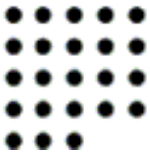
Mini Projekt Alternative Rechnerarchitekturen und Programmiersprachen:

Parallele Berechnung einer Fast Fourier Transform (FFT) mit CUDA und Python

Christoph Kuhr

WS 2013/14



Fachhochschule Köln
Cologne University of Applied Sciences

1 Einleitung

In diesem Projekt wird die Ausführung einer FFT Routine auf einer CPU mit der Ausführung auf einer GPU verglichen. Als CPU dient hierbei ein AMD Phenom X6 3,3Ghz und als GPU kommt eine NVidia GT 520 zum Einsatz.
Der Vergleich wird mit einer grafischen Python Anwendung durchgeführt. Hier kommen die Python Module Numpy, Scipy und Matplotlib zum Einsatz. Das Modul Matplotlib bietet die Möglichkeit Messdaten in Plots zu visualisieren. Bei Numpy handelt es sich um eine Bibliothek mit Funktionen zur numerischen Berechnung mathematischer Probleme, während Scipy mit Anwendungsbezogenen Berechnungen auf Numpy zugreift. Diese Module führen ihre Berechnungen auf der CPU aus. Der FFT Algorithmus, der seine Berechnungen auf der GPU ausführen soll, wird mit dem Modul pyFFT bereitgestellt. PyFFT bietet die Möglichkeit sowohl OpenCL, als auch CUDA kompatible Hardware anzusprechen. Hier wird pyFFT mit CUDA eingesetzt.
Die Algorithmen werden beide mit den gleich formatierten Audiodaten gespeist. Unmittelbar vor und nach der Ausführung des Algorithmus wird ein Zeitstempel, mithilfe der Betriebssystem eigenen time() Funktion erstellt und anschließend die Differenz festgehalten.
Die beiden FFT Algorithmen werden dann mit den gleichen FFT Größen ausgeführt. Für FFT Größen zwischen 256 und 32768 sollen Messungen durchgeführt und anschließend verglichen werden.
Als zweite Laufzeitanalyse Methode kommt die Software Visual Profiler zum Einsatz. Visual Profiler wird von NVidia bereitgestellt und bietet die Möglichkeit alle Aspekte für die Ausführung auf der Hardware zu analysieren.

2 CUDA by NVidia

Die Compute Unified Device Architecture, kurz CUDA, ist eine General-Purpose Parallel Computing Plattform (GPGPU) und Programmiermodel. In einem heterogenen System, bestehend aus einer CPU und einer GPU, können zeitintensive Berechnungen von der CPU auf die GPU ausgelagert werden. GPUs liegen bei nicht grafischen Anwendungen meist brach und können diese Aufgaben somit problemlos übernehmen. CUDA wird häufig in wissenschaftlichen Umfeldern eingesetz, in denen komlexe Simulationen erstellt werden müssen.

2.1 Programmiermodel

Das CUDA Programmiermodel ist für Hardware plattformabhängige Skalierbarkeit und Betriebssystemabhängigkeit designed worden. Demnach ist eine Applikation, die CUDA einsätzt, auf allen Hardware Plattformen ab einer gewissen Versionsnummer ausführbar und profitiert von Fortschritten der Hardware Technologie.

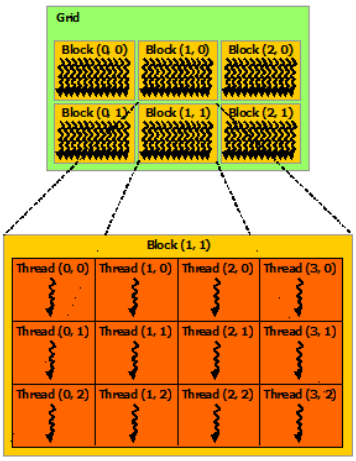


Figure 1: Grid of Thread Blocks [1]

Jede Plattform, von denen mehrere in einem System installiert sein können, wird als ein Grid betrachtet. Ein Grid besteht seinerseits aus Blöcken, die in ein, zwei oder drei Dimensionen möglich sind. Blöcke umfassen schließlich Threads, die ebenfalls bis zu drei dimensionen annehmen können. Dieses Design dient der besseren Verarbeitung von Vektoren, Matrizen und Volumen und wird in Figure 1 gezeigt.

