

Quantum key distribution fast post-processing

Focus on fast Privacy Amplification on GPUs

Laborjournal

Nico Bosshard, Forschungsmitarbeit FS 2021
 Betreuung: Dr. Esther Hänggi, Roland Christen
 Hochschule Luzern, Informatik, 11. Juni 2021

[Back](#) Jobs in run #20210607.4

Jobs	
✓ Build_PrivacyAmplification_On_Self_Hosted_Agent	
Initialize job	
Checkout nicoboss/PrivacyAmplification@master to s	
Get ShortGitHash Script	
Create build folder for glslang	
Execute CMake for glslang	
Build glslang	
Build GLSL Shaders	
Build PrivacyAmplification	
Build MatrixSeedServerExample	
Build SendKeysExample	
Build ReceiveAmpOutExample	
Build LargeBlocksizeExample	
Copy examples to Release	
[Vulkan] CalculateCorrectionFloat Unit Test	
[Vulkan] ElementWiseProduct Unit Test	
[Vulkan] BinInt2float Unit Test	
[Vulkan] ToBinaryArray Unit Test	
[Cuda] CalculateCorrectionFloat Unit Test	
[Cuda] SetFirstElementToZero Unit Test	
[Cuda] ElementWiseProduct Unit Test	
[Cuda] BinInt2float Unit Test	
[Cuda] ToBinaryArray Unit Test	
Large Blocksize Test Cuda	
Large Blocksize Test Vulkan	
Generate Vulkan Stats	
Generate Cuda Stats	
Upload Stats	
Publish Pipeline Artifact	
Post-job: Checkout nicoboss/PrivacyAmplification@master to s	
Finalize Job	

VL

Function	Cuda	Vulkan	Difference	Difference %
	Vulkan - Cuda	(Vu/Cu)*100-100		
wait_for_input_buffer [ms]	0.006	0.006	0	0
cleaned_memory [ms]	0.003	0	-0.003	-100 %
set_count_to_zero [ms]	0.007	0.172	0.165	2'357 %
binIntffloat_seed [ms]	2.782	3.043	0.261	9 %
binIntffloat_key [ms]	1.762	1.991	0.229	13 %
calculateCorrectionFloat [ms]	0.068	0.226	0.158	232 %
fft_key [ms]	7.225	6.283	-0.942	-13 %
fft_seed [ms]	7.425	6.638	-0.787	-11 %
setFirstElementToZero [ms]	0.094	0	-0.094	-100 %
elementWiseProduct [ms]	2.397	2.467	0.07	3 %
ifft [ms]	7.352	6.65	-0.702	-10 %
wait_for_output_buffer [ms]	0.005	0.003	-0.002	-40 %
toBinaryArray [ms]	1.725	1.612	-0.113	-7 %
Total [ms]	30.868	29.094	-1.774	-6 %
Speed [MBit/s]	4348.055	4613.213	265.158	6 %

Inhalt

I Woche 1 + 2	1
1. NVIDIA GPU Computing Toolkit CUDA v11.2.....	1
2. Updated libzmq to latest commit	1
3. Portierung von Cuda zu ROCm.....	1
4. Fixed atomic not compiling on Linux.....	2
5. Applied preprocessor definitions to PrivacyAmplification.cu and cleaned up the resulting code as HIPIFY has issues with them.....	2
6. Made PrivacyAmplification.cu compatible with HIPIFY	2
7. Diff PrivacyAmplification.cu with PrivacyAmplification.cu.hip.....	2
8. Added my hipify-clang build	3
9. Created a linux Makefile for the PrivacyAmplification HIP version	3
10. Edited PrivacyAmplification.h to make it HIPIFY compatible	3
11. Better PrivacyAmplification to HIP conversion	3
12. Diff PrivacyAmplification CUDA with PrivacyAmplification HIP	3
II Woche 3 + 4	4
13. Fehlerbehandlung HIP-kompatibel gemacht	4
14. Erster erfolgreicher HIP-Build.....	4
15. Von rocFFT zu hipFFT	5
16. Alles nochmals auf CentOS 8.....	6
17. Vuda testen	8
18. Vuda Fehler beheben.....	9
19. vkFFT für Cuda und Vulkan kompilieren	11
20. vkFFT updaten	11
21. vkFFT für Vulkan kompilieren.....	12
22. vkFFT mit Vuda kombinieren.....	12
23. GLSL und SPIR-V	12
III Woche 5 + 6	14
24. Subblock branch aufgeräumt und libzmq geupdatet	14
25. vkFFT geupdatet	14
26. Debugging im neues Visual Studio Projekt ermöglichen	14
27. Script um GLSL-Shaders zu bilden	15
28. Test unitTestCalculateCorrectionFloat verbessert	15

29.	PrivacyAmplification-Hilfsfunktionen in Vulkan.....	16
30.	unitTestCalculateCorrectionFloat erfolgreich auf Vulkan	16
31.	unitTestSetFirstElementToZero erfolgreich auf Vulkan	17
32.	unitTestElementWiseProduct erfolgreich auf Vulkan	18
33.	Fehlende Bibliotheken und Konfigurationsdateilogik	19
34.	unitTestBinInt2float erfolgreich auf Vulkan.....	20
35.	unitTestToBinaryArray erfolgreich auf Vulkan.....	22
36.	Hilfsfunktionen von vkFFT separiert.....	23
37.	Erster erfolgreicher vkFFT Aufruf	24
38.	Größenbeschränkung in vkFFT	24
39.	Alle Tests auf Vulkan erfolgreich!	27
IV	Woche 7 + 8.....	28
40.	Alle Tests auf AMD Radeon RX 5700 XT erfolgreich!.....	28
41.	Vulkan-spezifische Bibliotheken zu PrivacyAmplification portiert.....	29
42.	Vulkan und Cuda Projekt Kerncode zusammenführen.....	29
43.	cuFFT zurück vom C2C Test zu R2C/C2R	29
44.	VkFFT unterstützt neu R2C/C2R mit grossen Blöcken!.....	30
45.	Script zur sauberen Darstellung der Git History hinzugefügt	32
46.	Lizenzen aktualisiert.....	32
47.	Verlinken von glslang mit PrivacyAmplification VS - Projekt	32
48.	Angefangen Vuda and vkFFT miteinander kompatibel zu machen.....	33
49.	vkFFT mit Vuda Speichermanagement kompatibel machen	34
50.	VK_DEVICE_LOST testen	34
51.	Nicht zwingend notwendiger VkFFT-Code entfernt	35
52.	Meine verbesserte Vuda-Version zu PrivacyAmplification portiert.....	35
53.	Bereinigter vkFFT_helper nach PrivacyAmplification portiert	35
54.	GLSL-Shaders nach PrivacyAmplification portiert	36
55.	vkFFT-Steuercode nach PrivacyAmplification portiert	36
56.	PrivacyAmplification kompiliert mit Vulkan!	36
57.	Fehler mit VK_DEVICE_LOST behoben	36
58.	Alle Vuda-Dateien zum Visual Studio-Projekt hinzugefügt.....	36
59.	vkFFT in den PrivacyAmplification Algorithmus integriert	37
60.	Assert, RHash und VIDEO_SCHEDULER_INTERNAL_ERROR	37
61.	VIDEO_SCHEDULER_INTERNAL_ERROR BoD behoben.....	38

62.	PrivacyAmplification VS-Projekt kompilierbar für Cuda und Vulkan	39
63.	Nutzloses GPU-Pipelining entfernt	39
64.	FFT-Ergebnis-Debugging	39
65.	cudaMalloc und cudaCalloc repariert.....	40
66.	Ergebnis von vkFFt und cuFFT sind unterschiedlich	40
67.	0-Hz Test mit kleinem Zahlenbeispiel	40
68.	0-Hz Test mit realen, grossen Zahlen.....	40
69.	In-Place VkFFT.....	40
70.	Korrektes Vulkan IFFT-Ergebnis!.....	41
71.	PA läuft mit Cuda und mit Vulkan	41
72.	Gleiches Ergebnis in Cuda und Vulkan.....	41
73.	Gleiches Ergebnis in Cuda und Vulkan, alles auf GPU	41
74.	cudaMemset repariert	42
75.	Korrektes Ergebnis auch mit mehreren Blöcken	42
76.	PrivacyAmplification-Algorithmus funktioniert auf Vulkan!.....	42
77.	Erste Planung zum Aufteilen in Blöcke!	43
V Woche 9 + 10		44
78.	VkFFT Zero Padding	44
79.	VkFFT konfiguriert.....	44
80.	Benchmarking Tool	44
81.	SetFirstElementToZero durch VkFFT Zero Padding	46
82.	SetFirstElementToZero durch VkFFT Zero Padding	47
83.	Vuda: cudaMemset implementiert	47
84.	Vuda: cudaMemset optimiert	47
85.	set_count_one_to_zero vereint.....	47
86.	GLSL: Round anstelle von RoundEven	48
87.	VkFFT kann nun 64 bit.....	49
88.	ElementWiseProduct optimiert.....	49
89.	Cuda 11.3.....	49
90.	Fliesskommazahlen halber Präzision	49
91.	Test mit grossen Blockgrössen	51
92.	AMD-Probleme behoben	52
93.	Matrix in Blöcke aufteilen	53
VI Woche 11 + 12		62

94.	VkFFT aktualisiert.....	62
95.	Aufteilen des reciveData-Thread	62
96.	Keine ZeroMQ-Statusmeldungen	62
97.	Send_alice und send_bob zu sendData	62
98.	Kommunikationsfehler behoben.....	62
99.	Eingabe für mehrfache Verwendung des Seeds.....	63
100.	LargeBlockSizeExample hinzugefügt.....	63
101.	LargeBlockSizeExample mit Multithreading.....	63
102.	ReceiveAmpOutExample integriert	63
103.	ReceiveAmpOut	63
104.	ToBinaryArrayNoXOR.....	63
105.	Do_xor_key_rest.....	64
106.	Codeverbesserungen durch templates.....	64
107.	VkFFT für Mehrfach-Upload aktualisiert.....	64
108.	Problem bei Cuda behoben.....	65
109.	Speedtest.....	66
110.	PrivacyAmplification_Matrix_Splitten	66
111.	Keine horizontale und vertikale Aufteilung mehr	66
112.	Block-Split-Algorithmus vereinfacht.....	66
113.	Zweiter Block automatisiert	66
114.	Erster Block automatisiert.....	66
115.	Dritter Block automatisiert.....	67
116.	Alle vier Blöcke automatisiert!	67
117.	Blocktracker.....	67
118.	Zählen des aktuellen Schlüssels.....	67
119.	Die Seed-Berechnung funktioniert	67
120.	CurrentRowNr.....	67
121.	Amp_out automatisiert.....	67
122.	Horizontal XOR automatisiert	68
123.	Amplified_key automatisiert.....	68
124.	Test mit sample_size 64	68
125.	Gewünschte Länge.....	68
126.	Test mit sample_size 1024 und 3840 Blöcken	68
127.	Test mit sample_size 1024 und 16x16 Unterblöcke.....	68

128.	Blockgrößenkontrolle.....	68
129.	2^{14} -Testmatrix mit 2^{10} Blöcken	69
130.	2^{14} -Testmatrix mit 2^{11} Blöcken	69
131.	Endianness bei Seed und Key	69
132.	Gleiches Ergebnis bei Matrix mit Blockaufteilung.....	69
133.	Variablen anstelle von Arrays.....	69
134.	Fixed Pipeline Tests.....	70
135.	2^{14} Pipeline-Tests	70
136.	Modifiziertes Toeplitz-Hashing mit sample size of 16383	70
137.	Modifiziertes Toeplitz-Hashing mit sample size of 16384	70
138.	Simulierter Python-Algorithmus von Cuda funktioniert.....	70
139.	Sichtbare Eingabelänge	70
140.	Gleiches Ergebnis wie mit Python!	71
141.	First Split Block auf GPU funktioniert	71
142.	Block-Splitting in C ++	71
143.	Multiple Split Block Algorithmus auf GPU	71
144.	Chunk_size Kompatibilität.....	71
145.	Korrekter permuted_key_raw.....	72
146.	BinOut	72
147.	XOR mit key_rest	72
148.	Blockreihenfolge	72
149.	ZeroMQ	72
150.	Entfernung der Komprimierung	72
151.	Debugging zur Überprüfung von C ++.....	72
152.	GitLab-URL korrigiert	73
153.	Chunk-Splitting-Algorithmus funktioniert!	73
	VII Woche 13 - 16.....	74
154.	Aktualisierung des Zugriffstoken für GitLab.....	74
155.	Dockerfile mit neuester Cuda-Version und Accessss Token	74
156.	Downloadort des Dockers behoben	74
157.	Keine VkFFT beim Kompilieren mit Cuda.....	74
158.	Linux-Kompilierungsfehler behoben	74
159.	Atomics ist Linux kompatibel	75
160.	Zeitmessung neu mit high_resolution_clock	75

161.	Kommunikationsprotokoll aktualisiert	75
162.	SendKeysExample	75
163.	PrivacyAmplification-Algorithmus Test.....	75
164.	Kommunikationsprotokoll der LargeBlocksizeExample.....	75
165.	ToBinaryArray-Ergebnispipelinetest geflickt.....	76
166.	VkFFT aktualisiert.....	76
167.	VkFFT-Test mit Grösse von 2^{14}	76
168.	Problembehebung mit Blockgrösse von 2^{14}	76
169.	Ursache für das 2^{14} - Problem gefunden	76
170.	Standard wieder auf 2^{27} -Pipeline-Test gesetzt.....	76
171.	Vorbereitung Linux Docker-Test.....	77
172.	Keine Wiederverwendung des Toeplitz Matrix Seeds	77
173.	Azure Pipeline: GLSL-shaders	77
174.	Azure Pipeline: Code-Eintrückungen.....	77
175.	Azure Pipeline: cmake als einzelner Task.....	77
176.	Azure Pipeline: cmake in zwei Tasks aufgeteilt.....	77
177.	UnitTestToBinaryArray repariert	78
178.	Memoryproblem bei Vuda	78
179.	Fehler beim Speichermanagement behoben.....	78
180.	UnitTestBinInt2float repariert.....	78
181.	Azure Pipeline: Komplilieren der GLSL-shaders	78
182.	Azure Pipeline: Unit Tests for Cuda	78
183.	Vulkan: Limite für unitTestBinInt2float	79
184.	Speedtest für Vulkan und Cuda	79
185.	Terminierung des Programms verbessert.....	79
186.	Problem mit zu kurzen Unit-Tests behoben	79
187.	Speedtest Zeitmessung optimiert	79
188.	Speedtest mit 10 Runden pro Test	79
189.	Azure-Artefakte geupdatet	80
190.	Speedtest mit Aufwärmung	80
191.	Negativer Speedtest-Index.....	80
192.	Beispiele für Azure-Pipelines.....	80
193.	Array von pointer anstelle Zeigerarithmetik.....	80
194.	Tests für LargeBlocksizeExample.....	81

195.	Problembehebung beim LargeBlocksizeExample-Test.....	81
196.	Standard für gpu_device_id_to_use.....	81
197.	Taskkill.....	81
198.	Problembehebung in stderr	81
199.	Speedtest mit sample_size von 2^{27}	81
200.	Fehlerbehebung beim Large Blocksize Test	82
201.	LargeBlocksizeExample Debug-Ausgabe.....	82
202.	Reparatur des Vulkan ToBinaryArray	82
203.	Fehlerbehebung.....	82
204.	VkFFT 2^{14} -Test.....	82
205.	VkFFT 2^{14} -Test mit erstem Block.....	83
206.	VkFFT 2^{14} - zero padding Fehler behoben.....	83
207.	Pipelinetests Präzisionskorrektur	83
208.	Neuplanung bei Komprimierungsfaktoränderung bei Vulkan	83
209.	Cuda Speicherzuweisung.....	84
210.	Vereinheitlichung der Komprimierungsfaktorberechnung.....	84
211.	Seed caching	84
212.	Static Seed	84
213.	Seed caching bei Cuda	84
214.	Verify_ampout.....	84
215.	Azure-pipeline: LargeBlocksizeTest	85
216.	Pipeline-Tests wieder mit 2^{27}	85
217.	Arbeitsanzeige	85
218.	VkFFT_2_pow_14_vulkan_testing zusammengefügt.....	85
219.	Speicherberechnung Vulkan	85
220.	Anpassungen beim Config.....	85
221.	AzureCLI und azure-devops Installation	86
222.	VkFFT aktualisiert für bessere Fehlerbehandlung	86
223.	Kompatibilitätsüberprüfung und Grafikkartenanzeigen.....	86
224.	Speedtest repariert.....	86
225.	PLAN_CUFFT template entfernt	86
226.	Linux Makefile für LargeBlocksizeExample	87
227.	LargeBlocksizeExample-Warnungen behoben.....	87
228.	PrivacyAmplification Vulkan- Algorithmus auf Linux!	87

229.	Ausführbare Dateien als ausführbar markiert	87
230.	Speedtests auf Linux	87
231.	Run.sh führt Cuda aus.....	87
232.	Statischer Speedtest	88
233.	CudaSetDevice erstellt.....	88
234.	Vuda-Memorymanagement überarbeitet	88
	Zusammenfassung der Forschungsmitarbeit.....	89
	Grafikkartenunabhängigkeit - Vulkan:	89
	Grosse Blöcke	90
	«Open source»-Umgebung mit Testinfrastruktur	90
	Benchmarking-Toolkits	93
	Schlusswort	95
	Abbildungsverzeichnis.....	96

I Woche 1 + 2

1. NVIDIA GPU Computing Toolkit CUDA v11.2

Ich habe das Privacy Amplification Projekt auf die neuste Version des NVIDIA GPU Computing Toolkit portiert. Dies bedeutete auch dass ich das die gesamte NVidia Entwicklungsumgebung auf all meinen Geräte, Windows und Linux, auch auf diese Version updaten musste.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/f1afac240de796476e72b29be230bc8456ae7e26>

2. Updated libzmq to latest commit

Ich habe unser libzmq Repository mit dem neusten offiziellen libzmq Repository geupdatet. Davon habe ich einen neuen build erstellt, welcher ab jetzt in meinem Projekt verwendet wird.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/6895803986bedba6c0b27394b42444270b870486>

3. Portierung von Cuda zu ROCm

Bis jetzt läuft das Programm nur auf NVIDIA-Grafikkarten, da es in CUDA programmiert wurde.

Das Ziel ist es das Programm in HIP mit Hilfe des ROCm-Frameworks umzuschreiben. Damit das Programm auf allen Grafikkarten, egal von welchem Hersteller, läuft. Das ROCm ist eine Opensource Bibliothek, während CUDA closeSource ist. Dies ist vor allem für AMD-Grafikkarten interessant. Mit ROCm wird CuFFT durch rocFFT ersetzt, welches auch opensource ist.

