the number of sample points per iteration, which is used in the next five iterations. This means the final integral result is calculated from the integrals of the last five iterations.
- For common integrands, the recommended number of iterations is smaller than 50. Nonetheless, here the number of iterations is high so that memory allocation patterns or any memory leaks can be inferred more reliably from the plots.

With this information, we can make the following observations from the plots:

- Without gradient calculation, the memory usage goes back to approximately zero after each iteration in the plot, and every fifth iteration the memory requirement per iteration increases. This indicates that torchquad has no memory leak and the sample points, which require the most memory, are released as soon as the integral result for the current iteration is available and the VEGAS map and stratification adaptation has finished.
- When calculating gradients, PyTorch has to remember the sample points used to calculate the final result for the backwards pass, which explains the approximately stair-shaped look in each group of five iterations.
- Since every fifth iteration old solutions are removed, the memory usage goes down significantly when calculating gradients. It may not reach zero because the `VEGASStratification` indirectly refers to the sample points of the previous iteration; however, this is not a memory usage problem since these points are released after the first iteration in a group of five iterations, which is visible in the plot where the height of the first step of each stair is risen to the second step's height.



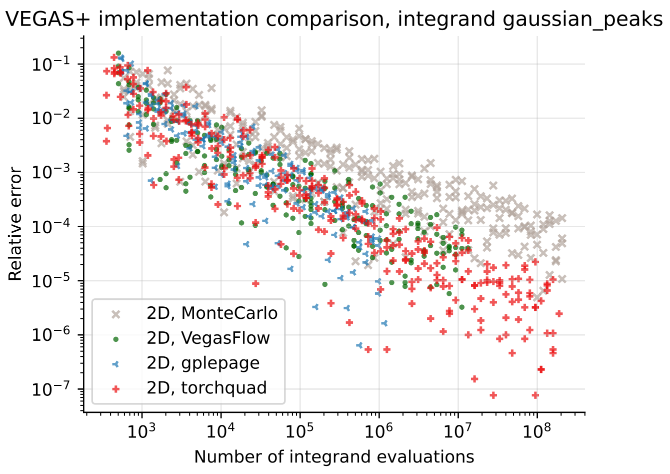


Figure 4.13: Convergence behaviour comparison between different VEGAS+ implementations for the `vegas_peak` and `gaussian_peaks` integrands. Many VEGAS+ algorithm parameters differ between the implementations, so the plot does not show if an implementation is better or worse than another one. However, it shows that all VEGAS+ implementations converge faster than Monte Carlo integration.

4.4 VEGAS+ Accuracy Comparisons

To validate that torchquad's VEGAS+ implementation converges faster than Monte Carlo integration for integrands where the VEGAS+ algorithm performs well, a script has been written which integrates the `vegas_peak` and `gaussian_peaks` integrands with different Python3 VEGAS+ implementations and Monte Carlo for various numbers of function evaluations multiple times. The `vegas_peak` integrand is based on an example function from G. Peter Lepage's vegas tutorial[10] and defined for the domain $[0,1]^4$:

$$\text{vegas\_peak}(x) = e^{-100\left((2x_1-1.5)^2+\sum_{i=2}^{4}(x_i-0.5)^2\right)} \tag{4.2}$$

The other VEGAS+ implementations which are compared to torchquad are VegasFlow [CC20a][CC20b], which is a VEGAS+ implementation optimized for TensorFlow, and G. Peter Lepage's vegas [Lep21], denoted gplepage here, which is the original implementation of VEGAS+. MonteCarlo corresponds to torchquad's Monte Carlo implementation. For torchquad's `VEGAS` the measurement script sets the

[10]G. Peter Lepage's example integrand: `https://vegas.readthedocs.io/en/latest/tutorial.html#basic-integrals` (Accessed: 2022-03-17)

floating-point precision to 32 bit, for VegasFlow it uses the `VegasFlowPlus` integrator and sets the number of iterations to seven, and for G. Peter Lepage's vegas it executes five iterations to adapt the VEGAS map and then seven iterations with `alpha=0.1` to calculate an integral result. Other configurations are left at their default value and differ between the implementations. The plots in Figure 4.13 show the measurement results; all VEGAS+ implementations converge faster than Monte Carlo for the `vegas_peak` and more complicated `gaussian_peaks` integrands and have similar accuracies although they are configured differently. Since the numbers of iterations, VEGAS map intervals and stratification hypercubes, and other VEGAS+ algorithm configuration has a high impact on the accuracy, the scatter plots do not represent the best possible accuracies for the implementations; therefore, the plots do not show trend lines which approximate the measurements.

5 Conclusion and Future Work

The previous chapter has shown that PyTorch, JAX and TensorFlow all utilize the GPU for parallel numerical computations in torchquad. Furthermore, autoray does not lead to performance limitations nor hinders the use of some backend-specific featues such as automatic differentiation, compilation of numerical computations, and profiling and debugging tools. With torchquad's composite Boole quadrature, a high number of integrand evaluations and a cheap three-dimensional integrand, JAX and TensorFlow are more than seven times as fast as PyTorch when code is compiled whereas in eager execution mode, PyTorch is more than twice as fast as the other backends.