2.2 Speicherhierarchie

Der Speicher einer Plattform ist hierarchisch aufgebaut. So hat jeder Thread Lese- und Schreibzugriff auf seinen privaten lokalen Speicher. Jeder Block hat einen geteilten Speicher, auf den alle seine Threads Lese- und Schreibzugriff haben. Darüber gibt es noch den globalen, einen konstanten und einen Textur Speicher, die ihre Daten auch über das Ausführen eines Kernels hinaus halten.

2.3 Laufzeit

Mit Kernel wird in CUDA der von einem Thread auszuführende Code genannt, der vom Programmierer erstellt wird. Die Berechnungen des Kernels werden in der oben beschriebenen Architektur parallel verbarbeitet. Diese Kernels werden vom Gerätetreiber Just-in-time in Binärcode übersetzt. Der Grund hierfür ist, dass bereits bestehende Applikationen auch von Neuerungen im Gerätetreiber profitieren können sollen. Der Binärcode wird wird jedoch gecached und bei weiteren Aufrufen dieses Codes nicht neu Kompiliert, sondern aus dem Cache gelesen.

3 Fast Fourier Transform

3.1 Diskrete Fourier Transformation:

Die Diskrete Fourier-Transformation bildet ein endliches zeitdiskretes Signal, zum Beispiel ein abgetastetes Audiosignal, auf ein periodisches, diskretes Spektrum ab.

$$X[m] = \sum_{n=0}^{N-1} x[n] \cdot e^{-j2\pi \frac{mn}{N}} \quad [2]$$

Nach einer Reihe Umformungen ergibt sich eine Linearkombination, ein rekursiven Ansatz der sich zur effizienteren Implementierung in Computersystemen eignet. Bei der Berechnung der Fourieranalysie mit hilfe des FFT Algorithmus, wird die Anzahl Multiplikationen durch das speichern von Zwischenergebnissen und somit auch der Rechenaufwand reduziert. Das FFT Verfahren wurde 1965 von James Cooley und John W. Tukey entwickelt und wird auch als Radix-2 Algorithmus bezeichnet. Neben dem Radix-2-Algorithmus existieren noch weiterere Algorithmen zur Berechnung einer FFT. Bei der konkreten Implementierung der FFT mithilfe des Radix-2 Algorithmus, wird die DFT in zwei Teile, geradzahlige $a[n]$ und alle ungradzahlige Abtastwerte $b[n]$, aufgesplittet. Weiterhin wird der komplexe Faktor $W_N = e^{-j\frac{2\pi}{N}}$, der auch als *twiddle factor* bezeichnet wird, abgespalten. Der *twiddle factor* ist nur noch von N, der Blockgröße der FFT abhängig.

$$X[m] = A[m] + W_N^m \cdot B[m] \quad [2]$$

Das folgende Diagramm (Figure 2) veranschaulicht den Rechenprozess und wird aufgrund seiner optischen Form, die mit einem Schmetterling vergleichbar ist, mit Butterfly Diagramm oder Algorithmus bezeichnet.
Für eine weitere parallelisierung eignet sich dieser Algorithmus besonders gut, da die DFT für gerad- un ungradzahlige Abtastwerte auf viele parallele Berechnungen aufgeteilt werden kann. Bei einer Blockgröße von 1024 liesse sich die FFT in 16 x 16 x 4 einzelne FFTs aufteilen und parallel berechnen.

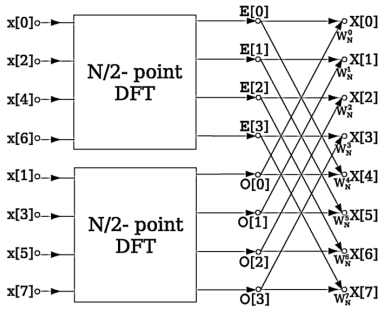


Figure 2: Butterfly Algorithmus [3]

4 pyFFT

Es gibt eine bereits fertiges Python Modul das den FFT Algorithmus mit CUDA bzw. OpenCL berechnet lässt. Es ist auf einem offiziellen Python Server verfügbar [4].
Dieses Modul nutzt seinerseits die Module pyCUDA und PyOpenCL, sowie die Bibliothek zur numerischen Berechnung, Numpy.