Als Konvertierungsunterstützung habe ich HIPIFY verwendet. Um HIPIFY verwenden zu können musste ich dies zuerst selber kompilieren. Dies war alles andere als einfach, da für die neuste Cuda-Version LLVM 11.1 oder neuer verwendet werden musste, für welche kein Packet für Ubuntu 20.04 vorhanden ist. Somit musste ich dies auch selbst kompilieren. Generell war die Installation von ROCm sehr schwierig, da der mit Ubuntu 20.04

vorinstallierte Kernel inkompatibel mit ROCm ist. Um das Problem zu beheben musste ich 5.6-oem installieren und alle neueren Kernel deinstallieren.

4. Fixed atomic not compiling on Linux

Bei der Vorbereitung von ROCm ist mir aufgefallen, dass der subblock branch des Projekts auf Linux nicht kompiliert. Der Fehler lag an dem initialisieren von atomaren Variablen.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/d75a923f3f48e383d115e054ae4fb47e017b658c>

5. Applied preprocessor definitions to PrivacyAmplification.cu and cleaned up the resulting code as HIPIFY has issues with them

HIPIFY hatte grosse Probleme mit den Preprozessorenmakros. Eine Idee, dieses Problem zu beheben war den Preprocessor über meinen Code laufen zu lassen und den erhaltenen Output zu speichern, aufzuräumen und als Input für HIPIFY zu verwenden. Dies behob viele Fehler, führte aber zu unschönem Code und es blieben noch einige Fehler ungelöst. Deswegen suchte ich eine andere Lösung.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/44116291ebf718fe3035188fa8c199259d05243b>

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/3744fc2b6e738b41fed593cd61f498d3d1adda63>

6. Made PrivacyAmplification.cu compatible with HIPIFY

Manuell behob ich nun alle HIPIFY-Fehler in PrivacyAmplification.cu. Danach lief HIPIFY über meinen Code erfolgreich.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/cd533a4e5ebdc4d9746a74879caebc949f324a36>

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/5084a41baae0cff701ef6a7d2f378ff87cd0def8>

7. Diff PrivacyAmplification.cu with PrivacyAmplification.cu.hip

Ich habe den Cuda Code mit dem generierten ROCm Code verglichen um zu schauen, was HIPIFY an dem Code verändert hat.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/6aa635f4643179ccc38b9d4db50c949d62d71ab2>

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/d40b38998de4437e345a80115b49b4673e58e456>

8. Added my hipify-clang build

Da es recht kompliziert war HIPIFY zu kompilieren, habe ich mich entschieden die so erhaltenen ausführbare Datei dem GitRepository hinzu zu fügen.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/b791f100555616221ecd65a624be897c615d0292>

9. Created a linux Makefile for the PrivacyAmplification HIP version

Das Cuda Linux Makefile ist logischerweise nicht mit HIP kompatibel. Somit musste für ROCm ein eigenes Makefile erstellt werden.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/d23d772e5c5db62c35ed9017da0f59c369747b3b>

10. Edited PrivacyAmplification.h to make it HIPIFY compatible

Beim Komplizieren ist mir aufgefallen, dass die Headerdatei PrivacyAmplification.h noch nicht mit HIPIFY in ROCm kompatiblen Code konvertiert wurde. Ich editierte die Datei um sie mit HIPIFY kompatibel zu machen.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/d9dcf118ee006fc1dfa199da99d79e71241371d2>

11. Better PrivacyAmplification to HIP conversion

Bessere Konvertierung der PrivacyAmplification nach ROCm, diesmal auch mit konvertiertem Header. Dazu wurde der Header vor der Konvertierung in PrivacyAmplification.cu zusammengeführt und nach der Konvertierung erneut aufgeteilt.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/562a2a8fc3b476affe8236da7ccff769c49585a2>

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/9c0497fe32b440c59f92e7ff343d85ec5dd9da55>

12. Diff PrivacyAmplification CUDA with PrivacyAmplification HIP

Ich habe den Cuda Code mit dem generierten ROCm Code verglichen um zu schauen, was HIPIFY an dem Code verändert hat.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/2dce020973cb4e1fcab7c558f83f88b5c261ff87>

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/914e3446e5622f809e6d68c1857e74ff6cfadd3d>

II Woche 3 + 4

13. Fehlerbehandlung HIP-kompatibel gemacht

Die Fehlerbehandlung wurde neu geschrieben, um sie HIP-kompatibel zu machen. Dies da HIP sowie rocFFT andere error codes als CUDA und cuFFT haben. Fehlerbehandlung ist immer etwas sehr Bibliothekspezifisches.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/2794a29a00caf17566b762d6a8b59b738625f3c3>

14. Erster erfolgreicher HIP-Build

Mein PrivacyAmplifion Programm auf ROCm kompilieren zu können war ein grosses Ziel. Nach dem Beheben der hunderten von Kompilierfehlern war dies nun möglich. Die meisten HIPIFY Fehler passierten in der cuMemcpyToSymbol zu hipMemcpyToSymbol, da ich dort undokumentierte CUDA Funktionen verwendet habe. Auch alle min() und max()Befehle gingen nicht mehr. Ich ersetze diese durch ein min_template und max_template. Auch das Makefile musste natürlich recht angepasst werden. Ich bin jedoch überglücklich, dass es schlussendlich geklappt hat.

```

nico@castlebuntu:~/Documents/QKD/privacyamplification/PrivacyAmplification$ make
/opt/rocm/bin/hipcc -O2 -g -std=c++17 -c PrivacyAmplification_ hip.cpp -o PrivacyAmplification_ hip.o
PrivacyAmplification_ hip.cpp:105:5: warning: ignoring return value of function declared with 'nodiscard'
attribute [-Wunused-result]
    hipMemcpy(x, rdata.data(), real_bytes, hipMemcpyHostToDevice);
PrivacyAmplification_ hip.cpp:159:5: warning: ignoring return value of function declared with 'nodiscard'
attribute [-Wunused-result]
    hipMemcpy(cdata.data(), x, complex_bytes, hipMemcpyDeviceToHost);
PrivacyAmplification_ hip.cpp:175:5: warning: ignoring return value of function declared with 'nodiscard'
attribute [-Wunused-result]
    hipFree(x);
3 warnings generated when compiling for gfx803.
PrivacyAmplification_ hip.cpp:105:5: warning: ignoring return value of function declared with 'nodiscard'
attribute [-Wunused-result]
    hipMemcpy(x, rdata.data(), real_bytes, hipMemcpyHostToDevice);
PrivacyAmplification_ hip.cpp:159:5: warning: ignoring return value of function declared with 'nodiscard'
attribute [-Wunused-result]
    hipMemcpy(cdata.data(), x, complex_bytes, hipMemcpyDeviceToHost);
PrivacyAmplification_ hip.cpp:175:5: warning: ignoring return value of function declared with 'nodiscard'
attribute [-Wunused-result]
    hipFree(x);
3 warnings generated when compiling for host.
/opt/rocm/bin/hipcc -O2 -g -std=c++17 -c yaml/Yaml.cpp -o yaml/Yaml.o
/opt/rocm/bin/hipcc -O2 -g -c sha3/sha3.c -o sha3/sha3.o
/opt/rocm/bin/hipcc -O2 -g -std=c++17 PrivacyAmplification_ hip.o yaml/Yaml.o sha3/sha3.o -lm -o PrivacyAmplification_ hip -lzmq -L/opt/rocm/lib -lrocfft -L/home/nico/Documents/QKD/hipFFT/build/library -lhipfft
nico@castlebuntu:~/Documents/QKD/privacyamplification/PrivacyAmplification$ █

```

Abbildung 1: PA erfolgreich auf ROCm kompiliert.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/688eccf0a783ff666b5a528f923c8af1be41ba3d>

15. Von rocFFT zu hipFFT

Wir wollen rocFFT nur für AMD GPUs benutzen da es schlechter als cuFFT ist. Durch den Wechsel von rocFFT zu hipFFT, einer Marshalling-Bibliothek, die entweder rocFFT oder cuFFT als Backend verwendet, kann genau dies realisiert werden.

Das Builden von hipFFT war alles andere als einfach da sehr viele Abhängigkeiten benötigt werden, wie unter anderem CUDA, cuFFT, rocM, rocFFT, libboost-dev, libboost-program-options1.71-dev, libgtest-dev, libfftw3-dev, amdhip64. Danach kann wie folgt mit cmake ein Linux Makefile erstellt werden: cmake -DCMAKE_CXX_COMPILER=g++ -DCMAKE_BUILD_TYPE=Release -DBUILD_CLIENTS_TESTS=ON -DBUILD_CLIENTS_SAMPLES=ON -DBUILD_WITH_LIB=CUDA -L ..

```

nico@castlebuntu:~/Documents/QKD/hipFFT/build$ make
[ 3%] Building CXX object library/CMakeFiles/hipfft.dir/src/hcc_detail/hipfft.cpp.o
[ 7%] Linking CXX shared library libhipfft.so
[ 7%] Built target hipfft
[ 11%] Building CXX object clients/samples/CMakeFiles/hipfft_1d_z2z.dir/hipfft_1d_z2z.cpp.o
[ 14%] Linking CXX executable ../staging/hipfft_1d_z2z
[ 14%] Built target hipfft_1d_z2z
[ 18%] Building CXX object clients/samples/CMakeFiles/hipfft_3d_z2z.dir/hipfft_3d_z2z.cpp.o
[ 22%] Linking CXX executable ../staging/hipfft_3d_z2z
[ 22%] Built target hipfft_3d_z2z
[ 25%] Building CXX object clients/samples/CMakeFiles/hipfft_1d_d2z.dir/hipfft_1d_d2z.cpp.o
[ 29%] Linking CXX executable ../staging/hipfft_1d_d2z
[ 29%] Built target hipfft_1d_d2z
[ 33%] Building CXX object clients/samples/CMakeFiles/hipfft_2d_z2z.dir/hipfft_2d_z2z.cpp.o
[ 37%] Linking CXX executable ../staging/hipfft_2d_z2z
[ 37%] Built target hipfft_2d_z2z
[ 40%] Building CXX object clients/samples/CMakeFiles/hipfft_2d_d2z.dir/hipfft_2d_d2z.cpp.o
[ 44%] Linking CXX executable ../staging/hipfft_2d_d2z
[ 44%] Built target hipfft_2d_d2z
[ 48%] Building CXX object clients/samples/CMakeFiles/hipfft_3d_d2z.dir/hipfft_3d_d2z.cpp.o
[ 51%] Linking CXX executable ../staging/hipfft_3d_d2z
[ 51%] Built target hipfft_3d_d2z
[ 55%] Building CXX object clients/samples/CMakeFiles/hipfft_planmany_2d_z2z.dir/hipfft_planmany_2d_z2z.cpp.o
[ 59%] Linking CXX executable ../staging/hipfft_planmany_2d_z2z
[ 59%] Built target hipfft_planmany_2d_z2z
[ 62%] Building CXX object clients/samples/CMakeFiles/hipfft_planmany_2d_r2c.dir/hipfft_planmany_2d_r2c.cpp.o
[ 66%] Linking CXX executable ../staging/hipfft_planmany_2d_r2c
[ 66%] Built target hipfft_planmany_2d_r2c
[ 70%] Building CXX object clients/samples/CMakeFiles/hipfft_setworkarea.dir/hipfft_setworkarea.cpp.o
[ 74%] Linking CXX executable ../staging/hipfft_setworkarea
[ 74%] Built target hipfft_setworkarea
[ 77%] Building CXX object clients/tests/CMakeFiles/hipfft-test.dir/gtest_main.cpp.o
[ 81%] Building CXX object clients/tests/CMakeFiles/hipfft-test.dir/simple_test.cpp.o
[ 85%] Building CXX object clients/tests/CMakeFiles/hipfft-test.dir/accuracy_test.cpp.o
[ 88%] Building CXX object clients/tests/CMakeFiles/hipfft-test.dir/accuracy_test_1D.cpp.o
[ 92%] Building CXX object clients/tests/CMakeFiles/hipfft-test.dir/accuracy_test_2D.cpp.o
[ 96%] Building CXX object clients/tests/CMakeFiles/hipfft-test.dir/accuracy_test_3D.cpp.o
[100%] Linking CXX executable ../staging/hipfft-test
[100%] Built target hipfft-test
nico@castlebuntu:~/Documents/QKD/hipFFT/build$ 

```

Abbildung 2: hipFFT erfolgreich kompiliert.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/cb9263163151d92fcca3a965649c28c7bf8bbbbe>

16. Alles nochmals auf CentOS 8

Mein kompiliertes PrivacyAmplification Programm crashed andauernd auf Ubuntu mit der AMD RX 5700 XT GPU. Nach binärsuchartigem Crash ermitteln stellte sich heraus, dass das Planen von hipFFT den Crash verursacht. Dies war sehr merkwürdig da hipFFT von ROCm selbst programmiert wurde und somit definitiv ROCm kompatibel sein müsste. Da erinnerte ich mich, dass ich Ubuntu auf Kernel 5.6 OEM downgradien musste da ROCm inkompatibel mit dem neusten Linux Kernel ist. Das Problem könnte sein, dass ich dort eventuell etwas falsch gemacht habe oder der Kernel 5.6 nicht richtig unterstützt wird. Ich entschied alles nochmals auf CentOS 8, welches offiziell von ROCm unterstützt wird und auf Kernel 4.18 läuft, zu installieren. All die Grafiktreiber und Abhängigkeiten wieder zu installieren ging wesentlich länger als gedacht. Auch hatte ich bei AnyDesk, meinem Remote Desktop Tool,

das Problem, dass ich kein Passwort für unbeaufsichtigten Zugriff setzen konnte, was ich durch Wechsel von Wayland auf x11 beheben konnte.

Es stellte sich jedoch leider heraus, dass auch auf CentOS meinen für ROCm kompilierten PrivacyAmplification Algorithmus nicht lief. Nach einigen Researchen stellte sich heraus, dass Rom Navi GPUs erst ab Herbst 2021 unterstützt, da diese momentan noch nicht in Rechenzentren verwendet werden. Auch stellte sich heraus, dass APUs nur inoffiziell unterstützt werden und einige Funktionen dort nicht existieren. Die billigste Vega GPU mit ROCm Unterstützung ist etwa \$500 was meiner Meinung nach zu teuer für eine jahrealte Grafikkarte ist. Jedoch wäre es spannend, meinen Algorithmus auf Vega auszuprobieren. Nicht nur um zu schauen, ob ROCm funktioniert, sondern Vega Grafikkarten nutzen mit HBM2 den schnellsten existierenden Speicher, welcher normalerweise nur Enterprise GPUs haben was die Geschwindigkeit meines Algorithmus wesentlich verbessern könnte. Dennoch waren mir die jedoch zu teuer und ich warte lieber bis ROCm Navi unterstützt. Des Weiteren hat sich herausgestellt, dass ROCm auch mit CUDA noch einige Schwierigkeiten hat. Hier ein Beispiel eines offiziellen ROCm Beispiels kompiliert für NVidia GPUs. Wie man sehen kann, funktionieren nicht mal deren offizielle Beispiele was nicht gerade für ihre Bibliothek spricht:

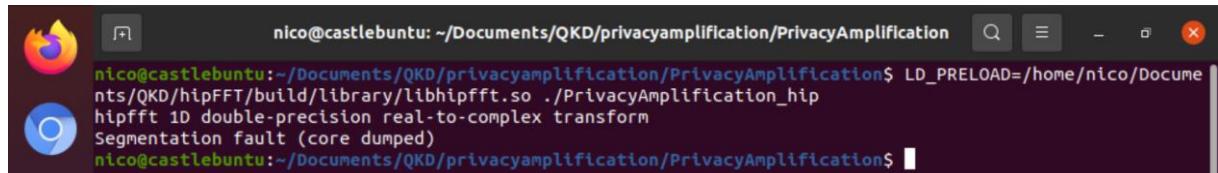
```

root@centos:/opt/rocm-4.0.0/hip/samples/0_Intro/bit_extract
File Edit View Search Terminal Help
[root@centos bit_extract]# nvidia-smi

+-----+
| NVIDIA-SMI 460.32.03    Driver Version: 460.32.03    CUDA Version: 11.2 |
| Persistence-M| Bus-Id     Disp.A | Volatile Uncorr. ECC | | | | |
| Fan  Temp  Perf  Pwr:Usage/Cap| Memory-Usage | GPU-Util  Compute M. |
|          |          |          |          |          |          | MIG M. |
+-----+
| 0  GeForce RTX 207... Off | 00000000:01:00.0 On |                    N/A | | | | |
| 0%   45C    P8    28W / 215W |      571MiB / 7979MiB |      9%     Default |
|          |          |          |          |          |          | N/A |
+-----+
+-----+
| Processes:
| GPU  GI CI      PID  Type  Process name          GPU Memory |
| ID   ID          ID   ID               Usage          |
+-----+
| 0   N/A N/A    2353    G  /usr/libexec/Xorg        200MiB |
| 0   N/A N/A    2751    G  /usr/bin/gnome-shell    368MiB |
+-----+
[root@centos bit_extract]# ./bit_extract
info: running on device #0 GeForce RTX 2070 SUPER
info: allocate host mem ( 7.63 MB)
info: allocate device mem ( 7.63 MB)
info: copy Host2Device
info: launch 'bit_extract_kernel'
info: copy Device2Host
info: check result
mismatch detected.
256: 00000000 =? 00000001 (Ain=00000100)
error: 'unknown error'(999) at bit_extract.cpp:95
[root@centos bit_extract]#

```

Abbildung 3: ROCm Nvidia CentOS 8



```

nico@castlebuntu: ~/Documents/QKD/privacyamplification/PrivacyAmplification$ LD_PRELOAD=/home/nico/Documents/QKD/hipFFT/build/library/libhipfft.so ./PrivacyAmplification_hip
hipfft 1D double-precision real-to-complex transform
Segmentation fault (core dumped)
nico@castlebuntu:~/Documents/QKD/privacyamplification/PrivacyAmplification$ 

```

Abbildung 4: PA auf AMD RX 5700 XT ausgeführt.

