JOHANNES KEPLER
UNIVERSITY LINZ

Submitted by
Michael Fritzenwallner

Submitted at
Institute for Communications Engineering and RF-Systems

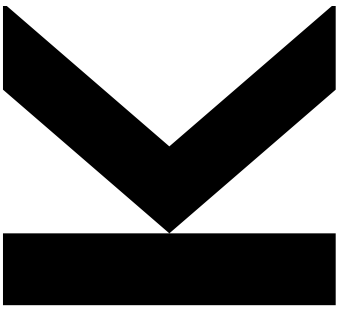
Supervisor
Assoz. Univ-Prof. DI Dr. Reinhard Feger

Co-Supervisor
DI Dr. Thomas Wagner

September 2023

GPU Programmierung im Kontext von Python mit Fokus auf die Fast Fourier Transformation

GPU Programing in the context of Python with a focus on the FFT algorithm



Bachelor Thesis

to obtain the academic degree of

Bachelor of Science

in the Bachelor's Program

Mechatronik

JOHANNES KEPLER
UNIVERSITY LINZ
Altenbergerstraße 69
4040 Linz, Österreich
www.jku.at
DVR 0093696

Statutory declaration

I hereby declare that the thesis submitted is my own unaided work, that I have not used other than the sources indicated, and that all direct and indirect sources are acknowledged as references. This printed thesis is identical with the electronic version submitted.

Place, Date

Signature

Zusammenfassung

Die Programmiersprache Python hat sich in den letzten Jahren als beliebte Sprache für wissenschaftliche Berechnungen und Datenverarbeitung etabliert. Python ist jedoch aufgrund seiner Eigenschaften als dynamisch typisierte und interpretierte Sprache grundsätzlich schlecht für rechenintensive Aufgaben wie die Signalverarbeitung geeignet. In Python ist es beispielsweise nicht möglich, alle Kerne moderner CPUs effektiv zu nutzen, da die Sprache einen global interpreter lock (GIL) verwendet.

Um diese Leistungseinschränkungen zu umgehen werden im Umfeld von Python verschiedene Werkzeuge und Bibliotheken entwickelt, die es einfach ermöglichen nativen, kompilierten Code in Python zu verwenden. Bekannte Beispiele sind Numpy und Scipy, die Python-Interfaces zu hoch optimierten und oft parallelisierten C und Fortran Bibliotheken bieten.

Ein besonders vielversprechender Ansatz zur weiteren Verbesserung der Leistung ist der Einsatz von Grafikprozessoren (GPUs) für berechnungsintensive Aufgaben, die sich für eine parallele Ausführung eignen. In den letzten Jahren wurden GPUs in Verbindung mit Python intensiv im Bereich des Deep Learning eingesetzt, wodurch eine Vielzahl von Python-Paketen entstanden ist, die einen einfachen Zugriff auf die GPU ermöglichen. In dieser Arbeit untersuchen wir den Einsatz von GPUs zur Beschleunigung der Signalverarbeitung in Python, wobei der Schwerpunkt auf der fast Fourier transform (FFT) liegt. Wir untersuchen geeignete Bibliotheken, die Python-Benutzern zur Verfügung stehen, wie PyTorch (ein Framework, das häufig in Bereich Deep-Learning verwendet wird und eine Vielzahl an Operationen auf GPUs auslagern kann) oder CuPy (eine Bibliothek ähnlich zu Numpy, die Berechnungen auf der GPU ausführt). Weiters wird eine eigene GPU-beschleunigte FFT-Routine in CUDA-C implementiert und in ein Python-Programm integriert.

Die experimentellen Ergebnisse zeigen, dass GPU-beschleunigte Bibliotheken erhebliche Geschwindigkeitsvorteile bei der Berechnung der FFT gegenüber CPU-basierten Implementierungen bieten können. Dies gilt insbesondere für die Berechnung von mehrdimensionalen Transformationen oder einer großen Anzahl an Transformationen gleichzeitig. Die Ergebnisse dieser Arbeit sind zwar auf eine bestimmte Hardware- und Softwarekonfiguration beschränkt und umfassen nur einen kleinen Teil der relevanten Algorithmen, können aber als Ausgangspunkt für weitere Untersuchungen zur GPU-beschleunigten Signalverarbeitung im Kontext von Python dienen.

Abstract

The Python programming language has emerged as the first choice for scientific computing and data processing due to its ease of use and vast ecosystem of libraries and modules. Python, however, is inherently unsuited for compute-intensive tasks such as signal processing because of its overhead as an interpreted language. Furthermore it is unable to effectively utilize all cores of modern central processing units (CPUs) due to the global interpreter lock (GIL). To address this performance limitation, various tools and libraries have been developed within the Python ecosystem to optimize performance. Tools commonly used in the domain such as Numpy and Scipy provide Python interfaces to highly optimized and often parallelized C and Fortran libraries.

One particularly promising area for further improvements is the integration of Graphics Processing Units (GPUs) for computation tasks, which lend themselves to parallel execution. In recent years GPUs have been extensively used in conjunction with Python in the field of Deep Learning. This led to a number of Python packages, which offer accessible ways to leverage the GPU's computing capabilities. In this thesis, we investigate the use of GPUs to accelerate signal processing in Python, with a focus on the FFT algorithm. We explore suitable libraries available to Python users, such as PyTorch (a framework commonly used in deep-learning that offers offloading tasks to GPUs) or CuPy (a GPU aware replacement for Numpy). We also implement our own GPU-accelerated FFT using idiomatic CUDA-C and integrate that routine into a Python program.

Our experimental results demonstrate that GPU-accelerated libraries can provide substantial speedups in FFT calculations compared to CPU based implementations. This is especially true for batched transforms or transforms of higher dimensions. While our work is limited to a specific set of hardware and software configurations and small subset of algorithms, our findings can serve as a starting point for further exploration of GPU-accelerated signal processing in Python.

List of Acronyms

AI artificial intelligence
AOT ahead of time
API application programming interface
AST abstract syntax tree
AVX advanced vector instructions

BLAS basic linear algebra subprograms

CPU central processing unit

DFT discrete Fourier transform
DIF decimation in frequency
DSP digital signal processor
DtoH device to host

FFT fast Fourier transform
FLOPS floating point operation per second
FMA fused multiply-add
FPGA field programmable gate array
FPU floating point unit

GIL global interpreter lock
GLSL OpenGL shader language
GPC GPU processing cluster
GPGPU general-purpose computing on graphics processing units
GPU graphics processing unit
GTC GPU technology conference

HIP heterogeneous interface for portability
HLSL high level shader language
HPC high performance computing
HtoD host to device

JIT just in time

LLVM Low Level Virtual Machine

MKL Intel math kernel library
Mutex mutually exclusive

NVML Nvidia management Library
NVRTC Nvidia runtime compiler

OpenCL open computing language

PEP Python enhancement proposal
PTX Parallel Thread Execution

ROCm Radeon open compute platform

SIMD same instruction, multiple data
SM streaming multiprocessor

TPC texture processing cluster
TPU tensor processing unit

1 Introduction

Python is a high-level programming language, which has become very widespread in recent years and can currently be considered to be the most widely used general programming language [1]. Python owes its popularity to its readability, conciseness, vast ecosystem of modules and low barrier of entry. It has been widely adopted in many fields of research and science. This use case in science and engineering fostered and benefitted greatly from what is often called the "scientific Python stack", a set of essential packages, which enable the fast execution of common numerical tasks. The package Numpy [2] is the most influential of those packages, since it defines a general and versatile class for arrays of arbitrary dimension: `ndarray`[3].

1.1 Properties and Limitations of CPython

The advantages of Python such as its dynamic-typing and high level of abstraction come at a cost. This section discusses this cost by examining the CPython version of Python. It is the most popular implementation of Python and is usually considered the reference implementation. Before Python code is run, it is compiled into so-called Python bytecode, which is then interpreted by the Python interpreter. This turns the simple multiplication of two numbers, which could otherwise be expressed in a single x86 CPU instruction, into a lengthy process of checking which types are involved and garbage-collection.

For example a function which multiplies two Python objects and the bytecode it gets compiled to is displayed in the following Listing 1.1. The produced bytecode can be inspected using the `dis`-module [4]:

```
1 def multiply(a,b):
2     return a*b
```

```
1 2       0 LOAD_FAST                0 (a)
2         2 LOAD_FAST                1 (b)
3         4 BINARY_MULTIPLY
4         6 RETURN_VALUE
```

Listing 1.1: Function definition and corresponding bytecode

The bytecode shown on the right is interpreted by code defined in CPython's `ceval.c`, which is located at the core of the interpreter and consists of a very large switch-case-statement that determines how to handle the individual commands like `LOAD_FAST` and `BINARY_MULTIPLY` (so-called opcodes) [5].

Listing 1.2 shows the interpreter code executed for `BINARY_MULTIPLY` in detail. In this case, it deals with the multiplication between the objects `a` and `b`. It takes the two values using `POP`, which removes the uppermost object from the stack and returns a pointer to that object and `TOP`, which retrieves a pointer to the uppermost object on the stack. The pointers were stored on the stack by the preceding `LOAD_FAST` op-codes. Subsequently the result is calculated by calling `PyNumber_Multiply` (displayed in Listing 1.2), which is defined in CPython's `abstract.c`. `PyNumber_Multiply` is actually a misnomer because it handles the multiplication of all Python-objects, not just numbers. Finally the result is stored on top of the stack using `SET_TOP`, which does not grow the stack, but overwrites the top with a new value. So after calling `POP`, `TOP` and `SET_TOP`, the original arguments of the functions, which were on top of the stack are now no longer on the stack at all. The op-code `RETURN_VALUE` then retrieves it from the top of the stack and returns it back to the caller of the function.

```
1618 case TARGET(BINARY_MULTIPLY): {
1619         PyObject *right = POP();
1620         PyObject *left = TOP();
1621         PyObject *res = PyNumber_Multiply(left, right);
1622         Py_DECREF(left);
1623         Py_DECREF(right);
1624         SET_TOP(res);
1625         if (res == NULL)
1626                 goto error;
1627         DISPATCH();
1628 }
```

Listing 1.2: Excerpt from CPython's source code (ceval.c from Python 3.9 [5])

`PyNumber_Multiply` in turn calls the function `binary_op1` [6, lines 848-881 in abstract.c] and passes it as arguments the two PyObjects to be multiplied and the slot where the desired binary operation is located.

```
1046 PyObject *
1047 PyNumber_Multiply(PyObject *v, PyObject *w)
1048 {
1049         PyObject *result = binary_op1(v, w, NB_SLOT(nb_multiply));
1050         if (result == Py_NotImplemented) {
1051                 PySequenceMethods *mv = Py_TYPE(v)->tp_as_sequence;
1052                 PySequenceMethods *mw = Py_TYPE(w)->tp_as_sequence;
1053                 Py_DECREF(result);
1054                 if  (mv && mv->sq_repeat) {
1055                         return sequence_repeat(mv->sq_repeat, v, w);
1056                 }
1057                 else if (mw && mw->sq_repeat) {
1058                         return sequence_repeat(mw->sq_repeat, w, v);
1059                 }
1060                 result = binop_type_error(v, w, "*");
1061         }
1062         return result;
1063 }
```

Listing 1.3: Excerpt from CPython's source code (`abstract.c` from Python 3.9 [6])

The function `binary_op1` finally calls the appropriate function the objects point to for multiplication. Objects containing floating-point numbers for example point to the function in Listing 1.4. Note that all floating point numbers in Python are always double precision floating point numbers and all integers are always of infinite precision. Additionally, everything in Python is an object and needs to be boxed within a PyObject as done on line 569 of Listing 1.4. As a consequence a floating point number in Python does not require 8 bytes of memory as expected from a double-precision floating point number, but 24

bytes because the PyObject has several other properties that need to be stored besides the double value it represents.

```
562 static PyObject *
563 float_mul(PyObject *v, PyObject *w)
564 {
565     double a,b;
566     CONVERT_TO_DOUBLE(v, a);
567     CONVERT_TO_DOUBLE(w, b);
568     a = a * b;
569     return PyFloat_FromDouble(a);
570 }
```

Listing 1.4: Excerpt from CPython's source code (`floatobject.c` from Python 3.9 [7])

Also, noteworthy are the calls to `Py_DECREF`, these calls indicate that the variables `left` and `right` (which are no longer on the stack and contained pointers to the function arguments `a` and `b`) are no longer needed in this context and that their respective reference counters should be decremented by one. Once the reference count of an object reaches zero, it will become de-allocated (referred to as being garbage-collected). This way of keeping track of objects being in use has far-reaching consequences for the performance of Python code, which will be discussed in Section 1.1.1.

By downloading the Python source code and compiling it with debug-options enabled, we can generate a debuggable binary and use it to step-through the interpreter's code as it is executed. This allows us to fully trace what happens on execution of the `BINARY_MULTIPLY` op-code. The interdependencies of functions involved is displayed in a condensed form in Figure 1.1.

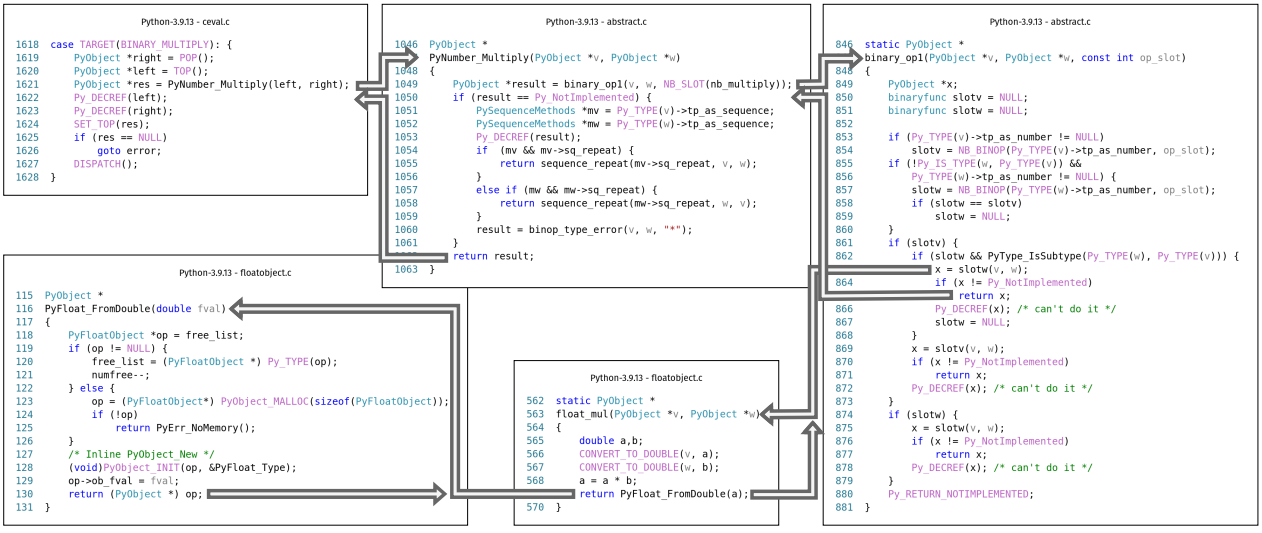


Figure 1.1: Call graph of the functions involved in the `BINARY_MULTIPLY` op-code executed for a pair of floating point objects

From this short exploration of CPython's source code it is already evident that multiplying two numbers in Python consists of more than a hundred lines of compiled C-Code and is certainly not as fast as the single x86 CPU-instruction, it ultimately boils down to (note the similarity between the highlighted line in

Listing 1.4 and line two in the left hand-side of Listing 1.6). A function that multiplies two floating point numbers in C or C++ and the resulting assembly code is displayed in Listing 1.6. The dynamic nature of the language, however, is also very powerful and allows Python to perform multiplication between all kinds of objects with the same function. If, for example, one of the factors of the multiplication can be interpreted as a sequence and the other one as a positive integer, the output is that sequence repeated by the value of the other factor (lines 1105-1116 in Listing 1.3), as demonstrated in Listing 1.5. This would not be possible to be expressed as concisely in a language like C/C++, where functions need to have typed signatures. Adding type-annotations to Python functions like specified in a Python enhancement proposal (PEP) (PEP 484 [8]), also does not change this behavior.

```
1 >>> multiply('Abc',5)
2 'AbcAbcAbcAbcAbc'
3 >>> multiply(5,'Abc')
4 'AbcAbcAbcAbcAbc'
```

Listing 1.5: Example of the function applied to non-numeric data

```
1 double multiply(double a, double b) {
2         return a * b;
3         }
```

```
1 multiply:
2         mulsd   xmm0, xmm1
3         ret
```

Listing 1.6: C function definition and corresponding assembly output for x86-64; compiled with clang 14.0.0 using compiler flag `-O2`

1.1.1 Concurrency in Python - The Global Interpreter Lock

Many programming languages offer the functionality to spread work over several threads to parallelize and effectively accelerate the execution of data. Python offers the threading-module [9] to perform tasks *concurrently* (in an inter-leaved fashion), but not to perform them in a truly parallel way (separate hardware performing separate tasks at the same time).

Assuming we have a list with two integer entries and would like to increment both entries a specified number of times. In a first step we will do this sequentially, as shown in Listing 1.7:

```
>> x = [0,0]
>> count = 100_000_000
>> for i in range(count):
>>         x[0] += 1
>> for i in range(count):
>>         x[1] += 1

>> display(x)

[100000000, 100000000]
```

Listing 1.7: Increasing the entries sequentially

This example took around 16 seconds to run, which equates to a counting frequency of 12.8 Mhz, not satisfactory considering CPU clock frequencies are ranging in the order of several Gigahertz. We investigate the threading-module and write code, which increases both values at concurrently as shown in Listing 1.8.

We define two threads, start them both with the `start()` function calls and then wait for them to conclude with the `join()` function calls. Without the `join()` function calls, the program would immediately display an intermediate result like `[316865, 160535]` without waiting for the two thread to finish their tasks of incrementing the values the specified number of times. Waiting for all threads to finish is a form of synchronization, which is an important concept in parallel computing.

```
>> def increase(var,idx, limit):
>>         for i in range(limit):
>>                 var[idx] += 1
>>         print(f"finished incrementing {idx}")

>> c = [0,0]
>> t1 = threading.Thread(target=increase, args=(x, 0, count))
>> t2 = threading.Thread(target=increase, args=(x, 1, count))
>> t1.start()
>> t2.start()
>> t1.join()
>> t2.join()

>> display(x)

finished incrementing 1
finished incrementing 0

[100000000, 100000000]
```

Listing 1.8: Concurrently increasing both entries

This code is only slightly faster at 13.7 seconds and this speed-up cannot be attributed to the fact that

we use two-threads. In fact the statement-sequence `t1.start()` → `t1.join()` → `t2.start()` → `t2.join()` would be exactly as fast without pretending to increment both values at the same time. The reason why one cannot achieve speedups of computationally intensive tasks by utilizing Python's threading library is the global interpreter lock (GIL). The GIL ensures that only one thread at a time can interpret Python bytecode. At first this might seem like an arbitrary restriction, but it is a measure that eases Python's internal bookkeeping of reference counts. If two threads were to modify the `refcnt` of a Python object at the same time, a so-called data-race (or race condition) might occur and the result could be incorrect. Bugs due to data-races can be notoriously hard to track down and debug [10].

The GIL might be removed from Python at some point in the future. Various attempts to remove it have been undertaken so far [11], [12].

The presence of the GIL, however, does not guarantee, that code utilizing the threading-module is free of data races. An example that displays a data-race is presented in Listing 1.9 If we task both threads with incrementing the first entry we would expect the entry to hold the value 200_000_000 after the following code concludes:

```
>> def increase(var,idx, limit):
>>         for i in range(limit):
>>                 var[idx] += 1
>>         print(f"finished incrementing {idx}")

>> c = [0,0]
>> t1 = threading.Thread(target=increase, args=(x, 0, count))
>> t2 = threading.Thread(target=increase, args=(x, 0, count))
>> t1.start()
>> t2.start()
>> t1.join()
>> t2.join()

>> display(x)

finished incrementing 0
finished incrementing 0

[125290242, 0]
```

Listing 1.9: Concurrently increasing the first entry

But we received an incorrect value of 125_290_242 and we would have yielded further, most-likely incorrect, values if repeatedly executed. The result of the code is non-deterministic. This happens because threads waiting to execute byte-code will request the currently running thread to be suspended (often referred to as releasing the GIL) after waiting a set amount of time (this amount of time can be set globally by calling `sys.setswitchinterval()`, by default the value is set to 5 milliseconds). After such a request has been raised, the currently running thread will not process any further op-codes.

To understand why this behavior is problematic in this situation, one needs to inspect the bytecode of the `increase`-function, which can be seen in Listing 1.10:

- At position 18 the instruction `BINARY_SUBSCR` loads the object from var[idx] onto the top of the stack.
- At position 20 the instruction `LOAD_CONST` loads the constant value 1 onto the top of the stack.
- At position 22 the instruction `INPLACE_ADD` adds the upper two elements of the stack together and puts the result onto the top of the stack, this calculates the value `var[idx] + 1`.
- At position 24 the instruction `ROT_THREE` rotates the top three elements of the stack, so the order of elements (index, container and value) are in the expected order for the next operation.
- At position 26 `STORE_SUBSCR` puts the calculated value at the correct position within the container.

```
2           0 LOAD_GLOBAL              0 (range)
            2 LOAD_FAST                2 (limit)
            4 CALL_FUNCTION            1
            6 GET_ITER
      >>    8 FOR_ITER                20 (to 30)
           10 STORE_FAST               3 (i)

3          12 LOAD_FAST                0 (var)
           14 LOAD_FAST                1 (idx)
           16 DUP_TOP_TWO
           18 BINARY_SUBSCR
           20 LOAD_CONST               1 (1)
           22 INPLACE_ADD
           24 ROT_THREE
           26 STORE_SUBSCR
           28 JUMP_ABSOLUTE            8

4     >>   30 LOAD_GLOBAL              1 (print)
           ... code related to the print statement omitted
           44 LOAD_CONST               0 (None)
           46 RETURN_VALUE
```

Listing 1.10: Bytecode of the increase-function

The section ranging from position 18 to 26 forms a critical section. If the thread is forced to release the GIL while in this area, the value of `var[idx]` might be changed by other threads and once this threads resumes to be active, it will not be aware of that fact and overwrite the entry with the wrong value [5], [13], [14].

It is also noteworthy how Python compiles our input into bytecode but does not perform any optimization at all, whereas other compiler-language combinations (like C compiled with Clang or GCC) would first transform our input into an intermediate representation and perform optimizations on that before emitting executable code. A possible optimization here would be to replace the for-loop with a `var[idx] += limit` statement. Python cannot perform these optimizations because at the time the `increase`-function is declared, nothing is known about the arguments it will receive. It could be an object of arbitrary type with an arbitrarily overloaded `__add__`-method.

One could still make the code work using a mutually exclusive (Mutex) lock guarding the critical section

(see Listing 1.11), which again is a common pattern in parallel computing, but does not at all help with performance in this case. To the contrary, it dramatically decreases performance, because now the two threads still demand the GIL from one another but cannot actually perform any work if they acquire the GIL but fail to acquire the lock. If that happens, the processor sits idle until the thread, which holds the lock gets another turn. Using this version of the `increase`-function causes the runtime to rise to 2 minutes and 17 seconds, albeit returning the expected, correct result.

```
1 lock = threading.Lock()
2
3 def increase(var,idx, limit):
4         for i in range(limit):
5                 lock.acquire()
6                 var[idx] += 1
7                 lock.release()
```

Listing 1.11: Increase function with a lock guarding the critical section

This example demonstrates why even such a simple task as counting can become difficult when using multiple threads in Python as well as in languages with truly parallel multi-threading, further discussions can be found by McKenney [15, Chapter 5].

1.2 Circumventing the Limitations

After having discussed the shortcomings (many layers of abstraction and indirection which are inefficient, the dynamic nature of the language which prevents compile-time optimizations and the GIL, which prevents compute-bound tasks from benefitting from multiple threads) of the CPython-interpreter for scientific computing, one might wonder how Python has become so popular in that field.

When looking at the sources of Python packages commonly used in scientific programming like Python it quickly becomes obvious that most of the offered computationally intensive functionality is not defined using Python code, but using C/C++ or Fortran code that is called from Python. Or in case of basic and ubiquitous building blocks of scientific programming like matrix-matrix-multiplications Numpy preferably delegates the computation to an existing basic linear algebra subprograms (BLAS) [16] compatible installation if possible [17].

BLAS-Libraries are generally very well optimized and tailored to specific hardware. In fact many hardware-vendors offer their own implementations of BLAS. For CPUs there is OpenBLAS [18], the Intel Intel math kernel library (MKL) [19] and BLIS (AMD's BLAS implementation) [20] to name just a few.

Those libraries offer a range of common operations in linear algebra such as (but not limited to) the following functions:

- vector operations (Level 1 - $\mathcal{O}(n)$)
  - $x = \alpha x$, scaling a vector
  - $y = \alpha x + y$, adding vectors
  - $\langle x, y \rangle$, dot-product between vectors
  - $||x||_2$, calculating the magnitude

- matrix-vector operations (Level 2 - $\mathcal{O}(n^2)$)
  - $x = \mathbf{A}x$, matrix-vector multiplication
  - $x = \mathbf{A}^{-1}x$, inverse matrix vector multiplication
  - $y = \alpha\mathbf{A}x + \beta y$, matrix-vector multiplication plus vector
- matrix-matrix operations (Level 3 - $\mathcal{O}(n^3)$)
  - $\mathbf{C} = \alpha\mathbf{A}\mathbf{B} + \beta\mathbf{C}$, General matrix-matrix multiplication
  - special cases for symmetric, triangular and hermitian matrices
  - $y = \alpha\mathbf{A}x + \beta y$, Matrix-vector multiplication plus vector
- and variations of those, see [16] for a complete list.

1.2.1 Comparing BLAS-Implementations

Naturally Numpy's performance for those operations is heavily dependent on the BLAS implementation it was linked against. When installing the binary distribution of Numpy from anaconda [21] it can be linked against either OpenBLAS or MKL, if installed from PyPI it is linked against OpenBLAS according to Numpy's documentation.

For the sake of comparing the default Numpy distribution, which uses OpenBLAS, against a Numpy-installation linked against a different library, we download the sources for both Numpy and AMD's version of BLIS (because the test is performed on AMD hardware) and compiled them. One has to be careful when doing so, because both Numpy and BLIS need some compilation configuration for best performance and one might end up without any BLAS acceleration in Numpy if done wrongly. This must be avoided since Numpy's fallback algorithms are not as sophisticated and do not claim to deliver comparable performance.

The following setups were compared:

- Blis compiled with openmp:

```
./configure --enable-cblas --enable-threading=openmp zen2
make
```

- Blis compiled with pthreads:

```
./configure --enable-cblas --enable-threading=pthreads zen2
make
```

- BLIS compiled with no multi-threading:

```
./configure --enable-cblas zen2
make
```

- Default Numpy installed from pip including OpenBLAS

- Numpy not linked against any BLAS library

Explanation of arguments:

- `-enable-cblas` tells BLIS to build the necessary interface to call it from c-code, which Numpy is subsequently able to utilize.
- `-enable-threading` is necessary to enable multithreading, one can choose between OpenMP [22] (a rather high level abstraction to use multiple threads using compiler pragmas around sections that can be run in parallel) or pthreads [23] (short for POSIX threads, a more low-level standard originating from Unix systems), which are the two most common ways of adding multi-threading capabilities to programs with OpenMP being the preferred option because it allows for fine-grained control over which thread is handled by which CPU-Core.
- `zen2` tells BLIS to optimize for AMD's Zen2 Architecture.

Checking whether all different implementations do in fact produce correct results is a non-trivial task, because usually Numpy's results are our trusted source of truth. So for verification purposes we performed the calculations on a set of random but reproducible matrices (setting the seed of the random number generator ahead of execution) with the known good Numpy distribution and saved the complete results as binary files, which could later be read and compared with the other results (using `numpy.all_close`).

Caution: We are testing multithreaded functions that deal with (double precision) floating point numbers. The order of operations is not deterministic and since addition of floating point numbers is not associative, the results may vary slightly from run to run and from implementation to implementation. Calculating hashes based on the result and comparing these hashes might seem easy and convenient at first but does not work due to the mentioned reasons.

Figure 1.2 and Table 1.1 show that there are significant differences between the two different BLAS implementations and that it can be very beneficial to check whether the used Numpy installation is actually using an optimized and correctly configured BLAS-backends. The BLIS backend we compiled from source with OpenMP and optimizations for the specific CPU-Architecture enabled was up to 40% faster on small matrices (100×100) and about 15% faster for larger matrices (4000×4000). It can also be seen that not multithreading the computation slows it down by a factor of 4, which is a speed-up within the expected range for a CPU with 6 cores, and we can conclude that the OpenBLAS library bundled with Numpy must also take advantage of multithreading to achieve its performance. In fact this can be verified by looking up the shared library dependencies of the `libopenblas` binary inside the directory containing the numpy extension using `ldd` (unix tool for **l**isting **d**ynamic **d**ependencies) and indeed we can recognize that is linked against pthreads. More important than having a multithreading-enabled BLAS-backend is having any correctly set-up at all. If Numpy cannot be linked against a BLAS library during the compilation process it will use the algorithms defined in its own C-codebase, which take anywhere between tens and hundreds times as long as the already suboptimal algorithms provided by OpenBLAS library in this example.

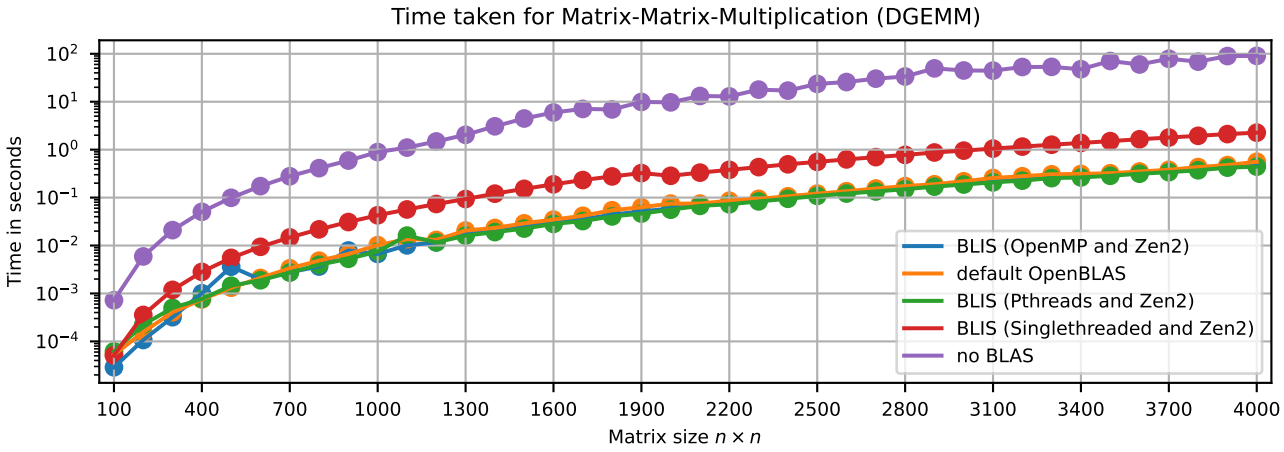


Figure 1.2: Result of comparing the different BLAS implementations

Note: This is not meant to be a comprehensive and fair benchmark comparing OpenBLAS and BLIS. It is a demonstration that displays the importance of the BLAS backend when using Numpy and highlights the fact that even large projects do not reinvent the wheel, but rely on existing and proven solutions for common problems.

BLAS Backend	Relative Time compared to OpenBLAS		
matrix sizes	100x100	600x600	4000x4000
OpenBLAS	1	1	1
BLIS OpenMP	0.54	0.927	0.835
BLIS pthreads	1.17	0.895	0.789
BLIS singlethreaded	0.96	4.463	4.038
no working BLAS installation	13.7	75.07	161.6

Table 1.1: GEMM-Performance using different BLAS Backends

1.2.2 Interfacing C-Code

As explored in the previous section Python relies on calls to functions and libraries implemented in other languages, usually low-level, compiled languages like C/C++ or Fortran, for enhancing its numeric computing capabilities. Because of this need to interface with code in written in other languages the Python ecosystem offers several packages which simplify this process or offer the functionality to take Python code and transpile it to one of those languages and compile it either just in time (JIT) or ahead of time (AOT).

The advantages of using compiled code are the ability to efficiently express algorithms in lower-level languages, which don't have to deal with the abstractions of the CPython interpreter and the ability to write multi-threaded code (since the interpreter is not involved, neither is the GIL) and compiled code greatly benefits from compile-time optimization techniques like auto-vectorization, loop-unrolling and function-inlining among others.

Because of the relative ease involving linking libraries written in other programming languages to Python, Python is often called a "glue language". Using this approach the Python language allows users to be

more productive than the low-level languages would, while still reaching C-level performance using C-Code, which is hidden from the user.

Tool commonly used to interface libraries written in C/C++:

- Cython [24]

  Cython (not to be Confused with CPython) is a programming language and compiler that allows users to annotate regular Python code with static type declarations. Cython then takes that code and compiles it to C and subsequently a binary shared library which is embedded into an importable module.

- Numba [25]

  Numba is a package that takes decorated Python functions and tries to compile them to LLVM-IR (an intermediate representation used in the low level virtual machine compiler framework) and subsequently binary code using the Low Level Virtual Machine (LLVM) compiler [26]. The LLVM compiler can target many architectures, it includes backends for Nvidia and AMD GPUs, which enables Numba to compile Python code to GPU Kernels.

- PyBind11 [27]

  PyBind11 is a package which allows users to easily write bindings for existing C/C++ code and make it accessible from within Python.

- Pythran [28]

  Pythran is a project that compiles Python Code to C++ Code, which is then compiled to a native binary. Not unlike Cython it is ahead of time compiled and produces an importable module. Whereas the Cython language is a superset of the Python language, Pythran supports a subset of Python and therefore ensures backwards-compatibility. Pythran code never interacts with the Python interpreter (this is optional in both Numba and Cython). Despite its name Pythran is unrelated to Fortan.

1.2.3 Case Study: Writing a matrix-multiplication algorithm using the mentioned libraries

In a further step we explore and compare the mentioned libraries by implementing the naive non-tiling matrix-matrix-multiplication algorithm. It should take two two-dimensional Numpy-arrays $\mathbf{A}, \mathbf{B}$ as arguments and return the result $\mathbf{AB}$.

1.2.3.1 Pure Python The simplest version of the algorithm would consist of three nested loops expressed using pure Python code, as shown in Listing 1.12. This is a very slow way of performing this task because of the reasons mentioned before.

```
def mat_mul(a,b):
        result = np.zeros((a.shape[0], b.shape[1]), dtype=np.float64)
        for x_dim in range(result.shape[0]):
                for y_dim in range(result.shape[1]):
                        for k in range(result.shape[0]):
                                result[x_dim,y_dim] += a[x_dim, k]*b[k,y_dim]
        return result
```

Listing 1.12: Matrix-Matrix Multiplication algorithm expressed using Python

1.2.3.2 Cython Cython (the programming language) is meant to be a superset of Python, i.e. almost all valid Python code is also valid Cython code, but not vice-versa. It allows users to annotate code with type annotations, which in this case means we can specify that the arguments passed to the function are two-dimensional Numpy-arrays. An example thereof is shown in Listing 1.13. Additionally, so-called typed memoryviews can be defined as visible on lines 7-9. These allow the compiled function to read and or write the memory areas underlying the PyObjects without interacting with the Python interpreter at all [29]. This enables Cython to compile the function to C-Code and in a further step to a callable native library.

The annotations require minimal modifications to the previous Python-code and the function achieves a significant performance gain. This is an acceptable trade-off as long as the types of the arguments and return value are known and fixed. If we also wanted a similar function for single-precision floating point numbers or integers, the whole code would have to be repeated with that modification either manually, which is bad for maintainability, or programmatically using a templating language (this is for example extensively done in the codebase of Numpy).

```
@cython.boundscheck(False)
@cython.wraparound(False)
def matmul_simple(numpy.ndarray[double, ndim=2] a, numpy.ndarray[double, ndim=2] b):

        result = np.zeros((arr_1.shape[0],arr_2.shape[1]), dtype=np.float64, order = 'C')

        cdef double[:, ::1] result_view = result
        cdef double[:, ::1] a_view = a
        cdef double[:, ::1] b_view = b

        for dim_x in range(result_view.shape[0]):
                for dim_y in range(result_view.shape[1]):
                        for k in range(result_view.shape[0]):
                                result_view[dim_x,dim_y] += a_view[dim_x,k]*b_view[k,dim_y]

        return result
```

Listing 1.13: Matrix-Multiplication using Cython

1.2.3.3 Cython using vector instructions and multithreading The strength of Cython lies within its ability to mix annotated Python code with user-written C-Code. In the example at hand we can replace the innermost loop that calculates dot-products between rows of $\mathbf{A}$ and columns of $\mathbf{B}$ with a more efficient function using AVX2-instructions. advanced vector instructions (AVX) and AVX2 are extensions to the x86-64 instruction set supported by recent Intel and AMD processors. These instructions enable the CPU to process 256-bit wide vectors (which fit 4 double precision floats) of data at a time with a single instruction. For example the instruction `_mm256_fmadd_pd` accepts three of those vectors, each containing four floating point numbers and calculates the following expression elementwise $a \times b + c$, with the vectors being called a, b and c respectively. This instruction is known as a fused multiply-add (FMA) instruction. Processing data in this way is called same instruction, multiple data (SIMD)-Programming because the same instructions are applied to multiple elements of data at once. Vector instructions however are not always easy to employ, they have high requirements to spatial memory-locality for optimal performance.

Figure 1.3 visualizes how arrays can be laid out differently in memory, which can have big impacts on performance. By default, Numpy creates row-major arrays, but this can be controlled with the `order` keyword argument of many array-creating functions in Numpy. CPUs retrieve memory contents one cacheline at a time, with a cacheline ranging in size from 32 to 256 bytes (64 bytes fitting 8 64-bit floats on the Zen 2 architecture)[30]. Knowing this we can see that calculating the dot-product between rows and columns of row-major aligned matrices is not very efficient, since consecutive elements of a column are located in memory locations far-apart for all but the smallest arrays. Transposing the second array or converting it using the Numpy function `asfortranarray` to minimize the cache misses before performing the inner loop dramatically increases performance.

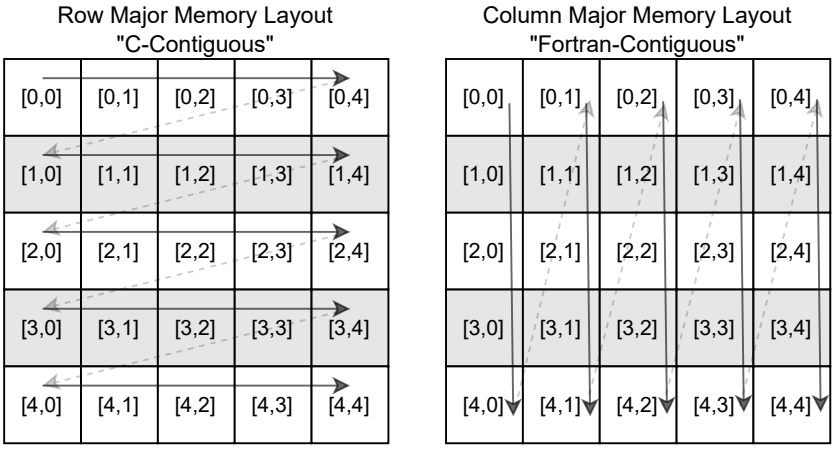


Figure 1.3: The two major ways of storing a 2D array in memory

Listing 1.14 show the adapted code with the innermost loop refactored into a specialized function, which calculates the dot-product. One also can see that the matrix $\mathbf{B}$ gets transposed (line 9) before a view into its memory is created to benefit from the changed layout in memory and that some memory `temp` is allocated (line 12) and a pointer to those memory regions passed to the dotproduct-function. The dotproduct-function will use this function to store temporary values in that memory, it is important to only allocate that memory once outside the function instead of allocating and freeing it in the function over and over again for performance reasons. It is also space for 12 times 64 bytes (12 whole cachelines) double-precision floating point numbers was allocated, even though we only ever want to store 12 256-bit (32-byte) AVX2 vectors. Dedicating a whole cacheline to each of the up to 12 threads improves performance, by avoiding false sharing. False sharing occurs when two or more threads try to access the

same cacheline at the same time, this is not possible and could lead to the program effectively being serialized [31].

Another change affects the outermost loop starting in line 17. Instead of `range` it now reads `prange(..., nogil=True)` , this indicates to Cython that the resulting for-loop in c-code should be annotated with an `omp for` directive, which means that this for-loop should be split across multiple threads (exact behavior can be controlled by various methods including environment variables). The `nogil=True` argument means that the global interpreter lock should be released for this section of the code, this means that no interaction with the Python interpreter can happen inside this section and that the Python interpreter could switch to other threads in the meantime.