PyFFT besteht aus vier bzw. fünf Komponenten, die in Figure 3 dargestellt sind. In der Komponente `kernel.py` sind alle Klassen und Funktionen, die zur Erstellung des Kernels und des Grids erforderlich sind, enthalten. Die Klasse GlobalFFTKernel beschreibt hierbei die Parameter und Funktionen, die für die Grid Ausführung erforderlich sind, während die Klasse LocalFFTKernel die Parameter und Funktionen beschreibt, die von Threads zur Ausgeführt benötigt werden. Nachdem diese Kernel bei der Ausführung kompiliert wurden, werden bei nachfolgenden Ausführungen nur noch die Parameter geändert, Funktionen hierfür sind in der Komponente `plan.py` enthalten. Die Klasse FFT_Plan greift bei der Ausführung auf die Parameter der Klasse _FFTParams zurück und führt das Grid und die Kernel aus. Sollten die GPU Resourcen nicht für eine Berechnung mit einer Ausführung reichen, so führt die Klasse FFT_Plan das Grid und die Kernel entsprechend häufig aus.
Der tatsächliche FFT Algorithmus, Radix-2, wird in der Komponente `kernel_helpers.py` hinterlegt.

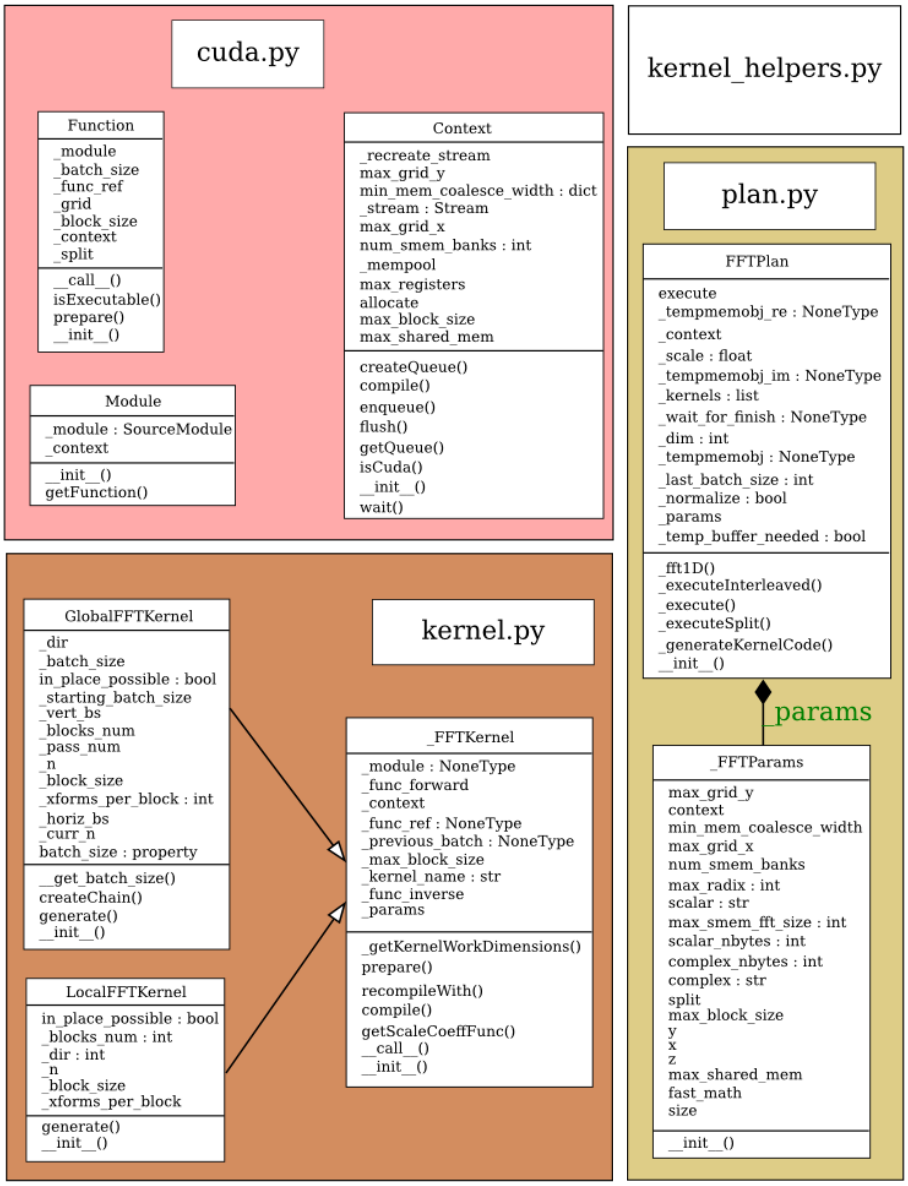


Figure 3: PyFFT Klassendiagramm

5 Python Audio Measurement Tool

Eine grafische Anwendung zur Visialisation der Daten wurde in Python mit Hilfe des Qt Frameworks entwickelt. Die Darstellung im oberen Plot wurde mit dem Python Modul Matplotlib erstellt und zeigt das Eingangsaudiosignal in Abhängigkeit von der Zeit. Bei dem Audiosignal handelt es sich um multifrequentes moduliertes Testsignal, dass mit der Zeit an Tonhöhe zunimmnt.

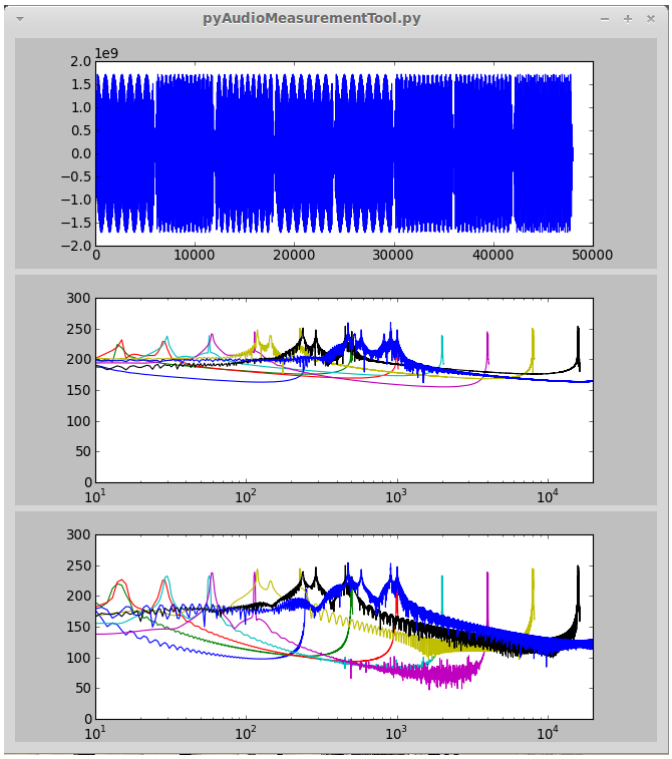