17. Vuda testen

Nach meiner Unzufriedenheit mit der ROCm Plattform suchte ich eine Alternative. Mir war es wichtig, dass damit jede aktuell auf dem Markt befindende sowie alle zukünftigen Grafikkarten mit ihrer vollen Geschwindigkeit unterstützt werden. Da stiess ich auf Vuda, einer Bibliothek, welche das CUDA Runtime API Interface bereitstellt, selbst jedoch die Vulkan API benutzt. Dies ist besonders interessant, da auch VkFFT, die momentan schnellste FFT Bibliothek für Grafikkarten, auf der Vulkan API basiert. Der Grund, dass ich nicht schon lange Vulkan nutze ist, da Vulkan viel komplexer als Cuda ist, da Vulkan nicht primär zum

Schreiben von GPU Anwendungen, sondern zur Interaktion von Gameengines mit der GPU entwickelt wurde. Eine Zeile Cuda entspricht also etwa 10 Zeilen Vulkan. Durch Vuda ist es mir möglich, weiterhin die gut lesbare Cuda Syntax zu nutzen, welche dann Vuda durch Funktionen und Templates automatisch in valide Vulkan Syntax umwandelt. Leider gibt es da einen Haken. Vulkan kennt keine Compute Kernels. Anstelle müssen Compute Shaders verwendet werden. Diese werden normalerweise beispielsweise zum Beleuchten von Szenen in Videogames verwendet, können aber auch zum Ausführen von Berechnungen auf Grafikkarten verwendet werden. Leider heisst dies aber auch, dass ich zur Nutzung von Vuda mein gesamter Grafikkartencode manuell von Cuda in eine Shadersprache umschreiben muss. Dies ist ziemlich viel Arbeit, aber die im letzten Semester geschriebenen Testcases werden mir hier sehr nützlich sein.

18. Vuda Fehler beheben

Als ich Vuda mit der neusten Vulkan Bibliothek ausprobiert habe, trat folgender Fehler auf sobald ich vuda::lauchKernel benutzte:

```

PrivacyAmplificationVulkan.cpp      utility  ▾
PrivacyAmplificationVulkan          std::pair<_Ty1, _Ty2>
236     negation<conjunction<is_convertible<_Other1, _Ty1>, is_convertible<_Other2, _Ty2>>::
237     int> = 0;
238     constexpr explicit pair<const pair<_Other1, _Other2>& _Right> noexcept
239     : is_nothrow_constructible_v<_Ty1, const _Other1&>&&
240     is_nothrow_constructible_v<_Ty2, const _Other2&> // strengthened
241     : first(_Right.first), second(_Right.second) {}
242 #endif // ^__ _HAS_CONDITIONAL_EXPLICIT __^
243
244 #ifndef _HAS_CONDITIONAL_EXPLICIT
245     template <class _Other1, class _Other2, <T> Geben Sie Beispielvorlagenargumente für IntelliSense an. >
246     : enable_if_t<conjunction<v_is_constructible<_Ty1, _Other1>, is_constructible<_Ty2, _Other2>>, int> = 0>
247     constexpr explicit conjunction<v_is_convertible<_Other1, _Ty1>, is_convertible<_Other2, _Ty2>>
248     : pair(pair<_Other1, _Other2>& _Right) noexcept(
249     : is_nothrow_constructible_v<_Ty1, _Other1&& is_nothrow_constructible_v<_Ty2, _Other2>) // strengthened
250     : first(_STD forward<_Other1>(_Right.first)), second(_STD forward<_Other2>(_Right.second)) {}
251 #else // __ _HAS_CONDITIONAL_EXPLICIT __^ / vvv ! _HAS_CONDITIONAL_EXPLICIT vvv
252     template <class _Other1, class _Other2,
253     enable_if_t<conjunction<v_is_constructible<_Ty1, _Other1>, is_constructible<_Ty2, _Other2>,
254     is_convertible<_Other1, _Ty1>, is_convertible<_Other2, _Ty2>,
255     int> = 0>
256     constexpr pair<pair<_Other1, _Other2>& _Right> noexcept(
257     : is_nothrow_constructible_v<_Ty1, _Other1&& is_nothrow_constructible_v<_Ty2, _Other2> // strengthened
258     : first(_STD forward<_Other1>(_Right.first)), second(_STD forward<_Other2>(_Right.second)) {})
259
260     template <class _Other1, class _Other2,
261     enable_if_t<conjunction<v_is_constructible<_Ty1, _Other1>, is_constructible<_Ty2, _Other2>,
262     negation<conjunction<is_convertible<_Other1, _Ty1>, is_convertible<_Other2, _Ty2>>>,
263     int> = 0>

```

Fehlerliste

Gesamte Projektmappe ▾ 1 Fehler 0 von 78 Warnungen 0 von 29 Mitteilungen Erstellen + IntelliSense ▾

C4996 vk::ResultValue<vk::UniqueHandle<vk::Pipeline>::operator<>': Implicit cast operators on vk::ResultValue are deprecated. Explicitly access the value as member of ResultValue.

Abbildung 5: Vuda-Fehler

Der Code in welchem der Fehler angezeigt wurde, war teil von C++ selbst was die Fehlersuche sehr kompliziert machte. Ich ging dem vuda::lauchKernel Aufruf Schritt für Schritt in allen Unterfunktionen nach bis ich die fehlerhafte Unterfunktion fand. Auch habe ich alle vulkan- Release vom letzten Jahr ausprobiert, bis ich herausfand ab welcher Version

der Fehler auftrat. Ich konnte den Fehler beheben. Dann eröffnete ich ein Issue mit der Fehlerbeschreibung und ein Pull- Request mit der Fehlerbehebung auf der offiziellen Vuda-GitHub repository. Mein Request wurde dankend angenommen.

The screenshot shows a GitHub issue page for issue #21. The title is "VUDA doesn't compile on Vulkan 1.2.154.1 and later if vuda::launchKernel is used". The issue is marked as "Closed" by "nicoboss" 6 days ago, with 1 comment. The comment contains code snippets from CentOS 8 and Visual Studio 2019 showing compilation errors related to Vulkan API usage. Below the comment, "nicoboss" mentioned the issue and linked pull request #22, which was merged. "jgbit" closed the issue yesterday. "jgbit" also commented yesterday, thanking the reporter. The right sidebar shows assignees, labels, projects, and milestones are not assigned. There are no linked pull requests.

Abbildung 6: Vuda Issue

The screenshot shows a GitHub pull-request page for pull request #22. The title is "Fixed compilation error on Vulkan 1.2.154.1 and later". The status is "Merged" by "jgbit" yesterday, merging commit f7e8ba7 into "jgbit:master". The pull request has 0 conversations, 1 commit, 0 checks, and 1 file changed. The commit message is "This fixes #21". The commit was made by "nicoboss" 6 days ago. The commit hash is 4638f05. "jgbit" merged the commit yesterday. The right sidebar shows the pull request was verified.

Abbildung 7: Vuda Pull-Request

19. vkFFT für Cuda und Vulkan kompilieren

```
administrator@CastlePeak MINGW64 ~/Documents/GitHub/HSLU/QKD/GPU/VkFFT/build_vulkan (master)
$ cmake ..
-- Building for: Visual Studio 16 2019
-- Selecting Windows SDK version 10.0.18362.0 to target Windows 10.0.20289.
-- The C compiler identification is MSVC 19.28.29912.0
-- The CXX compiler identification is MSVC 19.28.29912.0
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: C:/Program Files (x86)/Microsoft Visual Studio/2019/Enterprise/VC/Tools/MSVC/14.28.29910/bin/Hostx64/x64/cl.exe - skipped
-- Detecting C compile features
-- Detecting CXX compile features - done
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: C:/Program Files (x86)/Microsoft Visual Studio/2019/Enterprise/VC/Tools/MSVC/14.28.29910/bin/Hostx64/x64/cxx.exe - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Found Vulkan: C:/VulkansDK/1.2.170.0/Lib/vulkan-1.lib
-- Using Release VC++ CRT: MD
-- Using Release VC++ CRT: MD
-- Found PythonInterp: C:/Python39/python.exe (found suitable version "3.9.1", minimum required is "3")
-- Configuring done
-- Generating done
-- Build files have been written to: C:/Users/Administrator/Documents/GitHub/HSLU/QKD/GPU/VkFFT/build_vulkan
```

Abbildung 8: vkFFT build Vulkan

```
administrator@CastlePeak MINGW64 ~/Documents/GitHub/HSLU/QKD/GPU/VkFFT/build_cuda (master)
$ cmake ..
-- Building for: Visual Studio 16 2019
-- Selecting Windows SDK version 10.0.18362.0 to target Windows 10.0.20289.
-- The C compiler identification is MSVC 19.28.29912.0
-- The CXX compiler identification is MSVC 19.28.29912.0
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: C:/Program Files (x86)/Microsoft Visual Studio/2019/Enterprise/VC/Tools/MSVC/14.28.29910/bin/Hostx64/x64/cl.exe - skipped
-- Detecting C compile features
-- Detecting CXX compile features - done
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: C:/Program Files (x86)/Microsoft Visual Studio/2019/Enterprise/VC/Tools/MSVC/14.28.29910/bin/Hostx64/x64/cxx.exe - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Found CUDA: C:/Program Files/NVIDIA GPU Computing Toolkit/CUDA/v11.2 (found suitable version "11.2", minimum required is "9.0")
-- The CUDA compiler identification is NVIDIA 11.2.142
-- Detecting CUDA compiler ABI info
-- Detecting CUDA compiler ABI info - done
-- Check for working CUDA compiler: C:/Program Files/NVIDIA GPU Computing Toolkit/CUDA/v11.2/bin/nvcc.exe - skipped
-- Detecting CUDA compile features
-- Detecting CUDA compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: C:/Users/Administrator/Documents/GitHub/HSLU/QKD/GPU/VkFFT/build_cuda
```

Abbildung 9: vkFFT build CUDA

Ich habe vkFFT erfolgreich auf Windows für Vulkan und Cuda kompilieren können.

20. vkFFT updaten

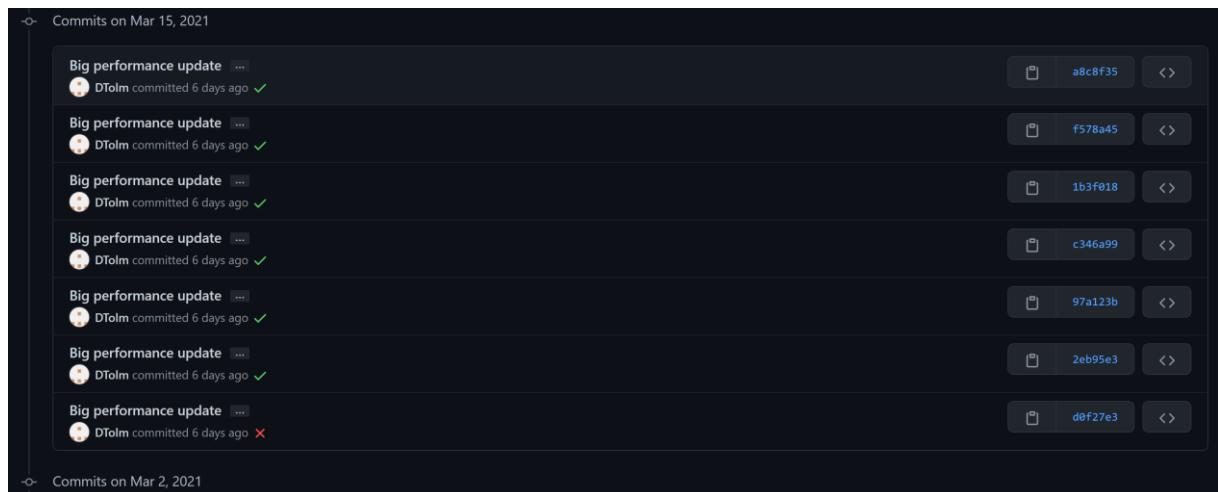
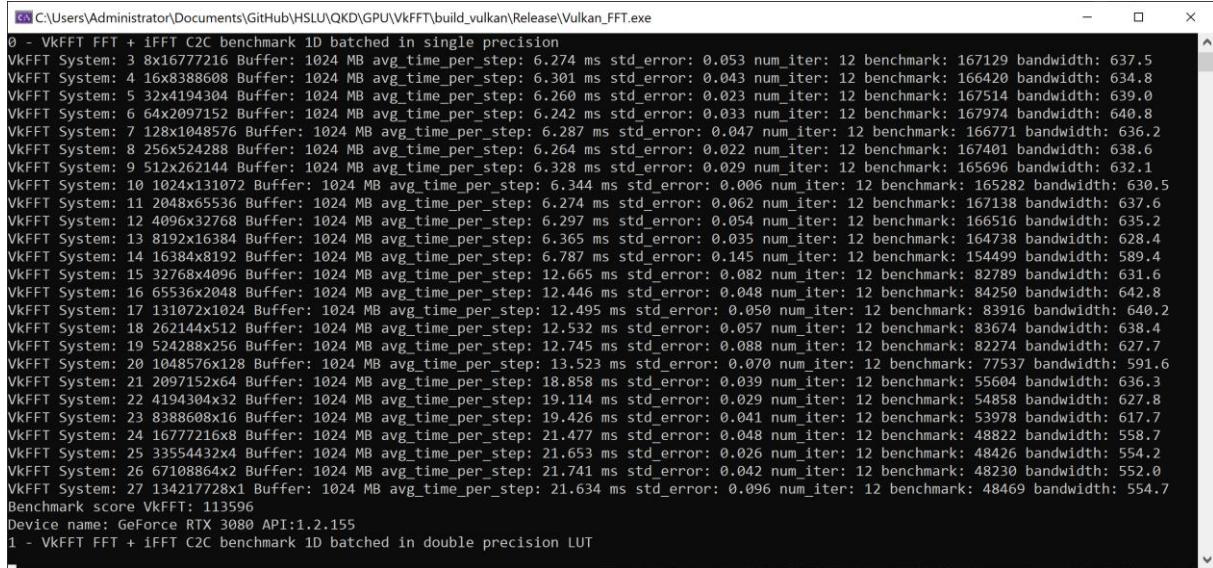


Abbildung 10: vkFFT update

Genau einen Tag, nachdem ich vkFFT eingerichtet hatte, gab es ein grosses Update mit gravierenden Verbesserungen. So musste ich das Ganze nochmals von vor neu einrichten.

21. vkFFT für Vulkan kompilieren



```
0 - VKFFT FFT + iFFT C2C benchmark 1D batched in single precision
VkFFT System: 3 8x16777216 Buffer: 1024 MB avg_time_per_step: 6.274 ms std_error: 0.053 num_iter: 12 benchmark: 167129 bandwidth: 637.5
VkFFT System: 4 16x8388608 Buffer: 1024 MB avg_time_per_step: 6.301 ms std_error: 0.043 num_iter: 12 benchmark: 166420 bandwidth: 634.8
VkFFT System: 5 32x4194304 Buffer: 1024 MB avg_time_per_step: 6.266 ms std_error: 0.023 num_iter: 12 benchmark: 167514 bandwidth: 639.0
VkFFT System: 6 64x2097152 Buffer: 1024 MB avg_time_per_step: 6.242 ms std_error: 0.033 num_iter: 12 benchmark: 167974 bandwidth: 640.8
VkFFT System: 7 128x1048576 Buffer: 1024 MB avg_time_per_step: 6.287 ms std_error: 0.047 num_iter: 12 benchmark: 166771 bandwidth: 636.2
VkFFT System: 8 256x524288 Buffer: 1024 MB avg_time_per_step: 6.264 ms std_error: 0.022 num_iter: 12 benchmark: 167401 bandwidth: 638.6
VkFFT System: 9 512x262144 Buffer: 1024 MB avg_time_per_step: 6.328 ms std_error: 0.029 num_iter: 12 benchmark: 165696 bandwidth: 632.1
VkFFT System: 10 1024x131072 Buffer: 1024 MB avg_time_per_step: 6.344 ms std_error: 0.006 num_iter: 12 benchmark: 165282 bandwidth: 630.5
VkFFT System: 11 2048x65536 Buffer: 1024 MB avg_time_per_step: 6.274 ms std_error: 0.062 num_iter: 12 benchmark: 167138 bandwidth: 637.6
VkFFT System: 12 4096x32768 Buffer: 1024 MB avg_time_per_step: 6.297 ms std_error: 0.054 num_iter: 12 benchmark: 166516 bandwidth: 635.2
VkFFT System: 13 8192x16384 Buffer: 1024 MB avg_time_per_step: 6.365 ms std_error: 0.035 num_iter: 12 benchmark: 164738 bandwidth: 628.4
VkFFT System: 14 16384x8192 Buffer: 1024 MB avg_time_per_step: 6.787 ms std_error: 0.145 num_iter: 12 benchmark: 154499 bandwidth: 589.4
VkFFT System: 15 32768x4096 Buffer: 1024 MB avg_time_per_step: 12.665 ms std_error: 0.082 num_iter: 12 benchmark: 82789 bandwidth: 631.6
VkFFT System: 16 65536x2048 Buffer: 1024 MB avg_time_per_step: 12.446 ms std_error: 0.048 num_iter: 12 benchmark: 84250 bandwidth: 642.8
VkFFT System: 17 131072x1024 Buffer: 1024 MB avg_time_per_step: 12.495 ms std_error: 0.050 num_iter: 12 benchmark: 83916 bandwidth: 640.2
VkFFT System: 18 262144x512 Buffer: 1024 MB avg_time_per_step: 12.532 ms std_error: 0.057 num_iter: 12 benchmark: 83674 bandwidth: 638.4
VkFFT System: 19 524288x256 Buffer: 1024 MB avg_time_per_step: 12.745 ms std_error: 0.088 num_iter: 12 benchmark: 82274 bandwidth: 627.7
VkFFT System: 20 1048576x128 Buffer: 1024 MB avg_time_per_step: 13.523 ms std_error: 0.070 num_iter: 12 benchmark: 77537 bandwidth: 591.6
VkFFT System: 21 2097152x64 Buffer: 1024 MB avg_time_per_step: 18.858 ms std_error: 0.039 num_iter: 12 benchmark: 55604 bandwidth: 636.3
VkFFT System: 22 4194304x32 Buffer: 1024 MB avg_time_per_step: 19.114 ms std_error: 0.029 num_iter: 12 benchmark: 54858 bandwidth: 627.8
VkFFT System: 23 8388608x16 Buffer: 1024 MB avg_time_per_step: 19.426 ms std_error: 0.041 num_iter: 12 benchmark: 53978 bandwidth: 617.7
VkFFT System: 24 16777216x8 Buffer: 1024 MB avg_time_per_step: 21.477 ms std_error: 0.048 num_iter: 12 benchmark: 48822 bandwidth: 558.7
VkFFT System: 25 33554432x4 Buffer: 1024 MB avg_time_per_step: 21.653 ms std_error: 0.026 num_iter: 12 benchmark: 48426 bandwidth: 554.2
VkFFT System: 26 67108864x2 Buffer: 1024 MB avg_time_per_step: 21.741 ms std_error: 0.042 num_iter: 12 benchmark: 48230 bandwidth: 552.0
VkFFT System: 27 134217728x1 Buffer: 1024 MB avg_time_per_step: 21.634 ms std_error: 0.096 num_iter: 12 benchmark: 48469 bandwidth: 554.7
Benchmark score VkFFT: 113596
Device name: GeForce RTX 3080 API:1.2.155
1 - VKFFT FFT + iFFT C2C benchmark 1D batched in double precision LUT
```

Abbildung 11: vkFFT ausgeführt.