This style of programming is far away from the high-level way of programming Python owes its popularity to. Additionally, manually managing memory allocations and pointers is an error-prone process that can lead to programs which either produce false results or outright crash. Automating the process of writing or relying on trusted, well-tested versions of these performance-critical low-levels parts of algorithms is therefore highly desirable.

```
1  @cython.boundscheck(False)
2  @cython.wraparound(False)
3  def matmul_transpose(numpy.ndarray[double, ndim=2] a, numpy.ndarray[double, ndim=2] b):
4
5      result = np.zeros((a.shape[0], b.shape[1]), dtype=np.float64, order = 'C')
6
7      cdef double[:, ::1] result_view = result
8      cdef double[:, ::1] a_view = a
9      cdef double[:, ::1] b_view = b.T.copy()
10
11     cdef double* temp
12     posix_memalign_wrapper(<void **> &inter, 64, 12*64)
13
14     cdef int dim_x
15     cdef int dim_y
16
17     for dim_x in prange(result_view.shape[0], nogil=True):
18         for dim_y in range(result_view.shape[1]):
19             result_view[dim_x,dim_y] = dot_prod_simd(
20                 &a_view[dim_x,0],
21                 &b_view[dim_y,0],
22                 result_view.shape[0],
23                 &temp[64*cython.parallel.threadid()]
24                 )
25
26     free(temp)
27     return result
```

Listing 1.14: Matrix-Multiplication using Cython with the innermost loop replaced

The function `dot_prod_simd` meanwhile is defined in a separate file and normal C-Code, unaware of its use in a Python context. This code can be seen in Listing 1.15. Explanation of the code:

- The instruction `_mm256_setzero_pd()` at line 7-9 initializes a 256-bit vector of packed doubles.
- The instruction `_mm256_load_pd(&arr_1[pos])` at line 13-14 loads 256-bit beginning from the specified memory address into a vector register.
- As mentioned earlier the instruction `_mm256_fmadd_pd(a0, b0, c0)` at line 15 performs a fused multiply add with the specified vectors ($\mathbf{c} = \mathbf{a} \cdot \mathbf{b} + \mathbf{c}$). This operation accumulates the result of the dot-product in the four entries within the $\mathbf{c}$ vector.
- The instruction `_mm256_hadd_pd(a, b)` at line 21 performs a horizontal add:

  $c[0] = a[0] + a[1]$

  $c[1] = b[0] + b[1]$

  $c[2] = a[2] + a[3]$

  $c[3] = b[2] + b[3]$

  This enables a way to calculate the sum of $\mathbf{c}$ more efficiently.
- The instruction `_mm256_store_pd(&inter[0], c0)` at line 22 stores the contents of $\mathbf{c}$ in the specified memory location.
- Finally the loop ranging from lines 26 to 30 takes care of the remaining elements that were not processed with the SIMD-instructions (this happens if the vector length is not a multiple of 4).

The downside of using AVX2-intrinsics is that the resulting code is less portable than before, since AVX2 instructions are an extension of x86-64, they are not supported by legacy hardware or different architectures like ARM or RISC-V, those architectures have their own set of vector instruction and the code might be easily portable, but it is not an automatic process. Trade-offs between code-portability and performance are a recurring theme in high-performance-computing.

```
double dot_prod_simd(const double *arr_1, const double *arr_2, unsigned int n, double *inter)
{
        unsigned int pos = 0;
        double *arr_1_ptr = arr_1;
        double *arr_2_ptr = arr_2;

        __m256d a0 = _mm256_setzero_pd();
        __m256d b0 = _mm256_setzero_pd();
        __m256d c0 = _mm256_setzero_pd();

        // equivalent for-statement: for(unsigned int i = 0; i < n/4;i++)
        while (pos + 4 <= 4 * (n / 4))
        {
                a0 = _mm256_load_pd(&arr_1[pos]);
                b0 = _mm256_load_pd(&arr_2[pos]);
                c0 = _mm256_fmadd_pd(a0, b0, c0);

                pos += 4;
        }

        c0 = _mm256_hadd_pd(c0, c0);
        _mm256_store_pd(&inter[0], c0);

        double result = inter[0] + inter[2];

        while (pos < n)
        {
                result += arr_1[pos] * arr_2[pos];
                pos++;
        }

        return result;
};
```

Listing 1.15: C-function for calculating the dot-product between vectors using SIMD-Instructions, abridged for readability

Tables 1.2 and 1.3 show the speed-up of using multiple threads (controlled by the environment variable `OMP_NUM_THREADS`) and unrolling the loop containing the AVX2-instructions. For this embarrassingly parallel problem performance scales very well with the number of threads, but manually unrolling the for-loop is also necessary to achieve maximum performance, apparently the compiler is unable to correctly unroll the loop (independently of whether we express the loop using a `while` or `for` statement) on its own.

We can recognize that large matrices benefit more from the parallelization and vectorization. Using all 12 threads and unrolling the loop speeds the calculation up by a factor of 5.7 for small matrices of size 100×100, while we can achieve a speedup of up to 13 for larger matrices like 1000×1000.

Matrix sizes: 100×100

# of cores	avx-count 1	2	3	4
1	1.00	1.37	1.46	1.60
2	1.59	1.89	1.93	1.94
3	2.22	2.62	2.64	2.66
4	2.85	3.37	3.39	3.36
5	3.43	4.00	4.07	4.07
6	3.88	4.52	4.57	4.56
7	4.20	4.83	4.88	4.97
8	4.12	5.18	5.16	4.99
9	4.81	4.67	5.41	4.74
10	5.08	4.84	4.86	5.81
11	5.17	5.86	5.81	5.80
12	4.78	5.81	5.35	6.00

Table 1.2: Relative speed-ups when using multi-threading and loop-unrolling

Matrix sizes: 1000×1000

# of cores	avx-count 1	2	3	4
1	1.00	1.95	1.93	2.65
2	1.79	2.78	2.87	2.82
3	2.67	4.07	4.14	4.13
4	3.54	5.42	5.44	5.46
5	4.36	6.65	6.72	6.76
6	5.16	7.92	7.97	7.98
7	5.98	9.05	9.16	9.00
8	6.58	10.05	10.38	10.16
9	7.52	11.14	10.90	11.22
10	8.21	12.08	12.28	12.33
11	8.69	12.23	13.15	13.18
12	8.48	11.92	12.57	12.35

Table 1.3: Relative speed-ups when using multi-threading and loop-unrolling

Having seen that utilizing multithreading and vectorization scales fairly well, we can compare our matrix-matrix-multiplication functions performance with that of Numpy's matmul. The results of that comparison can be seen in Figure 1.4. We can see that the version where we replaced the innermost loop with a C-function featuring AVX2-intrinsics is a lot faster than the most naive version and even slightly faster than BLAS for small matrices. It cannot compete with BLAS when tested with larger matrices because there are more sophisticated algorithms involved, namely blocked matrix-matrix multiplications, which optimize for fewer cache-misses. Those algorithms still have a time complexity of $\mathcal{O}(n^3)$, but make better use of the hierarchical nature of the memory-caches. Theoretically better algorithms like the Strassen-Algorithm are traditionally not found in BLAS implementations, because they are only superior for very large matrices [32]. A better asymptotic complexity does not necessarily translate to a better performance for all problem sizes, after all constant factors and terms of lower order are usually neglected when stating complexities (i.e. $\mathcal{O}(2n^2 + 4n)$ simplifies to $\mathcal{O}(n^2)$). The compiler failed to automatically vectorize the version written purely in Cython (same as in Listing **??**, except the outermost `range` was replaced with a `prange` to distribute the work among the available CPU-cores). Often flags like `-ffastmath` have to be passed to the compiler to indicate that the compiler is allowed to reorder operations on floating-point numbers, due to floating point number addition not being associative, which affects precision but is sometimes needed for vectorization. It does however not enable the compiler to vectorize the loop in this case.

1.2.3.4 Numba Numba takes a different approach to accelerating Python code. Unlike Cython it does not require a compilation step ahead of execution that has to be manually invoked, instead it only generates the required IR-Code and binaries once the function is called (JIT-compilation). This allows Numba to know the types of the passed arguments before compilation without the user explicitly defining them beforehand. The downside of this approach is that the first invocation of the function is slowed down due to the necessary compilation.

Numba works by decorating existing Python function. Decorators in Python are higher order functions, they take functions, transform them in some way and return the transformed functions. In case of Numba

the decorator ingests the Python function and analyzes its bytecode (using Python's `dis` package as discussed in Section 1.1) and tries to compile it to LLVM-IR and subsequently to native binary code.

```
@numba.jit(nopython=True)
def matmul_numba(a,b):
  result = np.zeros((a.shape[0], b.shape[1]), dtype=a.dtype)
  for x_dim in numba.prange(result.shape[0]):
        for y_dim in range(result.shape[1]):
          for k in range(result.shape[1]):
                result[x_dim, y_dim] += a[x_dim,k]*b[k,y_dim]

  return result
```

Listing 1.16: Matrix-Matrix multiplication example using Numba

Numba offers to dump the generated intermediate representations and assembly code for user inspection. However, interpreting these outputs is not always practical since they are not meant to be human readable and require some time and effort to reason about. Therefore Numba can, at times, appear to be a black box, with the generated code's performance being hard to predict given only the decorated function. users can only tweak the decorated function, benchmark the resulting native code and observe which changes improve performance and which ones do not, unlike working with Cython where it is possible to rewrite critical sections directly in C.

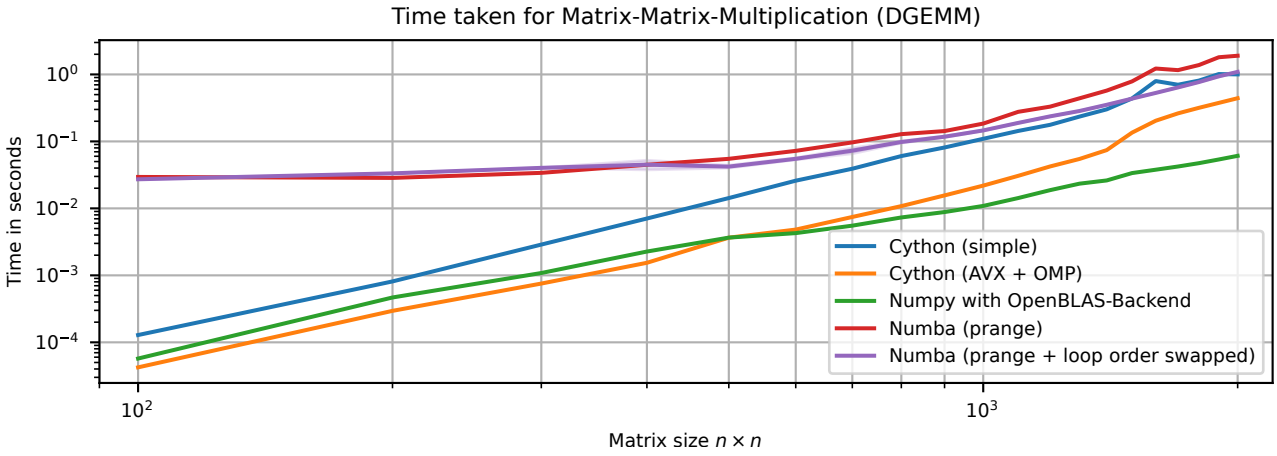


Figure 1.4: Result of comparing the different matrix-matrix-multiplication implementations

Figure 1.4 shows that the Numba functions seem to have a significant startup-overhead associated with them, which prevent them to be competitive when dealing with small matrices. We can rule out that this overhead is the time it takes Numba to jit-compile the function. The times used to create the plot were the lowest runtimes measured several runs. The jit-compile penalty however is only present the first time a Numba function is called with a new function signature and is about 0.6 seconds (effectively another order of magnitude larger than the fastest runtimes recorded for Numba functions) for this function. The figure does not include the pure Python implementation, as introduced in Section 1.2.3.1, since

presentating that version as a baseline would be disingenuous and misleading. That version is never used in practice and would be slower by a factor of approximately 400.

For larger matrices Numba only manages to compete with the most naive Cython implementation and is about an order of magnitude slower than the Cython version using AVX-intrinsics, even though the LLVM-compiler backend Numba is using would be free to also use vector instructions, it just fails to do so to the same extent.

1.2.3.5 Pybind11 Similarly to Cython, Pybind11 can be used to interface with arbirary C/C++-Code. Listing 1.17 for example shows how BLAS functions like `dgemm` (Double Precision General Matrix Multiplication) can be wrapped. The function takes two Numpy arrays `arr1` and `arr2`, requests information about their shapes and memory locations and passes that information to the `cblas_dgemm` function.

The application programming interface (API) CBLAS offers is very versatile, but a lot less comfortable to use than one would expect coming from high-level languages like Python. This is in part due to the fact that arrays in C are just pointers to the beginning of the memory location storing that array. A function that is passed one of such pointers has no way of knowing the size of the allocated memory block or the shape of the matrix that block of memory could represent. We therefore have to tell `cblas_dgemm` whether the used matrices are row-major aligned or column major aligned in memory and how many rows and columns they consist of. The performance of this approach is comparable to that of Numpy as discussed in Section 1.2.1.

Just as with Cython we have to take care of manually allocating and de-allocating memory on the heap with `malloc` and `free` or `new` and `delete` in the C/C++-Code. In this example this is done by the so-called capsule defined on lines 23 to 25. The code defined in that capsule is executed when the associated Python object is garbage collected. This is necessary to avoid memory-leaks. Pybind11's documentation on how to best return newly created Numpy arrays is unfortunately lacking and most information regarding this topic can only be found in an issue at GitHub [33]. This seems to be solved better in Cython where new Numpy arrays can simply be instantiated by calling `np.zeros` or similar functions, this however comes at a higher overhead than `malloc` and requires the GIL to be held.

```
1  py::array_t<double> dgemm_blis(pybind11::array_t<double> arr1, pybind11::array_t<double> arr2)
2  {
3    pybind11::buffer_info a = arr1.request();
4    pybind11::buffer_info b = arr2.request();
5
6    double *C = new double[a.shape[0] * b.shape[1]];
7
8    cblas_dgemm(CblasRowMajor,
9                CblasNoTrans,
10               CblasNoTrans,
11               a.shape[0],
12               b.shape[1],
13               a.shape[1],
14               1.0,
15               (double *)(a.ptr),
16               a.shape[1],
17               (double *)(b.ptr),
18               b.shape[1],
19               0.0,
20               C,
21               b.shape[1]);
22
23   py::capsule free_when_done(C, [](void *f){
24             double *C = reinterpret_cast<double *>(f);
25             delete[] C; });
26
27   return py::array_t<double>(
28       {a.shape[0], b.shape[1]},
29       (const double *)C,
30       free_when_done);
31 }
```

Listing 1.17: C++-function for calculating the dot-product between vectors using SIMD-Instructions, abridged for readability

Since Pybind11 (and Cython for that matter) can utilize arbitrary C/C++ Code, it can also be used to expose special hardware features to Python. To continue this example of matrix-matrix-multiplication one can also use Nvidia's cuBLAS to offload the calculation to a connected CUDA capable GPU. This is demonstrating in the following commented Listing 1.18:

- On lines 13-15 memory for the three arrays is allocated on the device memory.
- On lines 17 and 18 the memory contents of `arr_1` and `arr_2` are copied to device memory.
- On line 22 cuBLAS' version of DGEMM is called and unlike BLIS' DGEMM it always assumes arrays to be laid out in column major order in memory. Since we copied row-major-ordered matrices into the memory, but cuBLAS interprets them as column-major they are effectively transposed.

  We would like to calculate:

$$\mathbf{AB} = \mathbf{C}$$

but we only have $\mathbf{A}^T$ and $\mathbf{B}^T$ in column-major-ordered memory, instead of inefficiently re-ordering the matrix entries to get $\mathbf{A}$ and $\mathbf{B}$ in column-major-order, we can also calculate

$$\mathbf{B}^T\mathbf{A}^T = (\mathbf{AB})^T = \mathbf{C}^T.$$

The resulting $\mathbf{C}^T$ is also in column-major-order and therefore exactly the matrix $\mathbf{C}$ we wanted to calculate in row-major-order. It can be seen that the fact that the cuBLAS interface expects column-major-ordered matrices adds another level of complexity and additional opportunities for possible errors to occur.

- After calling DGEMM we don't need the matrices $\mathbf{A}$ and $\mathbf{B}$ on the GPU memory anymore and can free the memory using a `cudaFree` call.
- On line 39 host memory is allocated for storing the result of the calculation.
- On line 41 transfer the memory is transferred from the Device to the Host Memory.
- After copying the results from device memory to host memory, the memory containing $\mathbf{C}$ on the device can be freed as done on line 42.

```
1  template <typename T>
2  py::array_t<T> gemm_cublas(pybind11::array_t<T> arr1, pybind11::array_t<T> arr2)
3  {
4    cublasHandle_t handle;
5    cublasCreate(&handle);
6
7    pybind11::buffer_info a = arr1.request();
8    pybind11::buffer_info b = arr2.request();
9
10   T *devPtrA;
11   T *devPtrB;
12   T *devPtrC;
13
14   cudaMalloc(&devPtrA, a.shape[0] * a.shape[1] * sizeof(T));
15   cudaMalloc(&devPtrB, b.shape[0] * b.shape[1] * sizeof(T));
16   cudaMalloc(&devPtrC, a.shape[0] * b.shape[1] * sizeof(T));
17
18   cudaMemcpy(devPtrA, a.ptr, a.shape[1]*a.shape[0]*sizeof(T), cudaMemcpyHostToDevice);
19   cudaMemcpy(devPtrB, b.ptr, b.shape[1]*b.shape[0]*sizeof(T), cudaMemcpyHostToDevice);
20
21   cublasHandle_t handle;
22   cublasCreate(&handle);
23
24   pybind11::buffer_info a = arr1.request();
25   pybind11::buffer_info b = arr2.request();
26
27   T *devPtrA;
28   T *devPtrB;
29   T *devPtrC;
```

```
30
31    cudaMalloc(&devPtrA, a.shape[0] * a.shape[1] * sizeof(T));
32    cudaMalloc(&devPtrB, b.shape[0] * b.shape[1] * sizeof(T));
33    cudaMalloc(&devPtrC, a.shape[0] * b.shape[1] * sizeof(T));
34
35    cudaMemcpy(devPtrA, a.ptr, a.shape[1]*a.shape[0]*sizeof(T), cudaMemcpyHostToDevice);
36    cudaMemcpy(devPtrB, b.ptr, b.shape[1]*b.shape[0]*sizeof(T), cudaMemcpyHostToDevice);
37
38    const double alpha = 1.0;
39    const double beta = 0.0;
40
41    gemm_cuda(handle,
42              CUBLAS_OP_N,
43              CUBLAS_OP_N,
44              b.shape[1],
45              a.shape[0],
46              a.shape[1],
47              &alpha,
48              devPtrB,
49              b.shape[1],
50              devPtrA,
51              a.shape[1],
52              &beta,
53              devPtrC,
54              b.shape[1]);
55
56    cudaFree(devPtrA);
57    cudaFree(devPtrB);
58
59    T *C = new T[a.shape[0] * b.shape[1]];
60
61    cudaMemcpy(devPtrC, C, b.shape[1]*a.shape[0]*sizeof(T), cudaMemcpyDeviceToHost);
62    cudaFree(devPtrC);
63
64    py::capsule free_when_done(C, [](void *f){
65              double *T = reinterpret_cast<T *>(f);
66              delete[] C; });
67
68    return py::array_t<double>(
69        {a.shape[0], b.shape[1]},
70        (const T *)C,
71        free_when_done);
72  }
```

Listing 1.18: C++-function wrapping cuBLAS' DGEMM

Notice how the C++ templating system can be used to express the function in a type-agnostic way with the type `T` being used as a placeholder type. The function `gemm_cuda` (called in line 23, defined as shown in Listing 1.20), which is overloaded to handle both single and double precision floating point

numbers, then goes on to invoke the correct cublas-kernels. Employing the templating functionality in this fashion minimizes code-duplication.

The two resulting versions of the function can then be registered as functions of the Python module that will be compiled using pybind11 as shown in Listing 1.19. The exposed functions can then be imported in a Python script and offer the functionality to multiply two matrices similar to Numpy's matmul (albeit more basic since Numpy's version offers sophisticated options like treating matrices of orders higher than two as a stack of two-dimensional matrices) but using GPU-acceleration.

Figure 1.5 shows the result of comparing the performance of the pyBind11 wrapped cuBLAS implementation against Numpy's matmul function. The comparison was done using matrices of sizes ranging from 100x100 to 3000x3000. The times for cuBLAS include the time for copying the matrices from host memory to device memory and back (the function is used as defined in Listing 1.18). Despite that overhead the cuBLAS implementation is significantly faster than Numpy's matmul function for single precision inputs and about as fast for double precision inputs. This example introduces the GPU as a suitable device for offloading computationally expensive tasks. In the following chapters we will examine how this works in detail, how the GPU can be used to execute arbitrary code, why the single precision calculation was faster and how algorithms need to adapted for good performance on this specialized hardware.

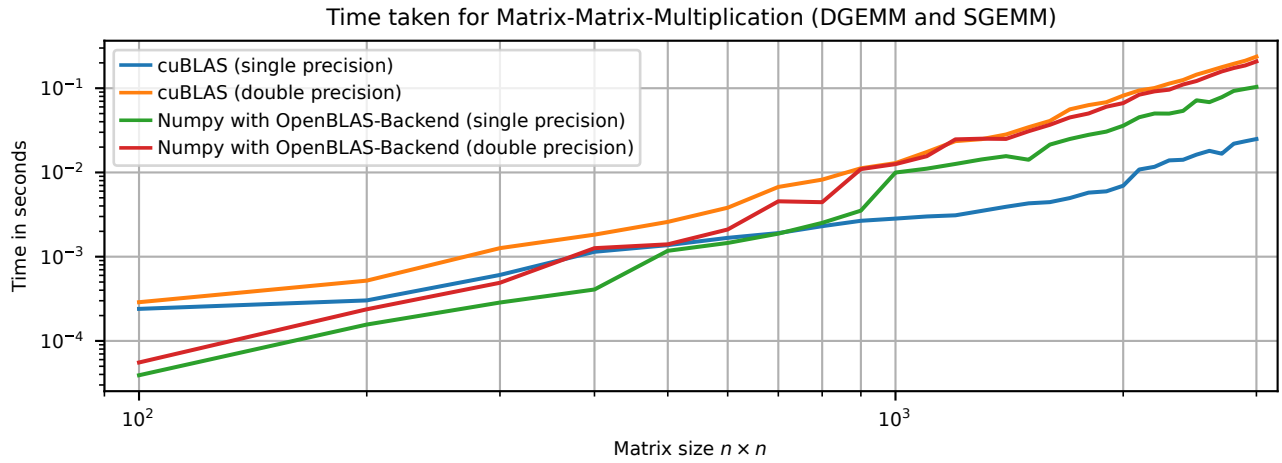


Figure 1.5: Result of comparing Numpy against the pyBind11 wrapped cuBLAS implementation

```
1 PYBIND11_MODULE(gpu_library, m)
2 {
3   m.def("gemm_cublas", &gemm_cublas<double>, py::arg("arr1").noconvert(),
  ↪  py::arg("arr2").noconvert());
4   m.def("gemm_cublas", &gemm_cublas<float>, py::arg("arr1").noconvert(),
  ↪  py::arg("arr2").noconvert());
5 }
```

Listing 1.19: Registration of the resulting Python extension's member function

```
/* ---
wrapper for DGEMM (double-precision general matrix-matrix multiplication)
--- */
void inline gemm_cuda(cublasHandle_t handle,
          cublasOperation_t transa,
          cublasOperation_t transb,
          int m, int n, int k,
          const double *alpha,
          const double *A, int lda,
          const double *B, int ldb,
          const double *beta,
          double *C, int ldc)
{

  cublasDgemm_v2(handle, transa, transb, m, n, k, alpha, A, lda, B, ldb, beta, C, ldc);

}

/* ---
wrapper for SGEMM (single-precision general matrix-matrix multiplication)
--- */
void inline gemm_cuda(cublasHandle_t handle,
          cublasOperation_t transa,
          cublasOperation_t transb,
          int m, int n, int k,
          const float *alpha,
          const float *A, int lda,
          const float *B, int ldb,
          const float *beta,
          float *C, int ldc)
{

          cublasSgemm_v2(handle, transa, transb, m, n, k, alpha, A, lda, B, ldb, beta, C, ldc);

}
```

Listing 1.20: Overloaded wrappers for `dgemm` and `sgemm`

2 GPGPU

After having outlined the limitations of Python and methods to overcome those by calling native compiled code this chapter focuses on the basics of utilizing the computing capabilities offered by the graphics processing unit (GPU) for general computations. The previous example (Listings 1.18 to 1.20) introduced this concept for matrix multiplication, which is a common operation in scientific computing. This chapter follows this approach further and provides an overview on how general-purpose computing on graphics processing units (GPGPU) works in principle, how algorithms can be adapted to be executed efficiently on these specialized many-core devices instead of sequentially on a CPU. Chapter 3 will then go on to discuss how to utilize the GPU's capabilities in Python, relating back to the first chapter.

2.1 Hardware structure of a GPU

Figure 2.1: Block diagram of the GA104 chip used in the RTX 3070 (as published in Nvidia's whitepaper on the ampere microarchitecture [34])

GPUs, which were traditionally used exclusively for rendering graphics to screens, have evolved to feature an ever-increasing number of processing cores and memory. The processing cores of GPUs are usually referred to as streaming multiprocessors (SMs) or shader cores and in comparison to CPU cores rather simple.

Figure 2.1 schematically displays the components of a modern GPU. It is connected to a host via a PCI Express interface and to its own, dedicated memory via several memory controllers. Host is the typical

term for the computer which controls the GPU, a single host can manage several GPUs. The GPU's dedicated memory usually ranges in capacity from a few to a few dozens of gigabytes distributed over several RAM-chips, which are placed in close proximity to the main chip of the GPU. The chip further includes 6 GPU processing clusters (GPCs), which in turn each contain multiple texture processing clusters (TPCs), finally these TPCs contain the SMs (their structure is outlined in Figure 2.2) and on-chip storage (like the L2-cache and lower levels of cache within the SMs).

The hierarchical nature of this design is of great importance when using these devices. The device can only communicate with the host over the relatively slow PCI-Express interface. Access to its own off-chip memory (typically referred to as global memory, because it is shared by the whole chip) is faster by several orders of magnitude, the on-chip memory is faster by another few orders of magnitude but very limited in size. Efficient programs have to take these differences in bandwidth and access latency into account.

Figure 2.2: Block diagram of the SMs within the GA104 chip used in the RTX 3070 (as published in Nvidia's whitepaper on the ampere microarchitecture [34]))

Each SM, as depicted in Figure 2.2, is divided into four so-called partitions, a ray-tracing core (not relevant for computing purposes), 128 kilobytes of shared memory and two FP64 floating point units (FPUs), which are omitted from the diagram supplied by Nvidia. These partitions each have two datapaths, one that can execute 16 FP32 instructions per clock and one that can execute either 16

FP32 or 16 INT32 instructions per clock. To summarize, each SM can execute up to 128 FP32 or two FP64 floating point operations per clock-cycle.

With the chip having a maximum clock-rate of 1725 Mhz one can calculate it's maximum compute performance:

$$\text{FLOPS}_{\text{max,FP64}} = 1725\ \text{MHz} \cdot 46\ \text{SM}_{\text{count}} \cdot 2\frac{\text{FLOP}}{\text{cycle}} \cdot 2 = 317 \times 10^9 \frac{\text{FLOP}}{s}$$
$$\text{FLOPS}_{\text{max,FP32}} = 1725\ \text{MHz} \cdot 46\ \text{SM}_{\text{count}} \cdot 128\frac{\text{FLOP}}{\text{cycle}} \cdot 2 = 20.3 \times 10^{12} \frac{\text{FLOP}}{s}$$

Note: the calculations includes the factor 2 to account for the GPU's ability to perform a fused-multiply-add (i.e. two floating point operations) in a single instruction and cycle.

This $64:1$ disparity between single and double-precision performance might seem like a large limitation in the usefulness of consumer-grade GPUs for scientific workloads, but it is still an appealing proposition, even for workloads requiring double-precision, as many algorithms are bandwidth limited (see Section 2.6 for an explanation of the roofline-performance model) and in many cases double precision not required. Alternatively, there are a number of techniques to achieve extended levels of precision by using multiple lower precision floating point numbers to represent a single number. Those techniques include Kahan summation, Knuth summation and others as discussed by Thall [35].

2.2 The CUDA Programming Language and Model

Traditionally GPUs were only used for rendering graphics to screens. These graphics related tasks consist of operations, which are performed on a large number of vertices or pixels at the same time. This drove the development of modern GPUs, which are now capable of performing many operations in parallel.

These operations are defined by shaders, which are programmed using so-called shader languages - there are various competing languages like the high level shader language (HLSL) or the OpenGL shader language (GLSL), but they all have similar C-like syntax. A vertex-shader for example displaces the vertices of a 3D-model depending on their position and the current time, which could be used to mimic waves in water by displacing the vertices as governed by a sine-function. The website shadertoy.com [36] showcases the possibilities of such shaders and their (often remarkably concise) source-code, which can be edited and re-compiled immediately to see the effects of modifications.

Around 2005 researchers started to employ GPUs for non-graphics uses like performing simulations and linear-algebra. Initially programs were written in shader languages and extensive knowledge about computer graphics was needed as laid out in a chapter on how to map problems onto GPU by M. Harris [37].

In 2007 Nvidia introduced CUDA [38], back then the term was an abbreviation for "computational unified device architecture", now it is often just the umbrella term for Nvidia's GPGPU offerings. CUDA encompasses a wide suite of tools like profilers, debuggers, compilers, but it is also the common name for the programming languages used to write programs which utilize the GPU in this CUDA framework. CUDA is not the only possible way to achieve this.

There exist official bindings for C/C++, Fortran and Python. Note that the official Python bindings do not allow users to express computational kernels in Python code at the time of writing, but there are other efforts trying to achieve that; see Section 3.1 for a detailed discussion. The C/C++ implementation can be considered the reference implementation.

2.2.1 Competing Programming Languages and Vendors

There are currently three major GPU vendors: Nvidia, AMD, and Intel (Intel is only recently trying to establish itself in the higher-performance segments relevant for scientific GPGPU), Nvidia is the dominant contender in this industry with a market share amounting to 81% of shipped dedicated GPUs in Q4 of 2021 [39]. AMD meanwhile has a strong presence in the market for integrated designs with their offerings being integrated in recent versions of popular gaming consoles as Microsoft's XBOX One branded consoles as well as Sony's Playstation 4 and 5. AMD also enjoys success supplying supercomputers used for scientific computing with their GPUs. The newly built supercomputer "Frontier" at Oak Ridge National Laboratory, for example, which claims the top spot among supercomputers worldwide [40], is equipped with four purpose built AMD Instinct GPUs (similar to the commercially available AMD Instinct 250X accelerators) per node [41].

The mentioned hardware manufacturers offer different languages and software stacks to facilitate programming their hardware, with code not necessarily portable between hardware from different manufacturers. Nvidia's proprietary CUDA language for example only supports Nvidia hardware, whereas competing offerings like open computing language (OpenCL) by a consortium of Vendors or, the much later introduced, Radeon open compute platform (ROCm) heterogeneous interface for portability (HIP) by AMD aim to support both AMD- and Nvidia-GPUs and in the case of OpenCL even non-GPU accelerators like field programmable gate arrays (FPGAs), digital signal processors (DSPs) or traditional CPUs (with OpenMP as backend).

A short, incomplete overview of the different languages and their respective ecosystems:

- CUDA - the ecosystem (consisting of the language, compilers, profilers and debuggers) created by Nvidia
- OpenCL - a more portable C-like language supporting many devices
- ROCm HIP - a recent addition to the landscape of GPGPU-languages, very similar to CUDA using the same terminology aiming to ease the transition for developers
- SYCL - a higher level C++ abstraction, initially on top of OpenCL, recent versions can also target HIP (hipSYCL)

CUDA has several advantages over its competitors like a comprehensive manual, sophisticated tools for profiling and debugging, support for Linux and Windows (AMDs ROCm only supports Linux at the time of writing) and a level of maturity not yet reached by ROCm [42]. This thesis therefore focuses on the use of CUDA in conjunction with Nvidia hardware. Regardless of this, the introduced concepts and findings also apply to competing hardware programmed with different languages. AMD even offers a set of tools called hipify [43], which aims to automatically translate programs written using CUDA to HIP C++. This enables those programs to work on AMD hardware.

A more comprehensive overview of the compatibility relationship between accelerated computing platforms, languages and hardware of the three major vendors is offered as a table by Herten [44].

2.2.2 Example CUDA kernel

First we need to introduce essential CUDA terminology.

- `grid` - A (up to) three-dimensional grid of blocks
- `block` - A (up to) three-dimensional block of threads, a block is the largest unit that is guaranteed to be executed in parallel. A block can at most consist of 1024 threads. The order in which blocks are executed is generally not known. Inter-block synchronization is therefore not supported.

- `warp` - A group of 32 threads that execute in lock-step on a SM. The threads within a warp can either perform an action or idle. The term "warp-divergence" refers to the situation where some threads within a warp exit earlier or take a different branch, which should be avoided for optimal performance.
- `thread` - Each thread is located in a warp, within a block, within a grid.

All threads execute the same instructions, but they do not necessarily operate on the same data. This is referred to as same instruction, multiple data (SIMD). To achieve this threads need to have a way of inferring their position within their warp, block or within the whole grid. For this the following variables are always within scope in code that is executed on the device:

- `gridDim` - a tuple of 3 ints specifying the dimensions of the grid; the contents of this variable is the same for all threads
- `blockDim` - a tuple of 3 ints specifiyng the dimensions of the blocks; the contents of this variable is the same for all threads, all blocks within a grid have the same dimensions
- `warpsize` - the size of a warp, 32 for all CUDA-capable GPUs, might however change in the future and is therefore queryable at runtime
- `blockIdx` - a tuple of 3 ints, which identifies the block the executing thread is in, values ranging from 0 to gridDim.$\{x|y|z\} - 1$
- `threadIdx` - a tuple of 3 ints, which identifies the executing thread within its block, values ranging from 0 to blockDim.$\{x|y|z\} - 1$

These variables can then be used by the threads to identify their share of work to be done at run-time. If, for example, one wants to program a kernel which increments every element of a vector by a given value that kernel could be implemented as seen in Listing 2.1.

```
1  __global__ void increment(float *input, int size, float x)
2  {
3          // tid uniquely identifies the thread among all threads
4          int tid = blockDim.x * blockIdx.x + threadIdx.x;
5
6          // do nothing if thread-id is outside of valid range
7          // accessing memory that is out of bounds can lead to segmentation faults
8          if(tid < size){
9                  input[tid] += x;
10         }
11 }
```

Listing 2.1: Example of a kernel which increments the values of an array

2.3 Occupancy

Whenever a kernel is launched the blocks within a grid are distributed and scheduled across the streaming multiprocessors of the GPU. To distribute this in an optimal way the blocks have to be shaped in a certain way. If, for example, a block consists of 1024 threads one can at most fit a single of those block on an SM at a time and reach a maximum occupancy of 66.67%. On the GPU we used throughout this thesis, one can therefore fit 70656 threads at a time (1536 threads per SM across 46 SMs). As a consequence

of this the GPU is only used to its potential when a workload consisting of hundreds of thousands of threads is launched. Small workloads are not well suited for GPGPU.
The number of threads a SM can fit at a time is however not the only constraint. There are also limitations regarding the shared memory per SM (the memory region indicated in Figure 2.2 is limited to a certain size) and the use of registers per SM, as well as the maximum use of registers per thread. The number of registers per thread or block is often referred to as register-pressure.
Higher occupancy is not necessarily always better, because there are often trade-offs involved like storing fewer data in sharing memory, which in turn might require more slow reads from global memory. Maximizing occupancy without compromising other performance aspects is a crucial step in optimizing performance, particularly for compute-bound kernels that benefit the most from a higher number of active threads.
Note: Not all blocks resident on a SM have to belong to the same grid or kernel. It's possible to launch one kernel with a blocksize of 1024 and another kernel with a blocksize of 512 and achieve perfect occupancy (given the need of other limited resources is also within the constraints).

2.4 Contextualizing Amdahl's Law

Amdahl's famous law [45] puts a limit to the benefit of distributing tasks over a number n of processors with r_s being the fraction of the program that has to be executed in a strictly serial fashion and can't be parallelized and r_p being the fraction of the program that benefits from parallelization; in this model r_s and r_p add up to one.

$$\text{Speedup} = \frac{1}{r_s + \frac{r_p}{n}}$$

This model shows that the ratio of $\frac{r_p}{r_s}$ has to be rather high for a meaningful speedup to be had. In all cases the speedup is limited by $\frac{1}{r_s}$. This model is simplified in the sense that it does not take into account the overhead of distributing the work to the processors or the overhead of communication between the processors. In practice the costs associated with these overheads can be significant and can even cause a slowdown as the problem is distributed among an excessive number of workers.
When offloading work to a specialized accelerator with a dedicated memory such as a GPU the parallelizable part of the problem can usually be solved very efficiently and quickly, but this comes at the cost of transferring the data to and from the accelerator. This memory copy operations can be interpreted as being part of the non-parallelizable parts of the program r_s which limit the speed-up.
Profiling tools like Nvidia's Nsight Compute and Nsight Systems [46] allow developers to profile and analyze code as it is executed on GPUs.

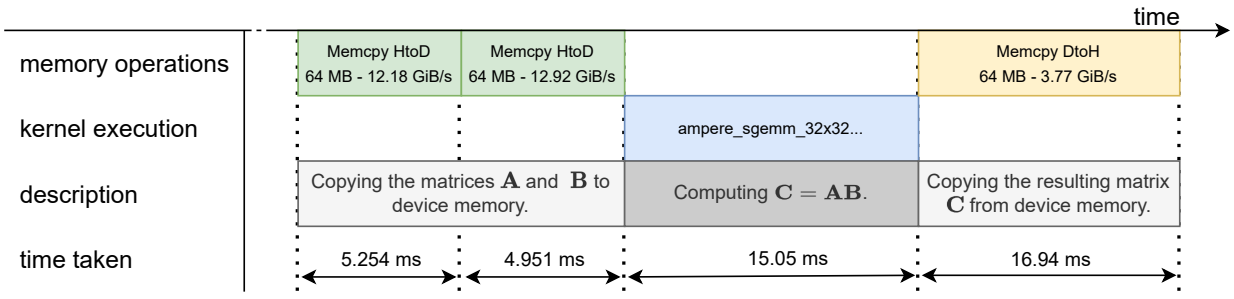


Figure 2.3: Timeline resulting from profiling the cuBLAS sgemm-kernel with Nsight Systems (the time axis in this illustration does not use a consistent scale)

Figure 2.3 depicts the timeline of the execution of a cuBLAS sgemm-kernel for inputs of size 4000×4000. In a first step both input matrices are copied to the device memory, this is abbreviated with host to

device (HtoD). Since the matrices are of size 4000×4000 and each element being 32 bits large (single precision) the total size of each of the three involved matrices in bytes is $4000 \cdot 4000 \cdot 4$ byte $= 64$ MB. Then the actual computation is performed and finally the result is retrieved from the device memory in a device to host (DtoH) memcopy operation. In the context of Amdahl's Law we recognize that the runtime of our execution is, in this case, not only determined by the number of floating point operation per second (FLOPS) our GPU can perform i.e. how fast the kernel actually executes, but by how long the data transfers to and from the GPU take. In this example we spend roughly 27 milliseconds waiting for memory contents to be copied and only about 15 milliseconds waiting for calculations to finish.
For optimal performance it is of utmost importance to keep the GPU's processors working and not starved for data to process. There are several ways to achieve this:

- Aiming for a high degree of arithmetic intensity (ratio of compute to I/O, measured in floating point operations per transferred byte): Transfer speeds to and from the GPU memory make it prohibitively expensive to perform a simple elementwise operation like squaring the elements of a vector on the GPU. The cost of the memory copy operation has to be outweighed by accelerating a large enough computation on the data.
- Pipelining and overlapping tasks: Multiple kernels can execute simultaneously and more importantly copy operations can take place while computations are executed (This will be discussed Section 2.5).
- Maximizing copy throughput: When judging copy speeds it is necessary to be aware of the theoretical maximum performance achievable on the used hardware. In the example depicted in Figure 2.3 an Nvidia RTX 3070 GPU was used. This GPU has a maximum memory data-rate of 14 GBps [34], this speed is almost reached in the HtoD-transfers but should also be achieved DtoH-transfers. One would expect the transfer rates to be equal in both direction, therefore the transfer of the resulting matrix $\mathbf{C}$ to the host memory should be about three-times as fast. This is a potential performance increase worth look into, CUDA offers the functionality to allocate so-called pinned memory on the host which is never swapped from memory for better performance.

2.5 CUDA Streams

CUDA Streams can be used to perform multiple tasks at the same time on a single GPU. Tasks within a single stream are always serialized i.e. performed one after another, as they were scheduled, but tasks issued to different streams can be executed at the same time if possible.
If one for example wants to perform more than a single matrix multiplication, one could schedule the individual, independent operations in different streams. This approach was tested as displayed in Listing 2.2 and the resulting behavior profiled with Nvidia Nsight Systems. The resulting timeline when using a single stream is displayed in Figure 2.4. This timeline looks exactly as expected and very similar to the timeline presented in Figure 2.3: every kernel-execution is preceded by two HtoD-Memcopy operations to copy the matrices, which are to be multiplied, into device memory (drawn in teal) and succeeded by a single DtoH-Memcopy operation to retrieve the result from the device (drawn in violet). There are large sections where the GPU is unable to perform computations because the data needed is not on the device yet, this situation is often referred to as GPU *starvation* and has to be avoided for maximum efficiency.