Figure 4: Python Audio Measurement Tool Qt GUI

Im mittleren Plot werden nun acht Spektren dieses Audiosignals angezeigt, welches mithilfe der Python Module Numpy und Scipy auf der CPU berechnet wurde. Jedes Spektrum zeigt hierbei eine andere FFT Blockgröße. Die Werte für die FFT Blockgröße sind 256, 512, 1024, 2048, 496, 8192, 16384 und 32768. Diese Werte werden genauso den FFT Analysen in den Spektren des unteren Plots zugrunde gelegt. Jedoch sind die Spektren im unteren Plot jene, die auf der GPU berechnet wurden.

5.1 Laufzeitanalyse

PyAudioMeasurementTool.py wurde 10 mal in Reihe ausgeführt und es wurden dabei folgende Zeiten für die Ausführung der FFT Algorithmen gemessen.

5.1.1 time() Funktion

CPU 256	100	103	103	102	103	103	119	104	111	103
CPU 512	162	118	115	135	113	113	111	116	113	121
CPU 1024	231	177	179	176	175	176	176	176	176	178
CPU 2048	399	336	299	327	300	300	298	301	312	299
CPU 4096	700	595	597	594	595	591	591	683	630	599
CPU 8192	1651	1548	1601	1645	1602	1595	1537	1541	1545	1674
CPU 16384	3436	3242	3276	3227	3276	3227	3211	3284	3249	3219
CPU 32768	6899	6921	7035	6948	6850	6661	6718	6861	6891	6831
GPU 256	1887	725	729	683	698	658	687	712	660	705
GPU 512	244	222	227	225	222	227	219	233	239	234
GPU 1024	219	224	219	219	219	222	228	335	227	220
GPU 2048	230	218	219	227	225	252	225	303	277	223
GPU 4096	272	250	252	256	252	252	252	305	255	254
GPU 8192	170	172	173	202	177	180	174	235	177	191
GPU 16384	297	303	336	328	311	314	314	350	322	321
GPU 32768	279	245	245	258	247	249	246	245	293	247