Ich habe vkFFT erfolgreich auf Windows für Vulkan zum Laufen gebracht und getestet.

22. vkFFT mit Vuda kombinieren

Um PA in Vulkan umschreiben zu können brauche ich ein Projekt, welches Vuda mit vkFFT kombiniert. Zudem müssen alle anderen PA-Abhängigkeiten und Projekteinstellungen in dieses Projekt übernommen werden.

23. GLSL und SPIR-V

Vuda unterstützt leider nur in der Standard Portable Intermediate Representation SPIR-V geschriebenen Shaders als GPU Code. Es kann jedoch nicht direkt in SPIR-V programmiert werden, da dies nur eine Zwischensprache ist. Nach vielem überlegen entschied ich mich für die OpenGL Shading Language GLSL welche die beste Kompilierung zu SPIR-V zu haben scheint. Die Kompilierung von GLSL nach SPIR-V werde ich mit

<https://github.com/KhronosGroup/glslang> realisieren. Leider ist GLSL anders als

beispielsweise OpenCL komplett inkompatibel mit Cuda und es existieren keine Tools die bei

einer Konvertierung helfen würden. Bei OpenCL wäre es beispielsweise möglich den Prozess mit <https://github.com/vtsynergy/CU2CL> zu automatisieren. Jedoch vermute ich, dass ich mit GLSL bessere Performance erreichen werde. Auch VkFFT nutzt im Hintergrund dynamisch generierten GLSL Code welcher beim Planen der FFT zur Laufzeit über das glslang C interface nach SPIR-V kompiliert wird. Da VkFFT die momentan schnellste FFT Bibliothek ist hat mich die Performance von GLSL und SPIR-V überzeugt.

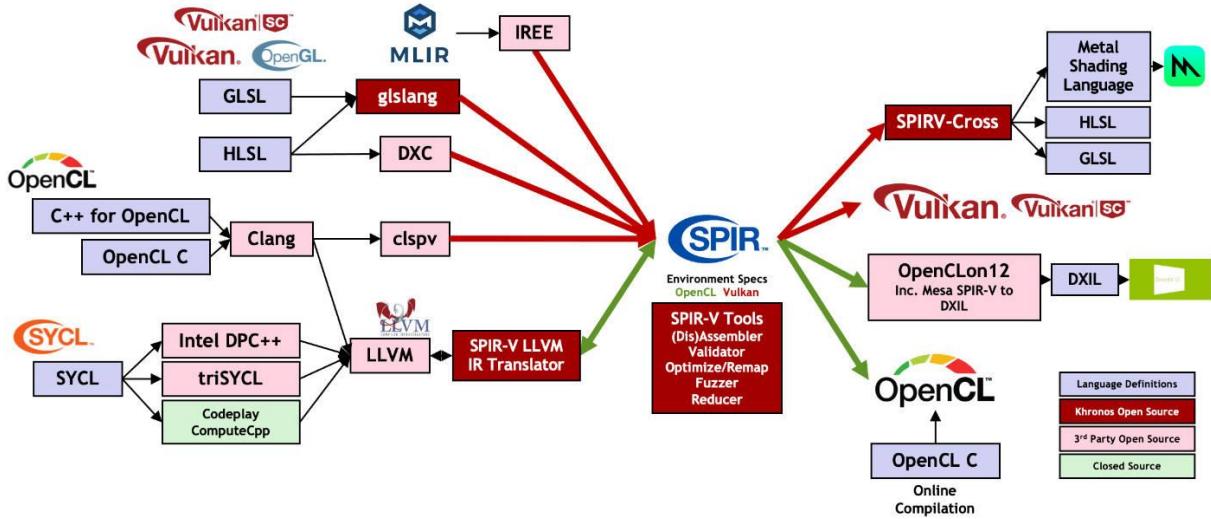


Abbildung 12: SPIR-V Umgebung

III Woche 5 + 6

24. Subblock branch aufgeräumt und libzmq geupdated

Ich habe eine stabile Version im main aus dem alten main und allen stabilen Teilen des Subblock branch erstellt. Der Codeteil mit den instabilen Elementen wurde in eine separate Datei verschoben. Die Änderungen vom instabilen zum stabilen Code sind in der git-history als diff ersichtlich. Es gab Probleme mit der, einige Wochen vorher aktualisierten ZeroMQ-Bibliothek. Einige Teile der Bibliothek wurden nicht korrekt aktualisiert. So habe ich das Ganze ZeroMQ nochmals neu aktualisiert und korrekt in das Visualstudio-Projekt eingebunden.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/662d585389208d9f31db4e372e36bcd574ab1941>

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/95655c26293b23d8fd353e7527730501b053db1>

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/35c565761be16d45deda2e8bab38ba2bcf89b50d>

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/a6e531034921e9005f6b37ab65feb13d4112fe29>

25. vkFFT geupdated

Da es sehr grosse Änderungen im vkFFT gab, habe ich mich entschieden diese Bibliothek vor der Verwendung zuerst auf die neuste Version up zu daten.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplificationvulkan/commit/8156e51e0df1d995af1bfd2b5a151d9e3f848656>

26. Debugging im neues Visual Studio Projekt ermöglichen

Um das Visual Studio Projekt für die Vulkan Version von meinem Privacy Amplification Algorithmus zu generieren, verwendete ich die CMakeLists von vkFFT. Leider stellte sich heraus, dass dieses nur ein Projekt mit release build erstellte. Da ich mindestens mein CPU-Code debuggen können will, und es durch die sehr komplexe Konfiguration von Abhängigkeiten nicht einfach möglich war ein Debug Preset zu erstellen, musste ich den Release Preset so editieren, dass Debugging möglich ist. Leider stellte sich dies alles andere als einfach heraus und nach Stundenlangem probieren aller möglichen Projekteinstellungen fand ich immer noch nicht heraus an was es lag. Durch langes Recherchieren meines Problems im Internet fand ich heraus, dass ich

<DebugInformationFormat>ProgramDatabase</DebugInformationFormat> mit einem Texteditor in das VCSProj-File schreiben muss. Sonst werden die generierten Debugsymbole nicht geladen. Danach ging das Debuging bestens.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplificationvulkan/commit/791cad8b77fe8acdc53ef4dbe5b9792685f2000a>

27. Script um GLSL-Shaders zu bauen

GLSD-Shaders können nicht mit Visual Studio kompiliert werden. Da nach jeder Änderung diese Schaders neu kompiliert werden müssen, habe ich ein Batch-Skript geschrieben, welches auf Knopfdruck automatisch alle Schaders in SPIR-V kompiliert.

```
1 :start
2 for %%f in (*.comp) do (
3     glslangValidator.exe -V %%~nf.comp -o build\%%~nf.spv
4 )
5 pause
6 goto start
```

Abbildung 13: Skript für GLSL-Shader zu kompilieren

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplificationvulkan/commit/f6baa304cbadf8c6b98f0b2bfc99f3b34ea54abe>

28. Test unitTestCalculateCorrectionFloat verbessert

Ich habe den unitTestCalculateCorrectionFloat verbessert, indem der systematische Test nun 16-mal schneller durchgeführt wird. Zudem wurde ein weiterer Test hinzugefügt wird, welcher speziell zum Testen realistischer Zahlen dient.

```

135     int unitTestCalculateCorrectionFloat() {
136         printf("Started CalculateCorrectionFloat Unit Test...");
137         if (!unitTestsFailedLocal) {
138             #if defined(_NVCC_)
139                 cudaStream_t CalculateCorrectionFloatTestStream;
140                 cudaStreamCreate(&CalculateCorrectionFloatTestStream);
141             #else
142                 const int CalculateCorrectionFloatTestStream = 0;
143             #endif
144             uint32_t* count_one_of_global_seed_test;
145             uint32_t* count_one_of_global_key_test;
146             float correction_float_dev_test;
147             uint32_t* sample_size_test;
148             cudaMallocHost((void**)&count_one_of_global_seed_test, sizeof(uint32_t));
149             cudaMallocHost((void**)&count_one_of_global_key_test, sizeof(uint32_t));
150             cudaMallocHost((void**)&correction_float_dev_test, sizeof(float));
151             cudaMallocHost((void**)&sample_size_test, sizeof(uint32_t));
152             *sample_size_test = pow(2, 6);
153             #if defined(_NVCC_)
154                 cudaMemcpy0Symbol(sample_size_dev, sample_size_test, sizeof(uint32_t));
155             #endif
156             for (uint32_t i = 0; i < *sample_size_test; ++i) {
157                 for (uint32_t j = 0; j < *sample_size_test; ++j) {
158                     *count_one_of_global_seed_test = i;
159                     *count_one_of_global_key_test = j;
160                     #if defined(_NVCC_)
161                         calculateCorrectionFloat KERNEL_ARG4(i, j, 0, CalculateCorrectionFloatTestStream)(count_one_of_global_seed_test, count_one_of_global_key_test, correction_float_dev_test);
162                     #else
163                         vuda::launchKernel("calculateCorrectionFloat.spv", "main", CalculateCorrectionFloatTestStream, i, j, count_one_of_global_seed_test, count_one_of_global_key_test, correction_float_dev_test, sample_size_test);
164                     #endif
165                     cudaStreamSynchronize(CalculateCorrectionFloatTestStream);
166                     uint64_t cpu_count_multiplied = *count_one_of_global_seed_test * *count_one_of_global_key_test;
167                     double cpu_count_multiplied_normalized = cpu_count_multiplied / (double)*sample_size_test;
168                     double count_multiplied_normalized_modulo = fmod(cpu_count_multiplied_normalized, 2.0);
169                     assertZeroThreshold(*correction_float_dev_test - count_multiplied_normalized_modulo, 0.0001, i * *sample_size_test + j);
170                 }
171             }
172             *sample_size_test = pow(2, 27);
173             #if defined(_NVCC_)
174                 cudaMemcpy0Symbol(sample_size_dev, sample_size_test, sizeof(uint32_t));
175             #endif
176             std::mt19937_64 gen(777);
177             std::uniform_int_distribution<uint32_t> distrib(pow(2, 25), pow(2, 27));
178             for (uint32_t n = 0; n < 4096; ++n) {
179                 *count_one_of_global_seed_test = distrib(gen);
180                 *count_one_of_global_key_test = distrib(gen);
181                 #if defined(_NVCC_)
182                     calculateCorrectionFloat KERNEL_ARG4(1, 0, CalculateCorrectionFloatTestStream)(count_one_of_global_seed_test, count_one_of_global_key_test, correction_float_dev_test);
183                 #else
184                     vuda::launchKernel("calculateCorrectionFloat.spv", "main", CalculateCorrectionFloatTestStream, 1, 0, count_one_of_global_seed_test, count_one_of_global_key_test, correction_float_dev_test, sample_size_test);
185                 #endif
186                 cudaStreamSynchronize(CalculateCorrectionFloatTestStream);
187                 uint64_t cpu_count_multiplied = *count_one_of_global_seed_test * *count_one_of_global_key_test;
188                 double cpu_count_multiplied_normalized = cpu_count_multiplied / (double)*sample_size_test;
189                 double count_multiplied_normalized_modulo = fmod(cpu_count_multiplied_normalized, 2.0);
190                 assertZeroThreshold(*correction_float_dev_test - count_multiplied_normalized_modulo, 0.0001, n);
191             }
192             #if defined(_NVCC_)
193                 cudaMemcpy0Symbol(sample_size_dev, &sample_size, sizeof(uint32_t));
194             #endif
195             printf("Completed CalculateCorrectionFloat Unit Test");
196             return unitTestsFailedLocal ? 100 : 0;
197         }
198     }

```

Abbildung 14: Neuer unitTest

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplificationvulkan/commit/1c70f516b3a5c7b89258854c6e3cdce54a277da1>

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplificationvulkan/commit/4fbba1929701ecc5eba5b082018872dcfde57855>

29. PrivacyAmplification-Hilfsfunktionen in Vulkan

Die meisten PrivacyAmplification-Hilfsfunktionen wurden von PrivacyAmplification zu PrivacyAmplificationVulkan herüberportiert. Diese werden für weitere Testfälle benötigt.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplificationvulkan/commit/bb3a2ad78b73eb7bb283cd92a3c07854e4b51188>

30. unitTestCalculateCorrectionFloat erfolgreich auf Vulkan

Erster Versuch, ein CUDA-Kernel als GLSL-Shader zu implementieren. Zuerst funktioniert dies nicht. Nach einigen Korrekturen konnte ich das CUDA-Kernel erfolgreich auf GLSL zum Laufen bringen. In einem nächsten Schritt wurde der Code so umgeschrieben, dass er sowohl auf CUDA und Vulkan läuft. Auf Cuda natürlich mit dem CUDA-Kernel und auf Vulkan mit dem GLSL-Shader. Der gleiche Code wird nun in PrivacyAmplification und PrivacyAmplification -Vulkan verwendet. Auf den nachfolgenden Abbildungen sieht man den Vergleich zwischen dem CUDA-Code für NVidia-Grafikkarten und dem Grafikkartenhersteller

unabhängigen GLSL-Code. Es war sehr anspruchsvoll GLSL, als neue Programmiersprache zu lernen. Es gab fast keine Beispiele für GLSL im Bereich Computing. Die Sprache ist auch allgemein schwieriger als CUDA. Mit expliziten und atomaren Datentypen gab es grosse Probleme. Unter den Standard-Datentypen gab es keinen 64-bit Integer.

```

279 __global__
280 void calculateCorrectionFloat(uint32_t* count_one_of_global_seed, uint32_t* count_one_of_global_key, float* correction_float_dev)
281 {
282     uint64_t count_multiplied = *count_one_of_global_seed * *count_one_of_global_key;
283     double count_multiplied_normalized = count_multiplied / (double)sample_size_dev;
284     double two = 2.0;
285     Real count_multiplied_normalized_modulo = (float)fmod(count_multiplied_normalized, two);
286     *correction_float_dev = count_multiplied_normalized_modulo;
287 }
288 #endif

```

Abbildung 15: CUDA-Code für calculateCorrectionFloat

```

1 #version 400
2 #extension GL_EXT_shader_explicit_arithmetic_types: enable
3
4 layout( local_size_x id = 0 ) in;
5 layout( constant_id = 1 ) const float64_t two = 2.0f;
6 layout(set = 0, binding = 0) readonly buffer A { uint32_t count_one_of_global_seed[]; };
7 layout(set = 0, binding = 1) readonly buffer B { uint32_t count_one_of_global_key[]; };
8 layout(set = 0, binding = 2) writeonly buffer C { float32_t correction_float_dev[]; };
9 layout(set = 0, binding = 3) readonly buffer D { uint32_t sample_size_dev[]; };
0
11 void main(void)
12 {
13     uint64_t count_multiplied = count_one_of_global_seed[0] * count_one_of_global_key[0];
14     float64_t count_multiplied_normalized = float64_t(count_multiplied) / float64_t(sample_size_dev[0]);
15     correction_float_dev[0] = float32_t(mod(count_multiplied_normalized, two)); //count_multiplied_normalized_modulo
16 }

```

Abbildung 16: GLSL-Code für calculateCorrectionFloat

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplificationvulkan/commit/b6e2814f3450882304a305c723552d5952621d2b>
<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplificationvulkan/commit/6e036086e2bbf7dd5e0c59d23d321ffb21be8064>
<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplificationvulkan/commit/1edcae2bc8e28f32568021f9ef0ab3c39c54fe1>
<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/48e5e1e7fa8164198aff9656f5843c150aed9bea>

31. unitTestSetFirstElementToZero erfolgreich auf Vulkan

Der CUDA-Code für setFirstElementToZero wurde erfolgreich in den GLSL-Code umprogrammiert. Der Testcase «unitTestSetFirstElementToZero» ist nun auch mit GLSL erfolgreich. Der gleiche Code wird nun in PrivacyAmplification und PrivacyAmplification - Vulkan verwendet.

```

327     __global__
328     void setFirstElementToZero(Complex* do1, Complex* do2)
329     {
330         if (threadIdx.x == 0) {
331             do1[0] = c0_dev;
332         }
333         else
334         {
335             do2[0] = c0_dev;
336         }
337     }

```

Abbildung 17: CUDA-Code für setFirstElementToZero

```

1 #version 460
2 #extension GL_EXT_shader_explicit_arithmetic_types: enable
3
4 layout( local_size_x_id = 0 ) in;
5 layout(set = 0, binding = 0) writeonly buffer A { f32vec2 do1[]; };
6 layout(set = 0, binding = 1) writeonly buffer B { f32vec2 do2[]; };
7
8 void main(void)
{
9
10    if (gl_GlobalInvocationID.x == 0) {
11        do1[0] = f32vec2(0.0, 0.0);
12    }
13    else
14    {
15        do2[0] = f32vec2(0.0, 0.0);
16    }
17}

```

Abbildung 18:GLSL-Code für setFirstElementToZe

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplificationvulkan/commit/b04dcb15221f8e663b60ef0a1ec44f13626924e8>
<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/287035592831b76345d50b652e918b323396d3aa>

32. unitTestElementWiseProduct erfolgreich auf Vulkan

Der CUDA-Code für ElementWiseProduct wurde erfolgreich in den GLSL-Code umprogrammiert. Der Testcase «unitTestElementWiseProduct» ist nun auch mit GLSL erfolgreich. Der gleiche Code wird nun in PrivacyAmplification und PrivacyAmplification - Vulkan verwendet.

```

385     __global__
386     void ElementWiseProduct(Complex* do1, Complex* do2)
387     {
388         uint32_t i = blockIdx.x * blockDim.x + threadIdx.x;
389         float r = pre_mul_reduction_dev;
390         Real do1x = do1[i].x / r;
391         Real do1y = do1[i].y / r;
392         Real do2x = do2[i].x / r;
393         Real do2y = do2[i].y / r;
394         do1[i].x = do1x * do2x - do1y * do2y;
395         do1[i].y = do1x * do2y + do1y * do2x;
396     }