As explained in Chapter 3 and Subsection 2.2.2, autoray wraps functions for numerical operations. Therefore, it is possible to replace functions from PyTorch or another single numerical backend with corresponding functions from autoray while the numerical computations which are performed during code executions do not change. The support for multiple numerical backends can require additional code changes, which differ between backends and sometimes include optional features such as support for non-default floating-point precision and function compilation. The VEGAS+ implementation uses in-place operations and methods which change object member variables and thus are non-pure. For PyTorch and NumPy this is not a problem. However, in-place operations are slow or unsupported with JAX and TensorFlow in eager execution mode and a code compilation, which does not work with non-pure functions, is needed to achieve a reasonable performance with these backends. We conclude that autoray can be employed for most numerical computations but the support for multiple backends can require compromises such as avoiding features which do not work with all targeted backends.

Future work may involve the investigation of support for more hardware in torchquad, for example AMD GPUs, TPUs and multi-GPU setups. Furthermore, there are numerous numerical backends in addition to the four backends considered in this thesis, for example Dask [Das16], which enables parallelization on high performance computing systems. Support for more backends in torchquad could be investigated; some of these backends may already work if their API is very similar to NumPy's API and others may be compatible after a few code changes. Furthermore, future work could include the addition of more features to torchquad's quadrature algorithms, for example support for integrands with multi-dimensional output and the possibility to reuse the VEGAS map and stratification in the VEGAS+ integrator for multiple integrations, which may be helpful in the context of stochastic gradient descent optimisation. Furthermore, low discrepancy sequences could be implemented as an optional replacement for the random number generator used in the Monte Carlo and VEGAS+ implementations, which may enable users to achieve a higher accuracy on average for certain integrands. It is also possible to add more quadrature algorithms to torchquad; for example, with certain integrands, a quadrature algorithm which uses Sparse Grids could be helpful to tackle the curse of dimensionality and can be parallelized well when using the Sparse Grid combination technique [Gar13].

List of Figures

List of Tables

Bibliography

[Ana21] Anaconda Inc. *Anaconda Software Distribution*. Version 2021.11. 2021. URL: https://anaconda.com.

[BBC61] R. Bellman, R. Bellman, and K. M. R. Collection. *Adaptive Control Processes: A Guided Tour*. Princeton Legacy Library. Princeton University Press, 1961. ISBN: 9780691079011.

[Bra+18] J. Bradbury, R. Frostig, P. Hawkins, et al. *JAX: composable transformations of Python+NumPy programs*. Version 0.2.5. 2018. URL: http://github.com/google/jax.

[CC20a] S. Carrazza and J. M. Cruz-Martinez. "VegasFlow: accelerating Monte Carlo simulation across multiple hardware platforms". In: *Comput. Phys. Commun.* 254 (2020). Implementation at https://github.com/N3PDF/vegasflow (Accessed: 2022-03-17), p. 107376. DOI: 10.1016/j.cpc.2020.107376. arXiv: 2002.12921 [physics.comp-ph].

[CC20b] J. Cruz-Martinez and S. Carrazza. *N3PDF/vegasflow: vegasflow v1.0*. Version v1.0. Feb. 2020. DOI: 10.5281/zenodo.3691926.

[CS17] P. Chadha and T. Siddagangaiah. *Performance Analysis of Accelerated Linear Algebra Compiler for TensorFlow*. May 2017. URL: https://parthchadha.github.io/xla_report.pdf.

[Das16] Dask Development Team. *Dask: Library for dynamic task scheduling*. 2016. URL: https://dask.org.

[Dav+21] M. Davis et al. *SnakeViz, an in-browser Python profile viewer*. Version 2.1.0. 2021. URL: https://jiffyclub.github.io/snakeviz.

[Gar13] J. Garcke. "Sparse Grids in a Nutshell". In: *Sparse grids and applications*. Ed. by J. Garcke and M. Griebel. Vol. 88. Lecture Notes in Computational Science and Engineering. extended version with python code https://ins.uni-bonn.de/media/public/publication-media/sparse_grids_nutshell_code.pdf. Springer, 2013, pp. 57–80. DOI: 10.1007/978-3-642-31703-3_3.

[Goo] Google. *XLA: Optimizing Compiler for Machine Learning*. URL: https://www.tensorflow.org/xla?hl=en.

[Gra+] J. Gray et al. *A lightweight python AUTOmatic-arRAY library*. Version 0.2.5. URL: `https://github.com/jcmgray/autoray`.

[GTM21] P. Gómez, H. H. Toftevaag, and G. Meoni. "torchquad: Numerical Integration in Arbitrary Dimensions with PyTorch". In: *J. Open Source Softw.* 6 (2021), p. 3439. URL: `https://joss.theoj.org/papers/10.21105/joss.03439`.