Figure 5: Laufzeitmessung mit time() Function

An dieser Messung lässt sich sehr deutlich die Just-in-time Kompilierung erkennen. Die erste Ausführung des Algorithmus auf der GPU, eine FFT der Blockgröße 256, dauerte 1887us. Alle weiteren FFTs der Blockgröße 256 brauchten im Durchschnitt ca. 695us. Auch hier ist auffällig, dass diese Berechnung deutlich länger dauert als bei allen folgenden. Der Grund hierfür is, dass für jede weitere Programmausfüuhrung der bereits kompilierte Kernel aus dem Cache geladen werden muss. Alle anderen Ausführungen des FFT Algorithmus liefen zwischen 170 und 336us.

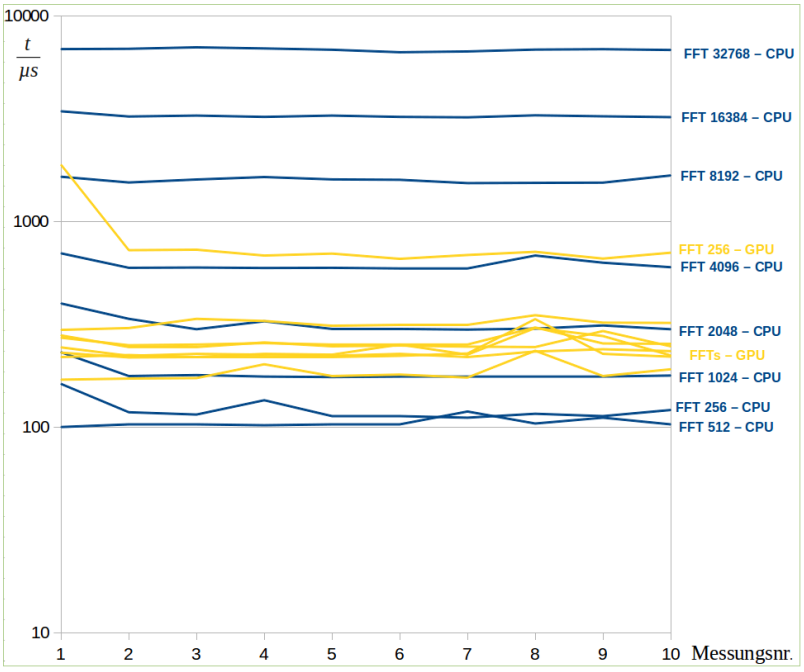


Figure 6: Statistische Auswertung der Laufzeiten

5.1.2 NVidia Visual Profiler

Der NVidia Visual Profiler bietet die Möglichkeit, die Performance des auf der GPU ausgeführten Kernels zu untersuchen. Hierfür wird das gesamte Programm 21 mal ausgeführt. Dabei werden unterschiedliche Parameter untersucht. Es werden zum Beispiel Warnungen über eine niedrige Auslastung wie in Figure 7 ausgegeben.

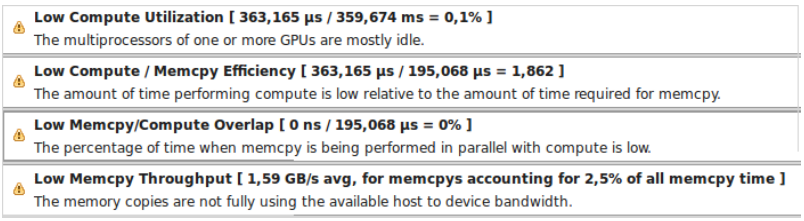


Figure 7: Laufzeit Warnungen

Der Visual Profiler basiert auf der weit verbreiteten Plattform Eclipse und bietet bekannte Ansichten und Tools. So gibt er zum Beispiel die Standard Ausgabe eines Programms in dem Reiter `Console` aus. Im Reiter `Details` sind alle Kopierbefehle zwischen CPU und GPU aufgelistet. Gezeigt werden die genaue Dauer jedes Kopierbefehls, die Menge der kopierten Daten und der Datendurchsatz. In Figure 8 links ist ein Screenshot dieses Reiters zu sehen, während diese Werte in Figure 8 rechts gemittelt für die ausgewählte GPU dargestellt werden.