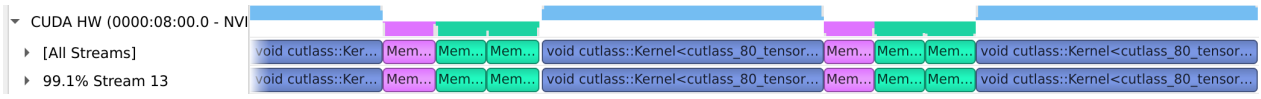


Figure 2.4: Timeline resulting from profiling a series of dgemm-kernel calls scheduled within a single stream

The timeline shown in Figure 2.5 shows the behavior when the operations are distributed over multiple streams (two in this case). The operations within a stream are unchanged, they still consist of a series

of kernel calls, with each kernel being preceded by two HtoD-operations and followed by a single DtoH-operation, but the two streams are now operating at the same time. The GPU streaming processors always have tasks to execute (no GPU starvation) and the copy engines can copy data to and from the host over the otherwise unused PCI-Express link in the meantime. This and related techniques are referred to as latency hiding, because the cost of moving the data to and from the GPU is now hidden and effectively gone.

Figure 2.5: Timeline resulting from profiling a series of dgemm-kernel calls distributed over 2 streams

When using more than two streams, even DtoH- and HtoD-memcpy operations can be performed at the same time as displayed in Figure 2.6.

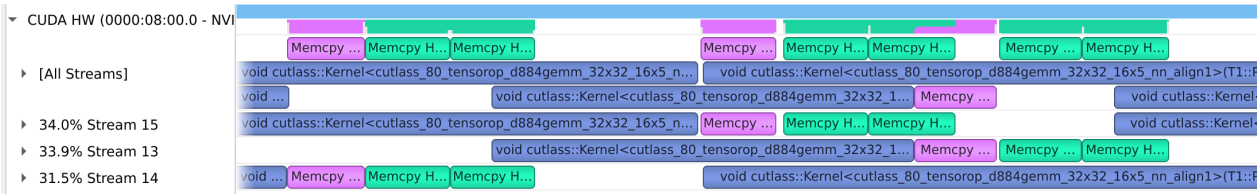


Figure 2.6: Timeline resulting from profiling a series of dgemm-kernel calls distributed over three streams

Of course the effectiveness of this technique heavily depends on the workload. It works especially well in cases where the duration of memcopy-operations and the duration of the actual kernel-execution is about equal. It works a lot less in cases where either the time taken by the kernel-execution (those cases are not problematic since they already benefit from the great performance of the GPU) or by the memory copy operations dominate.

2.5.1 Example containing streams

Listing 2.2 showcases the program that was profiled in Section 2.5. Unlike the previous CUDA examples like Listing 2.1, this example was written in Python and makes use of CUDA by utlizing the CuPy-package [47], which will be discussed in detail along similar alternatives in Section 3. For now, it is sufficient to call CuPy as a drop-in replacement for Numpy, which executes code on the GPU. The syntax and naming is very similar: `cp.matmul` is a function which multiplies two matrices which reside within the GPUs memory, just like `np.matmul` multiplies two matrices, which reside in the regular host-memory. The script itself is also not unlike a regular Numpy script:

- On line 1 the CuPy package is imported.
- On line 17 a number of CUDA-streams are created.
- On lines 20-23 random matrices `a` and `b` are created and space for the results is allocated.
- On lines 27-29 memory on the device is allocated. Note that only memory for one instance of the matrices `a`, `b` and `c` per stream is allocated. Not all inputs or results have to be stored on the GPU at the same time.
- On line 34 the stream to use for the next operations is set.
- On lines 36 and 37 the matrices `a` and `b` are loaded onto the device.
- On line 40 the matrices `a` and `b` are multiplied and the result stored in the memory allocated for the respective stream.
- This result is transferred back to the host in line 42.

This all is a rather straightforward process, except for the function `pinned_memory` defined on lines 5-10. Copying memory to and from the device in an asynchronous way i.e. interleaved with other scheduled operations, requires the host-memory involved in that operation to be pinned see CUDA Programming Guide v11.7.0 [48, section 3.2.6.3]. The concept of pinned memory is well explained in a blog post by Harris, even though the importance of using pinned memory seems to have waned as fast CPUs and fast memory are now more widespread than they were at the time of that blog post [49]. This is not a big restriction since data needed for high-performance calculations should not be swapped out of the main memory in any case, but Numpy does not make this assumption. By default, memory allocated by Numpy is not pinned, Numpy (or C's `malloc` for that matter) also has no problem with allocating memory for arrays which are larger than the total installed host-memory.
To deal with this CUDA offers the functions `cudaMallocHost` and `cudaHostAlloc`, which CuPy wraps using Cython and then in-turn offers to allocate memory using `cupy.cuda.alloc_pinned_memory`. Combined with Numpy's `frombuffer` it is possible to use that memory for Numpy arrays i.e. create Numpy arrays using pinned memory, which can be used for asynchronous memory transfers.

```
1  import cupy as cp
2  import numpy as np
3
4
5  def pinned_array(array):
6          # constructing pinned memory
7          mem = cp.cuda.alloc_pinned_memory(array.nbytes)
8          src = np.frombuffer(mem, array.dtype, array.size).reshape(array.shape)
9          src[...] = array
10         return src
11
12 # number of matrices to multiply
13 N = 100
14 # number of streams to utilize
15 streams = 3
16 # creating the streams
17 s = [cp.cuda.Stream() for i in range(streams)]
18
19 # creating matrices N matrices a and b on the host
20 a = [pinned_array(np.random.rand(400,400)) for i in range(N)]
21 b = [pinned_array(np.random.rand(400,400)) for i in range(N)]
22 # allocating memory for the results on the host
23 c = [pinned_array(np.empty(shape=(400,400))) for i in range(N)]
24
25 # allocating memory for the matrices a,b, and c
26 # each stream needs its own memory to avoid data-races
27 a_d = [cp.empty_like(a[0]) for i in range(streams)]
28 b_d = [cp.empty_like(a[0]) for i in range(streams)]
29 c_d = [cp.empty_like(a[0]) for i in range(streams)]
30
31
32 for i in range(N):
33         # determines which of the three streams should be used
34         s[i%streams].use()
35         # load contents into the device arrays
```

```
36          a_d[i%streams].set(a[i])
37          b_d[i%streams].set(b[i])
38
39          # calculate the result and store it in the respective device memory
40          c_d[i%streams] = cp.matmul(a_d[i%streams],b_d[i%streams])
41          # transfer the result back to the host
42          c_d[i%streams].get(out=c[i])
```

Listing 2.2: Example of CuPy-code making use of streams

2.6 The Roofline Performance Model

The roofline performance model proposed by Williams [50] provides a simple model to determine whether a kernel is compute- or memory-bound on a given hardware. It does so by relating the computational intensity of a kernel with the hardware's maximum FLOPS and memory bandwidth and therefore turns the rather vague statements made in the previous section into measurable figures and guides further optimization.
Figure 2.7 is an example of the roofline performance model visualized. The slope on the left-hand side indicates how fast the memory can supply the processor with data, whereas the horizontal lines indicate the peak computing performance of the chip. This figure tells us that peak performance cannot be achieved for simple tasks like multiplying every element of a single-precision vector by a fixed value. A kernel performing that task would need to transfer two 4-byte floating point numbers and perform a single floating point operation per element, resulting in a computational intensity of $1/8$ and a performance of less than 100 GFlops. This example would lead to a memory-bound kernel. Similarly, the displayed kernels **A** and **B** are memory-bound. The kernel **C** does reaches neither the memory-limit nor the performance limit. This can be due to several reasons, it either performs non-coalesced memory-accesses, which would significantly lower the achievable memory-bandwidth (as discussed in Section 2.7.1.3) or perform the "wrong" floating-point operations (as discussed in Section 2.1, peak performance can only be achieved by executing FMA-instructions). Those limitations could be modeled as additional ceilings to get a clearer picture of the effective bottleneck.

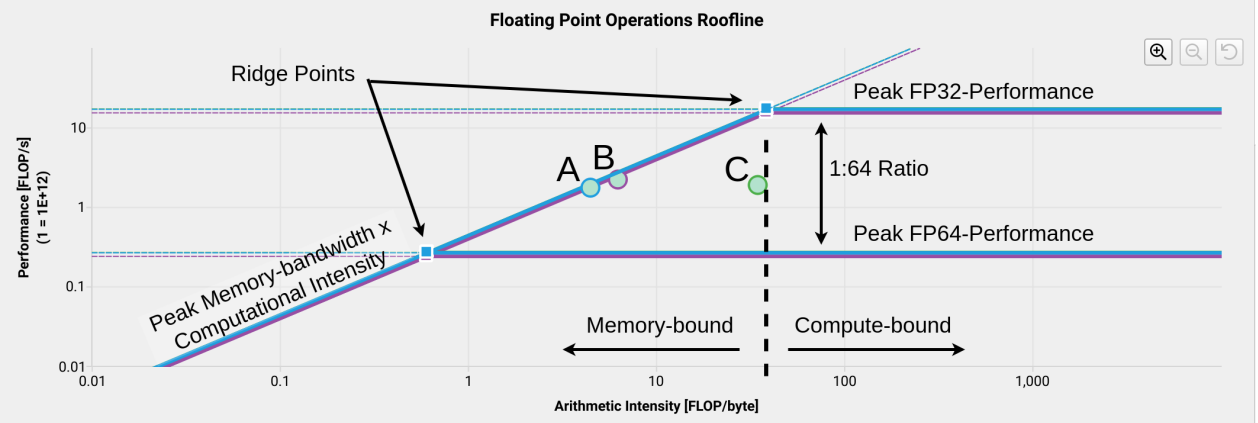


Figure 2.7: Three different kernels placed in an annotated roofline chart as presented by Nsight Compute

Note Figure 2.7 is drawn for a memory-bandwidth of 448 GBps, the bandwidth of the connection between the GPU and its dedicated off-chip memory. If one were to draw a roofline chart considering the bandwidth between the GPU and the host memory (a mere 14 GBps), the sloped line would be shifted down and the ridge points shifted right by about 1.5 orders of magnitude. This property, once-again,

demonstrates why offloading work to the GPU is only useful for certain kinds of tasks. Namely large tasks with superlinear time-complexities like $\mathcal{O}(n^3)$ for matrix-matrix multiplications or $\mathcal{O}(n \log n)$ in the case of the fast Fourier transform (FFT), resulting in large arithmetic intensities.

2.7 Case study: FFTs on the GPU

The fast Fourier transform (FFT) algorithm is ubiquitously used in many domains of science, especially in signal processing. In this case study we explore a number of different variations of this algorithm and try to adapt them to GPUs. The algorithms that will be examined and adapted in this section stem from the comprehensive book by Chu [51].
A more complete introduction to the discrete Fourier transform and a derivation of the FFT will be given in Section 4.

2.7.1 The Stockham auto-sort Algorithm

The first algorithm explored is the decimation in frequency (DIF) FFT for complex 1D inputs and outputs in natural order as outlined in Chapter 6 (specifically algorithm 6.1 "The (ordered) radix-2 DIF_{NN} FFT algorithm.") of the aforementioned book [51].
In a first step the algorithm was expressed typical Numpy syntax i.e. as few loops as possible, trying to express as much of the algorithm as possible using operations on slices of arrays instead of using explicit Python loops as displayed in Listing 2.3.

```
1  def fft_dif_it(x):
2      NumOfProblems = 1
3      ProblemSize = x.shape[0]
4      Distance = 1
5      res = np.empty_like(x, dtype=np.complex128)
6      y_l = np.empty(ProblemSize//2, dtype=np.complex128)
7      z_l = np.empty(ProblemSize//2, dtype=np.complex128)
8
9      res[:] = x
10
11     # log2(N) outer loops
12     while(ProblemSize > 1):
13         size = (ProblemSize//2)
14         for i in range(NumOfProblems):
15             omega = np.exp(2*1j*np.pi/(x.shape[0]))
16             omega_series = np.power(omega, -np.arange(ProblemSize//2)*NumOfProblems)
17
18             first = res[i:x.shape[0]//2:NumOfProblems]
19             second = res[i+x.shape[0]//2::NumOfProblems]
20
21             y_l[size*i:size*(i+1)] = first + second
22             z_l[size*i:size*(i+1)] = (first - second)*omega_series
23
24             res[i::2*NumOfProblems] = y_l[size*i:size*(i+1)]
25             res[i+Distance::2*NumOfProblems] = z_l[size*i:size*(i+1)]
26
27         NumOfProblems *= 2
28         Distance *= 2
```

```
29          ProblemSize //= 2
30
31      return res
```

Listing 2.3: Implementation of the Stockham FFT using Numpy

Next we expressed the very same algorithm using CUDA C as displayed in Listing 2.4. This algorithm does not operate on the input in-place, but requires a second array of equal size to store intermediate results in, which is why the function signature indicates pointers to two arrays of floating point numbers (`const float* input`, `float* output`). This is less obvious in the Numpy implementation (it uses the arrays `res` `y_l` and `z_l`), where we do not have to manually manage memory.

In this implementation every thread performs exactly one butterfly combination consisting of combining two complex numbers from the input array into two different complex numbers, which are then stored in the output array. It is noteworthy how the programming language is, unlike Numpy or Python in general, not aware of complex numbers, and we consciously have to load and store the real and complex parts of a number from consecutive memory addresses.

In this implementation we also encounter one of the limitations of CUDA. Whereas, in the Numpy implementation, we initiate new iterations, i.e. the loop defined on line 12, until $\log 2(N)$ iterations have been performed. Although `for` and `while` constructs are possible in CUDA kernel code, performing multiple iterations within a kernel call is still not as simple in the CUDA implementation. It would require all threads to have performed their butterfly combination, communicate that fact among all threads, swap the pointers for input and output and advance an iteration. CUDA however does not offer any way to synchronize more than a single block, which can be achieved by using the `__syncthreads()`-function, because it is not guaranteed that all blocks are processed at the same time. Since we want to be able to process problems larger than $N = 2^{11} = 2048$, we have to fall back on a different synchronization technique, namely consecutive kernel invocations with different arguments, i.e. swapping the pointers for the input and output arrays and incrementing the iteration. This procedure also highlights the computational complexity of the FFT algorithm, launching $\log 2(N)$ kenels with $\frac{N}{2}$ threads each and every thread performing a single butterfly leads us to the well known complexity of $\mathcal{O}(N \log(N))$.

```
extern "C" __global__
  void my_fft(const float* input, float* output, int iteration, int original_size, float
↪  omega_real, float omega_imag) {
    int tid = blockDim.x * blockIdx.x + threadIdx.x;

    int n_Problems = 1<<(iteration);
    int distance = 1<<iteration;
    int problem = tid/(original_size/(n_Problems*2));
    int posInProb = tid%(original_size/(n_Problems*2));

    float2 omega = make_float2(omega_real, omega_imag);
    float2 twiddle = raise_omega(omega, -posInProb*n_Problems);

    float2 first = make_float2(
       input[(posInProb*n_Problems+problem)*2],
       input[(posInProb*n_Problems+problem)*2+1]);
    float2 second = make_float2(
       input[(posInProb*n_Problems+problem)*2+original_size],
       input[(posInProb*n_Problems+problem)*2+1+original_size]);

    output[(posInProb*n_Problems*2+problem)*2] = first.x + second.x;
    output[(posInProb*n_Problems*2+problem)*2+1] = first.y + second.y;

    float2 toTwiddle = make_float2(first.x - second.x, first.y - second.y);
    complex_mult(&toTwiddle, twiddle);

    output[(posInProb*n_Problems*2+problem+distance)*2] = toTwiddle.x;
    output[(posInProb*n_Problems*2+problem+distance)*2+1] = toTwiddle.y;
  }
```

Listing 2.4: Implementation of the Stockham FFT using CUDA C

2.7.1.1 Benchmark Normally we would compare our CUDA implementation against the Numpy-baseline as defined in Listing 2.3, but since this is an FFT algorithm we have access to a wide selection of heavily optimized implementations from Numpy, scipy or FFTW3 [52]. Comparing it to our own Numpy implementation would not be very honest since that is not the baseline used in practice. For this first preliminary comparison we compare it against numpy.fft.fft, being well aware of the fact that, that function might not be the fastest CPU implementation.
Note that Numpy's FFT-routine does not handle single precision floats, they are cast to double precision before computation and the output is also of double precision. The comparison pictured in 2.8 also does not take the time it takes to transfer data to and from the GPU, which can be significant, into account. More elaborate comparisons for 1D FFT-transformations are presented in Section 4.4.1, but this first comparison already shows promising results, considering the naive GPU implementation.

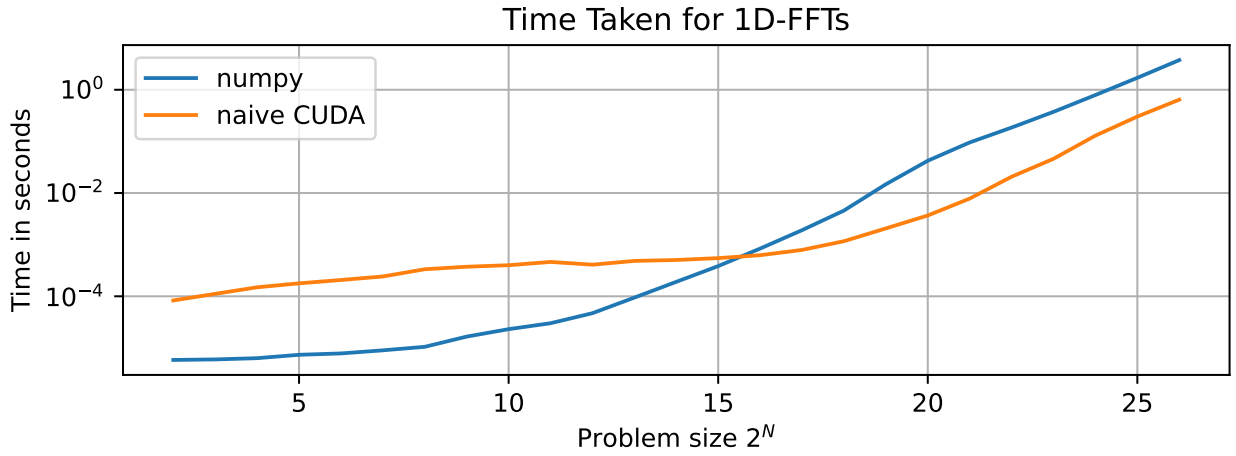


Figure 2.8: Comparison between the naive CUDA implementation developed in this section and Numpy for single precision 1D-FFTs

2.7.1.2 Memory access patterns

Additionally to timing the whole procedure we can also get timings for the individual iterations and memory transfers to and from the GPU using the Nvidia Nsight Profiler. In Table 2.1 the absolute runtimes of the individual kernel invocations for a problem of size $N = 2^{27}$ are displayed. This data reveals that there are great discrepancies between the run-times of the consecutive kernel invocations even though we can easily convince ourselves of the fact that the number of threads and the work each threads performs is unchanged.
The middle stage take significantly longer to execute than those at the beginning or end. For example the tenth stage takes almost 90 milliseconds, about 17-times as long as the fastest stage, which completes in 5.17 milliseconds.

iteration	Duration (ms)	rel. Duration
0	5.43	1.05
1	13.15	2.54
2	23.47	4.54
3	27.64	5.34
4	33.80	6.53
5	44.95	8.69
6	58.43	11.29
7	83.03	16.05
8	82.75	15.99
9	89.45	17.29
10	84.58	16.35
11	71.98	13.91
12	53.85	10.41
13	27.17	5.25
14	18.56	3.59
15	32.14	6.21
16	36.55	7.06
17	41.44	8.01
18	41.34	7.99
19	41.70	8.06
20	35.12	6.79
21	31.56	6.10
22	11.28	2.18
23	5.17	1.00
24	5.35	1.03
25	6.79	1.31
26	5.18	1.00

Table 2.1: Runtimes of the individual stages for a problem of size $N = 2^{27}$ and therefore 27 iterations

To understand why this happens it is necessary to look at the way this algorithm accesses the global memory, illustrated in Figure 2.9. In this figure the memory access pattern for a problem size of $N = 2^6 = 64$ is pictured (a rather small problem size, but it already displays the property that will be discussed and similar visualizations for larger problems are not feasible). To transform the input 6 consecutive kernel launches with 32 threads each are necessary. The operations are color-coded to indicate which thread is responsible for that operation, with colors ranging from yellow for thread 0 to green for thread 31. It is visually evident that the memory is accessed in well-behaved ways (desirable memory access would be described by consecutive threads accessing consecutive locations in memory, this behavior is described by the term *unit stride*) in the first few and last few stages, but in the middle stages the access and store strides are very unfavorable.

This leads us to the importance of efficient access of global (off-chip) memory. Due to the very nature of RAM chips reading consecutive bytes of memory (as long as they are located within the same row or page) is much faster than reading bytes from random positions. This topic was discussed in a blog post by Harris [53] in 2013. To confirm the relevancy of that post's finding for our modern hardware we decided to reproduce its experiment on strided memory access.

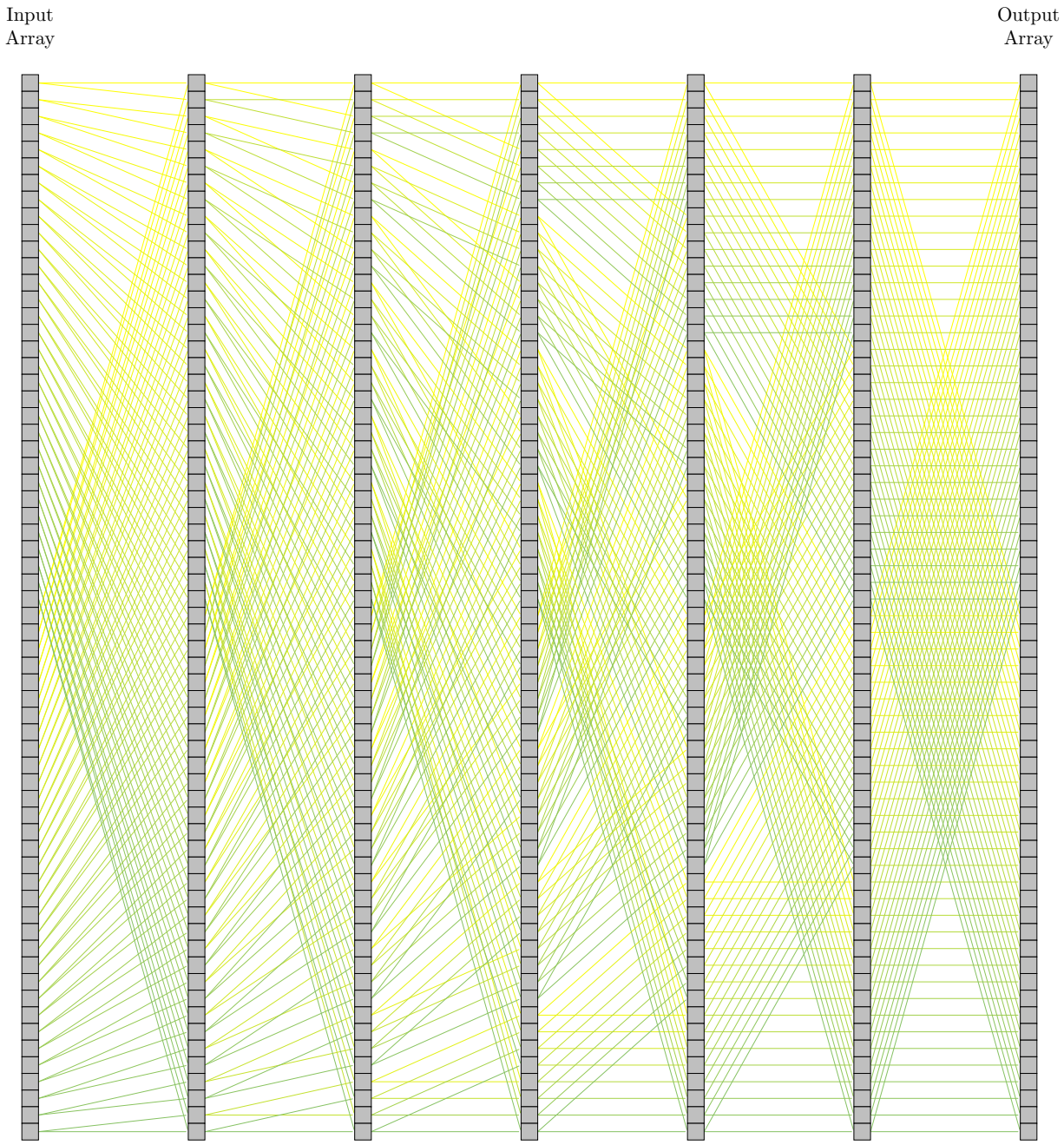


Figure 2.9: Memory access pattern of the Stockham FFT for a problem of size $N = 2^6 = 64$

2.7.1.3 Effect of strided access on memory bandwidth

In this experiment a large array of memory is allocated and subsequently a CUDA kernel (defined in Listing 2.5) is launched to increment elements of that array by one. Consecutive threads however do not necessarily increment neighboring elements of the array. Instead, the elements of the array are accessed and modified in a strided fashion, controllable by the parameter `s` passed to the kernel. This allows us to observe the varying performance of these operations.
In this timed benchmark there are always 4 megabytes of data read, modified and written back to global memory (note that 4 megabytes of data are a different number of elements for different data types, since

data types are not necessarily equal in size). The amount of data moved is then divided by the time taken to obtain a figure for the achieved throughput.

```
template <typename T>
__global__ void stride(T *a, int s)
{
    int i = (blockDim.x * blockIdx.x + threadIdx.x) * s;
    a[i] = a[i] + 1;
}
```

Listing 2.5: The CUDA kernel for benchmarking the strided memory accesses, uses templates to be able to handle a variety of different data types.

The described experiment was carried out for a number of different data types:

- `char` - size: 1 byte - Number of elements in 4 megabytes / Nr. of threads launched: 4194304
- `int_16` - size: 2 byte - Number of elements in 4 megabytes / Nr. of threads launched: 2097152
- `float` - size: 4 byte - Number of elements in 4 megabytes / Nr. of threads launched: 1048576
- `double` - size: 8 byte - Number of elements in 4 megabytes / Nr. of threads launched: 524288
- `2 packed doubles` - size: 16 byte - Nr. of elements in 4 MB / Nr. of threads launched: 262144
- `4 packed doubles` - size: 32 byte - Nr. of elements in 4 MB / Nr. of threads launched: 131072
- `8 packed doubles` - size: 64 byte - Nr. of elements in 4 MB / Nr. of threads launched: 65536

The packed doubles are a custom data type as defined in Listing 2.6. It was created to observe the performance of accessing larger consecutive structures of data. This is similar to accessing complex numbers which are essentially 2 packed floating point values which are usually accessed together.
The result of those measurements is visualized in Figure 2.10. It is evident that accessing memory in a strided fashion is associated with very high performance penalties. Memory bandwidth utilization is over 300 GB/s for all data types when accessing elements with a unit-stride, but not exactly the same because each thread is only accessing a single element and therefore a lot more threads are involved when processing the `char` types than when processing the `float` or `double` types and those additional threads incur performance overhead.
It can be observed that throughput decline with a $\frac{1}{\text{stride}}$ relationship (linear in the logarithmic plot) until every memory row accessed only contains a single type of the element. From this data we can conclude that in such a memory row one can fit 32 `chars`, 16 `int_16`s, 8 `floats` or 4 `doubles` (note the kinks in the respective graphs where performance stops to decrease with the $\frac{1}{\text{stride}}$ relationship), such a row of memory must be at least 32 bytes in sizes. The size of these rows however can vary from GPU model to GPU model, the results of a very similar experiment presented by Stephen Jones at Nvidia GPU technology conference (GTC) 2022 [54] indicate that this value is 64 bytes on the A100, a high-end accelerator for high performance computing (HPC) and AI workloads.
This dramatic drop in performance cannot (or to a much lesser extent) be observed when the data, every thread has to fetch, is almost as big or bigger than a single memory page. When accessing two packed doubles, which have a combined size of 32 bytes, the loss in performance when going from stride=1 to `stride=2` is still noticeable, but not when accessing 4 or 8 packed doubles at a time. Memory accesses for such large consecutive chunks of data are, by definition, always coalesced and don't suffer from the same dramatic performance degradation.

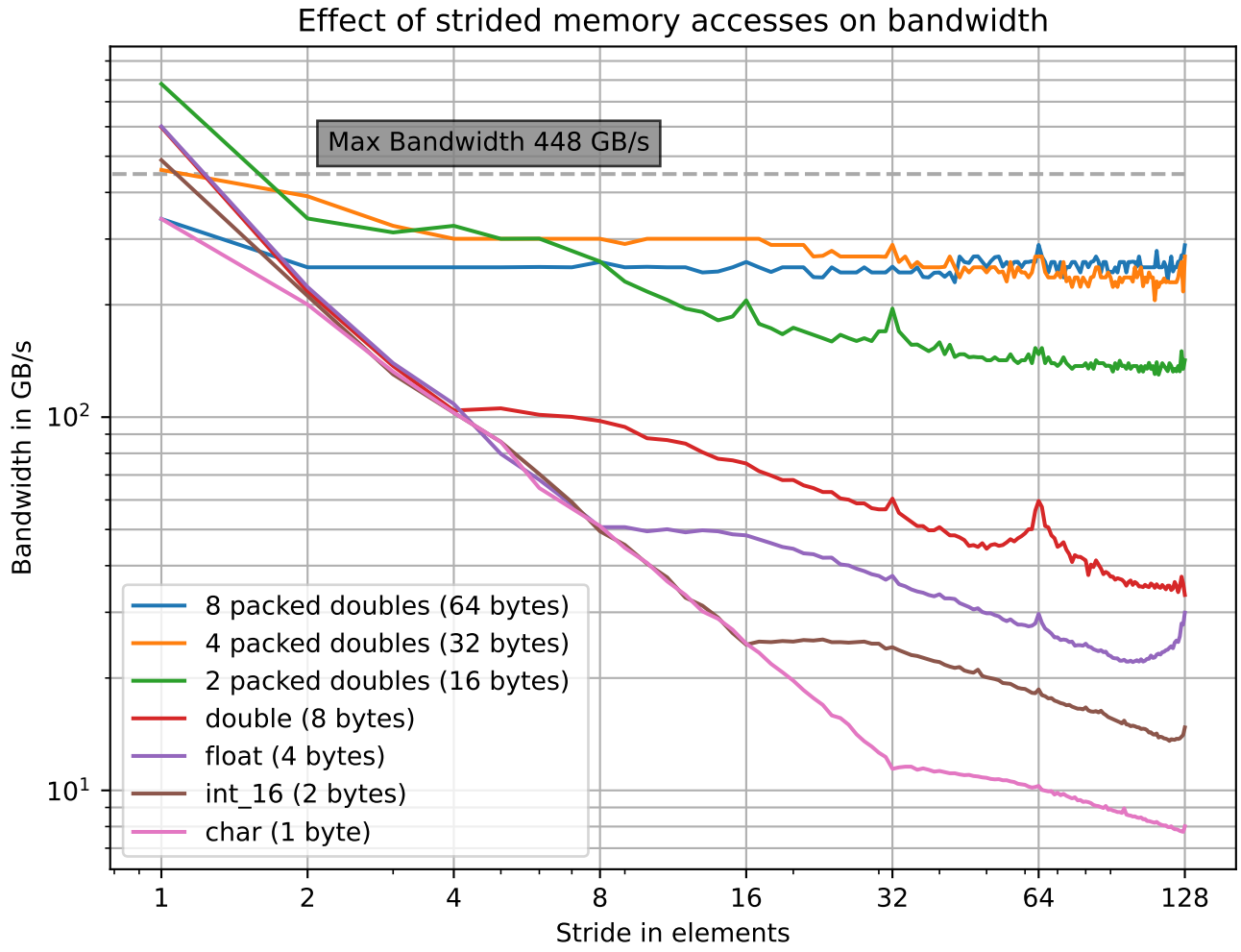


Figure 2.10: Effects on memory throughput when using strided memory access patterns.

One can also recognize that there are local maxima in performance which far from coalesced access patterns like accessing 2 packed doubles with a stride of 32 (and similarly for doubles at a stride of 64, floats at a stride of 128 and presumably for chars at a stride of 256). These maxima are very likely to originate from aligned accesses one layer higher in the hierarchy of memory: the elements are on the same position of consecutive pages of memory. The insights this experiment provides us with are crucial to understanding the differences in runtime of the different stages of the FFT-algorithm as presented in Table 2.1.

The fact that the benchmark reports slightly higher figures for the bandwidth when using unit-stride as stated possible by the datasheet (448 GB/s) is due to caching effects.

The code and data related to this section is located in the folder `coalesced`.

```
template <int n, typename T>
class packed
{
public:
    T vals[n];

    __host__ __device__ packed()
    {
        #pragma unroll
        for (size_t i = 0; i < n; i++)
        {
            vals[i] += 0;
        }
    };

    __host__ __device__ packed operator+(T x)
    {
        packed res;
        #pragma unroll
        for (size_t i = 0; i < n; i++)
        {
            res.vals[i] += x;
        }
        return res;
    };
};
```

Listing 2.6: Custom class `packed` to hold several packed values of a specified type. The qualifiers `__host__` and `__device__` are necessary to indicate to the compiler that versions for execution on the CPU and GPU must be generated. The compiler directive `#pragma unroll` is used to hint to the compiler that this loop should be unrolled in the compilation. This is possible because all template parameters must be known at compile time and it saves some memory on the device since the threads do not have to keep track of the variable `i`.

2.7.1.4 Kernel for problems of limited size

This knowledge, of course, raises the question of how fast a kernel, without those far-apart memory accesses, could be. As hinted at in Section 2.3 threads within a single block are guaranteed to be alive at the same time. This is due to the fact that they are all executed on the same SM and, as we can infer from the schematic of an SM (Figure 2.2), SMs offer 128 KB of fast shared memory threads on that SM can use for fast inter-thread communication and data exchange.
Each block can at most allocate 49152 bytes of shared memory, which would equal 12288 single precision floats or 6144 double precision floats. This amount can vary among chips with different compute capabilities, shared memory allocations greater than 48 KB require the use of dynamic shared memory instead of statically allocated shared memory as described in Section K.7.3 of the CUDA C Programming Guide 11.7 [48]. To process an FFT of size $N = 2048$ using the largest block possible consisting of 1024 threads using the proposed algorithm, we would need 2 arrays of size $2048 \cdot 2$ (due to the need to store both the complex and imaginary part). In total 8192 floating point values. Knowing this limitation we can at most perform an $N = 2048$ single precision or $N = 1024$ double precision FFT within a block.

Using the shared memory we can perform so-called coalesced reads from the global off-chip memory into the faster on-chip shared memory. This can be seen on lines 11-14 in Listing 2.7. Every thread reads four floats from adjacent memory positions, which was shown to be much more efficient in the previous section. The exact same behavior can be observed on lines 53-56 where the results are written back to global memory.

```
1  extern "C" __global__
2    void my_fft(const float* input, float* output, int original_size, float omega_real, float
   ↪  omega_imag) {
3      __shared__ float t1[1024*2*2];
4      __shared__ float t2[1024*2*2];
5
6      int tid = blockDim.x * blockIdx.x + threadIdx.x;
7      int iteration = 0;
8
9      // each thread loads 4 floats into the shared memory
10     // -- the real and imaginary parts of two complex64 values
11     t1[threadIdx.x*4] = input[tid*4];
12     t1[threadIdx.x*4+1] = input[tid*4+1];
13     t1[threadIdx.x*4+2] = input[tid*4+2];
14     t1[threadIdx.x*4+3] = input[tid*4+3];
15
16     float* input_;
17     float* output_;
18
19     input_ = t1;
20     output_ = t2;
21     int problemsize = 2048;
22     __syncthreads();
23     while(problemsize > 1){
24         int n_Problems = 1<<(iteration);
25         int distance = 1<<iteration;
26         int problem = threadIdx.x/(original_size/(n_Problems*2));
27         int posInProb = threadIdx.x%(original_size/(n_Problems*2));
28
29         float2 omega = make_float2(omega_real, omega_imag);
30         float2 twiddle = raise_omega(omega, -posInProb*n_Problems);
31         __syncthreads();
32
33         float2 first = make_float2(input_[(posInProb*n_Problems+problem)*2],
   ↪  input_[(posInProb*n_Problems+problem)*2+1]);
34         float2 second = make_float2(input_[(posInProb*n_Problems+problem)*2+original_size],
   ↪  input_[(posInProb*n_Problems+problem)*2+1+original_size]);
35
36         output_[(posInProb*n_Problems*2+problem)*2] = first.x + second.x;
37         output_[(posInProb*n_Problems*2+problem)*2+1] = first.y + second.y;
38
39         float2 toTwiddle = make_float2(first.x - second.x, first.y - second.y);
40         complex_mult(&toTwiddle, twiddle);
41
```

```
42          output_[(posInProb*n_Problems*2+problem+distance)*2] = toTwiddle.x;
43          output_[(posInProb*n_Problems*2+problem+distance)*2+1] = toTwiddle.y;
44
45          float* temp;
46          temp = input_;
47          input_ = output_;
48          output_ = temp;
49          problemsize /= 2;
50          iteration++;
51      };
52
53      output[tid*4] = input_[threadIdx.x*4];
54      output[tid*4+1] = input_[threadIdx.x*4+1];
55      output[tid*4+2] = input_[threadIdx.x*4+2];
56      output[tid*4+3] = input_[threadIdx.x*4+3];
57  }
```

Listing 2.7: Implementation of the Stockham FFT using CUDA C and shared memory

The kernel proposed here allows us to perform a great number (only limited by the size of global memory) of medium-sized 1D-FFTs with a single kernel launch. This is very useful when computing FFTs of higher-dimensions, which consist of repeated 1D-FFTs over different axis of the data.
It is also noteworthy that this kernel does not need a second array of size N within the global memory since all intermediate results are stored in shared memory. In a way this algorithm could be categorized as performing the computation in-place.
For benchmarking the benefit of this approach we compare the time needed for 1024 FFTs for a problem size of $N = 1024$. The kernel from Listing 2.4 requires 10240 (1024 FFTs each consisting of 10 iterations) kernel launches and performs a great number of unaligned memory accesses.
The optimized kernel calculates all 1024 FFTs in a kernel launch that lasts for 160 μs, while the less optimized version needs 289 ms for the computation of a *single* FFT of size 1024. It can therefore be assumed that 1024 of those computations would take almost 5 minutes. After all Figure 2.8 shows clearly that our previous kernel was slower than the Numpy function running on the CPU for all but the largest FFTs.
We can conclude that the optimized version is far better suited for applications involving the calculation of numerous medium-sized FFTs. A more elaborate comparison with the recommended reference implementation of FFTs on Nvidia GPUs (cuFFT) follows in Section 4.2.

2.7.2 Interactive Demonstration of the implemented algorithm using OpenCV

With an efficient implementation of the algorithm for batched DFTs of medium size, we can demonstrate this algorithm by employing it a real-time signal processing application. For this application we make use of OpenCV [55]. OpenCV is an important library for real-time computer vision. We utilize OpenCV for this application because it supports CUDA and OpenGL interoperability, which allows developers to use it to render the contents of CUDA arrays residing on the GPU's global memory directly to the screen without having to transfer those memory regions from the device memory to the hosts memory and displaying them like a traditional matrix, residing on the CPU, with tools like matplotlib [56]. This concept is referred to as zero-copy data exchange. The CUDA-OpenGL interoperability is described in section 3.2.13.1 (OpenGL Interoperability) of the CUDA Programming Guide v11.7.0 [48].
This demonstration application reads in an image from a file, calculates the corresponding two-dimensional frequency spectrum, applies a filter to that spectrum, performs the inverse FFT on the

filtered spectrum and displays the modified image as well as the filtered spectrum. The parameters of the filter are interactively modifiable by the user and the resulting image updated as fast as possible to convey a sense of the achieved real-time performance.

2.7.2.1 Compiling OpenCV

For this application to work as intended it is necessary to use an OpenCV version with both CUDA and OpenGL support. For the subsequent version of this program in Python, OpenCV also needs to be built with Python3 bindings. This is already a slightly exotic requirement from OpenCV and not all the distributed binaries of OpenCV might support this feature-set, since getting such version of OpenCV compiled is not trivial, we provide the options OpenCV was configured with before compilation. Note: OpenCV was compiled with the QT UI-framework instead of GTK because GTK proved to be incompatible with OpenGL. The OpenCV version used was release 4.6.0.
After compilation, it might be necessary to copy the resulting Python extension into the `site-packages` folder of the desired Python environment.

```
cmake \
-D CMAKE_BUILD_TYPE=RELEASE \
-D CMAKE_INSTALL_PREFIX=/usr/local\ \
-D INSTALL_C_EXAMPLES=ON \
-D INSTALL_PYTHON_EXAMPLES=ON \
-D OPENCV_GENERATE_PKGCONFIG=ON \
-D OPENCV_EXTRA_MODULES_PATH=../../opencv_contrib/modules \
-D BUILD_EXAMPLES=ON \
-D BUILD_opencv_world=ON \
-D WITH_CUDA=ON \
-D WITH_OPENGL=ON \
-D WITH_QT=ON \
-D WITH_GTK=OFF \
-D BUILD_opencv_cvv=OFF \
-D BUILD_opencv_python3=ON \
-D PYTHON3_EXECUTABLE=$(which python3) \
-D PYTHON_INCLUDE_DIR=$(python3 -c "from distutils.sysconfig import get_python_inc;
↪  print(get_python_inc())") \
-D PYTHON_INCLUDE_DIR2=$(python3 -c "from os.path import dirname; from distutils.sysconfig import
↪  get_config_h_filename; print(dirname(get_config_h_filename()))") \
-D PYTHON_LIBRARY=$(python3 -c "from distutils.sysconfig import get_config_var;from os.path import
↪  dirname,join; print(join(dirname(get_config_var('LIBPC')),get_config_var('LDLIBRARY')))") \
-D PYTHON3_NUMPY_INCLUDE_DIRS=$(python3 -c "import numpy; print(numpy.get_include())") \
-D PYTHON3_PACKAGES_PATH=$(python3 -c "from distutils.sysconfig import get_python_lib;
↪  print(get_python_lib())") \
-D PYTHON_DEFAULT_EXECUTABLE=$(which python3) \
-D HAVE_opencv_python3=ON \
-D BUILD_NEW_PYTHON_SUPPORT=ON \
..
make
sudo make install
```

Listing 2.8: Commands used to compile and install OpenCV

2.7.2.2 Program flow

Figure 2.11 outlines which steps are necessary to take the 2D-FFT of a grayscale image, filter it in the frequency domain and then transform it back to a viewable image.