```

Abbildung 19: CUDA-Code für ElementWiseProduct

```

1 #version 460
2 #extension GL_EXT_shader_explicit_arithmetic_types: enable
3
4 layout( local_size_x_id = 0 ) in;
5 layout(set = 0, binding = 0) buffer A { f32vec2 do1[]; };
6 layout(set = 0, binding = 1) readonly buffer B { f32vec2 do2[]; };
7 layout(set = 0, binding = 2) readonly buffer C { uint32_t pre_mul_reduction_dev
[]; };
8
9 void main(void)
10{
11    uint i = gl_GlobalInvocationID.x;
12    float32_t r = pre_mul_reduction_dev[0];
13    float32_t do1x = do1[i].x / r;
14    float32_t do1y = do1[i].y / r;
15    float32_t do2x = do2[i].x / r;
16    float32_t do2y = do2[i].y / r;
17    do1[i].x = do1x * do2x - do1y * do2y;
18    do1[i].y = do1x * do2y + do1y * do2x;
19}

```

Abbildung 20: GLSL-Code für ElementWiseProduct

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplificationvulkan/commit/be5e45489095ea6b3cf1e90ee1f9187f55740be3>
<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/4299be9427be99a89ec0261fdbf3d3b23f14fef1>

33. Fehlende Bibliotheken und Konfigurationsdateilogik

Ich musste einige fehlende Bibliotheken von PrivacyAmplification zu PrivacyAmplificationValken herüberportieren. Auch die Konfigurationsdatei logic musste

hinübergenommen werden. Da Linux Probleme mit dem Registerkeywort hat, habe ich diese entfernt.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplificationvulkan/commit/08fe2778ed47099241f408cde4e3a4264635f7a7>
<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplificationvulkan/commit/d783ca87745e7b6256da4e0b440435507ae4d7cc>
<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/010f4aee4c73111277346b506abe4068d2b6ec1a>

34. unitTestBinInt2float erfolgreich auf Vulkan

Der CUDA-Code für binInt2float wurde erfolgreich in den GLSL-Code umprogrammiert. Der Testcase «unitTestBinInt2float» ist nun auch mit GLSL erfolgreich. Der gleiche Code wird nun in PrivacyAmplification und PrivacyAmplification -Vulkan verwendet.

```
483  __global__  
484  void binInt2float(uint32_t* binIn, Real* realOut, uint32_t* count_one_global)  
485  {  
486      //Multicast  
487      Real h0_local = h0_dev;  
488      Real h1_reduced_local = h1_reduced_dev;  
489      __shared__ uint32_t binInShared[32];  
490  
491      uint32_t block = blockIdx.x;  
492      uint32_t idx = threadIdx.x;  
493      uint32_t maskToUse;  
494      uint32_t inPos;  
495      uint32_t outPos;  
496      maskToUse = idx % 32;  
497      inPos = idx / 32;  
498      outPos = 1024 * block + idx;  
499  
500      if (threadIdx.x < 32) {  
501          binInShared[idx] = binIn[32 * block + idx];  
502      }  
503      __syncthreads();  
504  
505      if ((binInShared[inPos] & intTobinMask_dev[maskToUse]) == 0) {  
506          realOut[outPos] = h0_local;  
507      }  
508      else {  
509          atomicAdd(count_one_global, 1);  
510          realOut[outPos] = h1_reduced_local;  
511      }  
512  }  
513 }
```

Abbildung 21: CUDA-Code für binInt2float

```

1 #version 460
2 #extension GL_EXT_shader_explicit_arithmetic_types: enable
3
4 shared uint32_t binInShared[32];
5
6 uint32_t intTobinMask_dev[32] =
7 uint32_t[] (
8     2147483648. //0b100000000000000000000000000000000000000000000000000000000000000

```

```

43 layout( local_size_x_id = 0 ) in;
44 layout( constant_id = 1 ) const float32_t h0_local = 0.0;
45 layout(set = 0, binding = 0) readonly buffer A { uint32_t binIn[]; };
46 layout(set = 0, binding = 1) writeonly buffer B { float32_t realOut[]; };
47 layout(set = 0, binding = 2) buffer C { uint count_one_global[]; };
48 layout(set = 0, binding = 3) readonly buffer D { float32_t h1_reduced_local[]; };
49
50 void main(void)
51 {
52     uint32_t block = gl_WorkGroupID.x;
53     uint32_t idx = gl_LocalInvocationID.x;
54     uint32_t maskToUse;
55     uint32_t inPos;
56     uint32_t outPos;
57     maskToUse = idx % 32;
58     inPos = idx / 32;
59     outPos = 1024 * block + idx;
60
61     if (idx < 32) {
62         binInShared[idx] = binIn[32 * block + idx];
63     }
64     barrier();
65
66     if ((binInShared[inPos] & intTobinMask_dev[maskToUse]) == 0) {
67         realOut[outPos] = h0_local;
68     }
69     else
70     {
71         atomicAdd(count_one_global[0], 1);
72         realOut[outPos] = h1_reduced_local[0];
73     }
74 }

```

Abbildung 22: GLSL-Code für binInt2float

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplificationvulkan/commit/6332499f89fe243219cff8f96e786bf0a01b7fde>

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/a5e64da794c8c6e49e9028784bf464e90ea8eef5>

35. unitTestToBinaryArray erfolgreich auf Vulkan

Der CUDA-Code für ToBinaryArray wurde erfolgreich in den GLSL-Code umprogrammiert.

Der Testcase «unitTestToBinaryArray» ist nun auch mit GLSL erfolgreich. Der gleiche Code wird nun in PrivacyAmplification und PrivacyAmplification -Vulkan verwendet.

```
626  __global__  
627  void ToBinaryArray(Real* invOut, uint32_t* binOut, uint32_t* key_rest_local, Real* correction_float_dev)  
628  {  
629      const Real normalisation_float_local = normalisation_float_dev;  
630      const uint32_t block = blockIdx.x;  
631      const uint32_t idx = threadIdx.x;  
632      const Real correction_float = *correction_float_dev;  
633  
634      __shared__ uint32_t key_rest_xor[31];  
635      __shared__ uint32_t binOutRawBit[992];  
636      if (idx < 992) {  
637          binOutRawBit[idx] = ((float2int(invOut[block * 992 + idx] / normalisation_float_local + correction_float) & 1)  
638              << ToBinaryBitShiftArray_dev[idx % 32]);  
639      }  
640      else if (idx < 1023)  
641      {  
642          #if XOR_WITH_KEY_REST == TRUE  
643          #if AMPOUT_REVERSE_ENDIAN == TRUE  
644              uint32_t key_rest_little = key_rest_local[block * 31 + idx - 992];  
645              key_rest_xor[idx - 992] =  
646                  (((key_rest_little) & 0xffff0000) >> 24) |  
647                  (((key_rest_little) & 0x00ff0000) >> 8) |  
648                  (((key_rest_little) & 0x0000ff00) << 8) |  
649                  (((key_rest_little) & 0x000000ff) << 24));  
650          #else  
651              key_rest_xor[idx - 992] = key_rest_local[block * 31 + idx - 992];  
652          #endif  
653      }  
654      __syncthreads();  
655  
656      if (idx < 31) {  
657          const uint32_t pos = idx * 32;  
658          uint32_t binOutLocal =  
659              (binOutRawBit[pos] | binOutRawBit[pos + 1] | binOutRawBit[pos + 2] | binOutRawBit[pos + 3] |  
660              binOutRawBit[pos + 4] | binOutRawBit[pos + 5] | binOutRawBit[pos + 6] | binOutRawBit[pos + 7] |  
661              binOutRawBit[pos + 8] | binOutRawBit[pos + 9] | binOutRawBit[pos + 10] | binOutRawBit[pos + 11] |  
662              binOutRawBit[pos + 12] | binOutRawBit[pos + 13] | binOutRawBit[pos + 14] | binOutRawBit[pos + 15] |  
663              binOutRawBit[pos + 16] | binOutRawBit[pos + 17] | binOutRawBit[pos + 18] | binOutRawBit[pos + 19] |  
664              binOutRawBit[pos + 20] | binOutRawBit[pos + 21] | binOutRawBit[pos + 22] | binOutRawBit[pos + 23] |  
665              binOutRawBit[pos + 24] | binOutRawBit[pos + 25] | binOutRawBit[pos + 26] | binOutRawBit[pos + 27] |  
666              binOutRawBit[pos + 28] | binOutRawBit[pos + 29] | binOutRawBit[pos + 30] | binOutRawBit[pos + 31])  
667              #if XOR_WITH_KEY_REST == TRUE  
668              ^ key_rest_xor[idx]  
669              #endif  
670              ;  
671          binOut[block * 31 + idx] = binOutLocal;  
672      }  
673  }
```

Abbildung 23: CUDA-Code für ToBinaryArray

```

27    layout( local_size_x_id = 0 ) in;
28    layout( constant_id = 1 ) const float32_t h0_local = 0.0;
29    layout(set = 0, binding = 0) readonly buffer A { float32_t invOut[]; };
30    layout(set = 0, binding = 1) writeonly buffer B { uint32_t binOut[]; };
31    layout(set = 0, binding = 2) readonly buffer C { uint32_t key_rest_local[]; };
32    layout(set = 0, binding = 3) readonly buffer D { float32_t correction_float_dev[]; };
33    layout(set = 0, binding = 4) readonly buffer E { float32_t normalisation_float_dev[]; };
34
35 void main(void)
36 {
37    const float32_t normalisation_float_local = normalisation_float_dev[0];
38    const uint32_t block = gl_WorkGroupID.x;
39    const uint32_t idx = gl_LocalInvocationID.x;
40    const float32_t correction_float = correction_float_dev[0];
41
42    if (idx < 992) {
43        binOutRawBit[idx] = (uint32_t)(round(invOut[block * 992 + idx] / normalisation_float_local + correction_float) & 1)
44                                << ToBinaryBitShiftArray_dev[idx % 32];
45    }
46    else if (idx < 1023) {
47    {
48        #if XOR_WITH_KEY_REST == TRUE
49        #if AMPOUT_REVERSE_ENDIAN == TRUE
50            uint32_t key_rest_little = key_rest_local[block * 31 + idx - 992];
51            key_rest_xor[idx - 992] =
52                (((key_rest_little) & 0xff000000) >> 24) |
53                (((key_rest_little) & 0x00ff0000) >> 8) |
54                (((key_rest_little) & 0x0000ff00) << 8) |
55                (((key_rest_little) & 0x000000ff) << 24));
56        #else
57            key_rest_xor[idx - 992] = key_rest_local[block * 31 + idx - 992];
58        #endif
59    #endif
60    }
61    barrier();
62
63    if (idx < 31) {
64        const uint32_t pos = idx * 32;
65        uint32_t binOutLocal =
66            (binOutRawBit[pos] | binOutRawBit[pos + 1] | binOutRawBit[pos + 2] | binOutRawBit[pos + 3] |
67             binOutRawBit[pos + 4] | binOutRawBit[pos + 5] | binOutRawBit[pos + 6] | binOutRawBit[pos + 7] |
68             binOutRawBit[pos + 8] | binOutRawBit[pos + 9] | binOutRawBit[pos + 10] | binOutRawBit[pos + 11] |
69             binOutRawBit[pos + 12] | binOutRawBit[pos + 13] | binOutRawBit[pos + 14] | binOutRawBit[pos + 15] |
70             binOutRawBit[pos + 16] | binOutRawBit[pos + 17] | binOutRawBit[pos + 18] | binOutRawBit[pos + 19] |
71             binOutRawBit[pos + 20] | binOutRawBit[pos + 21] | binOutRawBit[pos + 22] | binOutRawBit[pos + 23] |
72             binOutRawBit[pos + 24] | binOutRawBit[pos + 25] | binOutRawBit[pos + 26] | binOutRawBit[pos + 27] |
73             binOutRawBit[pos + 28] | binOutRawBit[pos + 29] | binOutRawBit[pos + 30] | binOutRawBit[pos + 31]);
74        #if XOR_WITH_KEY_REST == TRUE
75            ^ key_rest_xor[idx]
76        #endif
77        ;
78        binOut[block * 31 + idx] = binOutLocal;
79    }
80}

```

Abbildung 24: GLSL-Code für ToBinaryArray

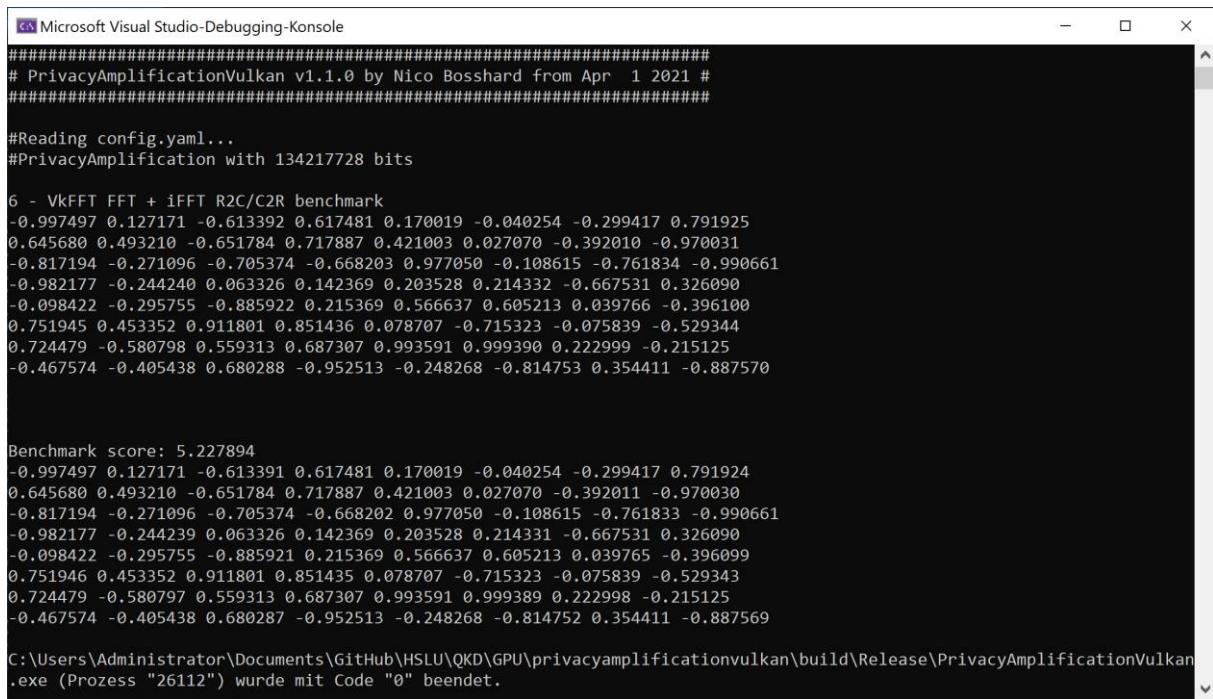
<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplificationvulkan/commit/0a7707da4d54c4f7971cde7d5125ac78eeef9b63>
<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplificationvulkan/commit/e011d3e1610bf1aacb0a37f31eec4b4ff40cf4bc>
<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplificationvulkan/commit/d9de493f6dfba53392002c102dfcc958befd972d>
<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplificationvulkan/commit/47ec6af442673210fe3c9bc5bebac07be4323294>
<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/0213c3d1f6ef188512db4b9f5dd66ed866faf28b>

36. Hilfsfunktionen von vkFFT separiert

Es wurden die erforderlichen Hilfsfunktionen von Vulkan_FFT.cpp separiert und in vkFFT_helper.h eingefügt. Dies führt zu einer Unabhängigkeit von Vulkan_FFT.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplificationvulkan/commit/de43db92745c141b9710a95e638f8171883ba24e>

37. Erster erfolgreicher vkFFT Aufruf



```
Microsoft Visual Studio-Debugging-Konsole
#####
# PrivacyAmplificationVulkan v1.1.0 by Nico Bosshard from Apr 1 2021 #
#####

#Reading config.yaml...
#PrivacyAmplification with 134217728 bits

6 - VkFFT FFT + iFFT R2C/C2R benchmark
-0.997497 0.127171 -0.613392 0.617481 0.170019 -0.040254 -0.299417 0.791925
0.645680 0.493210 -0.651784 0.717887 0.421003 0.027070 -0.392010 -0.970031
-0.817194 -0.271096 -0.705374 -0.668203 0.977050 -0.108615 -0.761834 -0.990661
-0.982177 -0.244240 0.063326 0.142369 0.203528 0.214332 -0.667531 0.326090
-0.098422 -0.295755 -0.885922 0.215369 0.566637 0.605213 0.039766 -0.396100
0.751945 0.453352 0.911801 0.851436 0.078707 -0.715323 -0.075839 -0.529344
0.724479 -0.580798 0.559313 0.687307 0.993591 0.999390 0.222999 -0.215125
-0.467574 -0.405438 0.680288 -0.952513 -0.248268 -0.814753 0.354411 -0.887570

Benchmark score: 5.227894
-0.997497 0.127171 -0.613391 0.617481 0.170019 -0.040254 -0.299417 0.791924
0.645680 0.493210 -0.651784 0.717887 0.421003 0.027070 -0.392011 -0.970030
-0.817194 -0.271096 -0.705374 -0.668202 0.977050 -0.108615 -0.761833 -0.990661
-0.982177 -0.244239 0.063326 0.142369 0.203528 0.214331 -0.667531 0.326090
-0.098422 -0.295755 -0.885921 0.215369 0.566637 0.605213 0.039765 -0.396099
0.751946 0.453352 0.911801 0.851435 0.078707 -0.715323 -0.075839 -0.529343
0.724479 -0.580797 0.559313 0.687307 0.993591 0.999389 0.222998 -0.215125
-0.467574 -0.405438 0.680287 -0.952513 -0.248268 -0.814752 0.354411 -0.887569

C:\Users\Administrator\Documents\GitHub\HSLU\QKD\GPU\privacyamplificationvulkan\build\Release\PrivacyAmplificationVulkan.exe (Prozess "26112") wurde mit Code "0" beendet.
```

Abbildung 25: Erfolgreicher vkFFT

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplificationvulkan/commit/dfe0ff3cc8a38cb04d40f5167be588e043779996>