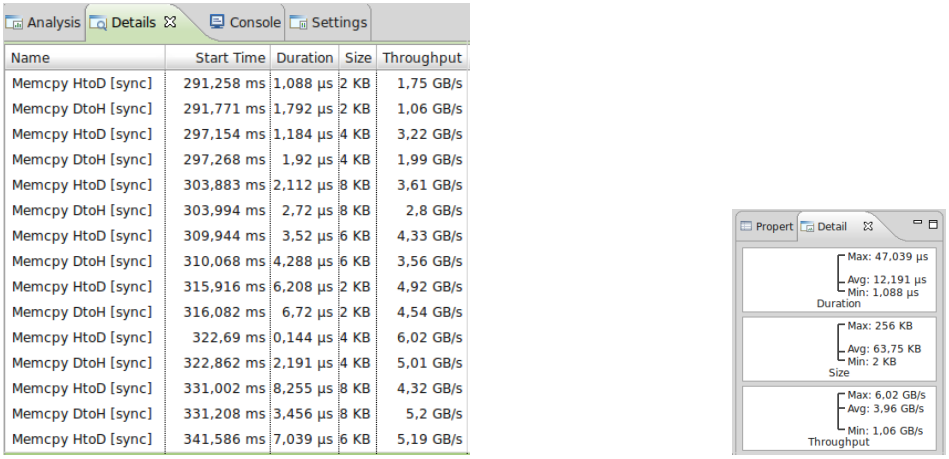


Figure 8: Details zu Memcpy Operationen und Programmlaufzeit

Im eigentlichen Arbeitsfenster wird eine Zeitleiste dargestellt, wie man in Figure 9 erkennen kann. Die darunterliegenden Zeilen zeigen die Dauer verschiedener Vorgänge. In der zweiten Zeile werden Aktivitäten des Gerätetreibers angezeigt. Dabei handelt es sich zum Beispiel um das erstellen des Gerätekontextes, um um das Kompilieren oder laden des Kernels. In der dritten Zeile wird der Overhead, der durch das Profiling des Programms entsteht dargestellt.
In der sechsten und siebten Zeile werden die Vorgänge MemCpy(HtoD) und MemCpy(DtoH) dargestellt. Hierbei handelt es sich um das Kopieren der Daten von der CPU auf die GPU

und zurück.
In der achten Zeile wird die Ausführungszeit des eigentlichen Kernels dargestellt.

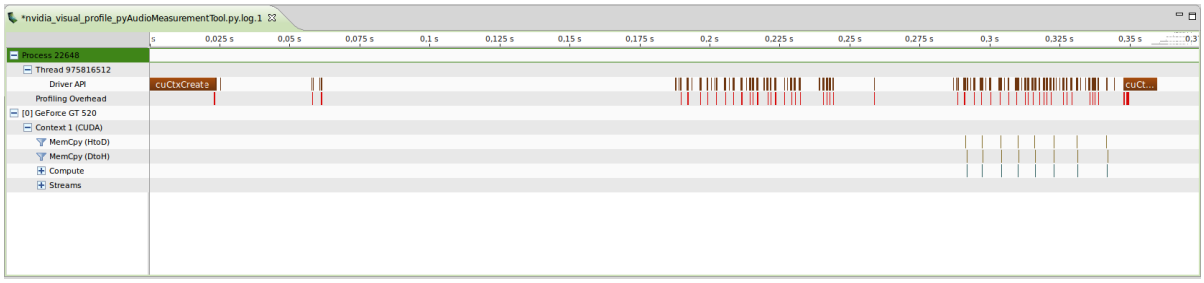


Figure 9: Laufzeitanalyse mit NVidia Visual Profiler

In Figure 10 sind die relevanten Bereiche aller FFT Blöckgrößen zusammengefasst.
In der zweiten Zeile der FFT 4096 lässt sich erkennen, dass der Kernel ein zweites mal geladen wird, also das Grid erneut ausgeführt wird und in der achten Zeile lässt sich dann die doppelte Ausführung beobachten. Bei einer Blockgröße von 32768 wird das Grid gar ein drittes mal geladen.