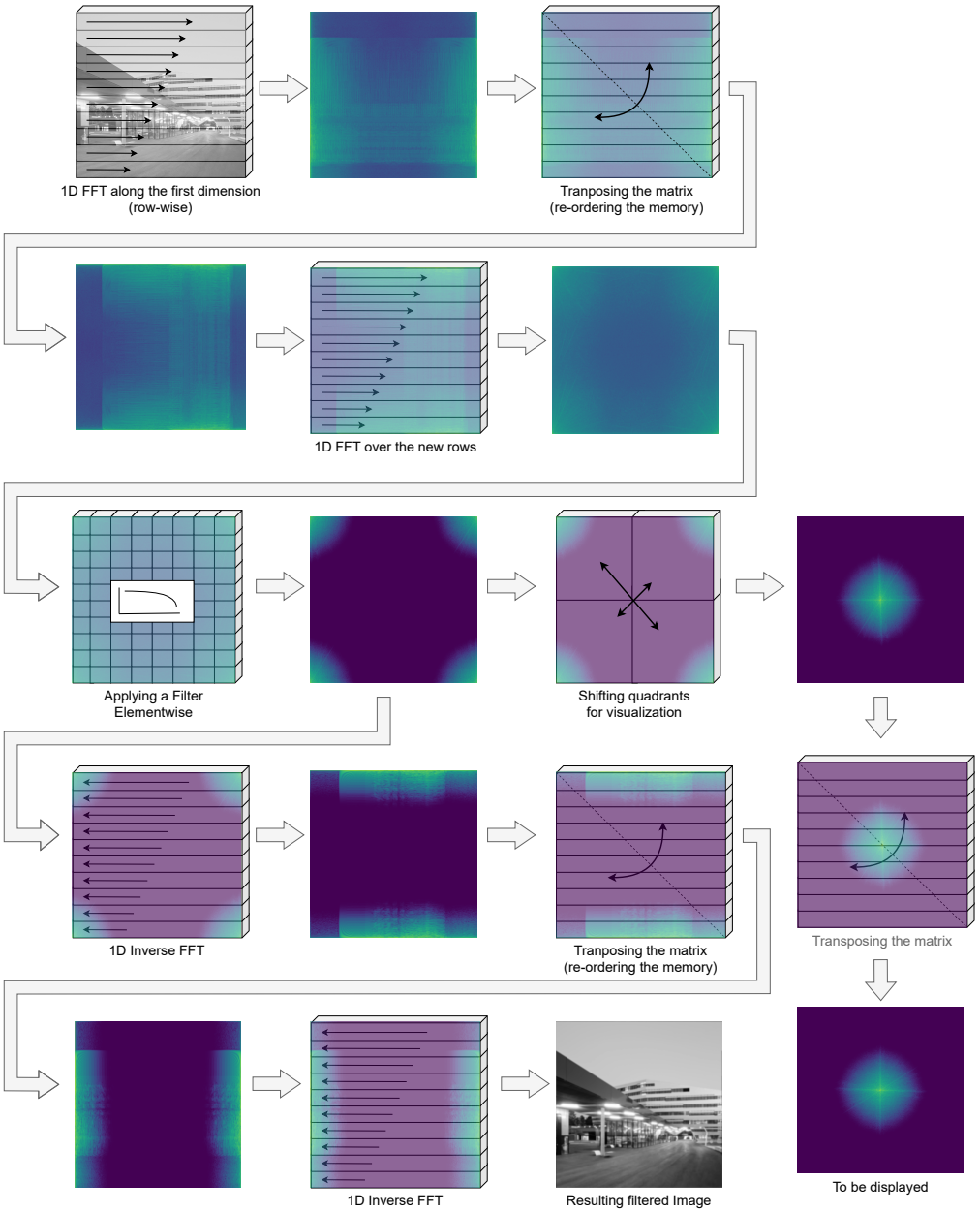


Figure 2.11: Schematic flow of the program, the labeled boxes indicate operations performed by CUDA-Kernels on the GPU. The two rightmost outputs from the bottom row are mapped to OpenGL-textures and displayed.

The procedure consists of 9 individual CUDA-Kernel launches (4 batched 1D FFTs, 3 Transpositions, 1 arbitrary filter and a fftshift to get the quadrants in the expected positions for visualization purposes). Some of these operations (namely the FFT, FFT-shift and Filter-Kernels) can be performed in-place, others like the kernel to transpose the matrix require separate memory buffers for input and output (it would be possible to define an in-place transposition kernel, but the kernel used is provided by OpenCV). This whole procedure is repeated for every change of the filter-parameters, even though the results of

the first three kernels are obviously independent of the choice of filter. This is done intentionally and indicates that this approach would be sufficiently fast to also process constantly changing input images in real-time i.e. videos (or in a more general sense: real-time 2D sensor data) instead of still images.

2.7.2.3 Adapting the FFT-Kernel for inverse FFT-calculations

The Kernel presented in Listing 2.7 had to be altered only slightly to be able to perform the inverse-FFT. The necessary changes are on the highlighted lines:

- line 1: changing the kernel to be templated, this allows us to easily generate kernels for various sizes of inputs and the template parameter `inverse` determines whether the kernel should perform the DFT or inverse DFT. Alternatively one could pass these parameters as functions arguments, but that practice comes with the downside of run-time overhead. In that case every thread would have to check at run-time if it is currently supposed to perform the regular or inverse transformation, whereas with template arguments it is known at run-time and the compiler creates entirely separate versions, which also benefit from better compile-time optimization.
- line 32: this line is responsible for calculating the twiddle-factors, in case of the inverse DFT, one has to use the opposite sign for the exponent when calculating the needed complex roots of unity.
- lines 54-57: Those lines are responsible for writing the results back to global memory. Depending on convention normalization factors have to be used either in the forward or backward transformations. Either the result of the forward or backward transformation has to be multiplied with $\frac{1}{N}$ or both results have to be scaled with $\frac{1}{\sqrt{N}}$. This version applies the normalization factor $\frac{1}{N}$ when performing the inverse transformation, this is consistent with the default choice of Numpy.

```
1  template <unsigned int size, bool inverse>
2  __global__ void my_fft_c2c(float *input, float *output, int original_size, float omega_real, float
   ↪  omega_imag)
3    {
4          __shared__ float t1[size * 2];
5          __shared__ float t2[size * 2];
6
7          int tid = blockDim.x * blockIdx.x + threadIdx.x;
8          int iteration = 0;
9
10         // each thread loads 4 floats into the shared memory
11         // -- the real and imaginary parts of two complex64 values
12         t1[threadIdx.x * 4] = input[tid * 4];
13         t1[threadIdx.x * 4 + 1] = input[tid * 4 + 1];
14         t1[threadIdx.x * 4 + 2] = input[tid * 4 + 2];
15         t1[threadIdx.x * 4 + 3] = input[tid * 4 + 3];
16
17         float *input_;
18         float *output_;
19
20         input_ = t1;
21         output_ = t2;
22         int problemsize = size;
23         while (problemsize > 1)
24         {
25
26           int n_Problems = 1 << (iteration);
```

```
27          int distance = 1 << iteration;
28          int problem = threadIdx.x / (original_size / (n_Problems * 2));
29          int posInProb = threadIdx.x % (original_size / (n_Problems * 2));
30
31          float2 omega = make_float2(omega_real, omega_imag);
32          float2 twiddle = raise_omega(omega, (inverse ? 1 : -1) * posInProb * n_Problems);
33          __syncthreads();
34          float2 first = make_float2(input_[(posInProb * n_Problems + problem) * 2],
    ↪  input_[(posInProb * n_Problems + problem) * 2 + 1]);
35          float2 second = make_float2(input_[(posInProb * n_Problems + problem) * 2 +
    ↪  original_size], input_[(posInProb * n_Problems + problem) * 2 + 1 + original_size]);
36
37          output_[(posInProb * n_Problems * 2 + problem) * 2] = first.x + second.x;
38          output_[(posInProb * n_Problems * 2 + problem) * 2 + 1] = first.y + second.y;
39
40          float2 toTwiddle = make_float2(first.x - second.x, first.y - second.y);
41          complex_mult(&toTwiddle, twiddle);
42
43          output_[(posInProb * n_Problems * 2 + problem + distance) * 2] = toTwiddle.x;
44          output_[(posInProb * n_Problems * 2 + problem + distance) * 2 + 1] = toTwiddle.y;
45
46          float *temp;
47          temp = input_;
48          input_ = output_;
49          output_ = temp;
50          problemsize /= 2;
51          iteration++;
52         };
53        __syncthreads();
54        output[tid * 4] = input_[threadIdx.x * 4] * (inverse ? 1. / size : 1.);
55        output[tid * 4 + 1] = input_[threadIdx.x * 4 + 1] * (inverse ? 1. / size : 1.);
56        output[tid * 4 + 2] = input_[threadIdx.x * 4 + 2] * (inverse ? 1. / size : 1.);
57        output[tid * 4 + 3] = input_[threadIdx.x * 4 + 3] * (inverse ? 1. / size : 1.);
58  }
```

Listing 2.9: Implementation of the Stockham FFT using CUDA C and shared memory

Note that all FFT Kernels in this example assume the inputs to be complex-valued, even though they are clearly not. The first FFT-kernel for example which consumes the images' pixels ranging from 0-255 (data type: `unsigned char`) casts those values to floats and interprets them as the real part of a complex value and sets the imaginary part to zero. This is inherently inefficient since it is known that the transformation of real-valued inputs has specific properties like symmetry. Using these properties it is possible to reduce the computation of a real-valued FFT of size N to a complex-valued FFT of size N/2 as explained in detail in chapter 14.1 by Chu [51].
This optimization was not employed in this example to not limit the generality of the application to real-valued inputs like images.

2.7.2.4 Filter Kernel

The sliders depicted in the right-hand images control the parameters of the applied filter, which is a 2D-gaussian function:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{bmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{bmatrix} \begin{pmatrix} x - \frac{n_x}{2} \\ y - \frac{n_y}{2} \end{pmatrix}$$
$$f(x,y) = \exp\left(-\left(\frac{\left(x' - \frac{n_x}{2}\right)^2}{\sigma_x^2} + \frac{\left(y' - \frac{n_y}{2}\right)^2}{\sigma_y^2}\right)\right)$$

Of course there are many other valid filters which could be used to alter the spectrum, this is a basic example for a low-pass filter.
This filter is applied by the kernel depicted in Listing 2.10.

```
1  template <unsigned int size>
2  __global__ void filter(float *input, float *output, float s_x, float s_y, float angle)
3  {
4          int tid = blockDim.x * blockIdx.x + threadIdx.x;
5
6          // lambda function to calculate filter,
7          auto coeff = [&s_x, &s_y](int y, int x)
8          {
9                  return expf(-(
10                         ((x - size/2) / s_x) * ((x - size/2) / s_x) + ((y - size/2) / s_y) * ((y -
   ↪  size/2) / s_y)));
11         };
12
13         // each thread handles a pixel
14         // the offset of 512 and modulo take care of shifting the quadrants
15         int x = (threadIdx.x + size/2) % 1024;
16         int y = (blockIdx.x + size/2) % 1024;
17
18         float cosval;
19         float sinval;
20         sincosf(angle, &sinval, &cosval);
21         float x_ = (x - size/2) * cosval - (y - size/2) * sinval;
22         float y_ = (x - size/2) * sinval + (y - size/2) * cosval;
23         float c = coeff(x_ + size/2, y_ + size/2);
24         output[tid * 2] = c * input[tid * 2];
25         output[tid * 2 + 1] = c * input[tid * 2 + 1];
26 }
```

Listing 2.10: Source code of the filter-kernel

The filter-function itself is encapsulated in a lambda function, which is then called by every thread for its respective pixel. This design choice enables a more flexible implementation of the filter, since the filter-function can be changed easily by changing the lambda function.
This kernel can be optimized as will be discussed in the following section Section 2.7.2.5.

2.7.2.5 Resulting images

Figures 2.12 and 2.13 display two filtered sample and their respective frequency spectra.
Every time on of the sliders' values changes, the procedure outlined in Figure 2.11 is re-calculated and the updated results are displayed.

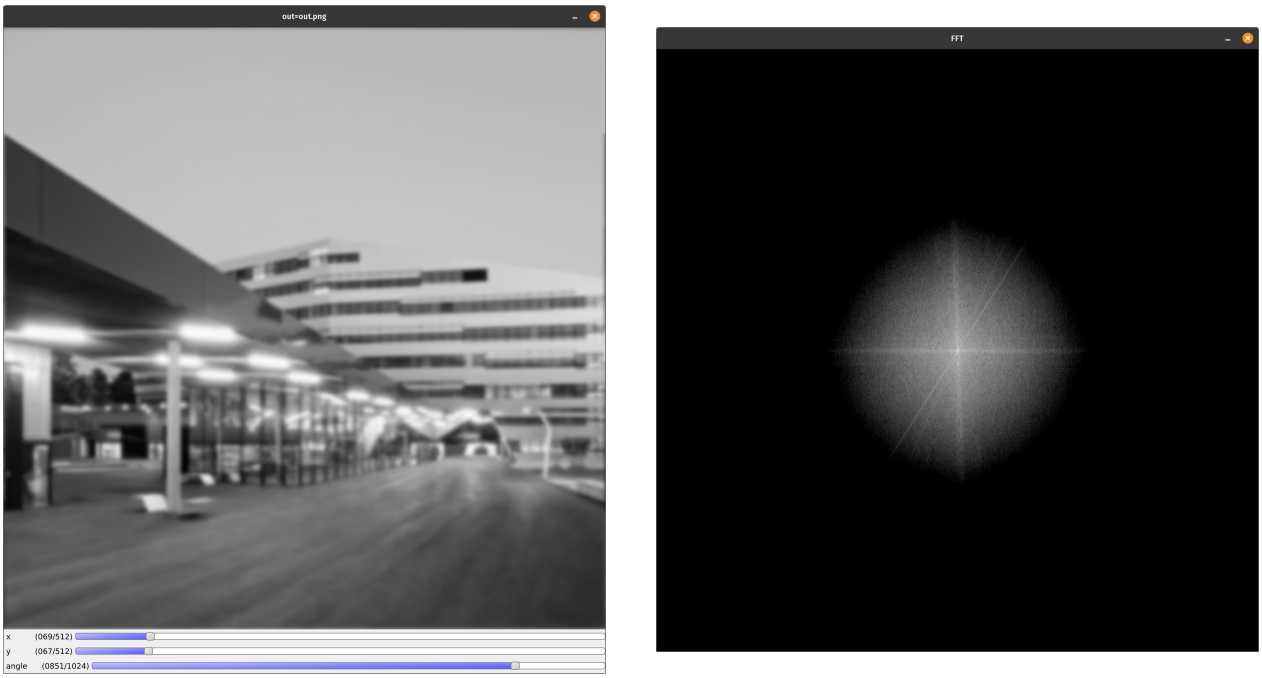


Figure 2.12: Image with a uniform low-pass filter applied and the corresponding frequency spectrum

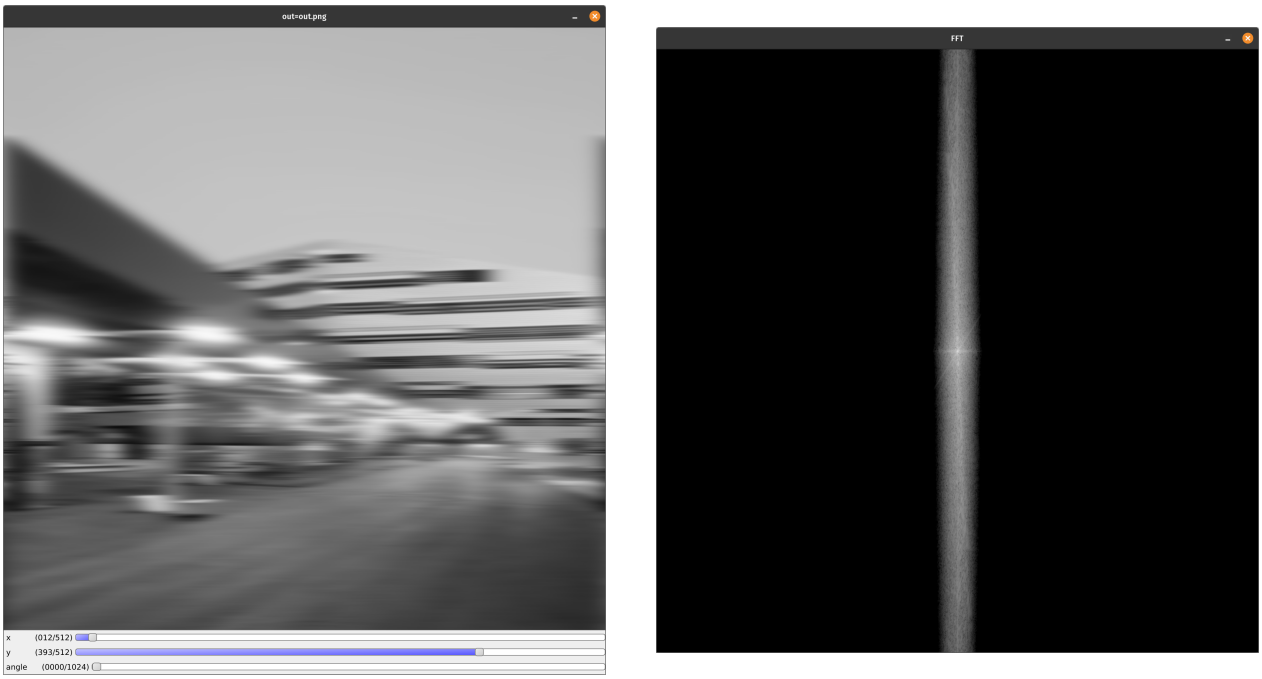


Figure 2.13: Image with a low-pass filter, which preserves vertical frequencies but filters high horizontal frequencies, applied and the corresponding frequency spectrum

2.7.2.6 Performance

Profiling the program with Nsight Systems reveals how long the whole procedure and each individual kernel takes when a re-calculation is triggered by a user-made change of the filter-parameters. Table 2.2, lists those timings and we can recognize that, first of all the calculations are clearly fast enough, for real-time user interaction. The combined run-times of the 9 kernels add up to 940μs, that is however not the actual time elapsed between the invocation of the first kernel to the conclusion of the last kernel, because there are short phases in between the kernels. Nsight Systems reports that time as 1050μs, so we can recognize that the used hardware is able to perform this calculations hundreds of times per second for this choice of algorithms and input sizes (a 1024 x 1024 grayscale image).

	Name	Duration	Percentage	Description
0	FFT-uc2c	155.486 μs	16.5	Computes the 1D-FFT over the rows of the input image; takes input of data type unsigned char and outputs complex64
1	Transpose	42.176 μs	4.5	Transposes the matrix of complex64 values
2	FFT-c2c	158.046 μs	16.8	Computes the 1D-FFT over the new rows; operates in-place on the complex64 matrix
3	Filter	179.325 μs	19.1	Applies the gaussian filter elementwise to every element of the matrix
4	Shift	29.952 μs	3.2	Shifts the four quadrants and calculates the log10 of the absolute values and outputs an unsigned char matrix; only for visualization purposes
5	Transpose	12.192 μs	1.3	Transposes the matrix of unsigned characters
6	FFT-c2c-inverse	157.246 μs	16.7	Performs the inverse FFT on the filtered spectrum, operates in-place on the complex64 matrix
7	Transpose	42.720 μs	4.5	Transposes the complex64 matrix
8	FFT-c2uc	162.717 μs	17.3	Performs the final inverse FFT, outputs a matrix of unsigned characters

Table 2.2: Runtime of the individual kernels within the update-procedure. The kernels listed in this table correspond with those illustrated in Figure 2.11

A surprisingly large share of the time is needed by the filter-kernel, which should be rather simple and embarrassingly parallel. A closer look at that kernels source-code (see Listing 2.10) reveals some potential performance pitfalls. One glaring flaw is the fact that the value of angle is passed to this kernel and then the kernel calculates the sine and cosine of that angle. While nothing is inherently wrong with this approach of calculating the sine and cosine within the kernel, but it can be questioned, if it is truly necessary to perform the same calculation in all 2^{20} threads instead of directly passing the sine and cosine to the kernels.
To a lesser extent, the same argument could be made for the parameters s_x and s_y representing the

standard deviations in the x and y-directions. These values are only used once and squared, so passing the kernel s_x^2 and s_y^2 directly, would save two floating point operations per thread and could be more efficient as well.
The run-times presented in Table 2.2 were recorded with a preliminary version (see Listing 2.11 for a section of that kernel), which was not templated with the parameter `size`, but where the size was hardcoded into the source like:

```
9 return expf(-(((x - 512.) / s_x) * ((x - 512.) / s_x) + ((y - 512.) / s_y) * ((y - 512.) / s_y)));
```

Listing 2.11: Source code of the filter-kernel during the prototyping stages

And it was not possible to reproduce the run-time with the templated version, furthermore the templated version did provide a far better performance (around 43μs) without incorporating any of the possible improvements, like passing the sine and cosine values directly to the kernel. This naturally led to the question of why such a seemingly small change like replacing `512.` with size/2 leads to a 4-fold speedup. Further testing concluded that the performance degradation was entirely due to how the compiler treats literal-floating point numbers. If a number has no suffix (like `512.`) it is treated as a double-precision floating point number, as a consequence of this, most of the other variables in the expression will also be promoted to double-precision and double-precision floating point operations will be performed, which are far more time-consuming than single-precision or int32 operations. It is therefore good practice to either avoid literal values like that completely or at least specify the intended data-type with an appropriate suffix (`f` or `F` for single-precision, `l` or `L` for long-double, there is no suffix for regular double-precision). When using the appropriate data type (either single-precision float or int32), the code does not benefit from passing in the values for sine or cosine, since the kernel is memory-bound at that point.

2.7.2.7 Kernel fusing

The procedure showcased in Figure 2.11 lists two consecutive kernels which are somewhat similar, both the filter and the shift kernel operate purely elementwise and they even share some of their computation since the filter kernel also calculates the shifted positions to apply the filter correctly. Since there is no limitation that prevents a kernel from writing to two distinct arrays (i.e. having two outputs) one might try to fuse the two kernels. The resulting kernel should have a significantly higher computational intensity than the two individual kernels, since fewer data has to be retrieved from global off-chip memory.

```
1  template <unsigned int size>
2  __global__ void shift(float *input, uchar *output)
3  {
4          int tid = blockDim.x * blockIdx.x + threadIdx.x;
5
6          // each thread handles a pixel
7          int x = (threadIdx.x + size/2) % size;
8          int y = (blockIdx.x + size/2) % size;
9          // arbitrary scaling factor to get a decent contrast
10         // when displaying values in the range [0,255]
11         float scaling_factor = 30;
12         output[y * blockDim.x + x] = scaling_factor * log10(sqrt(input[tid * 2] * input[tid * 2] +
   ↪  input[tid * 2 + 1] * input[tid * 2 + 1]));
13 }
```

Listing 2.12: Source code of the shift-kernel

Listing 2.12 shows the source of the shift-kernel, all it does is calculate the new position of a pixel in lines 7 and 8 and then calculates $\log_{10}(|f(x,y)|)$ with $f(x,y)$ being the complex-valued value of the spectrum at the specified index after the filter has been applied.
The two kernels presented in Listings 2.10 and 2.12 can be combined (this process is referred to as kernel fusing) into a single kernel as shown in Listing 2.13.

```
1  template <unsigned int size>
2  __global__ void fused_filter_shift(float *input, float *output, uchar *output_viz, float s_x,float
   ↪  s_y, float angle)
3  {
4          int tid = blockDim.x * blockIdx.x + threadIdx.x;
5
6          // lambda function to calculate filter,
7          auto coeff = [&s_x, &s_y](int y, int x)
8          {
9                  return expf(-(
10                         ((x - (float)(size)/2) / s_x) * ((x - (float)(size)/2) / s_x) + ((y -
   ↪  (float)(size)/2) / s_y) * ((y - (float)(size)/2) / s_y)));
11         };
12
13         // each thread handles a pixel
14         // the offset of 512 and modulo take care of shifting the quadrants
15         int x = (threadIdx.x + size/2) % size;
16         int y = (blockIdx.x + size/2) % size;
17
18         float cosval;
19         float sinval;
20         sincosf(angle, &sinval, &cosval);
21         float x_ = (x - (float)(size)/2) * cosval - (y - (float)(size)/2) * sinval;
22         float y_ = (x - (float)(size)/2) * sinval + (y - (float)(size)/2) * cosval;
23         float c = coeff(x_ + (float)(size)/2, y_ + (float)(size)/2);
24         float real_part = c * input[tid * 2];
25         float imag_part = c * input[tid * 2 + 1];
26         output[tid * 2] = real_part;
27         output[tid * 2 + 1] = imag_part;
28
29         output_viz[y * blockDim.x + x] = 30.f * log10(sqrt(real_part * real_part + imag_part *
   ↪  imag_part));
30         }
```

Listing 2.13: Source code of the fused-kernel

Comparing the resulting new kernel with the two separate kernels used before reveals significant performance benefits. The two individual kernels needed:

- filter-kernel $43\ \mu$s
- shift-kernel $29\ \mu$s

Whereas the new kernel only needs $48\ \mu$s for both operations, that is a speedup of 33%. This is an expected result for memory-throughput-bound kernels. One could also fuse this kernel with the preceding FFT-Kernel and following inverse FFT-Kernel and doing so will save some execution time, but at that point the kernel will become unwieldy and very specific to this application, which is in conflict with the

principles of modular, reusable and testable code. Fusing an FFT-Kernel with a Transposition-Kernel would not be possible as easily, since those kernels both make heavy use of local memory to enable strided memory access, but each kernel in a different way. The FFT-Kernel keeps a single row of data in local memory, unlike the Transposing-Kernel, which decomposes the input-matrix in many small tiles and processes those tiles within the shared memory of a SM.
Just like it usually pays of to perform as many operations on the GPU before retrieving the data back to the host-memory it is also advantageous to perform as many operations on the data as long as it is in the chip's memory. As discussed in Section 2.4 this is a recurring theme in GPGPU-Computing even though the respective bandwidths are an order of magnitude apart in bandwidth: PCI-Express between the Host and this specific GPU (the PCI-Express standard could support higher bandwidths) at 14 GB/s; The Bus-System between the off-chip memory and the streaming multiprocessors at 448 GB/s.
Despite the new kernel featuring a higher computational intensity than the two kernels it stems from, an analysis with Nsight Compute reports that this kernel is still memory bound and would not benefit from reducing the number of floating point calculation it has to perform per thread.
The complete C++ source code which can be compiled with nvcc (nvidia c compiler) and a demonstration video, which shows the responsiveness of the program is located in the `opencv_demo` folder. This same program is also implemented using Python and its opencv bindings (see Section 3.6)

3 GPGPU and Python

After having discussed Python, its limitations and ways to circumvent them in Section 1 and the possibility of utilizing GPUs for scientific-numerical workloads in Section 2, this section will explore Python packages and projects which aim to enable Python users to enhance the performance of their programs or scripts using GPUs.

3.1 Available packages

It is possible to expose the functionality offered by the CUDA C/C++ to Python using tools like Cython and Pybind11 as discussed in Section 1.2.2 and showcased in the example involving cuBLAS presented in Section 1.2.3.5. The immense popularity of the Python language (especially the popularity of Python in the domain of artificial intelligence (AI) and machine learning, fields which utilize GPGPU to a great extent) has given rise to a number Python libraries which try to bridge this gap between Python and the traditionally lower-level CUDA-API. This section provides an overview over the most important and influential of those packages with a focus on their intended use, differences and interoperability.

CuPy

CuPy [47] is an open-source library, which is primarily developed by the Japanese company *Preferred Networks*, designed as a drop-in replacement for Numpy and SciPy. It aims to provide the same functionality, but accelerating that functionality by utilizing the GPU. One of CuPy's strengths is its ability to allocate memory and create n-dimensional arrays residing on the GPU in a way similar to the way Numpy is used for arrays on the host memory using functions like `cupy.zeros`, `cupy.empty` and `cupy.ones`.
CuPy makes heavy use of templated kernels (not in the sense of C++-Templates though) to create the required kernels and compile them just-in-time using Nvidia runtime compiler (NVRTC). By doing so it also enables users to write CUDA C/C++-kernels directly in Python as strings (it is also possible to change those strings, which define the kernels, by using f-strings or more sophisticated templating systems to instantiate arbitrarily complex kernels at run-time) and call them after they have been compiled. Since version 7.0 released in December 2019 CuPy offers support for AMD hardware using ROCm HIP. Alternatively, since version 9.0, released in April 2021, CuPy supports defining GPU-kernels in pure Python instead of C/C++, it is able to generate C/C++ by analyzing the function's abstract syntax tree (AST). CuPy relies on the previously discussed Cython library to interface with the CUDA-API.

Numba

As discussed in Section 1.2.2 Numba [25] is a package that can parse Python code and tries to compile it to equivalent code in the LLVM intermediate representation language, which can then optimized by LLVM. In a subsequent step the optimized IR-Code can be transformed to machine-executable code using the backends integrated into the LLVM compiler framework. Since LLVM has a backend for Nvidia GPUs, Numba is able to generate code executable on GPUs.
Additionally, to this method where regular Python code is compiled to GPU Kernels without much manual intervention, Numba also offers a dialect of Python referred to as "CUDA Python", in which CUDA-Kernels can be described very similarly to how they would be expressed in CUDA C/C++, but with Python Syntax.
Similarly to CuPy, Numba also offers functionality to allocate memory and create arrays within the device memory or copy existing Numpy arrays to the device memory using functions like `numba.cuda.device_array` or `numba.cuda.to_device`.
Numba added support for AMD hardware with version 0.40.0 in September 2018 and removed it with version 0.54.0 in August 2021

PyCUDA

PyCUDA [57] is a library developed and maintained by Andreas Klöckner that aims to provide a complete wrapper of the CUDA C/C++, additionally, it also includes bindings for the less frequently used parts of the API like the OpenGL interoperability, which both CuPy and Numba lack. From a usage point of view PyCUDA is very similar to using CuPy with user-defined kernels: memory can be allocated on the device, copied to and from the device and GPU-Kernels are defined using triple-quoted multi-line Python strings, which are compiled by NVRTC as needed. Unlike CuPy it does not feature any implementations for common operations like matrix-multiplications, matrix inversions or calculating the absolute value of elements in a matrix. There are however packages that are built on top of PyCUDA and implement a number of those algorithms (as listed on the homepage of the PyCUDA documentation [58]).
PyCUDA uses boost, a library similar to pybind11 for interfacing with the CUDA C/C++ API.

PyOpenCL

PyOpenCL [57] is a sister project to PyCUDA, aiming to provide a complete set of bindings for the OpenCL API. It is also developed and maintained by Andreas Klöckner. PyOpenCL uses pybind11 for interfacing with the OpenCL C/C++ API.

PyTorch

Meta AI's PyTorch [59] is one of the leading frameworks used in deep learning research leveraging GPGPU for acceleration. Besides uses in the artificial intelligence domain its GPU-accelerated tensor library can of course also be used to perform scientific computations. It features a Numpy-like interface for such tasks. Similarly to the other mentioned libraries PyTorch also exposes many features of the CUDA API (like streams) and ways to efficiently move data objects (PyTorch refers to its n-dimensional as tensors) between the host and device.
After initially only supporting Nvidia GPUs, PyTorch release 1.8 in March 2021 added support for AMD GPUs using ROCm.

Official CUDA-Python bindings

Starting from April 2021 [60] Nvidia began publishing its own official package [61] containing bindings for the CUDA-API using Cython. This set of bindings is designed to replace the individual bindings other packages like CuPy or Numba maintain and build on. As of time of writing Numba, for example uses these official bindings instead of its own, if it detects an installation of Nvidia's cuda-python package with plans to ultimately deprecate and remove its own bindings. It can be assumed that other packags will follow and that this will become a common layer of abstraction between Python packages and the CUDA driver.

JAX

JAX [62] is a framework from Google that makes use of the XLA [63] (accelerated linear algebra) compiler to just-in-time compile a series of operations into a single kernel (opposed to calling multiple pre-compiled or pre-defined kernels). JAX is predominantly used in conjunction with deep learning frameworks like PyTorch or TensorFlow, but it can also be on its own as a GPU-accelerated alternative to Numpy, whose API it closely follows. Additionally, to supporting GPUs and CPUs, JAX can also target tensor processing units (TPUs), hardware specifically designed to accelerate typical AI-workloads (i.e. large matrix multiplications).
JAX offers experimental support for ROCm.

3.2 Examples using the mentioned packages

To highlight the differences in functionality and programming approaches between the mentioned packages, we present a basic saxpy-kernel (saxpy is short for single precision $\mathbf{z} = \alpha\mathbf{x} + \mathbf{y}$, using the naming convention used by the BLAS standard, the double-precision variant would be referred to as daxpy) implemented with each of the different packages.

3.2.1 CuPy

Listing 3.1 shows one of the ways CuPy can be used. On lines 5-6 the inputs are created as traditional Numpy ndarrays and then transferred to the GPU on lines 9-10, at this point memory on the device is allocated. The allocation on line 12 is not strictly required, as that memory would have been allocated either way when executing line 15 needed but good practice. On line 15 the actual computation is executed and on line 18 that result is retrieved from the device to the host using a DtoH-memcopy.
As mentioned in the previously given description CuPy is utilizing pre-defined templated-kernels to instantiate the right kernel just-in-time. This is done to avoid having to define an exponentially increasing number of kernels (there would have to be kernel for every combination of data-types and operation involved), the code we defined on line 15, for example, would work the same if a, $\mathbf{x}$ and $\mathbf{y}$ were of type `float64`. The same code could be re-used for both saxpy and daxpy.
By setting the environment variable `CUPY_CACHE_SAVE_CUDA_SOURCE` to `TRUE` prior to execution CuPy logs the human-readable C++ source-code of the kernels it created to files.

```
1  import cupy
2  import numpy as np
3
4  # creating inputs on the host
5  x_host = np.arange(10_000, dtype=np.float32)
6  y_host = np.arange(10_000, dtype=np.float32)
7
8  # copying them to device memory
9  x = cupy.asarray(x_host)
10 y = cupy.asarray(y_host)
11 a = cupy.float32(5)
12 z = cupy.empty_like(x)
13
14 # calculating the result on the GPU
15 z = a*x + y
16
17 # retrieving the result from the GPU
18 z_host = cupy.asnumpy(z)
```

Listing 3.1: Source code of the saxpy-kernel using CuPy

In this case for example CuPy created two separate kernels, one for the multiplication of a and $\mathbf{x}$ and another one for the addition of $a\mathbf{x}$ and y. These kernels are presented in Listings 3.2 and 3.3. The kernels are very similar and both show signs (like the redundant semicolon in the second to last line) of being computationally generated from the same template.

```
#include <cupy/complex.cuh>
#include <cupy/carray.cuh>
#include <cupy/atomics.cuh>

typedef float in0_type;
typedef float in1_type;
typedef float out0_type;

extern "C" __global__ void
cupy_multiply__float_float32_float32(
        const float in0,
        const CArray<float, 1, 1, 1> _raw_in1,
        CArray<float, 1, 1, 1> _raw_out0,
        CIndexer<1, 1> _ind)
{
  ;
#pragma unroll 1
  CUPY_FOR(i, _ind.size())
  {
    _ind.set(i);
    const in1_type in1(_raw_in1[_ind.get()]);
    out0_type out0;
    out0 = in0 * in1;
    _raw_out0[_ind.get()] = out0;
    ;
  };
}
```

Listing 3.2: Generated kernel for the multiplication

```
#include <cupy/complex.cuh>
#include <cupy/carray.cuh>
#include <cupy/atomics.cuh>

typedef float in0_type;
typedef float in1_type;
typedef float out0_type;

extern "C" __global__ void
cupy_add__float32_float32_float32(
        const CArray<float, 1, 1, 1> _raw_in0,
        const CArray<float, 1, 1, 1> _raw_in1,
        CArray<float, 1, 1, 1> _raw_out0,
        CIndexer<1, 1> _ind)
{
  ;
#pragma unroll 1
  CUPY_FOR(i, _ind.size())
  {
    _ind.set(i);
    const in0_type in0(_raw_in0[_ind.get()]);
    const in1_type in1(_raw_in1[_ind.get()]);
    out0_type out0;
    out0 = in0 + in1;
    _raw_out0[_ind.get()] = out0;
    ;
  };
}
```

Listing 3.3: Generated kernel for the addition

By inspecting the CuPy source it becomes evident that both of these kernels were created by adapting the template defined in the cython file `_kernel.pyx` [64, lines 47-73] (as shown in Listing 3.4). As tested in Section 2.7.2.7 this way of creating separate kernels for different parts of this simple saxpy-operation is inefficient and comes at the cost of additional kernel-launch overheads and memory transfers. This is the default behavior of CuPy, but CuPy offers ways to circumvent that behavior with the `cupy.fuse` function decorator as used in Listing 3.5.

```
47 cdef function.Function _get_simple_elementwise_kernel(
48         tuple params, tuple arginfos, str operation, str name,
49         _TypeMap type_map, str preamble, str loop_prep='', str after_loop='',
50         tuple options=()):
51     # No loop unrolling due to avoid 64-bit division
52     module_code = string.Template('''
53     ${typedef_preamble}
54     ${preamble}
55     extern "C" __global__ void ${name}(${params}) {
56       ${loop_prep};
57       #pragma unroll 1
```

```
58        CUPY_FOR(i, _ind.size()) {
59          _ind.set(i);
60          ${operation};
61        }
62        ${after_loop};
63      }
64      ''').substitute(
65          typedef_preamble=type_map.get_typedef_code(),
66          params=_get_kernel_params(params, arginfos),
67          operation=operation,
68          name=name,
69          preamble=preamble,
70          loop_prep=loop_prep,
71          after_loop=after_loop)
72      module = compile_with_cache(module_code, options)
73      return module.get_function(name)
```

Listing 3.4: Underlying template

```
1  @cupy.fuse(kernel_name='saxpy')
2  def saxpy(a,x,y):
3      return a*x+y
```

Listing 3.5: The operation re-defined and decorated with the fuse-decorator

Decorating the operation with `cupy.fuse` indicates that the kernels should be fused into a single kernel. The resulting kernel is displayed in Listing 3.6. The code cannot be mistaken for handwritten code because of the many unnecessary it includes. The type-definitions in the device-functions `cupy_mulitply` and `cupy_add` are not used anywhere and similarly there is an excessive number of floats declared in the main kernel on line 39 (originally all of these declarations were on separate lines, here displayed on a single line for conciseness) then those floats are cast to other floats, which again is completely unnecessary. Such superfluous bloat in the code however is no cause for concern, modern compilers recognize that all those definitions and type-casts are not needed and the code that gets executed will not be affected by them.

```
1  #include <cupy/complex.cuh>
2  #include <cupy/carray.cuh>
3  #include <cupy/atomics.cuh>
4
5  __device__ void cupy_multiply(float &in0, float &in1, float &out0)
6  {
7    typedef float in0_type;
8    typedef float in1_type;
9    typedef float out0_type;
10
11   out0 = in0 * in1;
12 }
13
14 __device__ void cupy_add(float &in0, float &in1, float &out0)
```

```
{
  typedef float in0_type;
  typedef float in1_type;
  typedef float out0_type;

  out0 = in0 + in1;
}

extern "C" __global__ void
saxpy(const float v0,
      const CArray<float, 1, 1, 1> _raw_v1,
      const CArray<float, 1, 1, 1> _raw_v2,
      CArray<float, 1, 1, 1> _raw_v5,
      CIndexer<1, 1> _ind)
{
  ;
#pragma unroll 1
  CUPY_FOR(i, _ind.size())
  {
    _ind.set(i);
    const float &v1 = _raw_v1[_ind.get()];
    const float &v2 = _raw_v2[_ind.get()];
    float &v5 = _raw_v5[_ind.get()];
    // 2 operations
    float v3, v4, v0_0, v0_1, v0_2, v1_0, v1_1, v1_2;
    // op  # 0
    v0_0 = static_cast<float>(v0);
    v0_1 = static_cast<float>(v1);
    v0_2 = static_cast<float>(v3);
    cupy_multiply(v0_0, v0_1, v0_2);
    v3 = static_cast<float>(v0_2);
    // op  # 1
    v1_0 = static_cast<float>(v3);
    v1_1 = static_cast<float>(v2);
    v1_2 = static_cast<float>(v4);
    cupy_add(v1_0, v1_1, v1_2);
    v4 = static_cast<float>(v1_2);
    v5 = v4;
    ;
  };
}
```

Listing 3.6: The operation re-defined and decorated with the fuse-decorator

Another way of defining GPU calculations using CuPy is through the `RawKernel`-module as displayed in Listing 3.7. This module gives developers full control over the CUDA-Code that will be compiled by the compiler and allows for a fast, responsive developer experience. By utilizing an interactive Python environment like a Jupyter-notebook one can rapidly alter the kernel without having to set up a C/C++-Project and recompiling the whole program for every change made or losing application state. While this kernel is no faster than the kernel automatically generated by CuPy, there is something to be said

about the conciseness and simplicity of a handwritten kernel. The complete kernel fits within a few lines of code and is very readable.

```
1  import cupy
2  import numpy as np
3  import math
4
5  # manually defining the kernel in C/C++
6  saxpy_rawKernel = cupy.RawKernel(
7      """
8      extern "C" __global__
9      void saxpy(float a, float* x, float* y, float* z, long n) {
10         long idx = (blockDim.x * blockIdx.x + threadIdx.x);
11         if(idx < n){
12             z[idx] = a*x[idx] + y[idx];
13         }
14     }
15     ""","saxpy"
16 )
17
18 # defining a function to call the kernel with an appropriate grid
19 def saxpy(a,x,y,z):
20     n = x.shape[0]
21     blocksize = 512
22     gridsize = math.ceil(n / blocksize)
23     saxpy_rawKernel((gridsize,),(blocksize,),(a, x, y, z, n))
24
25 # creating inputs on the host
26 x_host = np.arange(10_000, dtype=np.float32)
27 y_host = np.arange(10_000, dtype=np.float32)
28
29 # copying them to device memory
30 x = cupy.asarray(x_host)
31 y = cupy.asarray(y_host)
32 a = cupy.float32(5)
33 z = cupy.empty_like(x)
34
35 # calling the function, after this function call z contains the result
36 saxpy(a,x,y,z)
37
38 # retrieving the result from the GPU
39 z_host = cupy.asnumpy(z)
40
41 # checking the result for correctness
42 np.allclose(a_host*x_host+y_host,z_host)
```

Listing 3.7: CuPy exampled using a handwritten CUDA-Kernel

The very same CUDA-Kernel can also be expressed using purely Python with a largely unchanged syntax (see Listing 3.8). The kernel defined on lines 3-7 is a drop-in replacement for the kernel defined on lines 6-16 in Listing 3.7 without any further changes except the needed import of cupyx. Note the

absence of any types. The type information is inferred at run-time, which means that this code could be resulted for double-precision or integers of various. This advantage comes at the cost of having to be very careful about the parameters passed to this function otherwise one might inadvertently work with double-precision numbers, which would slow the calculation down.

```
from cupyx import jit

@jit.rawkernel()
def saxpy_kernel(a, x, y, z, n):
    idx = jit.blockDim.x * jit.blockIdx.x + jit.threadIdx.x
    if (idx < n):
        z[idx] = a * x[idx] + y[idx]
```

Listing 3.8: CuPy exampled using a handwritten Python-CUDA-Kernel

When using this feature CuPy warns about this functionality being experimental and the possibility of the syntax possibly changing at a future point in time.