38. Größenbeschränkung in vkFFT

Es stellte sich heraus, dass in vkFFT die Real zu Komplex und die Komplex zu real FFT eine Limitierung von 2^{12} hat. Ich versuchte zuerst selber eine Lösung über den Umweg mit zweimal Komplex zu Komplex mit gleicher Geschwindigkeit zu finden. Ich versuchte ob es möglich ist die Realzahlen alternierend für den Real- und Komplexeil zu nutzen. Jedoch war dies bei der elementarweisen Multiplikation nicht möglich.

```

2 #!/usr/bin/python
3 import multiprocessing
4 from math import *
5
6 inputs = range(10)
7 def processitem(item1, item2):
8     for item3 in [-1.740287, 1.496323, -1.34196, 0.4814, 0.085331, -1.322489, -0.993072, -0.14655]:
9         #if item3 == item1: continue
10        for item4 in [0.022889, 0.268136, 1.912595, 0.725914, 2.110477, 0.772424, -1.463241, 0.206366]:
11            #if item4 == item2: continue
12            for op1 in ["+", "-", "*", "/"]:
13                for op2 in ["+", "-", "*", "/"]:
14                    for op3 in ["+", "-", "*", "/"]:
15                        for w1 in ["sin", "cos", "", ""]:
16                            for w2 in ["sin", "cos", "", ""]:
17                                for w3 in ["sin", "cos", "", ""]:
18                                    for w4 in ["sin", "cos", "", ""]:
19                                        formula = f"({w1}({item1}){op1}({w2}({item2}))({op2})({w3})({item3})({op3})({w4})({item4})"
20                                        value = eval(formula)
21                                        if(value > -2.74488 and value < -2.74487): #-2.74487468
22                                            print(item1, item2, item3, item4, formula + " = " + str(value))
23
24        continue
25    for item5 in [-1.740287, 1.496323, -1.34196, 0.4814, 0.085331, -1.322489, -0.993072, -0.14655]:
26        if item5 == item1: continue
27        if item5 == item3: continue
28        for item6 in [0.022889, 0.268136, 1.912595, 0.725914, 2.110477, 0.772424, -1.463241, 0.206366]:
29            if item6 == item2: continue
30            if item6 == item4: continue
31            for item7 in [-1.740287, 1.496323, -1.34196, 0.4814, 0.085331, -1.322489, -0.993072, -0.14655]:
32                if item7 == item1: continue
33                if item7 == item3: continue
34                if item7 == item5: continue
35                for item8 in [0.022889, 0.268136, 1.912595, 0.725914, 2.110477, 0.772424, -1.463241, 0.206366]:
36                    if item8 == item2: continue
37                    if item8 == item4: continue
38                    if item8 == item6: continue
39                    for op1 in ["+", "-", "*", "/"]:
40                        for op2 in ["+", "-", "*", "/"]:
41                            formula = f"({item1}{op1}{item2})({op2})({item3}{op1}{item4})({op2})({item5}{op1}{item6})({op2})({item7}{op1}{item8})"
42                            value = eval(formula)
43                            if(value > -2.74488 and value < -2.74487): #-2.74487468
44                                print(item1, item2, item3, item4, item5, item6, item7, item8, formula + " = " + str(value))
45
46 for item1 in [-1.740287, 1.496323, -1.34196, 0.4814, 0.085331, -1.322489, -0.993072, -0.14655]:
47     for item2 in [0.022889, 0.268136, 1.912595, 0.725914, 2.110477, 0.772424, -1.463241, 0.206366]:
48         processitem(item1, item2)
49 #Parallel(n_jobs=32)(delayed(processItem)(item1, item2) for item2 in [0.022889, 0.268136, 1.912595, 0.725914, 2.110477, 0.772424, -1.463241, 0.206366])

```

Abbildung 26: Alternativtests für R2C mit C2C

Dies gelang mir nicht. Darum habe ich mich direkt an den Programmierer von vkFFT gewendet.

R2C/C2R 1D 2^{12} limitation #23

(Open) nicoboss opened this issue 6 days ago · 1 comment

nicoboss commented 6 days ago • edited

I'm currently switching from cuFFT to VkFFT. I'm using 2^{27} large single precision 1D R2C/C2R transformations in cuFFT. In vkFFT single precision 1D R2C/C2R is unfortunately limited to 2^{12} . Are there any plans to support larger one-dimensional R2C/C2R transformations? If so, I would highly appreciate it. The limitation seems to have to do with the amount of available shared memory but I don't understand why it can't be dealt with using multiple axis uploads like done on C2C.

Until this is supported should I just use C2C and fill the complex part with zeros or are there any more efficient ways? It seems like a huge waste of memory access time having to read and write useless zeros from global memory. Could it be possible to only read the real part from memory and just hardcode the complex part to always be zero on FFT and discard the complex part on IFFT? I'm working on a science project about privacy amplification in quantum cryptography where performance is very important. Thanks a lot for making VkFFT. Without it I would be stuck on NVidia GPUs. I'm looking forward to GTC 2021.

DTolm commented 6 days ago

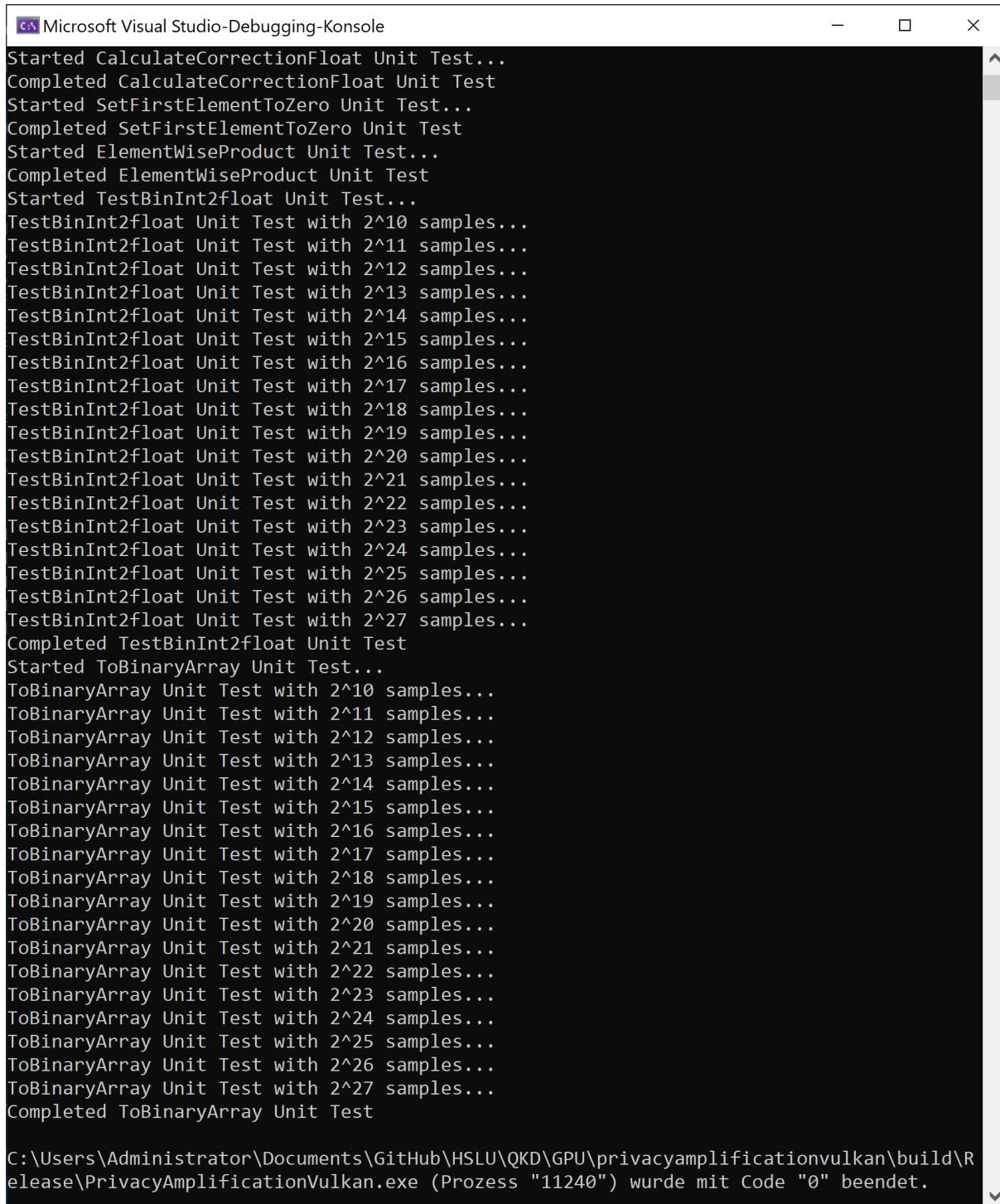
I have a long history with multiple upload R2C/C2R and I have spent a lot of time thinking on how to do this properly. The main problem is actually doing the switch step in multiple uploads, which exposes the symmetry. It gets shifted by one element for even sequences and this breaks the memory access pattern. Doing R2C by simple padding of complex part will only partially solve the issue. It will work for the first upload of the big sequence, while the consecutive will require full C2C memory. Unless there is also a way to expose the symmetry after the first upload. I really hope I will find a good solution one day. This is certainly on the to do list.
Best regards,
Dmitrii Tolmachev

Abbildung 27: Antwort des Programmierers auf Limitation

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplificationvulkan/commit/efdf09f5ddac80636cdcebb6babf97cec38bb761>

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplificationvulkan/commit/71cd0c6a5f8651a888bcd8d0a59143aa26b56ce0>

39. Alle Tests auf Vulkan erfolgreich!



The screenshot shows the Microsoft Visual Studio Debugging Console window. The title bar reads "Microsoft Visual Studio-Debugging-Konsole". The console output displays a series of test logs for Vulkan, indicating the start and completion of various unit tests. The tests include CalculateCorrectionFloat, SetFirstElementToZero, ElementWiseProduct, TestBinInt2float, ToBinaryArray, and several sub-tests for each, involving different sample sizes (2^10 to 2^27). The log concludes with the message "C:\Users\Administrator\Documents\GitHub\HSLU\QKD\GPU\privacyamplificationvulkan\build\Release\PrivacyAmplificationVulkan.exe (Prozess "11240") wurde mit Code "0" beendet.", which translates to "PrivacyAmplificationVulkan.exe (Process "11240") was terminated with code "0".

```
Started CalculateCorrectionFloat Unit Test...
Completed CalculateCorrectionFloat Unit Test
Started SetFirstElementToZero Unit Test...
Completed SetFirstElementToZero Unit Test
Started ElementWiseProduct Unit Test...
Completed ElementWiseProduct Unit Test
Started TestBinInt2float Unit Test...
TestBinInt2float Unit Test with 2^10 samples...
TestBinInt2float Unit Test with 2^11 samples...
TestBinInt2float Unit Test with 2^12 samples...
TestBinInt2float Unit Test with 2^13 samples...
TestBinInt2float Unit Test with 2^14 samples...
TestBinInt2float Unit Test with 2^15 samples...
TestBinInt2float Unit Test with 2^16 samples...
TestBinInt2float Unit Test with 2^17 samples...
TestBinInt2float Unit Test with 2^18 samples...
TestBinInt2float Unit Test with 2^19 samples...
TestBinInt2float Unit Test with 2^20 samples...
TestBinInt2float Unit Test with 2^21 samples...
TestBinInt2float Unit Test with 2^22 samples...
TestBinInt2float Unit Test with 2^23 samples...
TestBinInt2float Unit Test with 2^24 samples...
TestBinInt2float Unit Test with 2^25 samples...
TestBinInt2float Unit Test with 2^26 samples...
TestBinInt2float Unit Test with 2^27 samples...
Completed TestBinInt2float Unit Test
Started ToBinaryArray Unit Test...
ToBinaryArray Unit Test with 2^10 samples...
ToBinaryArray Unit Test with 2^11 samples...
ToBinaryArray Unit Test with 2^12 samples...
ToBinaryArray Unit Test with 2^13 samples...
ToBinaryArray Unit Test with 2^14 samples...
ToBinaryArray Unit Test with 2^15 samples...
ToBinaryArray Unit Test with 2^16 samples...
ToBinaryArray Unit Test with 2^17 samples...
ToBinaryArray Unit Test with 2^18 samples...
ToBinaryArray Unit Test with 2^19 samples...
ToBinaryArray Unit Test with 2^20 samples...
ToBinaryArray Unit Test with 2^21 samples...
ToBinaryArray Unit Test with 2^22 samples...
ToBinaryArray Unit Test with 2^23 samples...
ToBinaryArray Unit Test with 2^24 samples...
ToBinaryArray Unit Test with 2^25 samples...
ToBinaryArray Unit Test with 2^26 samples...
ToBinaryArray Unit Test with 2^27 samples...
Completed ToBinaryArray Unit Test

C:\Users\Administrator\Documents\GitHub\HSLU\QKD\GPU\privacyamplificationvulkan\build\Release\PrivacyAmplificationVulkan.exe (Prozess "11240") wurde mit Code "0" beendet.
```

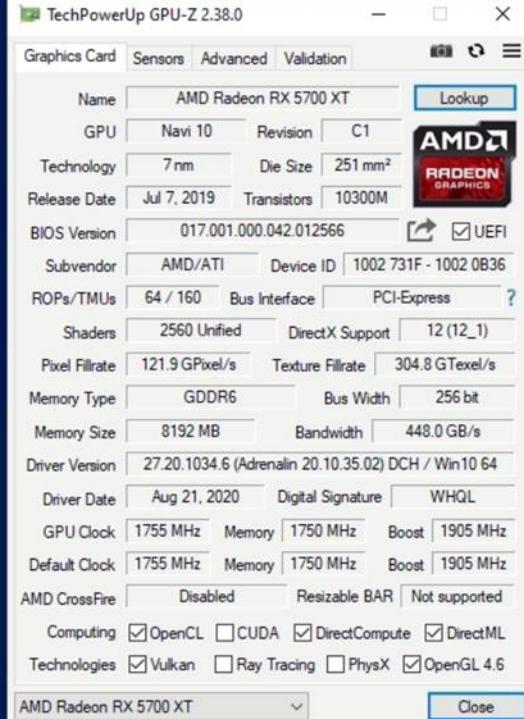
Abbildung 28: Erfolgreiche Tests

IV Woche 7 + 8

40. Alle Tests auf AMD Radeon RX 5700 XT erfolgreich!

```
Windows PowerShell
-0.098422 -0.295755 -0.885922 0.215369 0.566637 0.605213 0.039766 -0.396100
0.751945 0.453352 0.911801 0.851436 0.078707 -0.715323 -0.075839 -0.529344
0.724479 -0.580798 0.559313 0.687307 0.993591 0.999390 0.222999 -0.215125
-0.467574 -0.405438 0.680288 -0.952513 -0.248268 -0.814753 0.354411 -0.887570

Benchmark score: 66.699220
-0.997498 0.127171 -0.613391 0.617481 0.170019 -0.040254 -0.299417 0.791925
0.645680 0.493209 -0.651784 0.717887 0.421003 0.027070 -0.392010 -0.970030
-0.817194 -0.271096 -0.705374 -0.668203 0.977050 -0.108615 -0.761833 -0.990661
-0.982177 -0.244240 0.063326 0.142369 0.203528 0.214332 -0.667531 0.326090
-0.098422 -0.295755 -0.885922 0.215369 0.566637 0.605212 0.039766 -0.396100
0.751945 0.453352 0.911801 0.851436 0.078707 -0.715323 -0.075839 -0.529344
0.724479 -0.580798 0.559313 0.687307 0.993591 0.999389 0.222999 -0.215125
-0.467574 -0.405438 0.680288 -0.952513 -0.248268 -0.814752 0.354411 -0.887569
Started CalculateCorrectionFloat Unit Test...
Completed CalculateCorrectionFloat Unit Test
Started SetFirstElementToZero Unit Test...
Completed SetFirstElementToZero Unit Test
Started ElementWiseProduct Unit Test...
Completed ElementWiseProduct Unit Test
Started TestBinInt2float Unit Test...
TestBinInt2float Unit Test with 2^10 samples...
TestBinInt2float Unit Test with 2^11 samples...
TestBinInt2float Unit Test with 2^12 samples...
TestBinInt2float Unit Test with 2^13 samples...
TestBinInt2float Unit Test with 2^14 samples...
TestBinInt2float Unit Test with 2^15 samples...
TestBinInt2float Unit Test with 2^16 samples...
TestBinInt2float Unit Test with 2^17 samples...
TestBinInt2float Unit Test with 2^18 samples...
TestBinInt2float Unit Test with 2^19 samples...
TestBinInt2float Unit Test with 2^20 samples...
TestBinInt2float Unit Test with 2^21 samples...
TestBinInt2float Unit Test with 2^22 samples...
TestBinInt2float Unit Test with 2^23 samples...
TestBinInt2float Unit Test with 2^24 samples...
TestBinInt2float Unit Test with 2^25 samples...
TestBinInt2float Unit Test with 2^26 samples...
TestBinInt2float Unit Test with 2^27 samples...
Completed TestBinInt2float Unit Test
Started ToBinaryArray Unit Test...
ToBinaryArray Unit Test with 2^10 samples...
ToBinaryArray Unit Test with 2^11 samples...
ToBinaryArray Unit Test with 2^12 samples...
ToBinaryArray Unit Test with 2^13 samples...
ToBinaryArray Unit Test with 2^14 samples...
ToBinaryArray Unit Test with 2^15 samples...
ToBinaryArray Unit Test with 2^16 samples...
ToBinaryArray Unit Test with 2^17 samples...
ToBinaryArray Unit Test with 2^18 samples...
ToBinaryArray Unit Test with 2^19 samples...
ToBinaryArray Unit Test with 2^20 samples...
ToBinaryArray Unit Test with 2^21 samples...
ToBinaryArray Unit Test with 2^22 samples...
ToBinaryArray Unit Test with 2^23 samples...
ToBinaryArray Unit Test with 2^24 samples...
ToBinaryArray Unit Test with 2^25 samples...
ToBinaryArray Unit Test with 2^26 samples...
ToBinaryArray Unit Test with 2^27 samples...
Completed ToBinaryArray Unit Test
PS C:\Users\nico\OneDrive\Öffentlich\Dokumente\privacyamplificationvulkan\build>
```



The screenshot shows the GPU-Z 2.38.0 application window. The 'Graphics Card' tab is selected, displaying the following details for the AMD Radeon RX 5700 XT:

- Name: AMD Radeon RX 5700 XT
- GPU: Navi 10
- Technology: 7 nm
- Release Date: Jul 7, 2019
- BIOS Version: 017.001.000.042.012566
- Subvendor: AMD/ATI
- ROPs/TMUs: 64 / 160
- Shaders: 2560 Unified
- Pixel Fillrate: 121.9 GPixel/s
- Memory Type: GDDR6
- Memory Size: 8192 MB
- Driver Version: 27.20.1034.6 (Adrenalin 20.10.35.02) DCH / Win10 64
- Driver Date: Aug 21, 2020
- GPU Clock: 1755 MHz
- Default Clock: 1755 MHz
- AMD CrossFire: Disabled
- Computing Technologies: OpenCL (checked), CUDA (unchecked), DirectCompute (checked), DirectML (checked)
- Technologies: Vulkan (checked), Ray Tracing (unchecked), PhysX (unchecked), OpenGL 4.6 (checked)