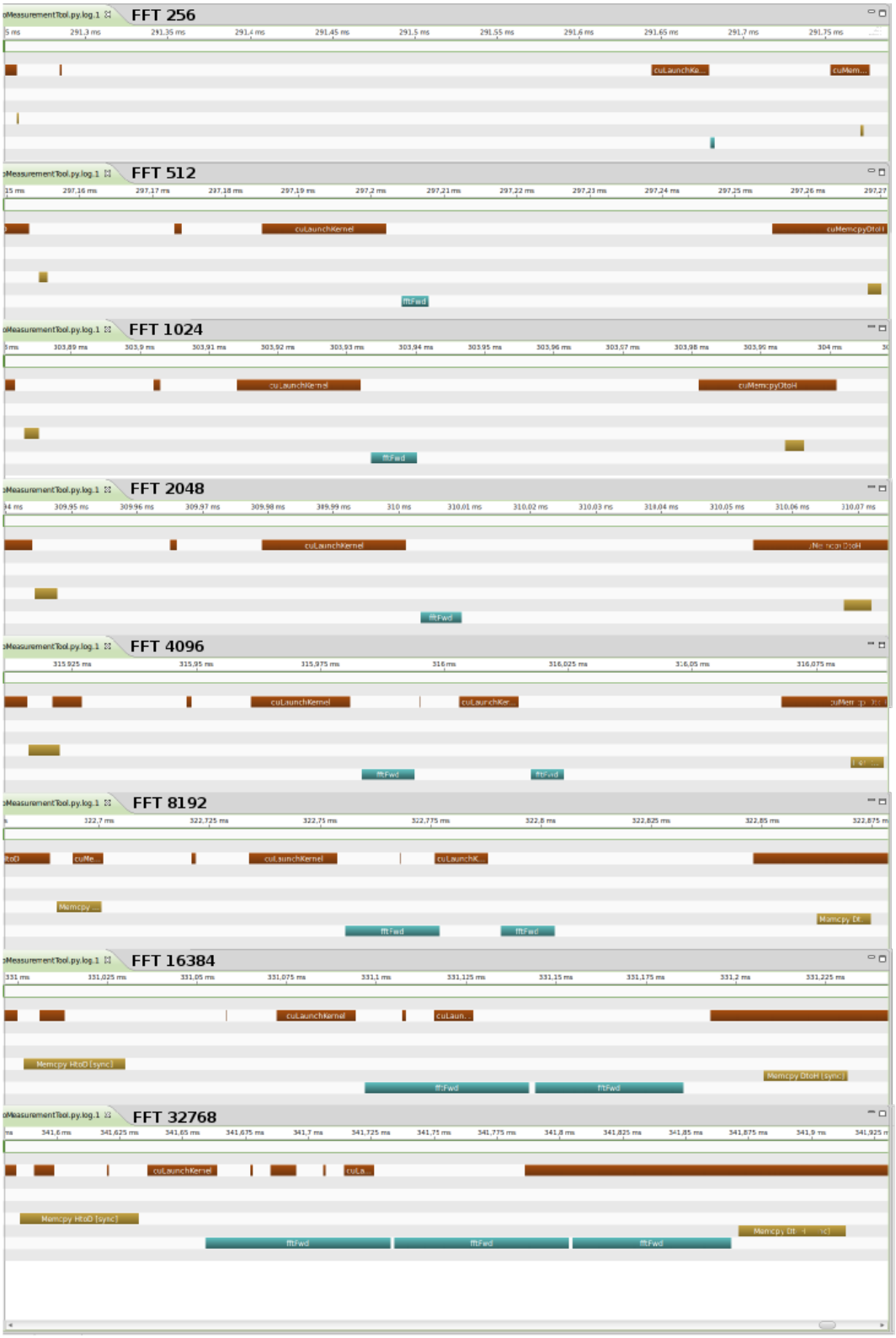


Figure 10: Laufzeiten der FFTs auf der GPU

6 Fazit

Mit Python und den eingesetzten Modulen lässt sich sehr schnell ein Prototyp entwickeln, der auf CUDA Hardware ausgeführt werden kann.

Das berechnen einer FFT mit CUDA ist dann effizient, wenn der zu berechnende Algorithmus häufig ausgeführt werden soll und größer als eine Blockgröße ¿ 2048 ist. Bei einer einzelnen Ausführung des Algorithmus ist die GPU Verarbeitung erst schneller, wenn die Blockgröße 8192 übersteigt.

References

[1] NVidia, *http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.htmlintroduction*, 15. Januar 2014.

[2] Martin Meyer, *Signalverarbeitung*, Vieweg 2006.

[3] Wikipedia, *http://en.wikipedia.org/wiki/File:DIT-FFT-butterfly.png*, 15. Januar 2014.

[4] PyFFT, *http://pythonhosted.org/pyfft/*, 15. Januar 2014.