3.2.2 Numba

Numba's `vectorize` and `guvectorize` decorators allow users to turn functions which operate on n-dimensional arrays into `ufuncs` (universal function) or generalized `ufuncs`. As universal functions operate element-wise on arrays, they are parallel by definition and prime candidates for acceleration using GPUs [65]. Listing 3.9 demonstrates the saxpy-kernel as an ufunc. Explicitly controlling the memory-flow between the host and the device by using functions like `numba_cuda.to_device` and `copy_to_host` is entirely optional but good practice. The `vectorize` decorator always generates a single kernel, un-fused operations are not a concern.

```
import numba
import numba.cuda
import numpy as np
import math

# defining the kernel as a typed-ufunc in Python
@numba.vectorize(
    [numba.float32(numba.float32,numba.float32,numba.float32)],
    target="cuda")
def saxpy(a, x, y):
    return a*x+y


# creating inputs on the host
x_host = np.arange(10_000, dtype=np.float32)
y_host = np.arange(10_000, dtype=np.float32)

# copying them to device memory
a = np.float32(5)
x = numba.cuda.to_device(x_host)
y = numba.cuda.to_device(y_host)
z = numba.cuda.device_array_like(x_host)
```

```
23
24 # calling the function, after this function call z contains the result
25 z = saxpy(a,x,y)
26
27 # retrieving the result from the GPU
28 z_host = z.copy_to_host()
29
30 # checking the result for correctness
31 np.allclose(a*x_host+y_host,z_host)
```

Listing 3.9: Numba exampled using an ufunc

Numba also offers a path to define CUDA-Kernels (as demonstrated in Listing 3.10) purely using Python-Syntax very similar to the previous CuPy-example (Listing 3.8) with the only difference being slight differences in the CUDA-specific syntax. A Numba CUDA-kernel for example is called with the following syntax `kernel_name[gridsize, blocksize, stream, shared_memory](kernel_arguments)` whereas a CuPy kernel is invoked using `kernel_name(gridsize, blocksize, kernel_arguments, shared_memory=0)`. There are however large differences between Numba and CuPy in the next steps towards generating executable code out of that Python code describing a kernel. CuPy generates C/C++-code, which is very close to the original Python kernel and easily inspectable, from the kernel, unlike Numba, which generates a series of intermediate representation including NVVM-IR [66] (a subset of the LLVM-IR designed to represent GPU-Kernels) and Parallel Thread Execution (PTX) Code [67] (Parallel Thread Execution). Both of these formats are the assembly languages for two different virtual machines and very-low level (essentially as low level as the x86-assembly language), for example, these languages lack the concepts of for-loops. For-loops and other features developers have grown to be accustomed to can only be expressed as conditional-jumps between different sections of the program.

```
1  import numba
2  import numba.cuda
3  import numpy as np
4  import math
5
6  # defining the kernel in Python
7  @numba.cuda.jit()
8  def saxpy_kernel(a, x, y, z, n):
9          # alternatively:
10         # idx = numba.cuda.blockDim.x * numba.cuda.blockIdx.x + numba.cuda.threadIdx.x
11     idx = numba.cuda.grid(1)
12     if (idx < n):
13         z[idx] = a * x[idx] + y[idx]
14
15 # defining a function to call the kernel with an appropriate grid
16 def saxpy(a,x,y,z):
17     n = x.shape[0]
18     blocksize = 512
19     gridsize = math.ceil(n / blocksize)
20     saxpy_kernel[gridsize,blocksize](a,x,y,z,n)
21
22
23 # creating inputs on the host
```

```
x_host = np.arange(10_000, dtype=np.float32)
y_host = np.arange(10_000, dtype=np.float32)

# copying them to device memory
a = np.float32(5)
x = numba.cuda.to_device(x_host)
y = numba.cuda.to_device(y_host)
z = numba.cuda.device_array_like(x_host)

# calling the function, after this function call z contains the result
saxpy(a,x,y,z)

# retrieving the result from the GPU
z_host = z.copy_to_host()

# checking the result for correctness
np.allclose(a*x_host+y_host,z_host)
```

Listing 3.10: Numba exampled using a handwritten CUDA-Kernel

Listing 3.11 displays a section of the resulting PTX-Code. And while it is entirely possible to reason about and convince oneself of the correctness of this code, it is a very time-consuming process and quickly becomes infeasable as more complex kernels (usually the compiler would transform the code more heavily, only very simple examples have this almost perfect injective mapping between lines written in the kernel and PTX-commands) are examined. As a consequence of this Numba is very opaque to users. The actual computation takes place on the highlighted line 42, the entire saxpy-function compiled down to a single FMA-instruction. Comments were added to explain the commands.
To understand the section presented it is also necessary to know in which registers the passed arguments were stored in.

- a is stored in `%f1`
- a pointer to x is stored in `%rd2`
- a pointer to y is stored in `%rd4`
- a pointer to z is stored in `%rd6`
- n is stored in `%rd8`

```
# read threadIdx.x into register r1
mov.u32                                 %r1,    %tid.x;
# read blockIdx.x into register r2
mov.u32                                 %r2,    %ctaid.x;
# read blockDim.x into register r3
mov.u32                                 %r3,    %ntid.x;
# calculate blockDim.x * blockIdx.x + threadIdx.x and store result in r4
mad.lo.s32                              %r4,    %r3,    %r2,    %r1;
# cast r4 to signed 64-bit integer and store result in rd1
cvt.s64.s32                             %rd1,   %r4;
# checks whether rd1 is greater than rd8, if it p1 becomes true
setp.ge.s64                             %p1,    %rd1,   %rd8;
# jump to label if p1 is true
@%p1 bra                                $L__BB0_2;

```

```
16 # casting rd1 to unsigned 32-bit integer and storing result in r5
17 cvt.u32.u64                          %r5,    %rd1;
18 # set p2 if r5 is less than zero
19 setp.lt.s32                          %p2,    %r5,    0;
20 # set rd9 to rd3 or 0 depending on p2
21 selp.b64                             %rd9,   %rd3,    0,     %p2;
22 # add rd9 and rd1 into rd10
23 add.s64                              %rd10,  %rd9,    %rd1;
24 # cast to global address space
25 cvta.to.global.u64                   %rd11,  %rd2;
26 # shift left by 2 bit and store result in rd12
27 shl.b64                              %rd12,  %rd10,   2;
28 # add rd12 and rd11 into rd13
29 add.s64                              %rd13,  %rd11,   %rd12;
30 # load the contents from rd13 into the register f2
31 ld.global.f32                        %f2,    [%rd13];
32 # the next line perform a similar calculation memory addresses to get y[idx]
33 selp.b64                             %rd14,  %rd5,    0,     %p2;
34 add.s64                              %rd15,  %rd14,   %rd1;
35 cvta.to.global.u64                   %rd16,  %rd4;
36 shl.b64                              %rd17,  %rd15,   2;
37 add.s64                              %rd18,  %rd16,   %rd17;
38 ld.global.f32                        %f3,    [%rd18];
39 # here the fused multiply-add is performed
40 # f2 contains x[idx], f1 is alpha and f3 is y[idx]
41 # the result is stored in f4
42 fma.rn.f32                           %f4,    %f2,     %f1,   %f3;
43 # followed by a calculation where to store the result in global memory
44 selp.b64                             %rd19,  %rd7,    0,     %p2;
45 add.s64                              %rd20,  %rd19,   %rd1;
46 cvta.to.global.u64                   %rd21,  %rd6;
47 shl.b64                              %rd22,  %rd20,   2;
48 add.s64                              %rd23,  %rd21,   %rd22;
49 # finally f4 is stored in z[idx]
50 st.global.f32                        [%rd23],%f4;
```

Listing 3.11: Section of the resulting PTX-Code

3.2.3 PyCUDA

PyCUDA does not offer as much as functionality CuPy or Numba: it does not support expressing CUDA-Kernels using purely Python, and it has no abstractions over CUDA (like shown with CuPy in Listing 3.1 or with Numba in Listing 3.9). What it does offer is a functionality very similar to that of CuPy shown in Listing 3.7, CUDA C/C++-Kernels defined in Python strings, that are compiled just-in-time and a truly complete set of bindings.
The syntax used is slightly different from the one used in CuPy. An example containing the saxpy-kernel is presented in Listing 3.12. Compared to CuPy it is a more verbose API, that closely follows the C/C++ API, one can for example allocate memory without associating a shape or datatype associated with it (line 37) or use a more convenient way of directly copying an existing Numpy array to the device memory (line 38). The way kernels are launched is also slightly different, with the grid- and blockdimension being

passed to the function as keyword-arguments. Performance-wise there are no differences between this library and CuPy, since both libraries are just thin wrappers around the CUDA-API.

```
import pycuda.driver as cuda
import pycuda.autoinit
from pycuda.compiler import SourceModule
import numpy as np
import math

# defining the kernel
mod = SourceModule(
    """
    extern "C" __global__
    void saxpy(float a, float* x, float* y, float* z, long n) {
        long idx = (blockDim.x * blockIdx.x + threadIdx.x);
        if(idx < n){
            z[idx] = a*x[idx] + y[idx];
        }
    }
    """
)
saxpy_kernel = mod.get_function("saxpy")

def saxpy(a, x, y, z):
    n = np.int64(x_host.shape[0])
    blocksize = 512
    gridsize = math.ceil(n / blocksize)
    saxpy_kernel(a, x, y, z, n, block=(blocksize, 1, 1), grid=(gridsize, 1, 1))

# creating inputs on the host
a = np.float32(5)
x_host = np.arange(10_000, dtype=np.float32)
y_host = np.arange(10_000, dtype=np.float32)
z_host = np.empty(10_000, dtype=np.float32)

# copying them to device memory
a = np.float32(5)
x = cuda.mem_alloc(x_host.nbytes)
y = pycuda.gpuarray.to_gpu(y_host)
z = cuda.mem_alloc(x_host.nbytes)
cuda.memcpy_htod(x, x_host)

# calculating the result on the GPU
saxpy(a, x, y, z)
# retrieving the result from the GPU
cuda.memcpy_dtoh(z_host, z)
# verifiying the result
np.allclose(z_host, a*x_host + y_host)
```

Listing 3.12: Source code of the saxpy-kernel using PyCuda

3.2.4 PyOpenCL

PyOpenCL takes an approach different from all the other introduced packages. Instead of targeting GPUs via the CUDA or ROCm ecosystems, this package targets the OpenCL platform, supporting GPUs from all major vendors and many CPUs. OpenCL uses similar concepts as CUDA, but a different terminology (as listed in Table 3.1). The most striking difference is the use of the term "shared memory" in CUDA and "local memory" in OpenCL for the same concept. Both of these terms refer to memory that is shared between threads or work-item within a block or work-group, from a different viewpoint the memory is not shared between the threads but local to a SM and its set of resident workers.

Category	CUDA term	OpenCL term
Execution	Kernel	Kernel
	Thread	Work-item
	Warp	Wavefront
	Block	Work-group
	Grid	N-D Range
	Stream	Command Queue
Memory	Global Memory	Global Memory
	Device Memory	Device Memory
	Constant Memory	Constant Memory
	Shared Memory	Local Memory
	Local Memory	Private Memory

Table 3.1: Terms of equal meaning in CUDA and OpenCL adapted from [68]

Listing 3.13 shows the saxpy-kernel implemented using this library. The basic structure, defining the kernel(s), copying data to the device, executing the kernel and retrieving the result from the device, of the code is unchanged, only the way those operations are described has changed slightly.

```
1  import numpy as np
2  import pyopencl as cl
3
4  ctx = cl.create_some_context()
5  queue = cl.CommandQueue(ctx)
6
7  # defining and compiling the kernel
8  prg = cl.Program(ctx, """
9  __kernel void saxpy(
10     const float a,
11         __global const float *x,
12         __global const float *y,
13         __global float *z)
14 {
15   int gid = get_global_id(0);
16   z[gid] = a*x[gid] + y[gid];
17 }
18 """).build()
19
```

```
20 knl = prg.saxpy
21
22
23 # creating inputs on the host
24 a = np.float32(5.0)
25 x_host = np.arange(10_000, dtype=np.float32)
26 y_host = np.arange(10_000, dtype=np.float32)
27 z_host = np.empty(10_000, dtype=np.float32)
28
29 # copying them to device memory
30 mf = cl.mem_flags
31 x = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=x_host)
32 y = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=y_host)
33 z = cl.Buffer(ctx, mf.WRITE_ONLY, z_host.nbytes)
34
35 # calculating the result on the GPU
36 global_size = x_host.shape[0]
37 local_size = 1
38 knl(queue, (global_size,), (local_size,), a, x, y, z)
39
40 # retrieving the result from the GPU
41 cl.enqueue_copy(queue, z, z_host)
42
43 # verifiying the result
44 np.allclose(z_host, a*x_host + y_host)
```

Listing 3.13: Source code of the saxpy-kernel using PyOpenCL

Since this package does not use CUDA, we can not make any assumptions about the kernel's performance. When using a library that targets CUDA we can be sure that the kernel always executes equally fast, no matter which package interfaces with the device driver's API, we only need to profile it once using one of the profiling tools CUDA provides us with (Nsight Compute and Nsight Systems).
When utilizing OpenCL those profilers cannot be used and to our knowledge there are no comparable profiler tools for OpenCL used with Nvidia hardware (for AMD GPUs there is rocprofiler [69]) available. This however does not imply that OpenCL code cannot be profiled at all, the kernel-calls such as the one on line 38 of Listing 3.13 returns a data-structure, which contains information about the time the task was queued, started and finished. Another possible work-around takes advantage of the fact that OpenCL-toolchain also produces an intermediate representation, which can be output to a file and analyzed. One could take that PTX-code, insert it into a CUDA program and profile the resulting program.

3.2.4.1 Porting a non-trivial kernel to OpenCL

To get a sense of the performance we translate the kernel used for the first (1D forward FFT of the rows of an image using shared/local memory) stage of the program discussed in Section 2.7.2. The process of porting consists of adapting the already written CUDA-Kernels by changing specific syntax (like swapping the idiomatic `int tid = blockDim.x * blockIdx.x + threadIdx.x` to `int gid = get_global_id(0)`) and making sure to replace all C++ features like templates, classes (including operator overloading) and lambda-functions. For the function the kernel calls (referred to as device-functions) only very few changes are needed (as displayed in Listings 3.14 and 3.15). The qualifier `__device__` is not needed in OpenCL, since re-use between device and host code is not supported in

OpenCL and OpenCL-Code is usually defined in its own source file and always JIT compiled to achieve portability. Aside from that there were slight modifications needed in the `raise_omega`-function, the interface of the `sincos` functions are not identical and OpenCL does not support taking the addresses of vector elements (as done in the CUDA-Code).

```
/*
Function to multiply the two complex numbers
↪ x1 and x2
The result is stored in x1, which is why x1
↪ needs to be passed by reference
*/
__device__ inline void complex_mult(float2*
↪ x1, float2 x2)
{
    float real_result = x1->x*x2.x-x1->y*x2.y;
    float imag_result = x1->x*x2.y+x2.x*x1->y;
    x1->x = real_result;
    x1->y = imag_result;
}

/*
Function to raise a complex number with
↪ unit-magnitude by an integer power.
Magnitude remains unchanged, phase is
↪ multiplied by the exponent
*/
__device__ inline float2 raise_omega(float2
↪ omega, int pow)
{
    float angle = atan2(omega.y, omega.x);
    angle *= pow;
    float2 retval;
    sincosf(angle, &retval.y, &retval.x);
    return retval;
}
```

Listing 3.14: Used device functions for CUDA

```
/*
Function to multiply the two complex numbers
↪ x1 and x2
The result is stored in x1, which is why x1
↪ needs to be passed by reference
*/
void complex_mult(float2 *x1, float2 x2)
{
    float real_result = x1->x*x2.x-x1->y*x2.y;
    float imag_result = x1->x*x2.y+x2.x*x1->y;
    x1->x = real_result;
    x1->y = imag_result;
}

/*
Function to raise a complex number with
↪ unit-magnitude by an integer power.
Magnitude remains unchanged, phase is
↪ multiplied by the exponent
*/
float2 raise_omega(float2 omega, int pow)
{
    float angle = atan2(omega.y, omega.x);
    angle *= pow;
    float cos;
    float sine = sincos(angle, &cos);
    return (float2)(cos,sine);
}
```

Listing 3.15: Used device functions for OpenCL

Porting the kernel itself is also a rather straightforward process of adapting syntax and finding equivalent functions in the respective libraries, the changed section are contrasted in Listings 3.16 and 3.17. The first and most important change between the two snippets lies in the fact that the CUDA-function is a templated-function, which enables it to express the forward and backward transform with the same function controlled by the boolean parameter `inverse`. This is not possible in OpenCL and would instead have to be implemented using multiple functions or templating the Python-string, representing the function, at run-time. Further changes affect platform specific syntax like `barrier(CLK_LOCAL_MEM_FENCE)` instead of `__syncthreads()` and `get_local_id(0)` instead of `threadIdx.x`.

```
template <unsigned int size, bool inverse>
__global__ void my_fft_u2c(
        const unsigned char * input,
        float *output,
        int original_size,
        float omega_real,
        float omega_imag)
{
  __shared__ float t1[size * 2];
  __shared__ float t2[size * 2];

  int tid = blockDim.x*blockIdx.x+threadIdx.x;
  int iteration = 0;

  t1[threadIdx.x*4] = input[tid*2];
  t1[threadIdx.x*4+1] = 0;
  t1[threadIdx.x*4+2] = input[tid*2+1];
  t1[threadIdx.x*4+3] = 0.;
  ...
  float2 omega = make_float2(omega_real,
↪  omega_imag);
  float2 twiddle = raise_omega(omega,
↪  (inverse?1:-1) * posInProb * n_Problems);
  __syncthreads();
  ...
  output[tid*4] = input_[threadIdx.x*4] *
↪  (inverse ? 1. / size : 1);
}
```

Listing 3.16: Kernel expressed in CUDA

```
__kernel void my_fft_u2c(
    __global const unsigned char *input,
    __global float *output,
    int original_size,
    float omega_real,
    float omega_imag)
{
  __local float t1[1024 * 2];
  __local float t2[1024 * 2];

  int gid = get_global_id(0);
  int iteration = 0;

  t1[get_local_id(0)*4] = input[gid*2];
  t1[get_local_id(0)*4+1] = 0;
  t1[get_local_id(0)*4+2] = input[gid*2+1];
  t1[get_local_id(0)*4+3] = 0;
  ...
  float2 omega = (float2)(omega_real,
↪  omega_imag);
  float2 twiddle = raise_omega(omega,
↪  -posInProb*n_Problems);
  barrier(CLK_LOCAL_MEM_FENCE);
  ...
  output[gid*4] = input_[get_local_id(0)*4];
  ...
}
```

Listing 3.17: Modified sections of the kernel for OpenCL

3.2.4.2 Performance comparison between OpenCL and CUDA

After having ported the FFT-Kernel it can be tested and benchmarked using PyOpenCL. The performance of the CUDA-Code is listed in Table 2.2 as 155 μs for a 1024 by 1024 input image, whereas the OpenCL-Code reports an execution time of 217 μs. This performance regression by about 40% can most-likely be attributed to the OpenCL compiler producing less optimized PTX-Code than NVCC.
To confirm this suspicion, the generated PTX-Code can be retrieved and integrated into a CUDA program. This allows us to profile both PTX-Codes (the one generated by OpenCL and the one generated by the Nvidia Compiler) with the same tools for reliable information. This experiment showed that the the regression in performance can in fact be attributed to worse PTX-Code being generated by the OpenCL toolchain, the kernel also takes around 220 μs when executed using the CUDA-API.

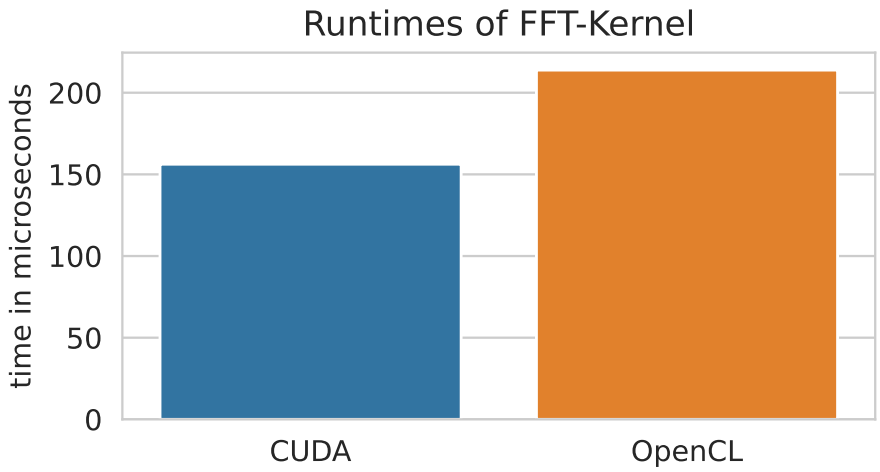


Figure 3.1: Comparing the run-times of the equivalent FFT-Kernels, both times were measured using Nsight Systems

Using Nsight Compute for a detailed analysis of both resulting kernels, which should perform exactly the same computations, reveals that they are structured differently (note: Nsight Compute cannot be used to accurately measure the run-times of the kernels due to its profiling overhead, Nsight Systems is able to perform this analysis). The version compiled using the OpenCL toolchain made much less use of fused FP32 operations like FMA (fused-multiply and addtion), which increased the overall number of executed instructions by 37.52 % and slowed the program down accordingly. Similar behavior could be observed whenever the PTX-Code is compiled by a different toolchain (e.g. NVCC, NVRTC, Clang, Numba, OpenCL). Even the same source code (or in this case almost identical versions expressed in slightly different dialects of C) is not guaranteed or likely to result in the same PTX-Code or performance.
This however does not imply that OpenCL is always worse than equivalent CUDA code. It is entirely possible that the OpenCL-Compiler could create PTX-Code as fast by employing additional optimization techniques.

3.2.5 PyTorch

Pytorch allows the creation and movement of its n-dimensional arrays (referred to as tensors) between different devices like the CPU or GPUs. This approach of using PyTorch as a GPU-aware drop-in replacement to Numpy is very similar to using CuPy for this task. Operations between Tensors located on the same GPU are automatically executed on that GPU using CUDA-Code. The saxpy example shown in Listing 3.18 executes three kernels:

- an arange-kernel to populate the $\mathbf{x}$-tensor
- a kernel to multiplies the $\mathbf{x}$ and a-tensors
- a kernel to add the result of $a\mathbf{x}$ and $\mathbf{y}$

So just like CuPy it executes several different elementwise kernels, which could easily be fused to improve efficiency and computational intensity.

```
1 import torch
2 import numpy as np
3
4 # creating inputs on the device
5 x = torch.arange(10_000, device="cuda:0", dtype=torch.float32)
6
```

```
# creating input on the host and copying it to the device
y_host = np.arange(10_000, dtype=np.float32)
y = torch.tensor(y_host, device="cuda")
# a is a tensor on the default device (the cpu)
a = torch.tensor(5.0)


def saxpy(a,x,y):
    return a*x+y

# calculating the result on the GPU
# the function itself is device-agnostic and
# calls either CPU- or device-Code depending on its inputs.
z = saxpy(a, x, y)

# copying result back to CPU and converting to Numpy
z_host = z.to(device="cpu").numpy()

# verifiying result
np.allclose(a.numpy()*x.to("cpu").numpy() + y_host, z_host)
```

Listing 3.18: Source code of the saxpy-kernel using PyTorch

With PyTorch the solution to unfused kernels is called TorchScript. TorchScript is a JIT-Compiler, which can be used to either trace PyTorch programs once and export them as a standalone module for use in C++-programs without any Python involved or to compile functions as they are called to benefit from compile-time optimizations. TorchScript is used by decorating the function. This use-case of TorchScript is showcased in Listing 3.19.

```
@torch.jit.script
def saxpy_jit(a,x,y):
    return a*x+y
```

Listing 3.19: Source code of the saxpy-kernel using PyTorch

When executing a Python file containing such a decorated function, one can set the environment-variable `PYTORCH_JIT_LOG_LEVEL` to have debug-information about the JIT-Compilation process reported. To receive the resulting CUDA-C++ one can set the variable to »`cuda_codegen`. This instructs PyTorch to log all the debug information related to the generation of the CUDA kernel displayed in Listing 3.20. The operations are properly fused, both multiplication and addition are performed in this kernel as visible on line 8, and should map to the efficient FMA-instruction. This can be confirmed by profiling the kernel with Nsight Compute and examining the instructions executed. Interestingly the code avoids defining the idiomatic global thread-id like `int tid = blockDim.x * blockIdx.x + threadIdx.x` and instead accesses the `blockIdx.x` and `threadIdx.x` multiple times over the course of the program and never accesses `blockDim.x`, but hard-codes it as 512. This is not an issue, since the compiler is smart enough to optimize away the repeating accesses to `threadIdx.x` and `blockIdx.x` and the fixed `blockDim.x` is also not an issue, because the kernel is only ever compiled for this set of inputs without any consideration for re-use and generality.

```
extern "C" __global__
void fused_mul_add(float* ty_1, float* ta_1, float* tx_1, float* aten_add) {
        {
        if ((long long)(threadIdx.x) + 512ll * (long long)(blockIdx.x)<100001l ? 1 : 0) {
            float v = __ldg(ta_1 + 0ll);
    float v_1 = __ldg(tx_1 + (long long)(threadIdx.x) + 512ll * (long long)(blockIdx.x));
    float v_2 = __ldg(ty_1 + (long long)(threadIdx.x) + 512ll * (long long)(blockIdx.x));
    aten_add[(long long)(threadIdx.x) + 512ll * (long long)(blockIdx.x)] = v * v_1 + v_2;
  }}
}
```

Listing 3.20: Source code of the saxpy-kernel using PyTorch

Note: the fact that the data type `long long` is used in Listing 3.20 is not of significance. When using the NVCC compiler this is an 8 byte integer, equivalent to a standard `long`. The same behavior is observed when using g++ or clang on regular C/C++ programs. This is compliant with the standards, which only guarantee `long long` to be as large as `long` and at least 8 bytes [70].

3.2.6 JAX

JAX is conceptually very similar to other frameworks designed for deep-learning frameworks like PyTorch or TensorFlow. Besides the use in the machine-learning domain it also duplicates Numpy's functionality and can be used as a GPU and TPU accelerated library for scientific computing. Major difference are the way JAX handles devices memory (by default 90% of the available device memory is pre-allocated at startup) and the precision of floating point numbers (by default single precision is *enforced* and extra steps are needed to enable double-precision support, this choice is owed to the limited device memory and performance disparity between single and double-precision floating point numbers on GPUs [71]). Listing 3.21 outlines how the saxpy-kernel could be defined using JAX.

```
import jax
import os
import numpy as np

# creating inputs on the device
# no device is specified, JAX always uses the GPU or TPU if available
x = jax.numpy.arange(10_000, dtype=jax.numpy.float32)
y = jax.numpy.arange(10_000, dtype=jax.numpy.float32)
a = jax.numpy.float32(5)

def saxpy(a,x,y):
        return a*x+y

# calculating the result on the GPU
# the function itself is device-agnostic and
# calls either CPU- or device-Code depending on its inputs.
z = saxpy(a,x,y)

# verifiying result
# copying results back to the host is done implicitly
np.allclose(z, a*x+y)
```

Listing 3.21: Source code of the saxpy-kernel using JAX

Just like PyTorch, JAX also compiles two kernels for the function call of `saxpy`, one for the multiplication of `a·x` and one for the addition of `y`. In fact JAX has no way of knowing that the multiplication will be followed by an addition at the time the bytecode to multiply `a` and `x` is executed (the path taken by the CPython interpreter is very similar to the outline in Figure 1.1). Interestingly those kernels consist of 10 blocks with 256 threads each, so only 2560 threads are launched for 10000 multiplications and additions respectively. This works because the kernels JAX compiled are vectorized, each thread is responsible for 4 operations and makes use of special vectorized instructions to load and store 4 floats at a time. This is beneficial because a lower number of warps has to be schedule, which is associated with a lower overhead and the vectorized memory accesses are very efficient. The downside of this approach is a lower occupancy caused by the smaller number of threads, especially on these relatively small inputs. A section of the generated PTX-Code is presented in Listing 3.22, with the vectorized section being highlighted.

```
32 ld.param.u64            %rd4, [fusion_param_0];
33 ld.param.u64            %rd5, [fusion_param_2];
34 cvta.to.global.u64      %rd1, %rd5;
35 ld.param.u64            %rd6, [fusion_param_1];
36 cvta.to.global.u64      %rd2, %rd6;
37 cvta.to.global.u64      %rd3, %rd4;
38 ld.global.nc.u32        %r6, [%rd3];
39 cvt.rn.f32.s32          %f1, %r6;
40 mul.wide.u32            %rd7, %r1, 4;
41 add.s64                 %rd8, %rd2, %rd7;
42 ld.global.nc.v4.f32     {%f2, %f3, %f4, %f5}, [%rd8];
43 mul.rn.f32              %f6, %f2, %f1;
44 add.s64                 %rd9, %rd1, %rd7;
45 mul.rn.f32              %f7, %f3, %f1;
46 mul.rn.f32              %f8, %f4, %f1;
47 mul.rn.f32              %f9, %f5, %f1;
48 st.global.v4.f32        [%rd9], {%f6, %f7, %f8, %f9};
```

Listing 3.22: Generated PTX-code of the multiplication kernel

To receive a single, fused kernel the function has to be decorated with `jax.jit`. The relevant section of the resulting kernel is shown in Listing 3.23. In the resulting code the `mul` and `add` instructions are both present in the same kernel but unlike PyTorch, JAX failed to converted them to FMA instructions, which would be a low-hanging optimization in this context.

```
32 ld.param.u64    %rd5, [fusion_param_0];
33 ld.param.u64    %rd6, [fusion_param_3];
34 cvta.to.global.u64      %rd1, %rd6;
35 ld.param.u64    %rd7, [fusion_param_1];
36 ld.param.u64    %rd8, [fusion_param_2];
37 cvta.to.global.u64      %rd2, %rd8;
38 cvta.to.global.u64      %rd3, %rd7;
39 cvta.to.global.u64      %rd4, %rd5;
40 ld.global.nc.f32        %f1, [%rd4];
41 mul.wide.u32    %rd9, %r1, 4;
```

```
add.s64         %rd10, %rd3, %rd9;
ld.global.nc.v4.f32     {%f2, %f3, %f4, %f5}, [%rd10];
mul.rn.f32      %f6, %f1, %f2;
add.s64         %rd11, %rd2, %rd9;
ld.global.nc.v4.f32     {%f7, %f8, %f9, %f10}, [%rd11];
add.rn.f32      %f11, %f6, %f7;
add.s64         %rd12, %rd1, %rd9;
mul.rn.f32      %f12, %f1, %f3;
add.rn.f32      %f13, %f12, %f8;
mul.rn.f32      %f14, %f1, %f4;
add.rn.f32      %f15, %f14, %f9;
mul.rn.f32      %f16, %f1, %f5;
add.rn.f32      %f17, %f16, %f10;
st.global.v4.f32        [%rd12], {%f11, %f13, %f15, %f17};
```

Listing 3.23: Generated PTX-code of the fused kernel

3.2.7 Official Python bindings

The official Python bindings for CUDA are very similar to PyCUDA in terms of functionality offered. They are wrappers around the CUDA API as it is offered in C/C++ and therefore do not offer any additional functionality like the ability to compile Python code to CUDA-Code. It also does not directly interact with established standards like DLPack or the CUDA Array Interface, which makes it harder use. It is not targeted at end-users, but rather at developers who want to integrate CUDA functionality into their own Python packages, without having to write a wrapper themselves.
Listing 3.24 shows how the saxpy-kernel could be called using this library. It contains a lot of boilerplate code, which is taken care of by the other libraries discussed in this section.

```
from cuda import cuda
import numpy as np
import common
from helper_cuda import checkCudaErrors, findCudaDeviceDRV
import ctypes

# initializing CUDA
checkCudaErrors(cuda.cuInit(0))
cuDevice = findCudaDeviceDRV()

# defining the kernel
saxpy_kernel_src = '''
extern "C" __global__ void saxpy(float a, float* x, float* y, float* z, long n) {
    long idx = (blockDim.x * blockIdx.x + threadIdx.x);
    if(idx < n){
        z[idx] = a*x[idx] + y[idx];
    }
}'''

# compiling the kernel
kernelHelper = common.KernelHelper(saxpy_kernel_src, int(cuDevice))
```

```
saxpy_kernel = kernelHelper.getFunction(b'saxpy')

N = 10_000
# creating inputs on the host
a = np.float32(5)
x_host = np.arange(N, dtype=np.float32)
y_host = np.arange(N, dtype=np.float32)
z_host = np.empty(N, dtype=np.float32)

d_x = checkCudaErrors(cuda.cuMemAlloc(x_host.nbytes))
d_y = checkCudaErrors(cuda.cuMemAlloc(y_host.nbytes))
d_z = checkCudaErrors(cuda.cuMemAlloc(z_host.nbytes))

checkCudaErrors(cuda.cuMemcpyHtoD(d_x, x_host.ctypes.data, x_host.nbytes))
checkCudaErrors(cuda.cuMemcpyHtoD(d_y, y_host.ctypes.data, y_host.nbytes))

# executing the kernel
kernelArgs = ((a, d_x, d_y, d_z, x_host.size),
              (ctypes.c_float, None, None, None, ctypes.c_long))
threads_per_block = (256, 1, 1)
blocks = ((x_host.size + threads_per_block[0] - 1) // threads_per_block[0], 1, 1)
checkCudaErrors(cuda.cuLaunchKernel(saxpy_kernel,
                                    *blocks,
                                    *threads_per_block,
                                    0, 0,
                                    kernelArgs, 0))

checkCudaErrors(cuda.cuMemcpyDtoH(z_host.ctypes.data, d_z, z_host.nbytes))

# verifying result
np.allclose(z_host, a*x_host + y_host)
```

Listing 3.24: Source code of the saxpy-kernel using the official CUDA bindings

3.3 Taxonomy of Packages

The discussed libraries operate on different levels of abstraction and stages of the compilation pipeline. Some of them (PyCUDA, PyOpenCL, the official cuda-python bindings) do not offer the functionality to compile Python code to GPU-executable code at all, but just forward C/C++ to the respective compiler and return functions callable from within Python. Figure 3.2 outlines how the compilation process of the remaining packages is designed. The packages can be categorized into two distinct groups. CuPy and PyTorch emit C/C++ and rely on NVCC to perform the later stages of the compilation process (compiling it to an intermediate representation, performing numerous optimization passes on that IR, and finally emitting an executable binary), whereas Numba and JAX forego the step of creating C/C++ representations and instead directly produce LLVM-IR compatible code, which is then injected into the regular NVCC compilation path at a later stage.
The importance of the LLVM project for modern compilers cannot be overstated. As NVCC itself is also based on LLVM [72], there is no path to compile CUDA Code without depending on components of LLVM.

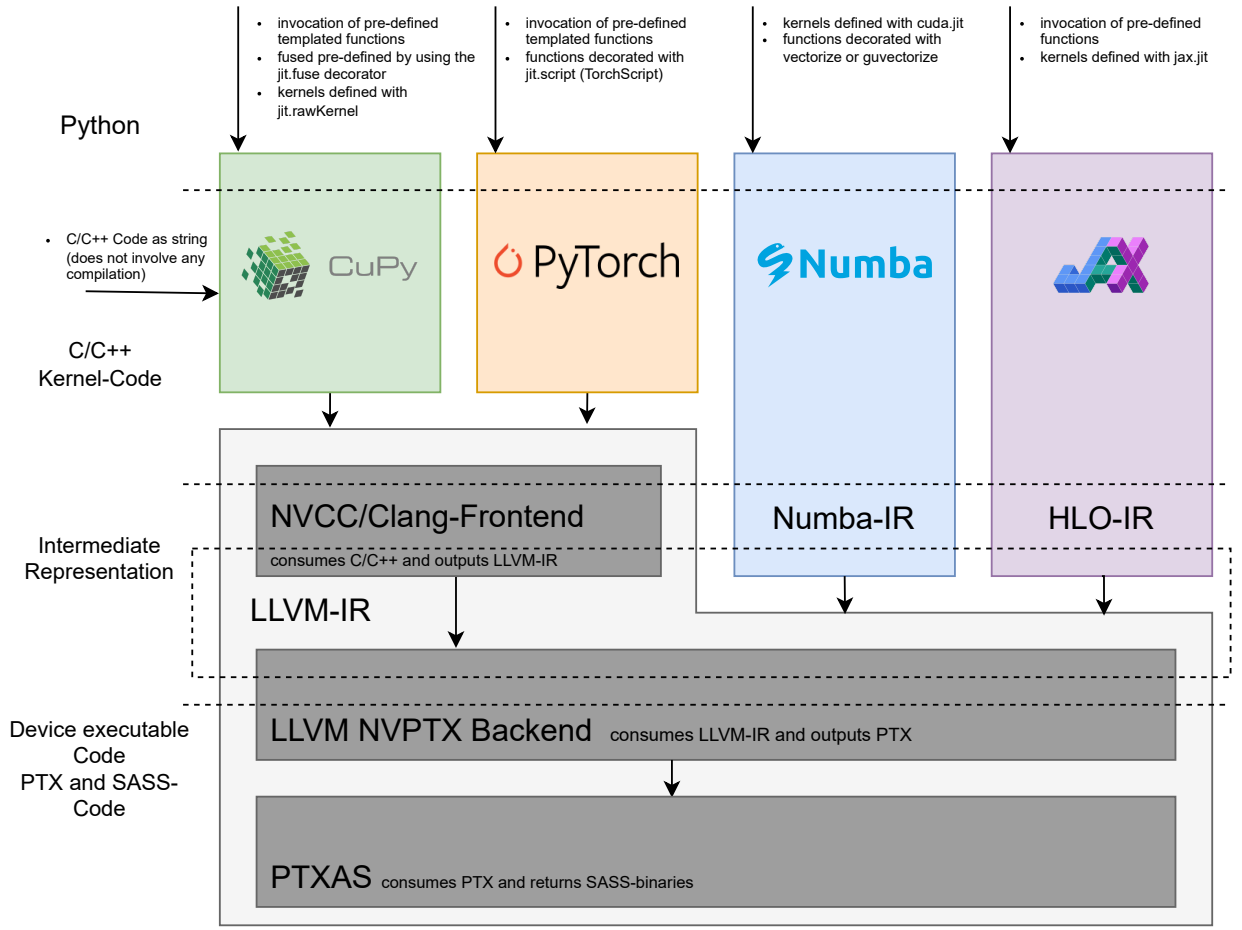


Figure 3.2: High-level overview over the compilation path of the discussed libraries for Nvidia targets; PyCUDA, PyOpenCL and the official python-cuda bindings were omitted from this graphic since those package do not offer compiling capabilities

3.4 Gauging their popularity

Table 3.2 compares the number of downloads of the discussed packages as retrieved from PyPI (Python Package Index) for the 6 month period ranging from February 2022 through July 2022. This is helpful to get a rough idea of the relative popularity of the individual packages. Numpy is included as a baseline, as virtually every Python installation used for scientific computing will have Numpy installed. The packages Numpy, Numba, torch and jax, which are the most downloaded packages on this list, are not exclusively used for GPU-accelerated usage. Numba, Pytorch and Jax can all be used without having access to a GPU and Numpy is designed solely for calculations on the CPU.
Of the remaining packages, which are solely used for accelerating workloads with GPUs, PyOpenXl is downloaded most often, suggesting a number of popular programs rely on it over the very similar PyCuda to support the wider range of device OpenCL is capable of targeting.
The relationship between the different flavors of CuPy-packages (ROCm and CUDA) is also of great interest. The version supporting exclusively CUDA is downloaded roughly 12 times as often as the alternative version which supports both Nvidia and AMD hardware. This is most likely due to the fact that the ROCm-version is in an experimental state of development.

package	number of downloads
numpy	683,099,856
numba	50,663,816
torch	49,346,636
jax	2,849,312
pyopencl	475,998
cupy-cuda	237,278
pycuda	192,625
cuda-python	40,589
cupy-rocm	19,203

Table 3.2: Download figures for the discussed figures from PyPi

Note that PyPI is not the only repository for Python packages and the presented numbers therefore do not represent the absolute usage numbers of those pacakges. Packages can also be installed by downloading the sources manually or via alternative popular repositories such as anaconde [21]. Nvidia's package containing the official Python bindings (listed in the table, PyPI and Github as `cuda-python`) for example do not even list retrieving the package from PyPI as a method to install the package on the projects Github page, even though Nvidia publishes the package on the index [73].