Abbildung 29: PrivacyAmplificationVulkan

41. Vulkan-spezifische Bibliotheken zu PrivacyAmplification portiert

Vulkan-spezifische Bibliotheken wie Vuda, VkFFT, glslang und half_lib wurden vom PrivacyAmplificationVulkan Projekt zum PrivacyAmplification portiert. Um als ersten Schritt die beiden Projekte wieder zu vereinen. Wieder zurück zu einem einzigen Projekt zu gehen ist sehr wichtig, da sonst alles doppelt gemacht werden muss. Zudem müsste es mit den korrekten Einstellungen möglich sein, alle Kompilierungsvorteile von Vulkan gegenüber Cuda zu behalten. Durch diesen Commit wurden dem Projekt 406'607 Zeilen Code und 2'472 neue Dateien hinzugefügt. Das Meiste davon stammte aus glslang, welches zur Kompilierung von GLSL shaders zur Laufzeit während des Planens von VkFFT benötigt wird.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/4c8a53098630d55949c1cf38c32368c319e3f251>

42. Vulkan und Cuda Projekt Kerncode zusammenführen

In einem weiteren Schritt um beide Projekte zu vereinen, musste auch der PrivacyAmplification Kerncode vereint werden. Dies ist der Code welcher die Cuda Kernels und GLSL Shaders aufruft. Durch die zuvor portierten Unit Tests war ein Grossteil der Arbeit und das Testen auf Korrektheit glücklicherweise schon gemacht. Die Hauptaufgabe war es also mit Preprocessorstatements den Programmfluss je nach der zu kompilierenden Plattform auf das richtige Compute-Kernel zu leiten. Da Vuda kein cudaMemcpyToSymbol kennt mussten für alle Konstanten eine Vuda kompatible Lösung gesucht werden. Auch alle Streams vom GPU Pipelining mussten umgeschrieben werden, um unter Vuda zu funktionieren.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/978ed6d2f3b4e13e2ced9bd34a15369386996989>

43. cuFFT zurück vom C2C Test zu R2C/C2R

Nach dem Test vor einigen Tagen stellte ich fest, dass der Algorithmus mit der Verwendung von cuFFT mit einer C2C – FFT nur noch etwas mehr als halb so schnell ist als mit R2C / C2R – FFT. Ich habe dies nun wieder rückgängig gemacht, da es dazu in Cuda absolut kein Grund gibt. Dies da, anders als VkFFT, Cuda R2C/C2R mit grosser Blockgrösse problemlos unterstützt.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/73e72b0126238a7474d388e19e24a65872d30904>

44. VkFFT unterstützt neu R2C/C2R mit grossen Blöcken!

R2C/C2R 1D 2^{12} limitation #23

 **Closed** nicoboss opened this issue 20 days ago · 3 comments

 **nicoboss** commented 20 days ago · edited

I'm currently switching from cuFFT to VkFFT. I'm using 2^{27} large single precision 1D R2C/C2R transformations in cuFFT. In vkFFT single precision 1D R2C/C2R is unfortunately limited to 2^{12} . Are there any plans to support larger one-dimensional R2C/C2R transformations? If so, I would highly appreciate it. The limitation seems to have to do with the amount of available shared memory but I don't understand why it can't be dealt with using multiple axis uploads like done on C2C.

Until this is supported should I just use C2C and fill the complex part with zeros or are there any more efficient ways? It seems like a huge waste of memory access time having to read and write useless zeros from global memory. Could it be possible to only read the real part from memory and just hardcode the complex part to always be zero on FFT and discard the complex part on IFFT? I'm working on a science project about privacy amplification in quantum cryptography where performance is very important. Thanks a lot for making VkFFT. Without it I would be stuck on NVidia GPUs. I'm looking forward to GTC 2021.

 **DToIm** commented 20 days ago

Owner  ...

I have a long history with multiple upload R2C/C2R and I have spent a lot of time thinking on how to do this properly. The main problem is actually doing the switch step in multiple uploads, which exposes the symmetry. It gets shifted by one element for even sequences and this breaks the memory access pattern.

Doing R2C by simple padding of complex part will only partially solve the issue. It will work for the first upload of the big sequence, while the consecutive will require full C2C memory. Unless there is also a way to expose the symmetry after the first upload.

I really hope I will find a good solution one day. This is certainly on the to do list.

Best regards,
Dmitrii Tolmachev

 **DToIm** commented 10 days ago

Owner  ...

Hello,

In the last update, I have added support for multi-upload R2C/C2R sequences - they should have the same limits as the C2C case now. It only works for even sequences for now (non-powers of 2 are supported). Updated sample_15 validates the results of R2C+C2R. If you have other ideas for improvement - feel free to share them!

Best regards,
Dmitrii

 1  1  1

 **nicoboss** commented 10 days ago · edited

Author  ...

Hello Dmitrii

Thank you so much. You put a lot of work into this! It is absolutely awesome and exactly what I needed. I'm so happy that VkFFT now supports large 1D R2C/C2R. I already tested it and it works perfectly. I will switch to VkFFT in the following days as now it has only advantages over cuFFT.

Best wishes,
Nico Bossard

  **nicoboss** closed this 10 days ago

Abbildung 30: Rückmeldung von Tolmachev

Dies kam absolut unerwartet. 10 Tage nachdem ich Dmitrii Tolmachev gefragt habe ob er multi upload R2C/C2R FFTs in VkFFT implementieren könnte, sah ich plötzlich diesen commit:



Abbildung 31: Gefixtes Issue

Dmitrii Tolmachev hat einfach so mal in 10 Tagen 5'129 Codezeilen geschrieben um genau das zu implementieren, was ich mir für dieses Projekt so wünschte. Ich war überglücklich! Sowas hatte ich noch nie erlebt. Es ist für mich absolut unverständlich wie er es überhaupt hingekriegt hat in 10 Tagen so viel Code zu schreiben. Unvorstellbar wie viele Stunden er dafür gearbeitet habe muss. Ich weiss ja aus eigener Erfahrung wie umständlich es ist undebuggbaren GLSL Code zu schreiben. Und er macht das Ganze noch zur Laufzeit durch Metaprogrammierung. Ich war so sprachlos, dass ihm unbedingt zeigen wollte wie sehr ich mich freue und seine Arbeit schätze. Alles Andere was ich an diesem Tag machen wollte wurde verschoben, da ich sofort mit Testen beginnen wollte. Das updaten auf diese VkFFT Version war recht einfach und am vkFFT_helper.h brauchte es nur eine kleine Anpassung um die in diesem Update auch noch hinzugefügten launchParams zu unterstützen. Durch die klare Trennung von meinem Code und der Schnittstelle zu VkFFT war dies auch kein Problem.

An dieser Stelle nochmals vielen Dank an Dmitrii Tolmachev für die Implementierung von R2C/C2R mit grossen Blockgrössen von bis zu 2^{32} in VkFFT.

<https://github.com/DTolm/VkFFT/commit/9590385361c21004f22fdd0e9dad6cc079a945df>

<https://github.com/DTolm/VkFFT/commit/c890960771cba714bf0610f8e76c5fa2174865b6>

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/0062c97dfb2aaaed7d53b493c03b0f6ce222b15a>

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplificationvulkan/commit/5d570a867e9dc153c8f734ca94919d51055dbe9c>

45. Script zur sauberen Darstellung der Git History hinzugefügt

Um besser die Bibliotheken und Lizenzen verwalten zu können und zum Nachschlagen was wann gemacht wurde, habe ich ein kleines Shell script zum Ausgeben einer sauber dargestellten git History erstellt.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/22405edce09d26663ee1e97eba18614d5ccc9f22>

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplificationvulkan/commit/aa9863ee0f161f30d20edd6d83df6f4b008fb4d8>

46. Lizenzen aktualisiert

Durch das zuvor erstellte Script war es mir nun möglich alle Lizenzen in der LICENSE Datei zu aktualisieren.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplificationvulkan/commit/dd0171fb3686f18f5d8fce5acf08c306b11d3dc4>

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplificationvulkan/commit/586e74668d6376be44fcf97fbacafb9df94fbb2c>

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/9f47ad4dea126b10d47dd06fcdd2b3704ee20ac6>

47. Verlinken von glslang mit PrivacyAmplification VS - Projekt

Bevor es mit dem Kombinieren von Cuda und Vulkan weitergehen kann muss erst mal glslang korrekt mit dem PrivacyAmplification Visual Studio Projekt verlinkt werden. Nach einigen Untersuchungen stellte sich heraus, dass das Hauptproblem mit der standardmäßig von glslang verwendete run-time library zu tun hat. Standardmäßig wird glslang mit /MD und /MDd kompiliert. Dies ist für das Linken über DLLs und nicht das statische Linken wie ich es hier verwende bestimmt. Durch das Anpassen von CMakeLists.txt gelang es mir von glslang ein mit /MT und /MTd buildendes Visual Studio Projekt zu erstellen. Dies konnte ich dann, wie gewünscht, über AdditionalDependencies statisch mit meinem PrivacyAmplification Visual Studio Projekt verlinken.

Option	Description
/MD	Causes the application to use the multithread-specific and DLL-specific version of the run-time library. Defines <code>_MT</code> and <code>_DLL</code> and causes the compiler to place the library name MSVCRT.lib into the .obj file. Applications compiled with this option are statically linked to MSVCRT.lib. This library provides a layer of code that enables the linker to resolve external references. The actual working code is contained in MSVCRversionnumber.DLL, which must be available at run time to applications linked with MSVCRT.lib.
/MDd	Defines <code>_DEBUG</code> , <code>_MT</code> , and <code>_DLL</code> and causes the application to use the debug multithread-specific and DLL-specific version of the run-time library. It also causes the compiler to place the library name MSVCRTD.lib into the .obj file.
/MT	Causes the application to use the multithread, static version of the run-time library. Defines <code>_MT</code> and causes the compiler to place the library name LIBCMT.lib into the .obj file so that the linker will use LIBCMT.lib to resolve external symbols.
/MTd	Defines <code>_DEBUG</code> and <code>_MT</code> . This option also causes the compiler to place the library name LIBCMTD.lib into the .obj file so that the linker will use LIBCMTD.lib to resolve external symbols.

Abbildung 32: Ausschnitt aus Dokumentation über Kompilierungsflags

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/bca8c5a18436db7f3ddcee87088e1d78118e2560>

48. Angefangen Vuda and vkFFT miteinander kompatibel zu machen

Um Vuda mit VkFFT kompatibel zu machen musste in einem ersten Schritt die Vulkan Instanz der beiden vereinigt werden. Da Vuda eine höhere Abstraktion aufweist als VkFFT entschied ich mich aus Vuda die Vulkan Instanz durch hinzufügen von weniger abstrakten Funktionen zu ermöglichen. Diese beinhaltet unter anderem: logical_device::GetQueue welches ein vk::Queue zurückgibt, logical_device::GetPool welches ein thrcmdpool* zurückgibt von welchem wiederum die neuen GetCommandPool und GetFence Fionktion welche vk::CommandPool und vk::Fence zurückgeben aufgerufen werden können. Diese so erhaltenen Werte können in das VkGPU struct gespeichert werden, wo sie von VkFFT verwendet werden können, ohne dass eine neue Vulkan Instanz erstellt werden muss.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplificationvulkan/commit/32514af05c3c3779cd23cd53a8e4c62714c66432>

49. vkFFT mit Vuda Speichermanagement kompatibel machen

Das Hauptproblem beim Zusammenführen ist, dass Vuda ein ganz andres Grafikspeichermanagementsystem verwendet als vkFFT. Es ist wichtig, dass dieses Speichermanagement kompatibel miteinander gemacht wird, da sonst eine Zusammenarbeit der beiden Bibliotheken unmöglich ist. Könnte ich diesen Stolperstein nicht lösen, müsste das AMD-Projekt, respektiv Vuda oder vkFFT aufgegeben werden.

Vuda nutzt einen selbstbalancierenden Binärbaum – vkFFT verwendet VkBuffer, der einen low level Speicherzugriff auf die Grafikkarte ermöglicht.

Glücklicherweise meisterte ich diese grosse Herausforderung. Der nachfolgende Codeausschnitt zeigt wie ein VkBuffer aus einem Vuda-Suchbaum extrahiert wird.

```
inline VkBuffer logical_device::GetBuffer(void* ptr)
{
    std::shared_lock<std::shared_mutex> lckResources(*m_mtxResources);
    const default_storage_node* dst_node = m_storage.search_range(m_storageBST_root, ptr);
    return dst_node->GetBuffer();
}
```

Abbildung 33: VkBuffer aus Vuda-Suchbaum extrahieren

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplificationvulkan/commit/89a18db8ca3f0b8cc658cdfb8bbaba6294bee09a>

50. VK_DEVICE_LOST testen

Um die Ursache der Fehlermeldung «VK_DEVICE_LOST-Test» zu finden, habe ich für 3 Stunden auf NVidia RTX 3080 und AMD Radeon RX 5700 XT den Code in einer Endlosschlaufe laufen lassen. Es gab keine Abstürze. Der Fehler tritt nur auf, wenn aktiv an diesem PC mit anderen Programmen parallel gearbeitet wird und die GPU gleichzeitig für verschiedene Aufgaben verwendet wird.

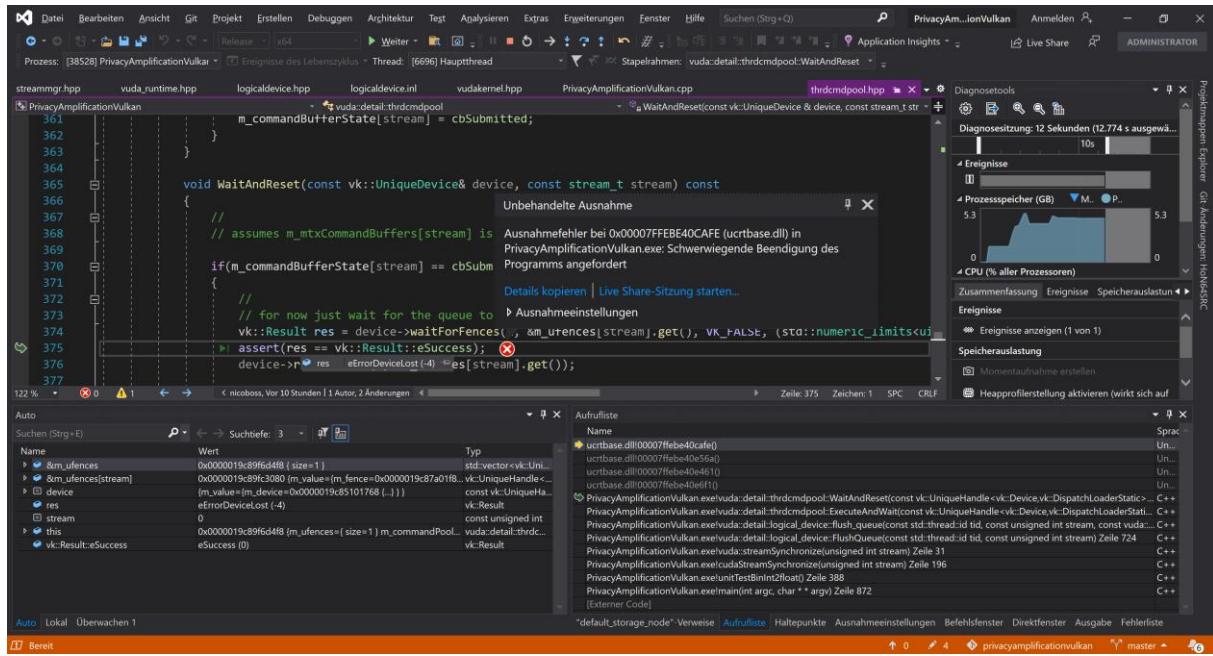


Abbildung 34: Fehlermeldung VK_DEVICE_LOST

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplificationvulkan/commit/cc2f94120766ea6e69979b419d46d14a48c91d24>
<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplificationvulkan/commit/dde330bd988abce7decf2931140ae3d00dcba5a4>

51. Nicht zwingend notwendiger VkFFT-Code entfernt

Um PrivacyAmplificationVulkan.cpp im Projekt PrivacyAmplificationVulkan aufzuräumen wurde aller nicht zwingend notwendiger VkFFT-Code entfernt.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplificationvulkan/commit/47a65fe95d25c20f41a55dc349e8c6c5f6119fbe>
<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplificationvulkan/commit/5fdfab1adfcab89345106b0ccce0636b132cc3d>
<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplificationvulkan/commit/57133d5eaffcb4e970811d587fba982c87170b18>

52. Meine verbesserte Vuda-Version zu PrivacyAmplification portiert

Ich habe meine verbesserte Vuda-Version von PrivacyAmplificationVulkan zur PrivacyAmplification portiert.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/2c5d9f675cce2ce01b70b9ac291cd6bddf4088>

53. Bereinigter vkFFT_helper nach PrivacyAmplification portiert

Portierung des bereinigten vkFFT_helper von PrivacyAmplificationVulkan nach PrivacyAmplification.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/cfbc7b3d26324c560eec7122724d173762d11c1e>

54. GLSL-Shaders nach PrivacyAmplification portiert

Portierung des GLSL-Shader von PrivacyAmplificationVulkan zu PrivacyAmplification.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/bce8d54441f29252b2e6cd937042c80e7deb3fa5>

55. vkFFT-Steuercode nach PrivacyAmplification portiert

Ich habe den vkFFT-Steuercode von PrivacyAmplificationVulkan nach PrivacyAmplification portiert.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/2a9e3ce2aa2ef4736f28f9558964e4fbaead9dc3>

56. PrivacyAmplification kompiliert mit Vulkan!

Nach vielen Änderungen, wie beispielsweise dem Auskommentieren von half_lib und rhash oder dem Implementieren von memset.comp, gelang es mir endlich die PrivacyAmplification mit Vulkan zu kompilieren!