[Har+20] C. R. Harris, K. J. Millman, S. J. van der Walt, et al. "Array programming with NumPy". In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: `10.1038/s41586-020-2649-2`.

[He19] H. He. *The State of Machine Learning Frameworks in 2019*. (Accessed: 2022-03-18). 2019. URL: `https://thegradient.pub/state-of-ml-frameworks-2019-pytorch-dominates-research-tensorflow-dominates-industry`.

[Hee+20] J. Heek, A. Levskaya, A. Oliver, et al. *Flax: A neural network library and ecosystem for JAX*. Version 0.4.1. 2020. URL: `http://github.com/google/flax`.

[Hen+20] T. Hennigan, T. Cai, T. Norman, and I. Babuschkin. *Haiku: Sonnet for JAX*. Version 0.0.3. 2020. URL: `http://github.com/deepmind/dm-haiku`.

[Jou+17] N. P. Jouppi, C. Young, N. Patil, et al. "In-Datacenter Performance Analysis of a Tensor Processing Unit". In: *SIGARCH Comput. Archit. News* 45.2 (June 2017), pp. 1–12. ISSN: 0163-5964. DOI: `10.1145/3140659.3080246`.

[KDH11] K. Karimi, N. G. Dickson, and F. Hamze. *A Performance Comparison of CUDA and OpenCL*. 2011. arXiv: `1005.2581 [cs.PF]`.

[Kre+21] H. Krekel, B. Oliveira, R. Pfannschmidt, F. Bruynooghe, B. Laugher, and F. Bruhin. *pytest*. Version 6.2.5. 2021. URL: `https://pytest.org`.

[Lep21] G. P. Lepage. "Adaptive Multidimensional Integration: VEGAS Enhanced". In: *Journal of Computational Physics* 439 (Aug. 2021). Implementation at `https://github.com/gplepage/vegas` (Accessed: 2022-03-17), p. 110386. ISSN: 0021-9991. DOI: `10.1016/j.jcp.2021.110386`.

[LP19] E. Lind and Ä. Pantigoso. "A performance comparison between CPU and GPU in TensorFlow". In: (June 2019). URL: `http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-260240`.

[Mar+15] Martín Abadi, Ashish Agarwal, Paul Barham, et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: `https://www.tensorflow.org`.

[MDA15] D. Maclaurin, D. Duvenaud, and R. P. Adams. “Autograd: Effortless gradients in numpy”. In: *ICML 2015 AutoML Workshop*. Vol. 238. 2015, p. 5.

[Mem+17] Z. Memon, F. Samad, Z. Awan, A. Aziz, and S. Siddiqi. “CPU-GPU Processing”. In: *International Journal of Computer Science and Network Security* 17 (Sept. 2017), pp. 188–193.

[NSW18] D. H. Noronha, B. Salehpour, and S. J. E. Wilton. *LeFlow: Enabling Flexible FPGA High-Level Synthesis of Tensorflow Deep Neural Networks*. July 2018. arXiv: `1807.05317 [cs.LG]`.

[Obe+21] M. Obersteiner et al. *sparseSpACE - the Sparse Grid Spatially Adaptive Combination Environment*. 2021. URL: `https://github.com/obersteiner/sparseSpACE`.

[Pas+19] A. Paszke, S. Gross, F. Massa, et al. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *NeurIPS*. 2019.

[PG+20] F. Pedregosa, P. Gervais, et al. *Memory Profiler, Monitor Memory usage of Python code.* Version 0.58.0. 2020. URL: `https://github.com/pythonprofilers/memory_profiler`.

[Poe21] PoetsAI. *Elegy: A High Level API for Deep Learning in JAX*. Version 0.8.1. 2021. URL: `https://github.com/poets-ai/elegy`.

[Qua21] Quansight Labs. *unumpy - NumPy, but implementation-independent*. 2021. URL: `https://github.com/Quansight-Labs/unumpy`.

[Sch+21] N. Schlömer, N. Papior, D. Arnold, J. Blechta, and R. Zetter. *quadpy, numerical integration (quadrature, cubature) in Python*. Sept. 2021. DOI: `10.5281/zenodo.5541216`. URL: `https://github.com/nschloe/quadpy`.

[Suh+21] A. Suhan, D. Libenzi, A. Zhang, et al. *LazyTensor: combining eager execution with domain-specific compilers*. 2021. arXiv: `2102.13267 [cs.PL]`.

[SVK20] P. Subramani, N. Vadivelu, and G. Kamath. “Enabling Fast Differentially Private SGD via Just-in-Time Compilation and Vectorization”. In: *arXiv preprint arXiv:2010.09063* (2020).

[Wu20] Y. Wu. *C++ Implementation of veGAs/veGAs+ algoRithm (CIGAR) for multi-dimension Integral.* 2020. URL: `https://github.com/ycwu1030/CIGAR`.