3.5 Interoperability between the frameworks

After having discussed a variety of packages and libraries which facilitate access to the GPU using the Python language it is important to clarify that users do not necessarily have to choose a single preferred package out of the available options. This section will introduce the two widely used interfaces used to exchange data residing on the GPU between the different libraries. It is for example possible to pass an object created with CuPy to a GPU-function that was defined by using Numba functionality (or vice-versa).
The two most important of those interface are the cuda array interface [74] and DLPack [75].

3.5.1 The CUDA Array Interface

The cuda array interface is a dictionary attribute of an object (stored under the name `__cuda_array_interface__`), which contains information about data stored on a CUDA-capable GPU. This dictionary must provide the following information:

- a tuple describing the shape of the n-dimensional array
- a typestring defining the type of the individual elements
- a pointer to the first element of the data (since Python lacks the concept of fixed size pointers this information is stored as a plain integer)

This is where an important concept of low-level programming becomes visible: Data in memory has no inherent meaning, but can be interpreted in different ways. It is entirely up to the user if an array of floating point numbers in memory should be treated as a one-dimensional, "flat list" of floating point numbers or as an n-dimensional grid of complex numbers each representated by two floating point numbers.
In optional entries the dictionary may also contain information about the strides (the numbers of bytes to skip to access neighboring elements in different dimensions) of the n-dimensional array and the stream in which the data may be altered and accessed to allow for synchronization with that stream before accessing the data. See Numbas documentation on the topic [74] for the detailed explanation of the interface.

The cuda array interface is supported by the following discussed packages (and a number of packages not discussed): Numba, CuPy, Jax, PyCUDA.

3.5.2 DLPack

DLPack is very similar to the aforementioned cuda array interface with the addition of also containing information about the device and context the data is associated with. I.e. it is applicable to a wide range of environments including, but not limited, to accelerated computing using CUDA, it also supports describing data residing within CPU memory, on GPUs in non-CUDA contexts (like OpenCL, ROCm or SYCL) or on DSPs and FPGAs.
A number of popular packages integrate DLPack in their offerings including Numpy, CuPy and PyTorch.

3.6 OpenCV Demonstration Program using Python

Even though many libraries, which utilize GPUs support either DLPack, the Cuda Array Interface or both, some of them choose to support neither. One of those libraries is OpenCV. As laid out in Section 2.7.2.1 OpenCV can be configured to provide Python bindings, this enables users to use OpenCV from within Python scripts. This allows us to express the C++ program introduced in Section 2.7.2 as a Python script.
This change from C++ to Python promises the well-known benefits of the Python language and its ecosystem:

- Interactive Jupyter notebooks enabling fast development cycles
- No explicit compilation step
- Less set-up code, compact notation

This section will compare and contrast the C++ and Python version of the demonstration program and showcase how to adapt libraries, which support neither DLPack nor the Cuda Array Interface.

3.6.1 Creation of matrices in the device memory

Listings 3.25 and 3.26 demonstrate how to allocate matrices of a specific size on the GPU within the OpenCV context using C++ and Python respectively. There is a fundamental difference in the construction of that matrix: In C++ one can choose to allocate the memory with a different library (in this case thrust was used) and hand the preallocated chunk of memory to OpenCV. This ensures that the memory, the matrix will be stored in, is actually a single continious area of memory, which is an assumption we made when writing the CUDA kernels like the Stockham FFT kernel in Listing 2.7. By default OpenCVs `GpuMats` are not necessarily continious but every row is aligned in memory depending on the specific hardware [76], this is not of great concern when dealing with rows, which have a known power-of-2 size, but in more general cases this has to be kept in mind.
In Python this practice of pre-allocating memory is not possible. While we have access to libraries which could allocate memory on the device (CuPy and PyTorch among others), it is not possible to pass the address of that memory to the `GpuMat` constructor, since that specific constructor signature (note how the prefix `CV_WRAP` is missing on line 137 of the header file `cuda.hpp` [77]; The details of this mechanism are discussed in the OpenCV documentation [78]) is not exposed to the Python bindings.

```cpp
int bsize = mx * my; // image size
// allocating array of type float2 for the
↪  specified number of pixels
// on the device
thrustDvec<float2> dev_b_pre(bsize);
// creating opencv matrix from the allocated
↪  memory
// this gives meaning to the memory
cuda::GpuMat dev_b(my, mx, CV_32FC2,
↪  dev_b_pre.data().get());
```

Listing 3.25: Creation of device matrices in C++

```python
// creating a opencv matrix with the given
↪  dimensions
// note that this uses a different constructor
dev_b = cv2.cuda.GpuMat(1024,1024,cv2.CV_32FC2)
```

Listing 3.26: Equivalent code in Python

3.6.2 Passing Matrices to GPU Kernels

In order to operate on the matrix with CUDA kernels, one has to pass the pointer to the matrix' memory to a kernel (displayed in Listing 3.28). Using C++ this is a straightforward process, since that address can be queried by taking the address of the `data` attribute of the GpuMat object. Using Python however this is not as easy, since the kernels themselves are wrapped by libraries like CuPy and those wrapped kernels do not take memory addresses (the memory address of the memory OpenCV allocated on the device is obtainable by querying the objects `cudaPtr()` method) as arguments but objects which implement either the CUDA Array Interface or DLPack. This means we have to construct such an object describing the matrix and subsequently pass that object to a wrapped kernel. To achieve this a class `GpuMatWrapper` as seen in Listing 3.27 was implemented. This class does nothing more than representing an object with a valid CUDA Array Interface dictionary pointing to the OpenCV GpuMat.

```python
class GpuMatWrapper:
        """
        Acts as a cuda array interface compatible wrapper aroung opencv GpuMats
        """
        def __init__(self, a: cv2.cuda_GpuMat):
                data_type = a.type()
                type_string = ""
                if data_type == 0:
                        type_string = '<u1'
                elif data_type == 13:
                        type_string = '<c8'
                self.__cuda_array_interface__ = {
                        'shape': a.size(),
                        'typestr': type_string,
                        'descr': [('real', '<f4'), ('imag', '<f4')] if data_type == 13 else [('',
                        ↪  type_string)],
                        'data': (a.cudaPtr(), False),
                        'version': 3
                }
```

Listing 3.27: Source code of the GpuMatWrapper class

Having that class enables us to operate on OpenCV GpuMats with all libraries discussed in Section 3.1 as one would expect (shown in Listing 3.29).

```
// standard notation of invoking a kernel in
↪  CUDA C/C++
// this kernel performs the 1D-FFT across the
↪  rows of the dev_a matrix
// and stores the result in the dev_b matrix
my_fft_u2c<1024, false><<<1024, 512>>>(
  (uchar *)dev_a.data,
  (float *)dev_b.data,
  1024, ...
);
```

Listing 3.28: Invoking a kernel in C++

```
# Wrapping the matrices in CuPy matrices
# this is a zero-copy operation
a_cp = cp.asarray(GpuMatWrapper(dev_a))
b_cp = cp.asarray(GpuMatWrapper(dev_b))

# Passing those objects to a CuPy Kernel
fft_stockham_cuda_1024_u2c(
  (1024,), (512,), # the launch parameters
  (a_cp, b_cp, 1024, omega.real, omega.imag) #
  ↪  the kernel parameters
)
```

Listing 3.29: Equivalent code in Python

With these adaptations in place porting the demonstration program from C++ to Python is a painless process. The resulting Python code is not only more conciser, but also easier to modify than it's C++ counterpart, while retaining equivalent performance. The programs can be found in the `opencv_demo`

4 FFTs and Windowing

4.1 The Discrete Fourier Transform

The discrete Fourier transform (DFT) is a mathematical operation that transforms a signal from the time (or spatial) domain to the frequency domain. It is a very important tool in signal processing and is used in many applications, such as audio processing, image processing and many more. It is given by the following equation:

$$X_k = \frac{1}{N} \sum_{n=0}^{N-1} x_n e^{-i\frac{2\pi}{N}kn} \tag{4.1}$$

where x_n is the n-th sample of the signal, N is the number of samples and k is the frequency bin. The preceding factor of $\frac{1}{N}$ is a normalization factor, which is merely a convention and the DFT is not necessarily defined like this in all implementations (in Numpy this can be controlled by the `norm` argument of the `fft`-functions). The only difference between the discrete fourier transform and its counterpart (the inverse discrete fourier) is the sign of the exponent and the normalization factor, to be consistent the product of the normalization factors has to be $\frac{1}{\sqrt{N}}$. The basic DFT is a matrix multiplication of the signal vector x with a matrix W of size $N \times N$ and therefore has a time complexity of $\mathcal{O}(N^2)$ (as displayed in Equation 4.2).

$$X = \frac{Wx}{N} = \frac{1}{N} \begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & e^{-i\frac{2\pi}{N}} & e^{-i\frac{4\pi}{N}} & \dots & e^{-i\frac{2\pi(N-1)}{N}} \\ 1 & e^{-i\frac{4\pi}{N}} & e^{-i\frac{8\pi}{N}} & \dots & e^{-i\frac{4\pi(N-1)}{N}} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & e^{-i\frac{2\pi(N-1)}{N}} & e^{-i\frac{4\pi(N-1)}{N}} & \dots & e^{-i\frac{2\pi(N-1)^2}{N}} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_{N-1} \end{pmatrix} \tag{4.2}$$

The matrix W is called the DFT matrix and contains roots of unity. Those roots of unity are called "twiddle-factors" in the context of FFT-transforms, since they rotate complex numbers when multiplied with them. The entries of the individual rows can be thought of phasors, which are rotating counter-clockwise as one moves along the columns of the matrix. This angular velocity increases as one moves down the rows of the matrix (it is a Vandermonde matrix). For a 8 by 8 matrix matrix would be:

$$W = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & e^{-i\frac{2\pi 1}{8}} & e^{-i\frac{2\pi 2}{8}} & e^{-i\frac{2\pi 3}{8}} & e^{-i\frac{2\pi 4}{8}} & e^{-i\frac{2\pi 5}{8}} & e^{-i\frac{2\pi 6}{8}} & e^{-i\frac{2\pi 7}{8}} \\ 1 & e^{-i\frac{2\pi 2}{8}} & e^{-i\frac{2\pi 4}{8}} & e^{-i\frac{2\pi 6}{8}} & e^{-i\frac{2\pi 8}{8}} & e^{-i\frac{2\pi 10}{8}} & e^{-i\frac{2\pi 12}{8}} & e^{-i\frac{2\pi 14}{8}} \\ 1 & e^{-i\frac{2\pi 3}{8}} & e^{-i\frac{2\pi 6}{8}} & e^{-i\frac{2\pi 9}{8}} & e^{-i\frac{2\pi 12}{8}} & e^{-i\frac{2\pi 15}{8}} & e^{-i\frac{2\pi 18}{8}} & e^{-i\frac{2\pi 21}{8}} \\ 1 & e^{-i\frac{2\pi 4}{8}} & e^{-i\frac{2\pi 8}{8}} & e^{-i\frac{2\pi 12}{8}} & e^{-i\frac{2\pi 16}{8}} & e^{-i\frac{2\pi 20}{8}} & e^{-i\frac{2\pi 24}{8}} & e^{-i\frac{2\pi 28}{8}} \\ 1 & e^{-i\frac{2\pi 5}{8}} & e^{-i\frac{2\pi 10}{8}} & e^{-i\frac{2\pi 15}{8}} & e^{-i\frac{2\pi 20}{8}} & e^{-i\frac{2\pi 25}{8}} & e^{-i\frac{2\pi 30}{8}} & e^{-i\frac{2\pi 35}{8}} \\ 1 & e^{-i\frac{2\pi 6}{8}} & e^{-i\frac{2\pi 12}{8}} & e^{-i\frac{2\pi 18}{8}} & e^{-i\frac{2\pi 24}{8}} & e^{-i\frac{2\pi 30}{8}} & e^{-i\frac{2\pi 36}{8}} & e^{-i\frac{2\pi 42}{8}} \\ 1 & e^{-i\frac{2\pi 7}{8}} & e^{-i\frac{2\pi 14}{8}} & e^{-i\frac{2\pi 21}{8}} & e^{-i\frac{2\pi 28}{8}} & e^{-i\frac{2\pi 35}{8}} & e^{-i\frac{2\pi 42}{8}} & e^{-i\frac{2\pi 49}{8}} \end{pmatrix} \tag{4.3}$$

Knowing the property $e^{-i\frac{2\pi(p+Nq)}{N}} = e^{-i\frac{2\pi p}{N}} \cdot e^{-i2\pi q} = e^{-i\frac{2\pi p}{N}}, \quad q \in \mathbb{Z}$ and designating $e^{\frac{-i2\pi}{N}}$ as ω_N, we can further simplify the matrix and the original Equation 4.1 (the normalization factor was omitted).

$$W = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \omega_8 & \omega_8^2 & \omega_8^3 & \omega_8^4 & \omega_8^5 & \omega_8^6 & \omega_8^7 \\ 1 & \omega_8^2 & \omega_8^4 & \omega_8^6 & \omega_8^0 & \omega_8^2 & \omega_8^4 & \omega_8^6 \\ 1 & \omega_8^3 & \omega_8^6 & \omega_8 & \omega_8^4 & \omega_8^7 & \omega_8^2 & \omega_8^5 \\ 1 & \omega_8^4 & \omega_8^0 & \omega_8^4 & \omega_8^0 & \omega_8^4 & \omega_8^0 & \omega_8^4 \\ 1 & \omega_8^5 & \omega_8^2 & \omega_8^7 & \omega_8^4 & \omega_8 & \omega_8^6 & \omega_8^3 \\ 1 & \omega_8^6 & \omega_8^4 & \omega_8^2 & \omega_8^0 & \omega_8^6 & \omega_8^4 & \omega_8^2 \\ 1 & \omega_8^7 & \omega_8^6 & \omega_8^5 & \omega_8^4 & \omega_8^3 & \omega_8^2 & \omega_8 \end{pmatrix} \tag{4.4}$$

Expressing the matrix in this way makes it clear that this matrix has special properties. Those properties are the symmetry and the alternating ones (ω^0) and minus ones (ω^4) in the $N/2+1$ th-column. The second property only emerges when N is a power of two.

$$X_k = \sum_{n=0}^{N-1} x_n \cdot \omega^{kn} \tag{4.5}$$

The Fast Fourier Transform

Using the mentioned properties of the matrix one can derive one version of the fast fourier transform (FFT) as follows. This is equal to the decimation in time formulation originally proposed by Cooley and Tukey in their 1965 paper [79] and as outlined by Chu [51]. It is recognized as one of the 7 key algorithms in high performance computing [80].

4.1.1 Derivation of the Radix-2 Decimation in Time FFT

Equation 4.5 can be rewritten as follows.

$$X_k = \sum_{n=0}^{\frac{N}{2}-1} x_{2n} \cdot \omega_N^{2kn} + \sum_{n=0}^{\frac{N}{2}-1} x_{2n+1} \cdot \omega_N^{(2n+1)k} \tag{4.6}$$

The factor ω_N^k can be extracted from the second summand.

$$X_k = \sum_{n=0}^{\frac{N}{2}-1} x_{2n} \cdot \omega_N^{2kn} + \omega_N^k \sum_{n=0}^{\frac{N}{2}-1} x_{2n+1} \cdot \omega_N^{2kn} \tag{4.7}$$

Using the property $\omega_N^{2kn} = \omega_{\frac{N}{2}}^{kn}$ it can be recognized that the original problem was divided into two equal sub-problems half as large as the original problem. Adding the results of the two sub-problems (which lends itself very well to FMA instructions) yields the result of the original problem, this combination is known as the butterfly operation.

$$X_k = \sum_{n=0}^{\frac{N}{2}-1} x_{2n} \cdot \omega_{\frac{N}{2}}^{kn} + \omega_N^k \sum_{n=0}^{\frac{N}{2}-1} x_{2n+1} \cdot \omega_{\frac{N}{2}}^{kn} \tag{4.8}$$

The original problem of complexity $\mathcal{O}(N^2)$ was divided into two smaller problems of $\mathcal{O}(\frac{N}{2}^2)$ each. This process can be repeated recursively until the sub-problems have a size of two. In this process one gets $\log_2(N)$ levels of recursion with N number of operations required per level, thus resulting in the overall complexity of $\mathcal{O}(N \log(N))$. All variations of the FFT are based on this principle of divide and conquer, their specific performance characteristics depend on implementation details (as investigated in Section 2.7).

4.2 The cuFFT Library

CuFFT [81] is Nvidia's proprietary offering for GPU accelerated FFT-transforms. It offers a C API and operates based on so-called plans, which are objects describing the parameters of the transform. The basic workflow consists of creating a plan, executing the plan and finally destroying the plan. Its API is closely modeled after FFTW3 [52]. Unlike FFTW3 however, cuFFT does not make it clear how it chooses a specific implementation for a given transform over another. It seems plausible that Nvidia conducts a wide range of tests on their hardware and ships a wisdom-file with the library, which contains the results of those tests. FFTW3 offers the functionality to build such databases of optimal plans for various transforms by running a large set of tests. The downside of this approach is that the wisdom-file is only valid for the hardware it was generated on (mostly depending on the CPU-memory pairing), so it is not possible to ship a single wisdom-file with the library. Nvidia has the advantage of being a vendor of a limited variation of self-contained devices (the GPU's computing capabilities as well as the paired amount and speed of memory is known), so they could determine that information beforehand. This means that there is little to no room for optimization on the user's side, since the library will always choose the same implementation for a given transform and users have no influence on that choice.
Libraries relying on cuFFT can only differentiate between them by making good use of the different API calls provided. Possible design choices which could make a library more efficient than another one are keeping a cache of recently used plans and reusing them, or using batched transforms to reduce the overhead of creating and destroying plans and using the correct transforms (i.e. using a Real to Complex transform instead of automatically converting the input to a Complex array and then performing a more general Complex to Complex transform).
CuFFT offers the following transform types:

- `C2C`: Complex to Complex
- `R2C`: Real to Complex
- `C2R`: Complex to Real
- `D2Z`: Double to Double Complex
- `Z2D`: Double Complex to Double

Besides the basic library Nvidia also offers a library called cuFFTXt (Xt stands for eXtended), with additional features including the functionality to distribute the workload over multiple GPUs and the ability to perform transforms on half-precision floating point numbers and a library called cuFFTDx. CuFFTXt offers additional features including the functionality to distribute the workload over multiple GPUs and the ability to perform transforms on half-precision floating point numbers. cuFFTDx is a library for integrating FFTs into user-written CUDA-kernels. This is useful to make individual kernels do more work, instead of having multiple kernels, with a cuFFT kernel in between. This is related to the idea of kernel fusion as discussed in Section 2.7.2.7. The size of transforms which can be performed with cuFFTDx however is rather limited (up to $2^{15} = 32768$ depending on the used data-types and CUDA capability of the GPU).

4.3 Comprehensive Benchmarks for various Implementations Available in Python

Benchmarking tools such as Gearshifft [82] play an important role in picking the right FFT implementation for a given problem. Additionally benchmarking is not as trivial as it might seem at first glance, since the performance of a given implementation is dependent on many factors, such as the size of the transform (powers of two, or a numbers with low prime factorizations are faster to transform than arbitrary sizes), the kind of hardware (CPU, GPU, FPGA, ASIC) used and the memory bandwidth connecting that hardware to the main memory. Measuring performance becomes even more complicated when dealing with heterogeneous systems, since the function calls often happen asynchronously and the time it takes to execute a given function is not necessarily the time it takes to complete the operation. Often the

functions are merely scheduled on the device and the executing program (in our case Python scripts) have to be explicitly instructed to wait for the operation to complete (synchronization). This connects with the concept of streams, as discussed in Section 2.5. Operations can be enqued on different streams and the order of execution is only ever guaranteed for operations on the same stream.
The tests conducted in this section were performed with a convenient set of scripts with the aim to test the various implementations of the available FFT (or other DFT) algorithms on performance (run-time and device memory usage). The aim is to provide users with vital information on which implementation works best on a given machine for a given workload. Having convenient access to such information is crucial when dealing with today's high heterogeneity of computing systems ranging from small, embeddable systems (like the Raspberry Pi series or CUDA-capable alternatives like Nvidia's Jetson series) over traditional desktop workstation to large distributed clusters, since the performance of the individual libraries dependent on many factors and predicting performance is non-trivial.
Unless otherwise stated each test was performed 50 times and the median time of that 50 tests reported, this median time is usually very close to the minimum time and significantly smaller than the maximum due to an increased cost of the first invocation caused by JIT-, or general driver overhead.

4.3.1 Libraries and Features tested

The benchmarks should evaluate both CPU- and GPU-accelerated implementations of (multi-dimensional, forward) fourier transforms for a variety of power of 2 and odd sizes with different levels of precision (single and double).
CPU libraries:

- Numpy's FFT libraries
- FFTW3 bindings
- Intel MKL

GPU accelerated (note: it is expected that some of these libraries rely on the cuFFT-implementation):

- CuPy
- PyVkFFT (using the OpenCL backend)[83], [84]
- PyTorch
- Jax

This analysis with many independent factors (library, direction, size/shape, kind of transform, precision) will quickly lead to many hundreds of possible configurations, which need to be efficiently testable and comparable.
Other noteworthy libraries include for example cuSignal [85] (a GPU-aware adaption of SciPy Signal); while very useful for signal processing, it was not tested because it is relying on CuPy (which in turn uses cuFFT) for the FFTs, which is already tested in this benchmark suite.

4.3.1.1 Architecture

This tool utilizes Nvidia Nsight Systems as the underlying profiler, since it is one of the few profilers which are able to track CPU and GPU usage as well as the usage of GPU-dedicated memory on the process level (the inability to profile system memory usage with this tooling is unfortunate, but acceptable since we are mostly interested in GPU resources and host side RAM is a comparatively abundant and cheap resource, albeit usually an order of magnitude slower). Other profilers like Scalene [86] (a Python specific profiler that allows users to track CPU, GPU and memory metrics) for example use Nvidia management Library (NVML) to periodically retrieve GPU metrics. We chose to use Nsight Systems because it can store all recorded information during a profiling session in an SQL-Lite database. The information stored in this database can then be programmatically queried, filtered and used for comprehensive reports and

analysis using well established tools from Python's ecosystem (pandas and matplotlib). Ultimately this section reports on the results of the execution of the defined algorithms with a selected set of libraries.

4.4 Results on local machine

First we are going to present the results of the benchmarking tool on a local machine with a Nvidia RTX 3070 GPU (GA104 with 8 GB of dedicated memory) and an AMD Ryzen 5 3600 CPU. Despite being listed in Section 4.3.1, the PyVkFFT library with the OpenCL backend was not tested on this machine, since it proved to be unstable and regularly crashed the system.

4.4.1 1D Transforms

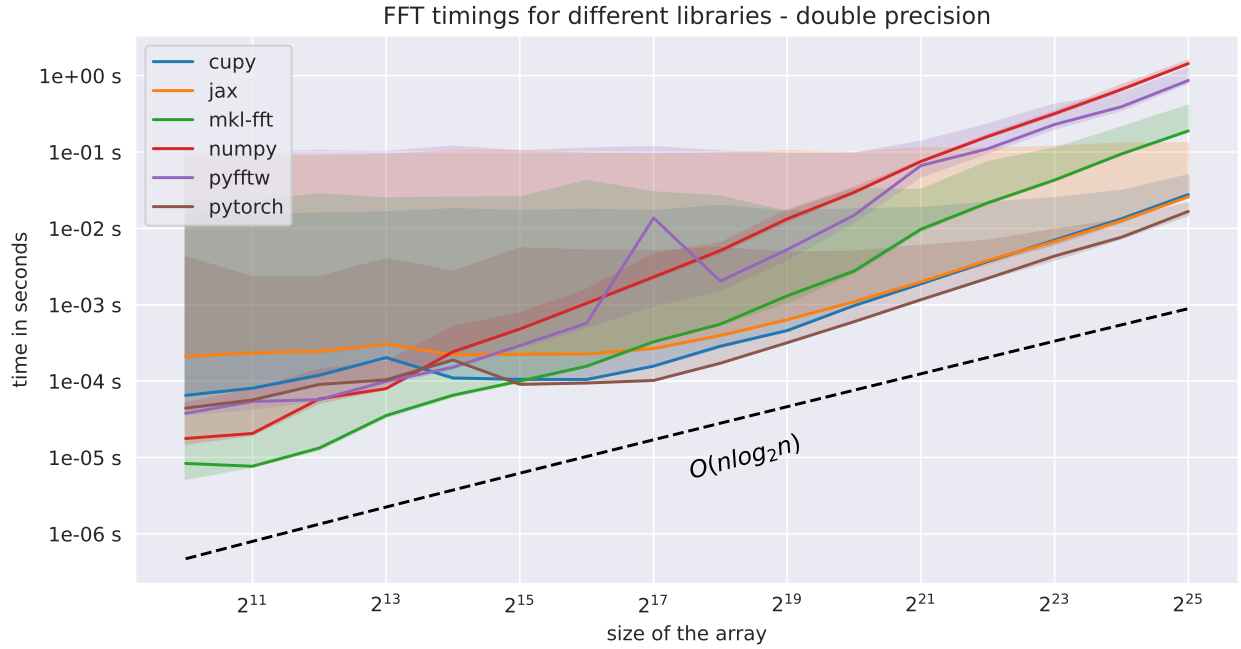


Figure 4.1: 1D FFT benchmarks for double precision, only execution time is shown (no memory transfer time), only power of 2 sizes were tested, performance between two datapoints is *not* linearly interpolatable

Figure 4.1 shows the performance of the different libraries for double precision. The shaded area indicates the complete range of the measured time taken for the execution of a single FFT. This range is especially wide for the GPU-accelerated libraries (CuPy, PyTorch and Jax), which is due to the cost of JIT-compilation of the kernels, which is necessary on the first run of a kernel. The line (located in the lower parts of the shaded area) indicates the median of the measured times.
The performances of the tested libraries can be roughly divided into two groups: the CPU-accelerated libraries (Numpy, pyfftw and mkl) and the GPU-accelerated libraries (CuPy, PyTorch and jax).
The GPU libraries are generally faster than the CPU libraries (at least for FFTs of sizes $2^{14} = 16384$ and larger) and show little differences in their relative performance with only PyTorch deviating from the trend by being slightly faster than CuPy and Jax. This small difference is likely due to the fact that PyTorch is using the same cuFFT implementation as CuPy and Jax, this suspicion can be confirmed by looking at the kernels called by the libraries. The kernels executed by the three libraries to perform a single 1D-transformation of large size (2^{25} - the largest datapoint recorded in Figure 4.2) are shown in the Tables 4.1 through 4.3.

It can be observed that all three libraries are using multiple (in this case three) invocations of a kernel called `regular_fft` to calculate the transformation, but they do so in different ways. CuPy and JAX both first convert the real-valued inputs to a complex-valued inputs using kernels called `cupy_copy__float64_complex128` and `convert_2` respectively followed by identical invocations of the `regular_fft` kernel.
PyTorch on the other hand starts directly with the `regular_fft` kernels and uses fewer blocks (although blocks of the same size) than the other two libraries. This directly translates into a lower number of threads launched and more work being done by each thread. This is likely the reason why PyTorch is slightly faster than the other two libraries.

Kernel Name	grid dimension	block dimension	thread count	occupancy
`cupy_copy__float64_complex128`	262144 x 1 x 1	128 x 1 x 1	33,554,432	100.00%
`regular_fft`	16384 x 1 x 1	8 x 16 x 1	2,097,152	25.00%
`regular_fft`	16384 x 1 x 1	8 x 16 x 1	2,097,152	25.00%
`regular_fft`	8192 x 1 x 1	8 x 64 x 1	4,194,304	66.67%

Table 4.1: Kernels called by the CuPy library for a double precision FFT of size $2^{25} = 33554432$

Kernel Name	grid dimension	block dimension	thread count	occupancy
`convert_2`	552 x 1 x 1	128 x 1 x 1	70,656	100.00%
`regular_fft`	16384 x 1 x 1	8 x 16 x 1	2,097,152	25.00%
`regular_fft`	16384 x 1 x 1	8 x 16 x 1	2,097,152	25.00%
`regular_fft`	8192 x 1 x 1	8 x 64 x 1	4,194,304	66.67%

Table 4.2: Kernels called by the Jax library for a double precision FFT of size $2^{25} = 33554432$

Kernel Name	grid dimension	block dimension	thread count	occupancy
`regular_fft`	8192 x 1 x 1	8 x 16 x 1	1,048,576	25.00%
`regular_fft`	8192 x 1 x 1	8 x 16 x 1	1,048,576	25.00%
`regular_fft`	8192 x 1 x 1	8 x 16 x 1	1,048,576	25.00%
`postprocessC2C_kernelMem`	32768 x 1 x 1	256 x 1 x 1	8,388,608	100.00%
`_fft_conjugate_copy_kernel`	16384 x 1 x 1	1024 x 1 x 1	16,777,216	66.67%

Table 4.3: Kernels called by the PyTorch library for a double precision FFT of size $2^{25} = 33554432$

The CPU libraries display a wider range of performance, which is due to the difference in backends used by the libraries. In this configuration the mkl-fft library is the fastest, followed by pyfftw and Numpy. Their respective backends are Intel's MKL, FFTW (linked against OpenBLAS) and Numpy's own adaptation of FFTPACK (called PocketFFT) [87].
The dashed line indicates the theoretical relationship of the FFT-algorithm and the size of the input data ($\mathcal{O}(n \log_2 n)$), the exact position of this line on the plot is arbitrary; important is how its slope relates to the slopes of the other lines, which stem from the measurements.

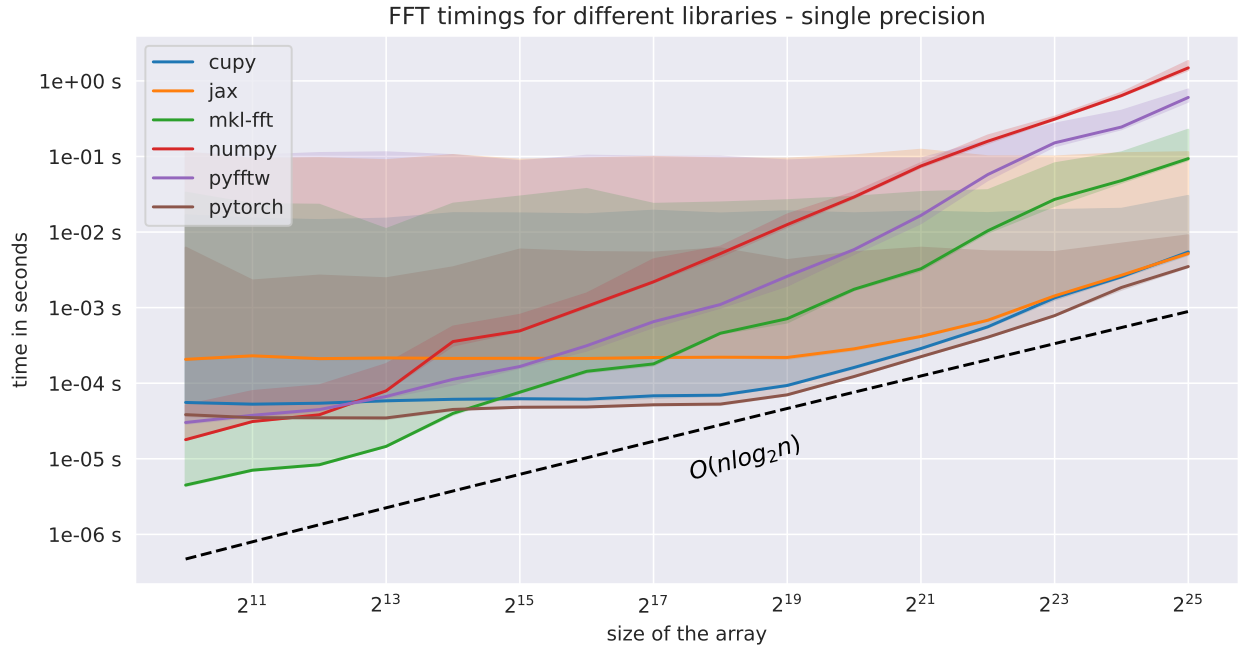


Figure 4.2: 1D FFT benchmarks for single precision

Figure 4.2 shows the performance of the different libraries for single precision. The differences in performance between single and double precision are especially pronounced for the GPU-accelerated libraries, while they are relatively small for the CPU libraries pyfftw and mkl-fft. Figure 4.3 offers a detailed view into the performance differences between the different levels of precision for two selected libraries (PyTorch executing on the GPU and PyFFTW on the CPU). Note that Numpy does not support single precision FFTs, it always returns a double precision array, even if the input array is single precision. Numpy is only included in Figure 4.2 for completeness.
Tables 4.4 and 4.5 shows the median execution times for the different libraries for all tested sizes and single and double precision. The table also shows the relative performance of the libraries compared to Numpy as a baseline.
Table 4.4 shows the median execution times for the CPU based libraries. It is of note that the alternatives libraries (pyfftw and mkl-fft) tend to a larger speedup the larger the input size, but only up to a certain point. MKL-FFT reaches a maximum speedup of 21.9 for a single precision FFT of size 2^{21} and a maximum speedup of 9.7 for a double precision FFT of size 2^{20}. The larger speedup for single precision problems is expected, since Numpy is always using double precision for its FFTs, so the baseline is slightly skewed in those cases. One can, however, observe that the difference in runtimes for single- and double precision problems is relatively small for CPU based libraries. The difference between single and double precision when using the mkl-fft library for example is roughly a factor of 2, and a factor of 1.5 for pyfftw.
Table 4.5 shows the median execution times for the GPU based libraries. The baseline for the speedups is once again Numpy, calculating double precision FFTs on the CPU. The speedups for the GPU based libraries are much larger than for the CPU based libraries, even for double precision problems even though modern GPUs prioritize single precision operations. The speedups get strictly larger for larger input sizes and reach values of up to 424.5 for a single precision FFT of size 2^{25} when using PyTorch and 85 for a double precision FFT of size 2^{25}. Note that this table only contains the executions times necessary for the calculation of the FFTs on the GPU, not the time needed to transfer the data to and from the GPU, which can be a bottleneck (as explained in the section discussing the roofline model 2.6).

	mkl-fft				pyfftw				numpy
	single		double		single		double		double
	median	speedup	median	s.up	median	s.up	median	s.up	median
2^{10}	4.5μs	3.0	8.4μs	1.1	30.2μs	-0.4	37.9μs	-0.5	17.7μs
2^{11}	7.1μs	3.4	7.7μs	1.7	37.7μs	-0.2	54.1μs	-0.6	20.6μs
2^{12}	8.3μs	3.6	13.3μs	3.4	44.8μs	-0.1	57.2μs	0.0	58.5μs
2^{13}	14.6μs	4.4	35.4μs	1.3	66.8μs	0.2	99.5μs	-0.2	79.9μs
2^{14}	39.9μs	7.9	65.4μs	2.7	112.7μs	2.2	151.9μs	0.6	242.7μs
2^{15}	76.1μs	5.5	100.2μs	3.8	166.4μs	2.0	292.5μs	0.6	481.1μs
2^{16}	143.8μs	6.2	157.0μs	5.7	313.4μs	2.3	572.6μs	0.8	1.0ms
2^{17}	180.2μs	11.2	327.3μs	6.1	652.4μs	2.4	13.7ms	-0.8	2.3ms
2^{18}	458.4μs	10.3	557.1μs	8.2	1.1ms	3.7	2.0ms	1.5	5.1ms
2^{19}	714.1μs	16.6	1.3ms	9.2	2.6ms	3.8	5.3ms	1.5	13.3ms
2^{20}	1.8ms	15.6	2.8ms	9.7	5.9ms	4.0	14.9ms	1.0	29.6ms
2^{21}	3.3ms	21.9	9.8ms	6.7	16.6ms	3.5	66.5ms	0.1	75.2ms
2^{22}	10.4ms	14.3	21.6ms	6.4	57.7ms	1.8	110.2ms	0.4	159.4ms
2^{23}	27.1ms	10.5	43.0ms	6.4	151.5ms	1.1	229.3ms	0.4	318.2ms
2^{24}	48.0ms	12.3	94.3ms	6.0	245.5ms	1.6	391.4ms	0.7	660.3ms
2^{25}	93.8ms	14.9	188.9ms	6.6	604.8ms	1.5	864.5ms	0.7	1.44s

Table 4.4: Median execution times and relative speedup compared to the Numpy baseline for the CPU based libraries for all tested sizes and single and double precision

	pytorch				cupy				jax			
	single		double		single		double		single		double	
	median	speedup	median	s.up	median	s.up	median	s.up	median	s.up	median	s.up
2^{10}	38.4μs	-0.5	44.3μs	-0.6	55.7μs	-0.7	65.2μs	-0.7	207.0μs	-0.9	210.5μs	-0.9
2^{11}	35.0μs	-0.1	56.7μs	-0.6	52.8μs	-0.4	80.9μs	-0.7	230.5μs	-0.9	234.0μs	-0.9
2^{12}	34.9μs	0.1	90.5μs	-0.4	54.4μs	-0.3	119.7μs	-0.5	211.4μs	-0.8	246.0μs	-0.8
2^{13}	34.7μs	1.3	103.9μs	-0.2	58.5μs	0.4	203.0μs	-0.6	216.3μs	-0.6	304.9μs	-0.7
2^{14}	45.0μs	6.9	189.1μs	0.3	61.3μs	4.8	110.0μs	1.2	213.1μs	0.7	220.8μs	0.1
2^{15}	48.1μs	9.2	90.6μs	4.3	62.4μs	6.9	105.5μs	3.6	214.0μs	1.3	224.5μs	1.1
2^{16}	48.5μs	20.4	94.4μs	10.1	61.5μs	15.9	105.1μs	8.9	212.5μs	3.9	226.0μs	3.6
2^{17}	51.9μs	41.4	102.0μs	21.6	68.2μs	31.2	157.1μs	13.7	218.9μs	9.0	267.2μs	7.6
2^{18}	53.0μs	97.0	171.6μs	28.9	69.5μs	73.7	287.0μs	16.9	220.9μs	22.5	395.7μs	12.0
2^{19}	70.2μs	177.8	318.5μs	40.8	93.4μs	133.5	459.4μs	28.0	219.4μs	56.2	636.1μs	19.9
2^{20}	122.7μs	236.4	600.2μs	48.4	161.6μs	179.2	973.6μs	29.4	285.1μs	101.1	1.1ms	26.1
2^{21}	224.5μs	332.2	1.2ms	63.6	287.9μs	258.9	1.9ms	39.1	415.8μs	179.0	2.0ms	36.8
2^{22}	406.4μs	390.7	2.2ms	70.8	557.7μs	284.4	3.7ms	42.3	681.2μs	232.7	3.8ms	40.9
2^{23}	783.7μs	397.0	4.3ms	72.2	1.3ms	230.4	7.0ms	44.3	1.4ms	217.0	6.7ms	46.1
2^{24}	1.8ms	344.5	7.7ms	85.2	2.6ms	248.6	13.4ms	48.3	2.7ms	237.0	12.6ms	51.5
2^{25}	3.5ms	424.1	16.7ms	85.2	5.5ms	271.7	27.7ms	50.8	5.2ms	284.7	25.9ms	54.5

Table 4.5: Median execution times and relative speedup compared to the Numpy baseline for the CPU based libraries for all tested sizes and single and double precision

4.4.1.1 Comparison of performance when using different levels of precision

Figure 4.3 shows the performance of two different libraries (PyTorch on the GPU and pyfftw on the CPU) over a range of different levels of precision. For PyTorch half (`torch.float16`), single and double precision was tested, while for pyfftw quad-precision was also tested. Note that the result of the half-precision transform has to be considered with caution, since it's very likely overflow the first entry (the sum of all entry elements, unless the transform is divided through the number of inputs) due to the limited range of half-precision floats [88]. PyFFTW quad-precision output will be the reference for evaluating the error of the other libraries.

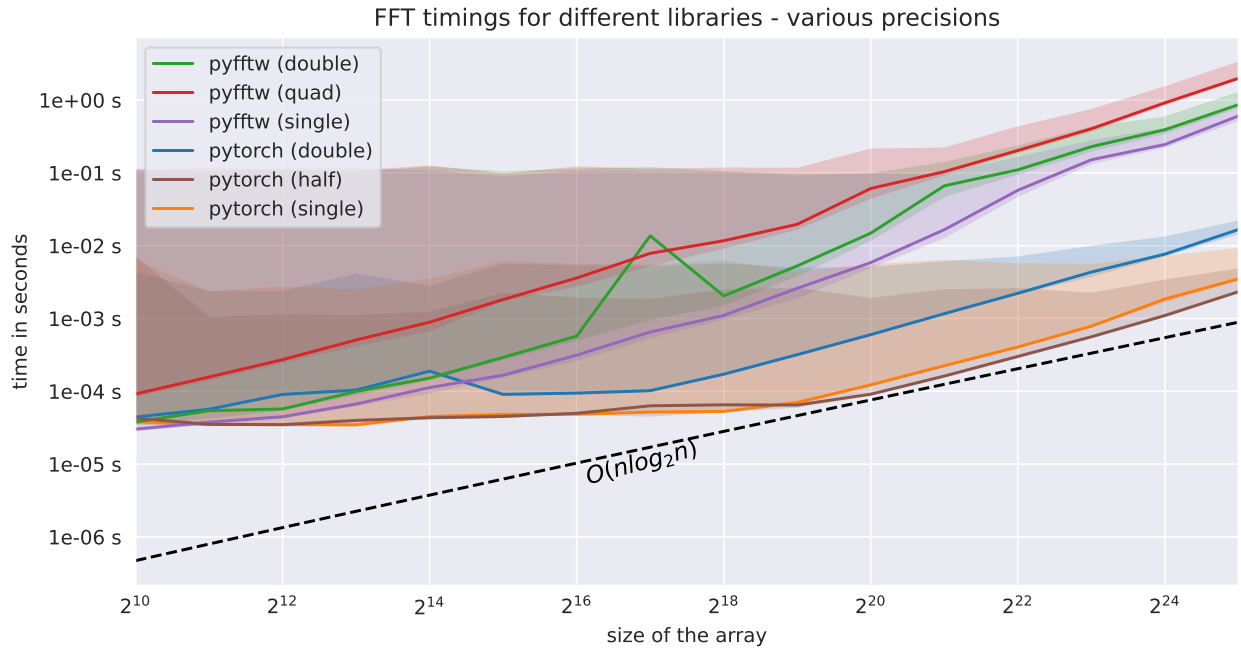


Figure 4.3: Comparison of performance when using different levels of precision

4.4.1.2 Memory usage of the GPU based libraries

The usage of GPU Memory is an important factor when evaluating GPU based libraries, since the amount of memory available on a GPU is often limited. Figure 4.4 shows the memory usage of the GPU based libraries for a single precision FFT of size 2^{25}.
The high relative memory usage (10 to 100 times the size of the input) for small problem sizes can be safely ignored, since that usage can be attributed to the usage of the CUDA runtime library. This memory usage is called device memory footprint and varies from CUDA version to CUDA version, this memory usage only becomes critical in the case of many separate processes using CUDA on the same GPU.
Of the GPU based libraries, JAX and CuPy have a significantly higher memory usage than the other library (PyTorch) for the same problem size. The exact numbers are provided in Table 4.6. It is clear that the relative memory usage cannot be below 2, since space has to be allocated for the input and output arrays. The relative memory usage of the GPU based libraries converges towards integer values as the problem size increases, which is expected since the memory usage is proportional to the size of the input array.
PyTorch uses 4 times the size of the input array for both single- and double precision problems. This memory is most likely distributed in the following way: the input array, the output array and double the

size of the input array for an intermediate array of the same size, but containing complex numbers and thus twice the size in memory of the input array.
CuPy and JAX use 9 or 8 times the size of the input array. This is most likely due to those libraries instantiaiting extra copies of the input at some point or, as seen in the analysis of the kernels executed (Tables 4.1-4.2), the creation of not necessarily needed complex-valued copies of the input array.
So even though the three tested libraries all share the same backend (cuFFT), the memory usage can vary significantly due to design choices made by the library developers.

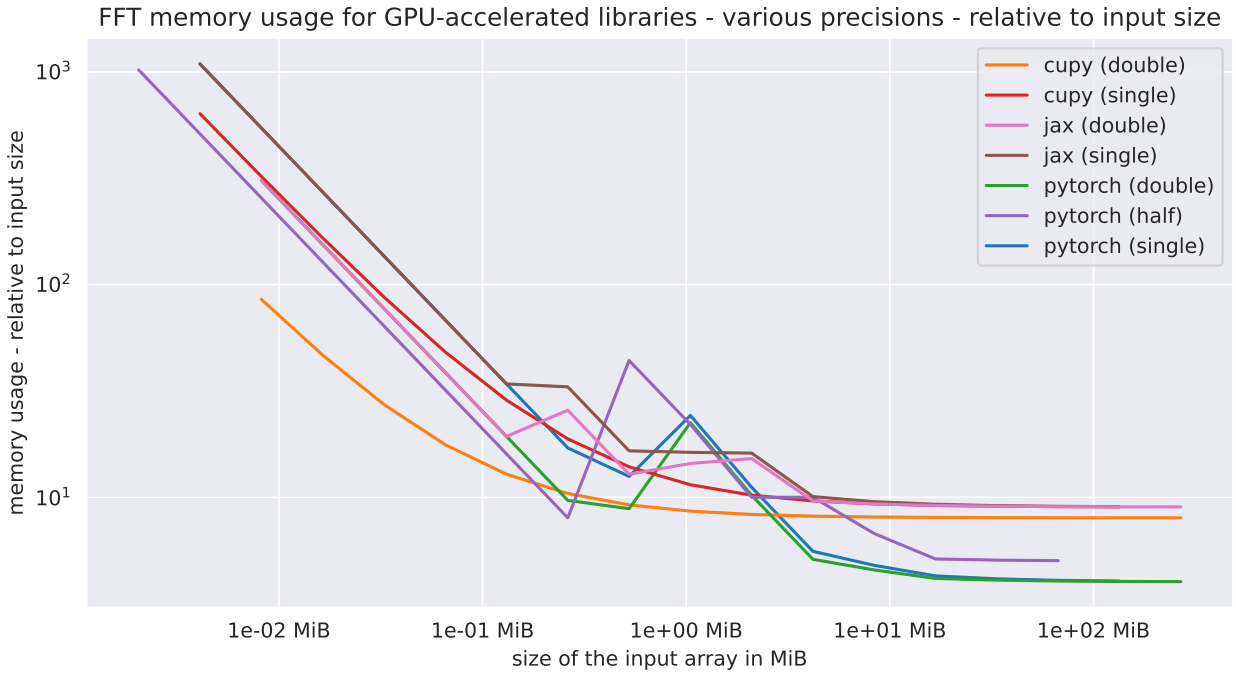


Figure 4.4: Relative Memory usage to size of input array

	pytorch				cupy				jax			
	single		double		single		double		single		double	
	mem. used	relative to input	mem. used	r.t.i.	mem. used	r.t.i.	mem. used	r.t.i.	mem. used	r.t.i.	mem. used	r.t.i.
2^{10}	4.3 MiB	1092.2	2.4 MiB	309.3	2.5 MiB	637.2	682.8 KiB	85.3	4.3 MiB	1092.2	2.4 MiB	309.3
2^{11}	4.3 MiB	546.1	2.4 MiB	154.7	2.5 MiB	323.1	746.8 KiB	46.7	4.3 MiB	546.1	2.4 MiB	154.7
2^{12}	4.3 MiB	273.1	2.4 MiB	77.3	2.6 MiB	166.1	874.8 KiB	27.3	4.3 MiB	273.1	2.4 MiB	77.3
2^{13}	4.3 MiB	136.5	2.4 MiB	38.7	2.7 MiB	87.5	1.1 MiB	17.7	4.3 MiB	136.5	2.4 MiB	38.7
2^{14}	4.3 MiB	68.3	2.4 MiB	19.3	3.0 MiB	48.3	1.6 MiB	12.8	4.3 MiB	68.3	2.4 MiB	19.3
2^{15}	4.3 MiB	34.1	2.4 MiB	9.7	3.6 MiB	28.6	2.6 MiB	10.4	4.3 MiB	34.1	6.4 MiB	25.7
2^{16}	4.3 MiB	17.1	4.4 MiB	8.8	4.7 MiB	18.8	4.6 MiB	9.2	8.3 MiB	33.1	6.4 MiB	12.8
2^{17}	6.3 MiB	12.5	22.4 MiB	22.4	7.0 MiB	13.9	8.6 MiB	8.6	8.3 MiB	16.5	14.4 MiB	14.4
2^{18}	24.3 MiB	24.3	20.4 MiB	10.2	11.5 MiB	11.5	16.6 MiB	8.3	16.3 MiB	16.3	30.4 MiB	15.2
2^{19}	22.3 MiB	11.1	20.4 MiB	5.1	20.5 MiB	10.2	32.6 MiB	8.2	32.3 MiB	16.1	38.4 MiB	9.6
2^{20}	22.3 MiB	5.6	36.4 MiB	4.6	38.5 MiB	9.6	64.6 MiB	8.1	40.3 MiB	10.1	74.4 MiB	9.3
2^{21}	38.3 MiB	4.8	66.4 MiB	4.2	74.5 MiB	9.3	128.6 MiB	8.0	76.3 MiB	9.5	146.4 MiB	9.2
2^{22}	68.3 MiB	4.3	130.4 MiB	4.1	146.5 MiB	9.2	256.6 MiB	8.0	148.3 MiB	9.3	290.4 MiB	9.1
2^{23}	132.3 MiB	4.1	258.4 MiB	4.0	290.5 MiB	9.1	512.6 MiB	8.0	292.3 MiB	9.1	578.4 MiB	9.0
2^{24}	260.3 MiB	4.1	514.4 MiB	4.0	578.5 MiB	9.0	1.0 GiB	8.0	580.3 MiB	9.1	1.1 GiB	9.0
2^{25}	516.3 MiB	4.0	1.0 GiB	4.0	1.1 GiB	9.0	2.0 GiB	8.0	1.1 GiB	9.0	2.3 GiB	9.0

Table 4.6: Memory usage of the GPU based libraries for various problem sizes, both in absolute and terms relative to the size of the input array

4.4.2 Batched 1D Transforms

As mentioned in Section 2.7.1.4 the performance of GPU algorithms are often greatly improved when problems fit into partitions of the memory, which share memory for fast synchronization (e.g. a single block of a CUDA kernel). To still fully utilize the GPU, the GPU can perform multiple FFTs in parallel, this step is natural when calculating FFTs of higher-dimensions, but can also be applied to 1D FFTs.

4.4.2.1 Constant transform size

Figure 4.5 shows the performance of batched 1D transforms with a constant transform size of 1024. The dashed line represents the linear continuation of the performance of the single transform of size 1024. The measured performance is better than one would expect from that linear continuation.
Especially remarkable are the plateaus in the performance (especially pronounced for PyTorch) which stem from the fact that all of the GPU's multiprocessors can be utilized at the same time. These plateaus are not as pronounced for the other libraries, since their overhead is higher and masks these effects. This means that it takes virtually the same amount of time to calculate 50 transforms as it does to calculate a single transformation, as long as all of the blocks of the kernel can be scheduled to be executed at the same time. As discussed in Section 2.1 the used GPU has 46 multiprocessors, which means that at least 46 blocks can be scheduled for simultaneous execution. In the example at hand, it seems that a jump occurs at every increase of the batch of size by 92 for the double-precision transform, which is the first time that not all blocks can be scheduled for execution at the same time. This can be confirmed by examining the kernels executed for the batch sizes of 92 and 93 (listed in Tables 4.7 and 4.8). For both settings the executed kernels are the same, but the number of blocks is different. When batching 93 the number of blocks scheduled for execution is 47, which is one more than the number of multiprocessors and the execution time doubles. This implies that performance can vary greatly even on slightly different GPU models. The next larger GPU model (RTX 3070 Ti), for example, has 48 multiprocessors, which means the jumps in performance would occur at slightly larger batch sizes and the performance would be twice as good as on the non Ti model for certain edge cases.
Regardless of the plateaus in execution time needed, the performance of the batched transforms is significantly better than the performance of the single transforms, since this are the situations GPUs excel in - calculating multiple relatively small tasks (which do not require communication across individual multiprocessors) of the same kind, which are independent of each other, in parallel. One can see that the time needed to calculate the transformations does not increase as fast as the batch size increases (it is a linear relationship, but its slope depends on the number of multiprocessors). Calculating 800 transforms in parallel takes about 1.5 times as long as calculating a single transform.

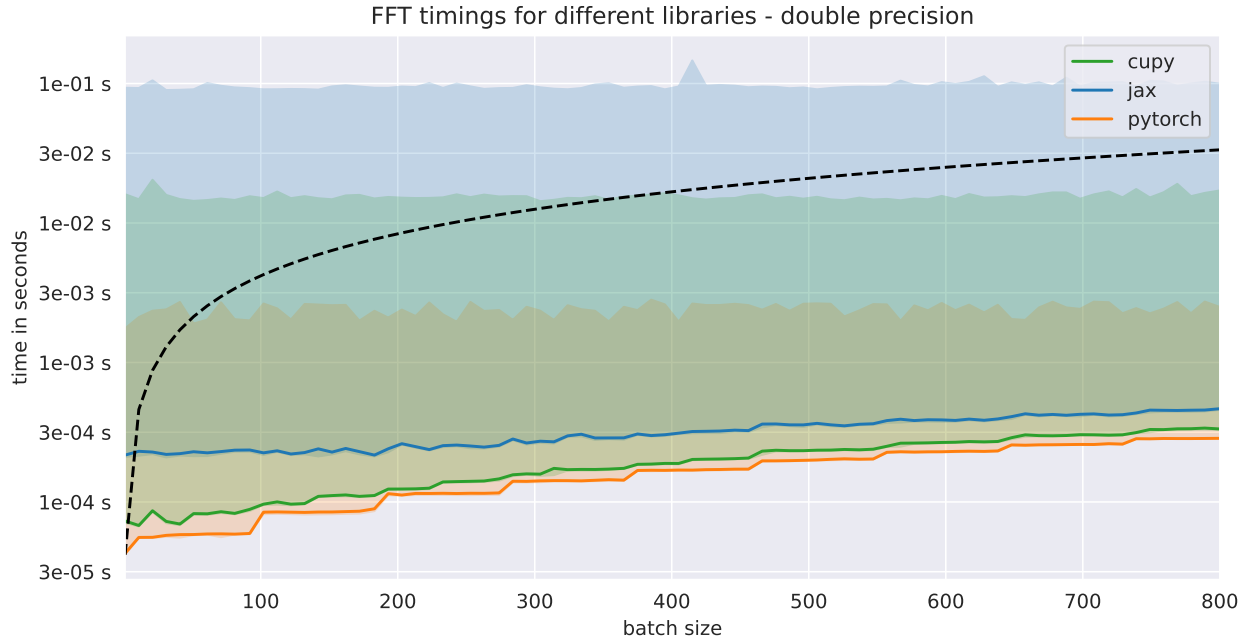


Figure 4.5: Performance of batched 1D transforms with a constant transform size of 1024 for double precision on a GPU (the steps are apparently not of equal size due to logarithmic scaling of the y-axis)

Kernel Name	grid dimension	block dimension	thread count	occupancy	execution time
`regular_fft_r2c`	46 x 1 x 1	64 x 2 x 1	5,888	41.67%	24.7 μs
`_fft_conjugate_copy_kernel`	46 x 1 x 1	1024 x 1 x 1	47,104	66.67%	4.0 μs

Table 4.7: Kernels executed for 92 batched transforms of size 1024, double precision PyTorch on the GPU

Kernel Name	grid dimension	block dimension	thread count	occupancy	execution time
`regular_fft_r2c`	47 x 1 x 1	64 x 2 x 1	6,016	41.67%	44.3 μs
`_fft_conjugate_copy_kernel`	47 x 1 x 1	1024 x 1 x 1	48,128	66.67%	5.0 μs

Table 4.8: Kernels executed for 93 batched transforms of size 1024, double precision PyTorch on the GPU

The batch size has less effect on the performance of the single precision transforms, since these transforms are an order of magnitude faster than the process of launching a kernel (at least for JAX and to a slightly lesser extent for CuPy; in the case of JAX all batch sizes take roughly the same amount of time). PyTorch again is the fastest library, but the plateaus are not as pronounced as in the case of the double precision transforms. This could be due to the fact that the GPU is not dedicated to those computations, but is also used for other tasks (like rendering the UI to the displays), which also put some load on the FP32 units of the GPU, whereas the FP64 units are not used at all by unless explicitly requested by GPGPU applications.

The benefit of using batched transforms is much less pronounced for the CPU based libraries, since CPUs do not have the capabilities for such massively parallel computations. Of course there is some benefit because a batched execution lends itself better to vectorization and parallelization than a single transform of such a small size (1024) and batching transformations is highly encouraged even when using CPU based libraries. Figure 4.7 shows the performance of the CPU based libraries for the same

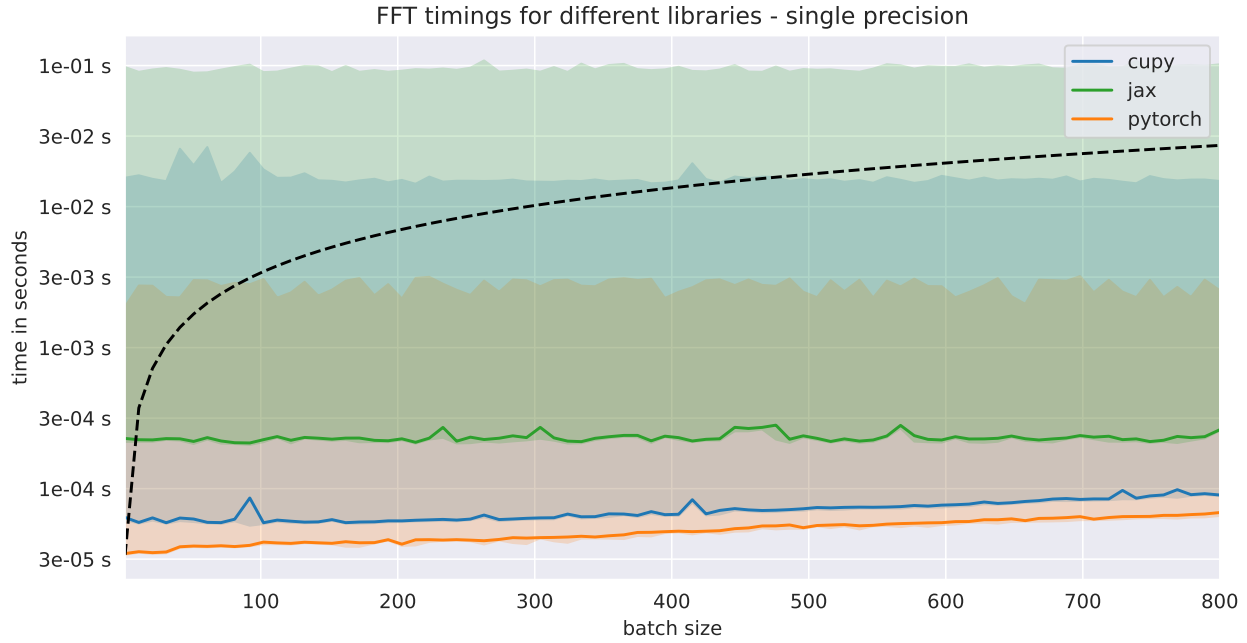


Figure 4.6: Performance of batched 1D transforms with a constant transform size of 1024 for single precision on a GPU

problem size as the GPU based libraries and double precision. The respective graphic for single precision since the differences between single- and double-precision are much less pronounced for the CPU based libraries. The absence of any plateaus in the performance of the CPU based libraries is expected but still noteworthy. Numpy's and MKL-FFTW's performances follow the linear continuation of the performance of the single transform of size 128 very closely, while pyFFTW's performance is a lot noisier. It can be as fast as MKL-FFTW, but in the median case it tends to be slower. In fact the median performance of pyFFTW is almost constant over the tested batch-sizes, which is a sign of a lot of overhead or bad multi-threading slowed down by other processes running on the same machine.

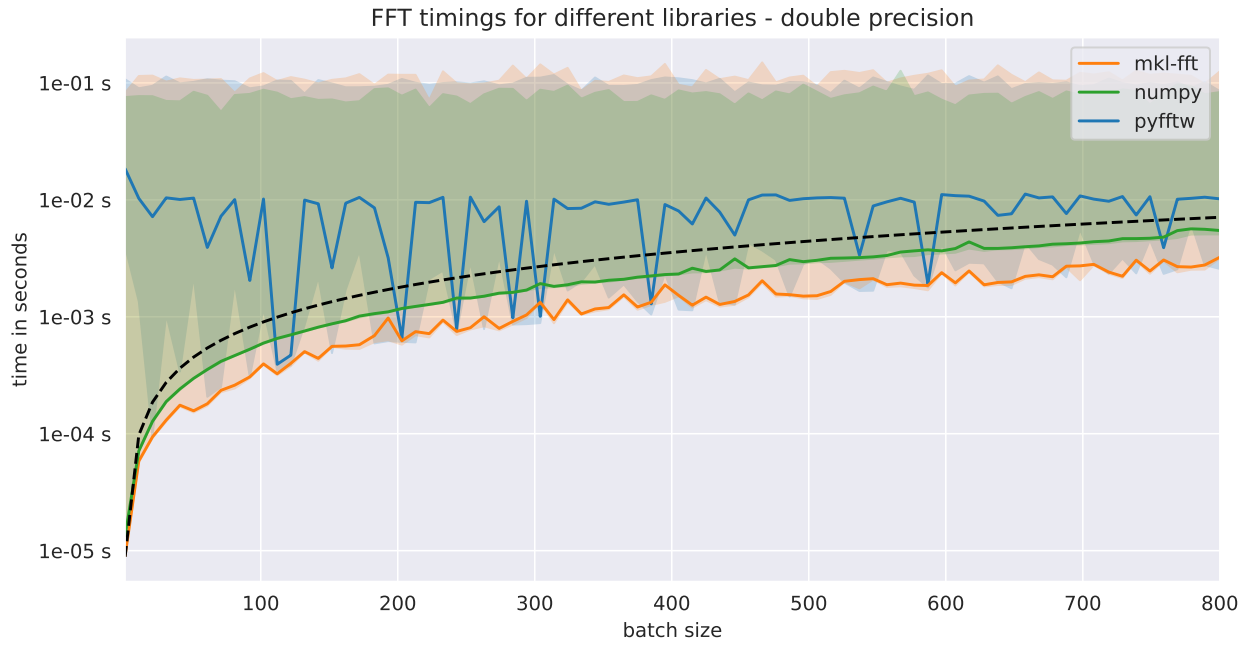


Figure 4.7: Performance of batched 1D transforms with a constant transform size of 1024 for double precision on a CPU

4.4.2.2 Constant batch size

A similar phenomenon to the plateaus observed in the previous section is expected when the batch size is held constant and the transform size is increased. In this case the jump would occur whenever a fewer number of blocks could be fitted onto a multiprocessor. This could happen due to a number of factors, namely the number of threads per block, the shared memory usage per block and the number of registers used per block. Measurements however show that this effect is not as visible as one might expect, because the effect of the varying occupancy is overshadowed by the fact that the performance of the FFT algorithm is highly dependent on the transform size. In the previous tests only power of two transform sizes were used, which are the most efficient transform sizes for the FFT algorithm. However, when testing with non-power of two transform sizes, entirely different algorithms and kernels are used such as the Bluestein algorithm, which allows for arbitrary transform sizes [89]. Those kernels also have different resource requirements, which leads to many plateaus as the transform size is increased.
Figure 4.8 shows the performance of 1D FFTs for all problem sizes between 1024 and 4096, which have a largest prime factor of 2, 3, 5 or 7 or are a prime number. This plot shows that the performance for problem sizes with low prime factors are faster than the performance for problem sizes, that happen to be prime numbers, by a factor of 2 to 3. Additionally, the performance for prime numbers exhibits a pronounced jump around 2048. Inspecting the respective kernels executed reveals the reason for this jump.

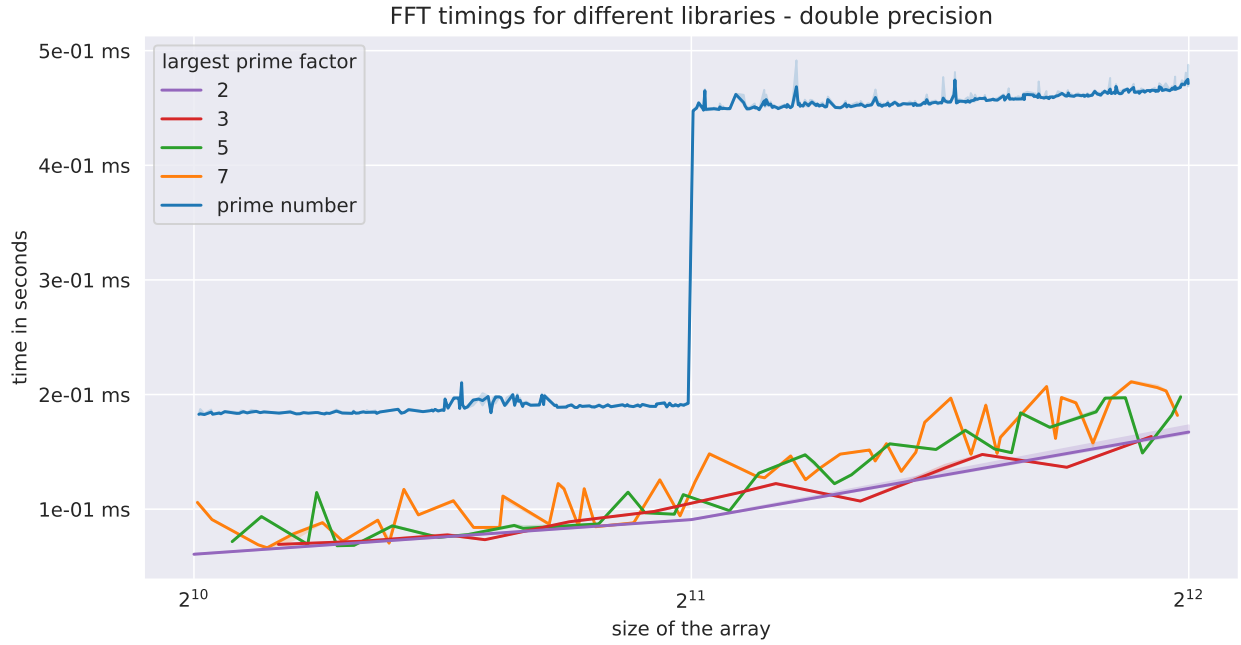


Figure 4.8: Performance of batched 1D transforms with a constant batch size of 92 for double precision on a GPU with PyTorch

As listed in Tables 4.9 and 4.10 the kernels executed for the prime number sizes 2039 and 2053 are different. For the larger prime number the kernel `regular_bluestein_fft` is replaced with 4 instances of the kernel `multi_bluestein_fft`. The kernel `regular_bluestein_fft` was launched with exactly 92 blocks (the same as the number of batched transforms), which suggests that this was the largest bluestein transform that could be calculated within a block on this particular GPU. Once a transform does not fit within a single block anymore the execution needs to be split into multiple kernels, since CUDA makes no guarantees about the order in which blocks within a kernel are executed. Separating different stages of a transform acts a way to introduce synchronization, as the order of the kernels is guaranteed. This introduces the drawback of additional round-trips of the data between the GPU registers and the global memory, which is why the performance of the multi-kernel execution is slower than the performance of the single kernel execution.

Kernel Name	grid dimension	block dimension	thread count	occupancy	execution time
`packR2C_Odd_kernel`	367 x 1 x 1	256 x 1 x 1	93,952	100.00%	8.1 μs
`regular_bluestein_fft`	46 x 1 x 1	256 x 1 x 1	11,776	16.67%	139.8 μs
`postprocessC2C_kernelMem`	184 x 1 x 1	256 x 1 x 1	47,104	100.00%	5.5 μs
`_fft_conjugate_copy_kernel`	92 x 1 x 1	1024 x 1 x 1	94,208	66.67%	7.1 μs

Table 4.9: Kernels executed for 92 batched transforms of size 2039, double precision PyTorch on the GPU

Kernel Name	grid dimension	block dimension	thread count	occupancy	execution time
`packR2C_Odd_kernel`	369 x 1 x 1	256 x 1 x 1	94,464	100.00%	7.9 μs
`multi_bluestein_fft`	184 x 1 x 1	32 x 8 x 1	47,104	50.00%	105.5 μs
`multi_bluestein_fft`	92 x 1 x 1	32 x 8 x 1	23,552	33.33%	84.8 μs
`multi_bluestein_fft`	184 x 1 x 1	32 x 8 x 1	47,104	50.00%	107.2 μs
`multi_bluestein_fft`	92 x 1 x 1	32 x 8 x 1	23,552	33.33%	94.2 μs
`postprocessC2C_kernelMem`	185 x 1 x 1	256 x 1 x 1	47,360	100.00%	6.4 μs
`_fft_conjugate_copy_kernel`	93 x 1 x 1	1024 x 1 x 1	95,232	66.67%	8.1 μs

Table 4.10: Kernels executed for 92 batched transforms of size 2053, double precision PyTorch on the GPU

4.4.3 Error analysis

The single precision FFTs on the GPU are faster than the equivalent double precision CPU based FFTs, in this section we will investigate the accuracy of the different libraries.
To judge the accuracy we will create a random signal containing values in the $[-1, 1]$ range using Numpy and then copy it to the GPU, if necessary, perform the FFT, copy the result back to the CPU and then compare the result to the result of a CPU based quad precision FFT based on the L2 error. This test is similar to the one conducted in [83], but also including different CPU based libraries as well as half-precision GPU-based FFTs where available (cuFFT only supports half-precision transforms of power-of-two sized transforms).
Figure 4.9 gives an overview of the error for all tested libraries and precision levels. As expected the error greatly depends on the used precision level, but cuFFT based libraries are not necessarily less accurate than their CPU based counterparts. The benchmarks do not display significant differences between the different libraries for the same precision level, except for Numpy. Numpy offers the best accuracy for single precision transforms out of all tested libraries, this is because Numpy's `fft.fft` routine always promotes inputs to doble precision and also always returns results of type complex-double.

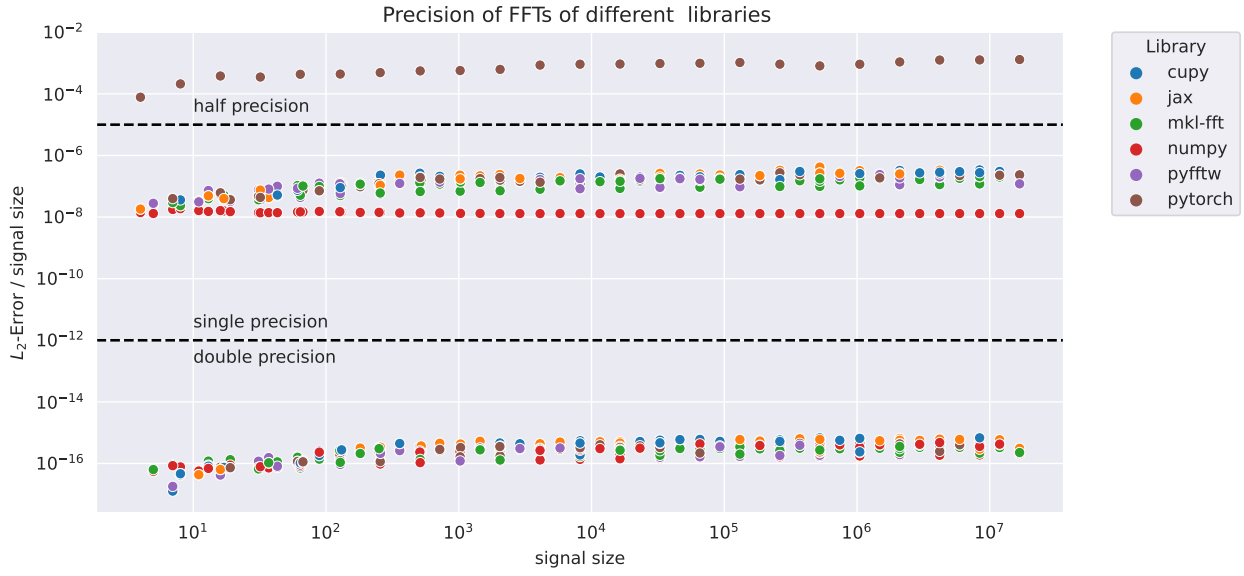


Figure 4.9: L2 error of the different libraries and precision levels

Figure 4.10 shows the error for single precision transforms on the GPU, which is probably the most relevant precision level for applications which could benefit from the GPU-acceleration. The figure shows that the difference between the different libraries is not zero, but it is small enough to be negligible for

most applications. In this particular test PyTorch is the most accurate GPU-based library. In general the error is smaller for power-of-two sized transforms.

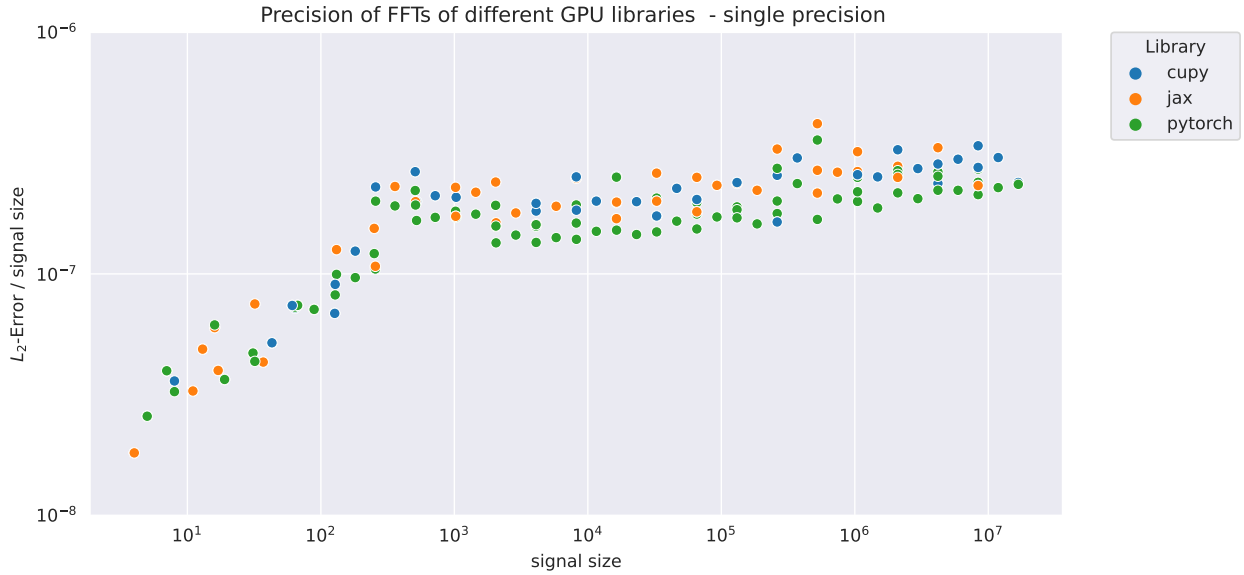


Figure 4.10: L2 error of the different libraries and precision levels for single precision transforms on the GPU

4.4.4 Effect of memory copy overhead on performance

The previous tests have shown that calculating FFTs is faster than CPU options for most cases (single as well as double precision, batched as well as single transforms), but these tests assumed that the data already resides on the GPU. This can be a justified assumption in some cases, on Nvidia's Jetson series for example the GPU has no dedicated memory, but a shared, physically unified memory with the CPU, therefore on that and similar platforms there is no need to transfer data to and from the GPU [90].
On most other platforms however the GPU has dedicated memory, which requires data to be copied to and from the GPU over the PCI-E bus. This can be a significant overhead, especially for small transforms, where the overhead of the memory copy can be larger than the actual computation time. Table 4.11 lists the performance and potential speedup for the different libraries, when the data is copied to and from the GPU. It is not entirely clear why CuPy performs best followed by PyTorch and then JAX. Investigations into the recorded profiles show that the amount of data copied is exactly the same across the libraries but the achieved bandwidth varies and that JAX distributes the copy-tasks across several streams, which might introduce additional overhead. The HtoD movements are generally faster (reaching 13 GB/s with all libraries) than the DtoH movements (PyTorch and JAX: 3.7 GB/s, CuPy: 7.2 GB/s). The difference is not explained by CuPy using pinned memory, since the DtoH movements appear as pageable for all libraries. It seems likely that CuPy uses a different memory allocator for its host-side memory, which might be write-able faster than the default allocator (might be achieved by using a different alignment, in order to allow CPU-SIMD instructions to copy the data).
It can be seen that the overhead of the memory copy gets relatively smaller the larger the transform size gets and that the speedups (again compared to a Numpy baseline) are non-existing for small transform sizes and low for even for large transforms. The speedups only reach values of up to 24 instead of the values seen in Table 4.5, where only the computation time is considered. This makes the case for using GPUs for single transforms weak, since similar speedups were also achieved by highly optimized CPU based libraries (MKL-FFT reached values in the 10-22 range for single precision transforms and values ranging from 5-10 for double precision transforms). The fact that data movement between across the

	pytorch				cupy				jax			
	single		double		single		double		single		double	
	median	speedup	median	s.up	median	s.up	median	s.up	median	s.up	median	s.up
2^2	94.1μs	-0.9	99.8μs	-0.9	139.5μs	-0.9	143.9μs	-0.9	413.1μs	-1.0	412.1μs	-1.0
2^3	95.4μs	-0.9	96.4μs	-0.9	139.9μs	-0.9	138.8μs	-0.9	410.4μs	-1.0	453.3μs	-1.0
2^4	103.5μs	-0.9	97.1μs	-0.9	143.4μs	-0.9	144.0μs	-0.9	416.7μs	-1.0	412.8μs	-1.0
2^5	97.9μs	-0.9	96.1μs	-0.9	138.3μs	-0.9	154.1μs	-0.9	402.4μs	-1.0	432.7μs	-1.0
2^6	97.0μs	-0.9	97.1μs	-0.9	136.1μs	-0.9	147.0μs	-0.9	399.4μs	-1.0	452.5μs	-1.0
2^7	97.9μs	-0.9	98.1μs	-0.9	141.3μs	-0.9	146.2μs	-0.9	414.4μs	-1.0	423.0μs	-1.0
2^8	96.7μs	-0.9	96.1μs	-0.9	139.5μs	-0.9	150.7μs	-0.9	411.8μs	-1.0	456.8μs	-1.0
2^9	96.2μs	-0.9	97.8μs	-0.9	146.0μs	-0.9	146.5μs	-0.9	422.2μs	-1.0	406.4μs	-1.0
2^{10}	170.6μs	-0.9	109.0μs	-0.8	150.2μs	-0.9	155.9μs	-0.9	428.2μs	-1.0	447.6μs	-1.0
2^{11}	98.3μs	-0.7	126.1μs	-0.8	143.4μs	-0.8	173.3μs	-0.8	411.3μs	-0.9	432.4μs	-0.9
2^{12}	101.3μs	-0.6	174.1μs	-0.7	151.0μs	-0.7	216.5μs	-0.8	440.6μs	-0.9	445.8μs	-0.9
2^{13}	109.0μs	-0.2	216.9μs	-0.6	154.8μs	-0.5	310.2μs	-0.7	455.8μs	-0.8	550.7μs	-0.8
2^{14}	134.5μs	0.2	355.3μs	-0.5	168.1μs	-0.0	229.6μs	-0.2	438.3μs	-0.6	469.3μs	-0.6
2^{15}	218.3μs	0.5	352.7μs	-0.0	181.9μs	0.9	275.0μs	0.2	485.0μs	-0.3	509.0μs	-0.3
2^{16}	313.0μs	1.2	543.4μs	0.3	223.0μs	2.1	346.7μs	1.0	547.1μs	0.3	611.3μs	0.1
2^{17}	502.0μs	2.0	869.3μs	0.8	301.7μs	4.0	514.1μs	2.1	685.6μs	1.2	836.3μs	0.9
2^{18}	823.1μs	3.3	1.6ms	1.3	422.9μs	7.3	889.6μs	3.1	741.1μs	3.8	1.6ms	1.3
2^{19}	1.4ms	5.4	2.9ms	2.3	716.7μs	11.7	1.7ms	4.6	1.2ms	6.5	3.1ms	2.1
2^{20}	2.7ms	9.5	5.5ms	4.3	1.4ms	19.5	3.6ms	7.0	2.7ms	9.4	4.6ms	5.3
2^{21}	5.1ms	13.6	12.2ms	5.1	3.0ms	24.3	8.8ms	7.5	4.0ms	17.8	17.1ms	3.3
2^{22}	11.5ms	12.2	23.9ms	5.4	7.6ms	18.9	17.4ms	7.8	15.8ms	8.6	34.0ms	3.5
2^{23}	22.7ms	12.6	47.4ms	5.7	15.3ms	19.2	33.9ms	8.4	31.6ms	8.8	67.0ms	3.8
2^{24}	45.3ms	13.2	93.9ms	5.9	30.6ms	20.0	67.1ms	8.7	63.7ms	9.1	132.6ms	3.9
2^{25}	92.9ms	14.5	194.3ms	6.6	62.1ms	22.2	148.7ms	8.9	125.2ms	10.5	268.0ms	4.5

Table 4.11: Performance of the different libraries for different transform sizes, when the data is copied to and from the GPU, related to Table 4.5

PCI-E bus amounts to $> 90\%$ of the total run-time for this task, also puts the discussed differences in run-time for power-of-two sized transforms in the best case and prime-sized transforms in the worst case into perspective. The run-time difference becomes negligible when the data is copied to and from the GPU, since the data movement is the dominant factor in the run-time. This relates back to the roofline model (Section 2.6), which explains how the arithmetic intensity of an algorithm relates to the performance of the algorithm.
The increase of the speed-up as the transform size increases can be explained by the increasing arithmetic intensity of the FFT algorithm as transform sizes grow ($\mathcal{O}(n \log(n))$), whereas the data needed to be copied only grows linearly. Taking the cost of memory copies into account, also increases the appeal of operating with lower precision, since the less memory needs to be copied and the arithmetic intensity automatically increases as it is defined as the ratio of the number of floating point operations to the number of bytes read from memory. Single precision transforms are almost exactly twice as fast as their double precision counterparts, which can be mostly attributed to the fact that only half the amount of data needs to be copied to and from the GPU. The relatively weak FP64 performance of the GPU is not reflected in the results, since the data movement is the dominant factor in the run-time. This was not the case in the results listed in Table 4.4, there are significant differences in the run-time for single and double precision transforms, but they do not matter in a memory bound regime. Every factor other than the interconnect speed becomes irrelevant.

4.4.4.1 Windowing

Applying windowing functions to the input data before performing the FFT is a common practice in signal processing. From the perspective of accelerated computing it is an embarrassingly parallel operation, which can be easily parallelized on the GPU. The run-time of such an operation is primarily defined by reading and writing the data to and from the GPU memory (memory bound operation). Ideally

the windowing operation should take place within the FFT-kernel, to avoid the overhead of a memory round trip (kernel fusion). CuFFT offers the possibility to use so-called callback functions to supply the FFT-Kernel with user-defined device functions, which are executed before or after the FFT is performed, when the data is still residing in the GPU's registers.
Unfortunately the discussed libraries do not offer the possibility to use callback functions, except for CuPy [91]. This test is therefore limited to a comparison between CuPy with and without the callback function. Another drawback of callback functions is the fact that they need to be written in CUDA C. Listings 4.1 and 4.2 display the two different approaches to window a signal with a Hann-Window.

```
__device__ cufftComplex CB_ConvertInputC(
  void *dataIn,
  size_t offset,
  void *callerInfo,
  void *sharedPtr)
{
  int len = *((int *)callerInfo);

  cufftComplex* floatIn = (cufftComplex*)dataIn;
  cufftComplex element = floatIn[offset];
  float hann_factor = (0.5f - 0.5f*
    cospif((2.f*(float)(offset))/(float(len)-1.f)));
  element.x = element.x*hann_factor;
  element.y = element.y*hann_factor;
  return element;
}
__device__ cufftCallbackLoadC d_loadCallbackPtr = CB_ConvertInputC;
```

Listing 4.1: Hann-Windowing via callback function in CUDA C

```
def cupy_hann(n):
  return ((cp.ones(n, dtype=cp.float32) - cp.cos(cp.linspace(0,
  ↪  2*cp.pi, n,  endpoint=False,dtype=cp.float32)))*0.5)

cp.fft.fft(cupy_hann(size)*a)
```

Listing 4.2: Hann-Windowing via multiplication with a CuPy array

The approach of using a callback function is more efficient, as it does not only avoid an additional memory round trip, but also avoids the need to store the windowing function in GPU memory. Another drawback of the approach outlined in Listing 4.1 is the fact that it requires a re-compilation of the cuFFT before the first invocation (after all the callback function will be compiled into the kernel). This re-compilation can take up to 20 seconds, which is something that should be kept in mind (the kernel does not need to be re-compiled for different transform sizes). The version listed in Listing 4.2 is more convenient, since it does not require the user to write CUDA C (note how the CUDA C code is specific to single precision transforms) code, but allocates additional memory on the GPU: the original input data, the windowing function and the windowed data. A third option is to calculate the windowing function once (either on the CPU or on the GPU), store it on the GPU and apply it to the input data by multiplication. Table 4.12 shows the results of benchmarks comparing the three different approaches.
The table lists a speedup of up to 400% for the callback function in comparison to the option using repeated calls to the windowing function. The version utilizing a callback is also a fair bit faster than the version using a cached window, with the upside of not having to construct and store the window function.
The tables 4.13 to 4.15 show the executed kernels by the respective variants tested for the transformation size of 2^{25}.

Kernel Name	grid dimension	block dimension	thread count	occupancy	execution time
`cupy_copy__float32_complex64`	262144 x 1 x 1	128 x 1 x 1	33,554,432	100.00%	964.3 μs
`regular_fft_callback`	8192 x 1 x 1	16 x 16 x 1	2,097,152	50.00%	1.4 ms
`regular_fft`	16384 x 1 x 1	8 x 16 x 1	2,097,152	50.00%	1.4 ms
`regular_fft`	8192 x 1 x 1	8 x 64 x 1	4,194,304	100.00%	1.3 ms

Table 4.13: Kernels executed in CuPy with the callback function

	cupy				
	function calls	cached window		callback	
	median	median	speedup	median	speedup
2^2	213.9μs	71.7μs	298.5 %	56.9μs	375.6 %
2^3	207.6μs	72.4μs	286.8 %	55.9μs	371.5 %
2^4	218.6μs	66.4μs	329.2 %	58.2μs	375.4 %
2^5	199.7μs	69.3μs	288.1 %	97.6μs	204.6 %
2^6	201.5μs	66.5μs	302.8 %	57.0μs	353.6 %
2^7	217.4μs	69.4μs	313.1 %	61.8μs	351.7 %
2^8	208.4μs	70.9μs	294.0 %	57.0μs	365.6 %
2^9	220.3μs	69.8μs	315.4 %	55.6μs	395.8 %
2^{10}	217.6μs	79.8μs	272.7 %	62.0μs	351.2 %
2^{11}	218.9μs	68.7μs	318.8 %	76.9μs	284.8 %
2^{12}	210.4μs	70.2μs	299.6 %	64.6μs	325.7 %
2^{13}	215.6μs	77.1μs	279.6 %	64.9μs	332.3 %
2^{14}	282.6μs	82.1μs	344.3 %	78.5μs	359.9 %
2^{15}	217.5μs	75.4μs	288.6 %	68.3μs	318.6 %
2^{16}	211.4μs	79.0μs	267.6 %	68.4μs	308.9 %
2^{17}	220.0μs	84.0μs	261.9 %	69.4μs	316.9 %
2^{18}	222.3μs	84.3μs	263.8 %	75.1μs	296.0 %
2^{19}	255.9μs	110.2μs	232.3 %	109.4μs	233.9 %
2^{20}	372.7μs	185.9μs	200.4 %	172.9μs	215.5 %
2^{21}	650.1μs	345.7μs	188.1 %	309.8μs	209.8 %
2^{22}	1.2ms	679.0μs	184.0 %	581.5μs	214.9 %
2^{23}	2.8ms	1.8ms	160.2 %	1.3ms	210.6 %
2^{24}	5.9ms	3.1ms	194.1 %	2.6ms	226.7 %
2^{25}	12.8ms	7.3ms	174.9 %	6.2ms	205.4 %

Table 4.12: Comparison of the run-time of the three different approaches to windowing

Table 4.13 lists the kernels executed when using the callback function. It can be seen that there is no extra kernel for the windowing, but the first FFT-kernel is altered to include the windowing function (the name `regular_fft_callback` reflects that change). Here the windowing comes at virtually no cost, as the kernel is executed in the same time as the original kernel.

Kernel Name	grid dimension	block dimension	thread count	occupancy	execution time
`cupy_fill`	262144 x 1 x 1	128 x 1 x 1	33,554,432	100.00%	372.6 μs
`cupy_linspace__float_float_float64`	262144 x 1 x 1	128 x 1 x 1	33,554,432	100.00%	675.6 μs
`cupy_copy__float64_float32`	262144 x 1 x 1	128 x 1 x 1	33,554,432	100.00%	995.6 μs
`cupy_cos__float32_float32`	262144 x 1 x 1	128 x 1 x 1	33,554,432	100.00%	649.7 μs
`cupy_subtract__float32_float32_float32`	262144 x 1 x 1	128 x 1 x 1	33,554,432	100.00%	975.7 μs
`cupy_multiply__float32_float_float32`	262144 x 1 x 1	128 x 1 x 1	33,554,432	100.00%	681.8 μs
`cupy_multiply__float32_float32_float32`	262144 x 1 x 1	128 x 1 x 1	33,554,432	100.00%	985.5 μs
`cupy_copy__float32_complex64`	262144 x 1 x 1	128 x 1 x 1	33,554,432	100.00%	1.0 ms
`regular_fft`	16384 x 1 x 1	8 x 16 x 1	2,097,152	50.00%	1.4 ms
`regular_fft`	16384 x 1 x 1	8 x 16 x 1	2,097,152	50.00%	2.1 ms
`regular_fft`	8192 x 1 x 1	8 x 64 x 1	4,194,304	100.00%	1.3 ms

Table 4.14: Kernels executed when repeatedly calling the cupy-hann function

Table 4.14 shows the kernels executed when repeatedly calling the windowing function. In this case there are many kernels, which need to be executed every time to construct the window function. This is not especially efficient but might occur in practice, as it is a very convenient. Note that this approach could

be sped up by fusing all the kernels preceding the FFT into a single kernel, this could be achieved with PyTorch's JIT compiler.

Kernel Name	grid dimension	block dimension	thread count	occupancy	execution time
`cupy_multiply__float32_float32_float32`	262144 x 1 x 1	128 x 1 x 1	33,554,432	100.00%	982.0 μs
`cupy_copy__float32_complex64`	262144 x 1 x 1	128 x 1 x 1	33,554,432	100.00%	1.0 ms
`regular_fft`	16384 x 1 x 1	8 x 16 x 1	2,097,152	50.00%	1.6 ms
`regular_fft`	16384 x 1 x 1	8 x 16 x 1	2,097,152	50.00%	1.6 ms
`regular_fft`	8192 x 1 x 1	8 x 64 x 1	4,194,304	100.00%	1.3 ms

Table 4.15: Kernels executed when re-using the result from cupy-hann

Table 4.15 shows the kernels executed when re-using the result from the windowing function. This is also a quite efficient approach, as the windowing function is only calculated once and then re-used. The multiplication with the input data is fast, as it is an element-wise operation, which is only memory bound. Re-using a window function, residing on the host's memory, however would be much slower, as the data would have to be transferred to the GPU.
In summary, the callback functions for FFTs are a way to gain performance, but its effect is not as pronounced as one might expect. It is unfortunate that not all libraries support this feature.

4.4.5 Transforms of higher dimensionality

As explored in Section 2.7.2 FFTs of higher dimensionality involve FFTs along the individual axis of the array. Between transforms along different axis it can be beneficial to re-order the data in memory (in 2D: transposing matrix), in order to improve the data locality and thus the performance of the subsequent FFT stages.
Tables 4.16 and 4.17 show the timings for 2D FFTs using CPU-based and GPU-based libraries, respectively. The tables only list the times and speedups for the execution, excluding the time for the data transfer. The listed size refers to the size of the input data, which is a 2D array of size $N \times N$ of real valued numbers.
The speedups achieved on the CPU (compared to the baseline of the Numpy FFT) are mostly in-line with the results for 1D FFTs. The highest speedup was achieved by MKL-FFT (28.8 in single precision and 20.7 in double precision).

	mkl-fft				pyfftw				numpy
	single		double		single		double		double
	median	speedup	median	s.up	median	s.up	median	s.up	median
2^2	8.1μs	1.4	8.4μs	0.5	25.7μs	-0.3	24.9μs	-0.5	12.8μs
2^3	8.7μs	0.5	8.8μs	0.5	27.0μs	-0.5	26.5μs	-0.5	13.1μs
2^4	9.2μs	0.7	9.5μs	0.5	31.2μs	-0.5	31.7μs	-0.5	14.6μs
2^5	12.5μs	0.6	13.7μs	0.4	34.4μs	-0.4	37.9μs	-0.5	19.6μs
2^6	45.0μs	-0.0	48.7μs	-0.1	51.1μs	-0.1	52.7μs	-0.2	44.3μs
2^7	89.4μs	0.8	123.3μs	0.3	83.1μs	0.9	119.0μs	0.3	160.2μs
2^8	381.2μs	1.3	393.3μs	1.3	394.2μs	1.2	1.2ms	-0.2	886.8μs
2^9	1.3ms	2.0	1.1ms	2.9	1.7ms	1.4	1.3ms	2.1	4.2ms
2^{10}	3.7ms	5.8	4.2ms	5.0	9.2ms	1.7	7.8ms	2.2	25.0ms
2^{11}	15.6ms	18.7	22.5ms	12.9	68.2ms	3.5	90.7ms	2.4	311.9ms
2^{12}	61.0ms	28.9	87.2ms	20.2	384.0ms	3.8	428.6ms	3.3	1.85s
2^{13}	250.4ms	28.8	347.5ms	20.7	2.56s	1.9	1.00s	6.5	7.53s

Table 4.16: Timings for 2D FFTs using CPU-based libraries

The GPU-based libraries manage to achieve even higher speedups than those measured for 1D FFTs. Single precision transforms achieved speedups of up to 640 and double precision transforms of up to 120. The equivalent test for 1D FFTs (see Table 4.5) achieved speedups of up to 420 and 85, respectively. The increase in speedup can be attributed to the better utilization of the GPU's streaming multiprocessors. A great number of smaller FFTs require less synchronization between the different threads and thus allow for a higher degree of parallelism.

	pytorch				jax				cupy			
	single		double		single		double		single		double	
	median	speedup	median	s.up	median	s.up	median	s.up	median	s.up	median	s.up
2^2	50.1μs	-0.6	47.9μs	-0.7	62.2μs	-0.7	62.5μs	-0.8	81.8μs	-0.8	80.5μs	-0.8
2^3	47.7μs	-0.7	48.5μs	-0.7	60.1μs	-0.8	54.0μs	-0.8	77.3μs	-0.8	78.0μs	-0.8
2^4	47.6μs	-0.7	51.6μs	-0.7	69.9μs	-0.8	74.5μs	-0.8	79.5μs	-0.8	79.5μs	-0.8
2^5	50.7μs	-0.6	58.7μs	-0.7	57.3μs	-0.7	89.8μs	-0.8	76.1μs	-0.7	88.9μs	-0.8
2^6	47.3μs	-0.1	67.9μs	-0.3	56.8μs	-0.2	101.5μs	-0.6	77.4μs	-0.4	99.2μs	-0.6
2^7	48.5μs	2.3	71.4μs	1.2	59.1μs	1.7	108.6μs	0.5	80.0μs	1.0	98.4μs	0.6
2^8	49.1μs	17.0	84.0μs	9.6	68.7μs	11.9	123.0μs	6.2	76.5μs	10.6	113.8μs	6.8
2^9	57.7μs	68.3	209.3μs	19.0	91.6μs	42.6	260.3μs	15.1	86.5μs	45.2	241.0μs	16.4
2^{10}	143.3μs	174.3	799.8μs	30.3	190.7μs	130.7	857.7μs	28.2	160.3μs	155.8	814.7μs	29.7
2^{11}	525.1μs	585.3	3.1ms	99.2	578.3μs	531.4	3.2ms	97.4	538.2μs	571.1	3.1ms	99.8
2^{12}	2.8ms	659.2	14.5ms	126.9	2.5ms	733.7	13.8ms	133.2	2.4ms	760.6	13.7ms	134.4
2^{13}	11.7ms	636.8	62.5ms	119.5	12.1ms	617.3	274.3ms	26.5	11.5ms	647.3	59.0ms	126.7

Table 4.17: Timings for 2D FFTs using GPU-based libraries

Transforms of higher dimensionality are also more likely to benefit from offloading to the GPU, as every number transferred is re-used in multiple FFTs. This increases the arithmetic intensity, which enables the GPU to achieve higher performance. Table 4.18 shows the performance of the different libraries for different transform sizes, when the data is copied to and from the GPU. In this scenario the GPU can offer significant speedups of up to a factor of 60 for single precision and 25 for double precision. The fastest library is again CuPy, as it achieves higher copy speeds than the other libraries. This result makes a strong case for GPGPU acceleration of large (or batched) FFTs of higher dimensionality. The GPU-based solution is faster than all CPU-based solutions, even for double precision, albeit not by a large margin (note that the baseline is Numpy, which is outperformed by libraries like MKL-FFT).

	pytorch				jax				cupy			
	single		double		single		double		single		double	
	median	speedup	median	s.up	median	s.up	median	s.up	median	s.up	median	s.up
2^2	111.6μs	-0.8	106.0μs	-0.8	278.7μs	-0.9	282.0μs	-0.9	172.3μs	-0.9	168.2μs	-0.9
2^3	113.1μs	-0.8	109.9μs	-0.8	287.2μs	-0.9	259.0μs	-0.9	167.7μs	-0.9	166.0μs	-0.9
2^4	108.1μs	-0.8	116.4μs	-0.8	267.7μs	-0.9	265.3μs	-0.9	170.7μs	-0.9	177.3μs	-0.9
2^5	107.9μs	-0.8	122.8μs	-0.8	293.3μs	-0.9	288.8μs	-0.9	167.3μs	-0.8	179.9μs	-0.9
2^6	115.2μs	-0.6	134.7μs	-0.6	272.3μs	-0.8	310.6μs	-0.8	171.6μs	-0.7	198.2μs	-0.7
2^7	133.0μs	0.3	297.0μs	-0.4	318.4μs	-0.4	379.4μs	-0.5	186.7μs	-0.0	226.8μs	-0.2
2^8	206.5μs	3.5	546.4μs	0.7	376.1μs	1.4	504.2μs	0.8	244.8μs	2.8	351.1μs	1.6
2^9	830.6μs	4.0	1.6ms	1.8	710.2μs	4.9	1.5ms	1.9	429.8μs	8.7	846.0μs	4.2
2^{10}	2.7ms	8.8	5.8ms	3.4	2.6ms	9.3	5.0ms	4.1	1.3ms	18.7	3.5ms	6.3
2^{11}	11.8ms	23.4	24.9ms	10.9	16.3ms	16.7	35.0ms	7.4	7.8ms	35.9	17.2ms	16.2
2^{12}	46.5ms	38.5	100.3ms	17.5	63.9ms	27.7	134.7ms	12.8	31.0ms	58.3	70.2ms	25.4
2^{13}	184.0ms	39.9	409.0ms	17.4	251.5ms	28.9	632.5ms	10.9	123.2ms	60.1	284.2ms	25.5

Table 4.18: Performance of the different libraries for different transform sizes, when the data is copied to and from the GPU, related to Table 4.17

5 Conclusion and Future Directions

In this section we summarize the main findings of the thesis and provide an outlook on future research directions and emerging trends in the field.

5.1 Key Findings

The tests conducted show that one can achieve tremendous speed ups for common tasks often occuring in signal processing pipelines. For especially well suited problems, those that can be broken down in thousands of smaller, independent problems, speed-ups in the range of 400 to 600 can be achieved. Even though large parts of those, often claimed, high speed-ups vanish, when taking account the overhead associated with copying the data to and from the GPU, offloading certain tasks to the GPU can be worth the additional effort and complexity. For such problems offloading to the GPU the overhead associated with copying the necessary data to and from the GPU is worth it, despite the overhead. We encourage readers to apply the introduced techniques and tools to their own problems, as the speedups can be significant and allow for otherwise impossible use-cases.

5.2 Implications for Signal Processing and GPU Programming

To achieve the best performance it is of utmost importance to perform as many consecutive steps of the signal processing pipeline on the GPU to eliminate the need for expensive data movements. This is a convenient requirement as the final step in many modern signal processing pipelines, especially in those real-time applications were performance is critical, is often a deep-learning model. Those models are often complex and require powerful GPU-like hardware to achieve the desired performance, it is therefore a logical step to move as much signal processing to the already available GPU. Such a situation, for example, is encountered in modern cars where the pre-processed data obtained from sensors like lidars and radars is used as input for deep-learning models to identify the position of surrounding vehicles and pedestrians. In this scenario the processed data would not needed to be transferred back to the host's memory and half of the costly memory transfers could be avoided completely.

5.2.1 New Frameworks

Apart from the frameworks discussed, there are other emerging or already existing solutions worth mentioning. This category of tools include Halide, DaCe and Triton.
Halide [92] is a library and domain specific language, which aims to separate the order of operations (schedule) from the defined operations themselves (algortihm). While Halide was originally designed as a library for image processing, it is very likely that the introduced concepts, which make it a popular library for accelerated image processing, would also apply to more generale signal processing.
DaCe [93] (short for data-centric parallel programming) is a set of tools for analyzing, re-ordering and optimizing graphs of computations needed. It is especially appealing due to its graphical views.
OpenAI's Triton [94] meanwhile is a programming language that treats blocks of threads as the lowest level of abstraction instead of individual threads. This is a different programming style that lends itself very well to writing kernels which optimize for cache re-use and memory coalescing. The drawback of this approach is that it does not allow for kernels, which require irregular memory access patterns like the FFT kernels discussed.
All of these tools have in common that they involve another layer of abstraction and an additional compilation step and try to seperate the expression of the algorithm from the actual algorithm in order to adapt it optimally to the hardware used. Additionally they can all be used to generate code for multiple backends (CUDA, OpenCL, x86) by emitting an LLVM compatible intermediate represenation. By using these tools programmers hand over fine-grained control over the kernel in exchange for a wider

range of automatic optimizations. The fact that multiple tools, aiming to solve very similar problems, exist, shows that there are shortcomings in the currently offered tools and that there is demand for more abstract, higher-level tools. It is likely that some of these approaches will mature and complement the current landscape of GPU programming tools.

5.2.2 Future Research Directions

Future further research in this area could focus on mixed-precision arithmetic and lower precision data-types (TF32, FP16, BF16), since newer GPU hardware is able to process vastly more data per time when dealing with these data-types, while FP64 performance is stagnating and not a priority.
Another possible direction would be to investigate the use of multiple GPUs in parallel. In such a setting the GPUs could either perform completely independent taks, or they could be used to perform the same task on different parts of the data. In any case, such a setup would require even more careful consideration of the data movements between the individual GPUs and the host's memory.

List of Figures

List of Tables

List of Listings

References

[1] "Tiobe index." (2022), [Online]. Available: https://www.tiobe.com/tiobe-index/ (visited on 02/10/2022).

[2] C. R. Harris, K. J. Millman, S. J. Van Der Walt, *et al.*, "Array programming with numpy," *Nature*, vol. 585, no. 7825, pp. 357–362, 2020.

[3] *Numpy.ndarray — numpy v1.22 manual*, https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html, (Accessed on 02/10/2022).

[4] *Dis — disassembler for python bytecode — python 3.9.10 documentation*, https://docs.python.org/3.9/library/dis.html, (Accessed on 02/11/2022).

[5] *Cpython/ceval.c at 3.9 · python/cpython*, https://github.com/python/cpython/blob/3.9/Python/ceval.c, (Accessed on 02/10/2022).

[6] *Cpython/abstract.c at 3.9 · python/cpython · github*, https://github.com/python/cpython/blob/3.9/Objects/abstract.c, (Accessed on 08/08/2022).

[7] *Cpython/floatobject.c at 3.9 · python/cpython · github*, https://github.com/python/cpython/blob/3.9/Objects/floatobject.c, (Accessed on 08/08/2022).

[8] *Pep 484 – type hints | python.org*, https://www.python.org/dev/peps/pep-0484/, (Accessed on 02/11/2022).

[9] *Threading — thread-based parallelism — python 3.9.10 documentation*, https://docs.python.org/3.9/library/threading.html, (Accessed on 02/11/2022).

[10] *Valgrind - detected errors: Data races*, https://valgrind.org/docs/manual/hg-manual.html, (Accessed on 02/11/2022).

[11] L. Hastings, *The gilectomy: How's it going?* https://us.pycon.org/2017/schedule/presentation/118/, Talk by Larry Hastings at the Pycon 2017 in Portland, OR, May 2017.

[12] S. Gross, *Python multithreading without the gil*, https://github.com/colesbury/nogil, (Accessed on 02/11/2022).

[13] *Cpython/ceval_gil.h at 3.9 · python/cpython*, https://github.com/python/cpython/blob/3.9/Python/ceval_gil.h, (Accessed on 02/11/2022).

[14] D. Beazley, *Understanding the python gil*, http://www.dabeaz.com/GIL/, Talk by David Beazley at Pycon 2010 in Atlanta, GE, Feb. 2010.

[15] P. E. McKenney, M. M. Michael, M. Gupta, P. W. Howard, J. Triplett, and J. Walpole, "Is parallel programming hard, and if so, why?," 2009.

[16] *Blas (basic linear algebra subprograms)*, http://www.netlib.org/blas/, (Accessed on 02/12/2022).

[17] *Numpy/matmul.c.src at 410a89ef04a2d3c50dd2dba2ad403c872c3745ac · numpy/numpy*, https://github.com/numpy/numpy/blob/410a89ef04a2d3c50dd2dba2ad403c872c3745ac/numpy/core/src/umath/matmul.c.src, (Accessed on 02/12/2022).

[18] *Openblas : An optimized blas library*, https://www.openblas.net/, (Accessed on 02/12/2022).

[19] *Accelerate fast math with intel® oneapi math kernel library*, https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html, (Accessed on 02/12/2022).

[20] *Amd/blis: Blas-like library instantiation software framework*, https://github.com/amd/blis, (Accessed on 02/12/2022).

[21] *Anaconda software distribution*, version Vers. 2-2.4.0, 2020. [Online]. Available: https://docs.anaconda.com/.

[22] R. Chandra, L. Dagum, D. Kohr, R. Menon, D. Maydan, and J. McDonald, *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[23] D. BUTTLAR, B. Nichols, D. Buttlar, J. Farrell, J. Farrell, and A. Oram, *PThreads Programming: A POSIX Standard for Better Multiprocessing* (A POSIX standard for better multiprocessing). O'Reilly Media, Incorporated, 1996, ISBN: 9781565921153. [Online]. Available: https://books.google.co.in/books?id=oMtCFSnvwmoC.

[24] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. S. Seljebotn, and K. Smith, "Cython: The best of both worlds," *Computing in Science Engineering*, vol. 13, no. 2, pp. 31–39, 2011. DOI: 10.1109/MCSE.2010.118.

[25] S. K. Lam, A. Pitrou, and S. Seibert, "Numba: A llvm-based python jit compiler," in *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, 2015, pp. 1–6.

[26] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis amp; transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, 2004, pp. 75–86. DOI: 10.1109/CGO.2004.1281665.

[27] W. Jakob, J. Rhinelander, and D. Moldovan, *Pybind11 – seamless operability between c++11 and python*, https://github.com/pybind/pybind11, 2017.

[28] S. Guelton, P. Brunet, M. Amini, A. Merlini, X. Corbillon, and A. Raynaud, "Pythran: Enabling static optimization of scientific python programs," *Computational Science & Discovery*, vol. 8, no. 1, p. 014 001, 2015.

[29] *Typed memoryviews — cython 0.29.28 documentation*, https://cython.readthedocs.io/en/stable/src/userguide/memoryviews.html, (Accessed on 02/17/2022).

[30] *Zen 2 - microarchitectures - amd - wikichip*, https://en.wikichip.org/wiki/amd/microarchitectures/zen_2, (Accessed on 02/18/2022).

[31] "Multithreaded programming with c++," in *Professional C++*. John Wiley & Sons, Ltd, 2021, ch. 27, pp. 915–967, ISBN: 9781119695547. DOI: https://doi.org/10.1002/9781119695547.ch27. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/9781119695547.ch27. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/9781119695547.ch27.

[32] J. Huang, T. M. Smith, G. M. Henry, and R. A. van de Geijn, *Implementing strassen's algorithm with blis*, 2016. DOI: 10.48550/ARXIV.1605.01078. [Online]. Available: https://arxiv.org/abs/1605.01078.

[33] *Not clear how to expose existing c++ vector as numpy array · issue #1042 · pybind/pybind11*, https://github.com/pybind/pybind11/issues/1042, (Accessed on 03/02/2022).

[34] *Nvidia-ampere-ga102-gpu-architecture-whitepaper-v1.pdf*, https://images.nvidia.com/aem-dam/en-zz/Solutions/geforce/ampere/pdf/NVIDIA-ampere-GA102-GPU-Architecture-Whitepaper-V1.pdf, (Accessed on 03/30/2022).

[35] A. Thall, "Extended-precision floating-point numbers for gpu computation," in *ACM SIGGRAPH 2006 Research Posters*, ser. SIGGRAPH '06, Boston, Massachusetts: Association for Computing Machinery, 2006, 52–es, ISBN: 1595933646. DOI: 10.1145/1179622.1179682. [Online]. Available: https://doi.org/10.1145/1179622.1179682.

[36] *Shadertoy beta*, https://www.shadertoy.com/, (Accessed on 06/19/2022).

[37] M. Harris, *Chapter 31. mapping computational concepts to gpus*, https://developer.nvidia.com/gpugems/gpugems2/part-iv-general-purpose-computation-gpus-primer/chapter-31-mapping-computational, (Accessed on 06/19/2022), 2005.

[38] *Nvidia cuda programming guide*, https://developer.download.nvidia.com/compute/cuda/1.0/NVIDIA_CUDA_Programming_Guide_1.0.pdf, (Accessed on 06/19/2022), Jun. 2007.

[39] J. P. Research, *Q4'21 sees a nominal rise in gpu and pc shipments quarter-to-quarter | jon peddie research*, https://www.jonpeddie.com/press-releases/q421-sees-a-nominal-rise-in-gpu-and-pc-shipments-quarter-to-quarter, (Accessed on 08/04/2022), Feb. 2022.

[40] *Ornl's frontier first to break the exaflop ceiling | top500*, https://www.top500.org/news/ornls-frontier-first-to-break-the-exaflop-ceiling/, (Accessed on 08/04/2022), May 2022.

[41] O. R. N. Laboratory, *Frontier supercomputer*, https://www.olcf.ornl.gov/frontier/, (Accessed on 08/04/2022), 2022.

[42] N. Otterness and J. H. Anderson, "AMD GPUs as an Alternative to NVIDIA for Supporting Real-Time Workloads," in *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*, M. Völp, Ed., ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 165, Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020, 10:1–10:23, ISBN: 978-3-95977-152-8. DOI: 10.4230/LIPIcs.ECRTS.2020.10. [Online]. Available: https://drops.dagstuhl.de/opus/volltexte/2020/12373.

[43] *Github - rocm-developer-tools/hipify: Hipify: Convert cuda to portable c++ code*, https://github.com/ROCm-Developer-Tools/HIPIFY, (Accessed on 08/04/2022).

[44] A. Herten, *GPU vendor/programming model compatibility table*, 2022.

[45] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, ser. AFIPS '67 (Spring), Atlantic City, New Jersey: Association for Computing Machinery, 1967, pp. 483–485, ISBN: 9781450378956. DOI: 10.1145/1465482.1465560. [Online]. Available: https://doi.org/10.1145/1465482.1465560.

[46] *Nvidia developer tools overview | nvidia developer*, https://developer.nvidia.com/tools-overview, (Accessed on 03/30/2022).

[47] R. Okuta, Y. Unno, D. Nishino, S. Hido, and C. Loomis, "Cupy: A numpy-compatible library for nvidia gpu calculations," in *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS)*, 2017. [Online]. Available: http://learningsys.org/nips17/assets/papers/paper_16.pdf.

[48] *Programming guide :: Cuda toolkit documentation*, https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, (Accessed on 07/11/2022).

[49] M. Harris, *How to optimize data transfers in cuda c/c++ | nvidia technical blog*, https://developer.nvidia.com/blog/how-optimize-data-transfers-cuda-cc/, (Accessed on 08/02/2022), Dec. 2012.

[50] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, pp. 65–76, Apr. 2009, ISSN: 0001-0782. DOI: 10.1145/1498765.1498785. [Online]. Available: https://doi.org/10.1145/1498765.1498785.

[51] E. Chu, *Inside the FFT black box : serial and parallel fast Fourier transform algorithms* (Computational mathematics series), eng. CRC Press, 2000, ISBN: 0849302706. [Online]. Available: http://media.obvsg.at/AC04278895-1001.

[52] M. Frigo and S. Johnson, "The design and implementation of fftw3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005. DOI: 10.1109/JPROC.2004.840301.

[53] M. Harris, *How to access global memory efficiently in cuda c/c++ kernels | nvidia technical blog*, https://developer.nvidia.com/blog/how-access-global-memory-efficiently-cuda-c-kernels/, (Accessed on 06/22/2022), Jan. 2013.

[54] S. Jones, *How cuda programming works*, https://www.nvidia.com/en-us/on-demand/session/gtcspring22-s41487/, (Accessed on 07/26/2022), Mar. 2022.

[55] G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, 2000.

[56] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007. DOI: 10.1109/MCSE.2007.55.

[57] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih, "PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation," *Parallel Computing*, vol. 38, no. 3, pp. 157–174, 2012, ISSN: 0167-8191. DOI: 10.1016/j.parco.2011.09.001.

[58] A. Klöckner, *Pycuda 2022.1 documentation*, https://documen.tician.de/pycuda/index.html, (Accessed on 08/04/2022).

[59] A. Paszke, S. Gross, F. Massa, *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[60] M. Nicely and K. Kraus, *Unifying the cuda python ecosystem | nvidia technical blog*, https://developer.nvidia.com/blog/unifying-the-cuda-python-ecosystem/, (Accessed on 08/06/2022), Apr. 2021.

[61] Nvidia, *Cuda python 11.7.1 documentation*, https://nvidia.github.io/cuda-python/, (Accessed on 08/06/2022).

[62] J. Bradbury, R. Frostig, P. Hawkins, *et al.*, *JAX: Composable transformations of Python+NumPy programs*, version 0.2.5, 2018. [Online]. Available: http://github.com/google/jax.

[63] *Xla: Optimizing compiler for machine learning | tensorflow*, https://www.tensorflow.org/xla, (Accessed on 08/08/2022).

[64] *Cupy/_kernel.pyx at v11 · cupy/cupy · github*, https://github.com/cupy/cupy/blob/v11/cupy/_core/_kernel.pyx, (Accessed on 08/08/2022).

[65] *Universal functions (ufunc) basics — numpy v1.23 manual*, https://numpy.org/doc/stable/user/basics.ufuncs.html, (Accessed on 08/09/2022).

[66] *Nvvm ir :: Cuda toolkit documentation*, https://docs.nvidia.com/cuda/nvvm-ir-spec/index.html, (Accessed on 08/09/2022).

[67] *Ptx isa :: Cuda toolkit documentation*, https://docs.nvidia.com/cuda/parallel-thread-execution/index.html, (Accessed on 08/09/2022).

[68] M. Harvey and G. De Fabritiis, "Swan: A tool for porting cuda programs to opencl," *Computer Physics Communications*, vol. 182, no. 4, pp. 1093–1099, 2011, ISSN: 0010-4655. DOI: https://doi.org/10.1016/j.cpc.2010.12.052. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0010465511000117.

[69] *Github - rocm-developer-tools/rocprofiler: Roc profiler library. profiling with perf-counters and derived metrics.* https://github.com/ROCm-Developer-Tools/rocprofiler, (Accessed on 08/11/2022).

[70] *Fundamental types - cppreference.com*, https://en.cppreference.com/w/cpp/language/types, (Accessed on 08/16/2022).

[71] *Jax - the sharp bits — jax documentation*, https://jax.readthedocs.io/en/latest/notebooks/Common_Gotchas_in_JAX.html, (Accessed on 08/24/2022).

[72] *Cuda llvm compiler | nvidia developer*, https://developer.nvidia.com/cuda-llvm-compiler, (Accessed on 08/25/2022).

[73] *Github - nvidia/cuda-python: Cuda python low-level bindings*, https://github.com/NVIDIA/cuda-python, (Accessed on 08/05/2022).

[74] *Cuda array interface (version 3) — numba 0.56.2+0.gd6731f6d2.dirty-py3.7-linux-x86_64.egg documentation*, https://numba.readthedocs.io/en/stable/cuda/cuda_array_interface.html, (Accessed on 09/29/2022).

[75] *Welcome to dlpack's documentation! — dlpack 0.6.0 documentation*, https://dmlc.github.io/dlpack/latest/, (Accessed on 09/29/2022).

[76] *Opencv: Cv::cuda::gpumat class reference*, https://docs.opencv.org/3.4/d0/d60/classcv_1_1cuda_1_1GpuMat.html, (Accessed on 12/21/2022).

[77] *Opencv/cuda.hpp at 2273af0166f5c4eb825802a187501874d20256f8 · opencv/opencv*, https://github.com/opencv/opencv/blob/2273af0166f5c4eb825802a187501874d20256f8/modules/core/include/opencv2/core/cuda.hpp, (Accessed on 12/22/2022).

[78] *Opencv: How opencv-python bindings works?* https://docs.opencv.org/4.x/da/d49/tutorial_py_bindings_basics.html, (Accessed on 12/22/2022).

[79] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex fourier series," *Mathematics of computation*, vol. 19, no. 90, pp. 297–301, 1965.

[80] K. Asanovic, R. Bodik, J. Demmel, *et al.*, "A view of the parallel computing landscape," *Commun. ACM*, vol. 52, no. 10, pp. 56–67, Oct. 2009, ISSN: 0001-0782. DOI: 10.1145/1562764.1562783. [Online]. Available: https://doi.org/10.1145/1562764.1562783.

[81] Nvidia, *Cufft | nvidia developer*, https://developer.nvidia.com/cufft, (Accessed on 10/11/2022).

[82] P. Steinbach and M. Werner, "Gearshifft – the fft benchmark suite for heterogeneous platforms," in *High Performance Computing*, J. M. Kunkel, R. Yokota, P. Balaji, and D. Keyes, Eds., Cham: Springer International Publishing, 2017, pp. 199–216, ISBN: 978-3-319-58667-0.

[83] D. Tolmachev, "Vkfft-a performant, cross-platform and open-source gpu fft library," *IEEE Access*, vol. 11, pp. 12 039–12 058, 2023. DOI: 10.1109/ACCESS.2023.3242240.

[84] *Github - vincefn/pyvkfft: Python interface to vkfft*, https://github.com/vincefn/pyvkfft, (Accessed on 02/13/2023).

[85] A. Thompson and M. Nicely, *cuSignal: The GPU-Accelerated Signal Processing Library*, version 21.08, Aug. 2021. [Online]. Available: https://github.com/rapidsai/cusignal.

[86] E. D. Berger, "Scalene: Scripting-Language Aware Profiling for Python," Jul. 2020. DOI: 10.48550/arXiv.2006.03879.

[87] *Numpy/readme.md at main · numpy/numpy*, https://github.com/numpy/numpy/blob/main/numpy/fft/README.md, (Accessed on 02/14/2023).

[88] *Cufft*, https://docs.nvidia.com/cuda/cufft/index.html, (Accessed on 02/14/2023).

[89] L. Rabiner and R. W. Schafer, "Bluestein's fft for arbitrary n on the hypercube," *Signal Processing*, vol. 9, no. 4, pp. 363–366, 1985.

[90] A. Brundyn, *Demystifying unified memory on jetson*, https://www.nvidia.com/en-us/on-demand/session/gtcspring22-se2600/?playlistId=playList-53110dbc-c11d-4619-b821-987015090afa, (Accessed on 02/27/2023), Mar. 2022.

[91] *Cupy.fft.config.set_cufft_callbacks — cupy 11.5.0 documentation*, https://docs.cupy.dev/en/stable/reference/generated/cupy.fft.config.set_cufft_callbacks.html, (Accessed on 02/27/2023).

[92] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," *SIGPLAN Not.*, vol. 48, no. 6, pp. 519–530, Jun. 2013, ISSN: 0362-1340. DOI: 10.1145/2499370.2462176. [Online]. Available: https://doi.org/10.1145/2499370.2462176.

[93] T. Ben-Nun, J. d. F. Licht, A. N. Ziogas, T. Schneider, and T. Hoefler, *Stateful dataflow multigraphs: A data-centric model for performance portability on heterogeneous architectures*, 2019. DOI: 10.48550/ARXIV.1902.10345. [Online]. Available: https://arxiv.org/abs/1902.10345.

[94] P. Tillet, H. T. Kung, and D. Cox, "Triton: An intermediate language and compiler for tiled neural network computations," in *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, ser. MAPL 2019, Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 10–19, ISBN: 9781450367196. DOI: 10.1145/3315508.3329973. [Online]. Available: https://doi.org/10.1145/3315508.3329973.