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/2b64c0c73e6ae3c025ab9698344e2f2e929251c6>

57. Fehler mit VK_DEVICE_LOST behoben

Ich fand nun einen Weg, um den Absturz von VK_DEVICE_LOST einigermassen zuverlässig zu reproduzieren. Das GPU-Pipelining ist in Vuda offenbar defekt. Dies ist aber eigentlich egal, da ich als Kerneleingang auf gepinnten Speicher umgestellt habe. Auf diesem kann immer vom CPU-Code zugegriffen werden, auch wenn GPU-Code am Laufen ist. Es war nur ein bisschen traurig, dass es so lange gedauert hat, bis ich den Fehler gefunden habe. Dies passierte, weil ich keinen Pipeline-Tests starten konnte, sondern nur die Unit-Tests, da ich mich mitten im Übergang nach Vulkan befand.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/e49b83a7916389e3f302c49cbc413e958e173ffd>

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/c1d9fd0ba72130bf19710ba729ff98a6e77043a1>

58. Alle Vuda-Dateien zum Visual Studio-Projekt hinzugefügt

Ich habe nun alle Vuda- Dateien zum PrivacyAmplification Visual Studio-Projekt hinzugefügt.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/f8090b442473249403f6eeabdd9471bf915f0f5>

59. vkFFT in den PrivacyAmplification Algorithmus integriert

Überall wo im PrivacyAmplification-Core-Algorithmus bei CUDA cuFFt verwendet wird, wird in Vulkan ab jetzt vkFFT verwendet. Alle notwendigen Hilfsfunktionen für vkFFT wurden programmiert.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/3c406890b20c33ec5e393fe46c8a76d4720f831b>

60. Assert, RHash und VIDEO_SCHEDULER_INTERNAL_ERROR

Ich habe Assert in Vulkan GPU-Code implementiert und RHash aktualisiert. Mein Programm lässt nun plötzlich meinen ganzen PC crashen. Folgende Errormeldung erscheint:
VIDEO_SCHEDULER_INTERNAL_ERROR

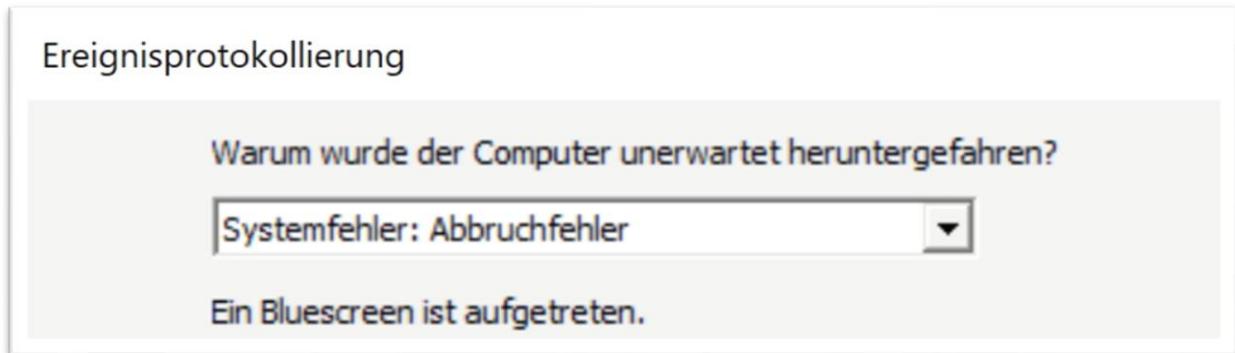


Abbildung 35: Ereignisprotokoll

Your computer crashed or a problem was reported

crash dump file: C:\Windows\Minidump\041521-13500-01.dmp

This was probably caused by the following module: [watchdog.sys](#) (watchdog+0x3BF3)

Bugcheck code: 0x119 (0x1, 0x14CA, 0x14E9, 0xFFFFFAD867CF5B000)

Error: [VIDEO_SCHEDULER_INTERNAL_ERROR](#)

file path: C:\Windows\system32\drivers\watchdog.sys

product: [Microsoft® Windows® Operating System](#)

company: [Microsoft Corporation](#)

description: Watchdog Driver

Bug check description: This indicates that the video scheduler has detected a fatal violation.

The crash took place in a Microsoft module. Your system configuration may be incorrect. Possibly this problem is caused by another driver on your system that cannot be identified at this time.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/76e773b6a49e016cfb5f52f09accba6a6ce382f6>

61. VIDEO_SCHEDULER_INTERNAL_ERROR BoD behoben

Da es unakzeptabel war, dass mein Programm andauernd mein Haupt PC crashte, erstellte ich eine neue VM in meiner Proxmox Umgebung. Dort installierte ich alle notwendigen Tools um solche Bluescreen of Death crashes zu reproduzieren und zu untersuchen. Nach langem Suchen mittels auskommentieren von Codeabschnitten und testen, wo der fehlerhafte Codeteil liegt, fand ich den Fehler heraus. Es lag an der fehlerhaften Nummerierung der bindigs des memory layouts des GLSL assert kernels.

```
5      5    layout(set = 0, binding = 0) readonly buffer A { uint32_t data[]; };
6      -layout(set = 0, binding = 2) readonly buffer B { uint32_t value[]; };
7      -layout(set = 0, binding = 3) writeonly buffer C { uint32_t returnValue[]; };
6    +layout(set = 0, binding = 1) readonly buffer B { uint32_t value[]; };
7    +layout(set = 0, binding = 2) writeonly buffer C { uint32_t returnValue[]; };
8      8
```

Abbildung 36: Video_Scheduler_Internal_Fehlerbehebung

Virtual Machine 118 (PrivacyAmplification-VIDEO-SCHEDULER-INTERNAL-ERROR-BoD) on node		
Summary	Add	Remove
>_ Console	Memory	16.00 GiB [balloon=0]
Hardware	Processors	64 (1 sockets, 64 cores) [host,flags=+aes]
Cloud-Init	BIOS	OVMF (UEFI)
Options	Display	none (none)
Task History	Machine	q35
Monitor	SCSI Controller	VirtIO SCSI single
Backup	Hard Disk (scsi0)	local-lvm:vm-118-disk-0,size=128G
Replication	Network Device (net0)	virtio=F6:6C:AB:10:B8:C6,bridge=vmbr0,firewall=1
Snapshots	EFI Disk	local-lvm:vm-118-disk-1,size=4M
Firewall	PCI Device (hostpci0)	22:00.0,rombar=0
Permissions	PCI Device (hostpci1)	23:00,pcie=1,x-vga=1

Abbildung 37: Hardware der Testing VM

Name	R...	Date/Status
└ ↻ Clean	No	2021-02-08 11:06:48
└ ↻ NewClean	No	2021-04-15 15:42:58
└ ↻ PrivacyAmplification	No	2021-04-16 13:42:31
└ ↻ NoMoreiCUE	No	2021-04-16 14:33:34
└ 🖥 NOW		

Abbildung 38: Snapshots der Testing VM

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/8761e5b88abd274ee1b3bd11c4cc314cf4de1392>

62. PrivacyAmplification VS-Projekt kompilierbar für Cuda und Vulkan

Die PrivacyAmplification-Codebasis kann jetzt für Cuda und Vulkan kompiliert werden. Dies wurde durch die Verwendung von zwei verschiedenen Visual Studio-Projekten in einer Visual Studio Solution gelöst.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/a3ec64242643c25dbf8c3720c8b3c4a731a322e3>

63. Nutzloses GPU-Pipelining entfernt

Das nutzlose GPU-Pipelining wurde entfernt, wodurch die Lesbarkeit des Codes erheblich verbessert wurde.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/dee44c450814d2b0f9e890a9f0f82a01f06b0b0a>

64. FFT-Ergebnis-Debugging

VkFFT scheint nach der Hälfte der Ausgabedaten aufgrund der Symmetrie der R2C-FFT's anzuhalten.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/17f41cbd8f7e6e3919ec315b0333ae4e332b5fc3>

65. cudaMalloc und cudaCalloc repariert

Ich habe den Fehler von cudaMalloc und cudaCalloc auf Vulkan behoben. Es wurde ein weiterer Debugging-Code für FFT-Ergebnisse hinzugefügt.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/4acdcd7249c00d66c96fd67d1536580b2d9304f0>

66. Ergebnis von vkFFt und cuFFT sind unterschiedlich

Testen, warum das IFFT-Ergebnis von vkFFT nicht mit dem von cuFFT übereinstimmt. Der Debugging-Code für das FFT-Ergebnis wurde vom main Branch entfernt, da sich dieser jetzt im fft_testing Branch befindet.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/5fa3159feb5381391512719f6faee616a5f0aa1a>

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/edada04a17de1816664efcd20d9e6a2fd0236f1b>

67. 0-Hz Test mit kleinem Zahlenbeispiel

Erster 0-HzTest mit den Zahlen von 1 bis 16, der zu einer Korrektur von (buffer_output [i] +136) / 16) führte.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplificationvulkan/commit/7fb29761fdccd77e0a501a7bfe074bc97c725a2d>

68. 0-Hz Test mit realen, grossen Zahlen

Ich habe nun den PrivacyAmplificationVulkan Core-Algorithmus mit realen, grossen Blockgrössen getestet, mit dem besonderen Schwerpunkt auf dem Testen von 0 Hz.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplificationvulkan/commit/7a6ca507f8750e969178d803d2bc301441e2ca8a>

69. In-Place VkFFT

Den Algorithmus habe ich nun in-Place VkFFT anstelle von out-Place umgeschrieben. Dies führt zu einer enormen GPU-Speichereinsparungen. Ich brauche jetzt nur noch etwa 1/3 des Speichers.

Ich verbesserte die Lesbarkeit der VkFFT-Konfiguration durch das Verschieben von einem Template in eine Inline-Funktion.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/3d3a85d336db1199a24daf13a27ddb8c41d2991>

70. Korrektes Vulkan IFFT-Ergebnis!

Der PrivacyAmplification-Kernalgorithmus wurde geändert, um ihn mit Vulkan und In-Place FFT kompatibel zu machen. Dieser Commit markiert den Meilenstein, an dem ich gute IFFT-Ausgabedaten erhalten habe. Das assert nach toBinaryArray schlägt jedoch weiterhin fehl. Ich habe auch alle FletcherFloat-Prüfsummen für das exaktere VkTTF-Ergebnis übernommen und die Toleranz erhöht, damit dieselben Prüfsummen auch für cuFFT verwendet werden können.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/397783f1b49e3561318a05ce94523e46a869b4df>

71. PA läuft mit Cuda und mit Vulkan

Der PrivacyAmplification-Kernalgorithmus wurde geändert, um ihn sowohl mit Cuda als auch mit Vulkan kompatibel zu machen. Nun läuft der Algorithmus auch wieder mit Cuda und nicht nur mit Vulkan.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/4d7871d22ea41cfa846ecbf26957f1afeb6913f0>

72. Gleiches Ergebnis in Cuda und Vulkan

Cuda und Vulkan haben genau das gleiche binäre Ergebnis, wenn toBinaryArray auf der CPU ausgeführt wird!

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/f607f57c36ee36c085300255958793b720f00439>

73. Gleiches Ergebnis in Cuda und Vulkan, alles auf GPU

Ich habe nun den PrivacyAmplification-Algorithmus vollständig, also auch den letzten Teil mit dem toBinaryArray ganz auf der GPU laufen lassen. Cuda und Vulkan haben genau das gleiche binäre Ergebnis für den ersten Block! Das Ergebnis meines Vulkan- Algorithmus wurde erfolgreich mit dem SHA3-256 Hash validiert! Es ist somit genau gleich wie mit dem Cuda-Algorithmus. Vorläufige Performancemessungen zeigen, dass der neue Algorithmus mindestens 100 Mbit/s schneller ist als der alte CUDA-Algorithmus.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/9cd3e5882706a0c81b929aa92f9c17bce9abcf5>

74. cudaMemcpy repariert

Der Fehler von Vulkan cudaMemset wurde behoben.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/9b86a2cb0be2fe1936b4bac73ca8d221b5e4e29a>

75. Korrektes Ergebnis auch mit mehreren Blöcken

Ich fand nun heraus, warum nur der erste Block der PA funktionierte. Es hat damit zu tun, dass die Memoryregion `di1`, welche zum Speichern des Keys verwendet wird, schmutzig wird, weil der Algorithmus die gleiche Speicherregion zum Speichern des IFFT-Ergebnisses verwendet. Dies habe ich temporär behoben durch das Reinigen des Speichers nach jedem Durchlauf. In Zukunft möchte ich dies mit `fft_zeropad_left` und `fft_zeropad_right` lösen.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/747078fa7ac322cf1fe9d33aae139a25ad444df7>

76. PrivacyAmplification-Algorithmus funktioniert auf Vulkan!

Voll funktionsfähiger PrivacyAmplification-Algorithmus auf Vulkan!

Abbildung 39: Voll funktionsfähiger PA-Algorithmus auf Vulkan

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/a0f4597c79c41f38b2fb0e40c4340dc77ab71b81>

77. Erste Planung zum Aufteilen in Blöcke!

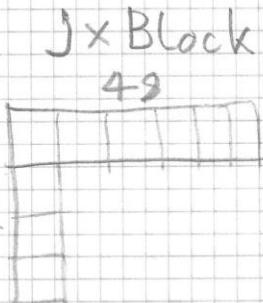
Annahme:

$$\text{Grafikkarte: } 2^4 = 16 = 2B$$

$$M \boxed{N} \quad N=M=B=2^3=8$$

$$n = j \cdot N + m \quad \text{und} \quad m = k \cdot M$$

$$\begin{array}{c} 6 \\ \downarrow \\ 48 \end{array} \quad \begin{array}{c} 4 \\ \uparrow \\ 8 \end{array} \quad \begin{array}{c} 32 \\ \downarrow \\ K \end{array}$$



key start + key rest aufteilen:

$$Z = n = 32 + 48 = 80$$

$$\text{key start} \quad \text{48-Elemente}$$

$$Z_a = Z_0 \dots Z_{n-m-1} = Z_0 \dots Z_{47}$$

Z_d in 6 8er Teile aufteilen

$$i=0 \rightarrow T_0 = Z_{d,0} = Z_0 \dots Z_7$$

$$i=5 \rightarrow T_5 = Z_{d,5} = Z_{40} \dots Z_{47}$$

key Rest Z_b in 4 8-er 32-Elemente

$$Z_b \rightarrow Z_{n-m} \dots Z_{n-1} = Z_{48} \dots Z_{79}$$

$$j=0 \rightarrow J_0 = Z_{b,0} = Z_{48} \dots Z_{55}$$

$$j=3 \rightarrow J_3 = Z_{b,3} = Z_{72} \dots Z_{79} \quad (6+4)-1$$

Seed aufteilen in 9 Blöcke:

$$r_5(i=5, j=0) = \left(\begin{smallmatrix} r_{40} & \dots & r_{43} \\ r_{47} & \dots & r_{41} \end{smallmatrix} \right)$$

$$r_3(i=0, j=3) = \left(\begin{smallmatrix} r_{-24} & \dots & r_{-31} \\ r_{-17} & \dots & r_{-23} \end{smallmatrix} \right)$$

$$\text{ifft}\left(\text{fft}(r_5) * \text{fft}(Z_{a,5})\right) \oplus Z_{b,3}$$

ifft
fft
o

Abbildung 40: Skizze der Planung der Blockaufteilung

V Woche 9 + 10

78. VkFFT Zero Padding

Durch das Benutzen des von VkFFT bereitgestellten Zero padding, ist das Säubern von di1 nicht mehr notwendig. Davor war dies wichtig, da durch Wiederverwendung der di1 Speicherregion durch das iFFT Resultat dreckig wurde.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/0616d1143a031b288a8a2a87da52210cbda5a86c>

79. VkFFT konfiguriert

Durch Verbesserungen der vkFFT-Konfigurationen konnte die Performance verbessert werden.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/04f3a17df9f39c671b99d8a17b3ac22fcb485e56>

80. Benchmarking Tool

Ich habe ein Benchmarking and Profiling-Tool implementiert, um die Bottlenecks meines PA-Algorithmus zu detektieren.

Um Sicherzustellen, dass ich keine falschen Resultate erhalte, habe ich den Vulkan validation layer ausgeschaltet.

Es wird an vielen Orten gemessen und nur Messwerte, die besser sind als die vorangegangenen angezeigt.

Benchmarking: Maximalwerte aus dutzenden von Durchläufen

Function	Cuda	Vulkan	Differnece C2-B2	Differnece % =((C2/B2)*100)-100
wait_for_input_buffer	0.006 ms	0.006 ms	0.000 ms	0.000 %
cleaned_memory	0.003 ms	0.000 ms	-0.003 ms	-100.000 %
set_count_key_to_zero	0.008 ms	0.211 ms	0.203 ms	2537.500 %
set_count_seed_to_zero	0.031 ms	0.222 ms	0.191 ms	616.129 %
binIntffloat_key	2.089 ms	1.949 ms	-0.140 ms	-6.702 %
binIntffloat_seed	3.160 ms	3.205 ms	0.045 ms	1.424 %
calculateCorrectionFloat	0.323 ms	0.169 ms	-0.154 ms	-47.678 %
fft_key	7.466 ms	6.561 ms	-0.905 ms	-12.122 %
fft_seed	7.881 ms	6.844 ms	-1.037 ms	-13.158 %
setFirstElementToZero	0.139 ms	0.171 ms	0.032 ms	23.022 %
elementWiseProduct	2.503 ms	2.555 ms	0.052 ms	2.078 %
ifft	7.372 ms	6.909 ms	-0.463 ms	-6.281 %
wait_for_output_buffer	0.003 ms	0.004 ms	0.001 ms	33.333 %
toBinaryArray	1.668 ms	1.877 ms	0.209 ms	12.530 %
Total	32.655 ms	30.687 ms	-1.968 ms	-6.027 %
Speed	4110.211 MBit/s	4373.825 MBit/s	263.614 Mbit/s	6.414 %

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/7f66373130dc9fa4956b54e8fa39546dbd7ebd8b>

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/0ce2caf83b02bfae4b9e7c9366b6bcd34ace486>

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/1961e8c06f409811d19af322e02cb56c3c0d3bf5>

81. SetFirstElementToZero durch VkFFT Zero Padding

Durch das Vulkan VkFFT zero padding kann bei der iFFT das erste Element durch zero padding übersprungen werden. Dadurch ist SetFirstElementToZero nicht mehr notwendig. Der Geschwindigkeitsgewinn ist auf der nachfolgenden Tabelle ersichtlich.

Geschwindigkeitsgewinn:

Function	Vulkan Old	Vulkan New	Difference C2-B2	Difference % = $((C2/B2) * 100) - 100$
wait_for_input_buffer	0.006	0.006	0.000	0.000
cleaned_memory	0	0	0.000	0
set_count_key_to_zero	0.211	0.064	-0.147	-69.668
set_count_seed_to_zero	0.222	0.059	-0.163	-73.423
binIntffloat_key	1.949	2.101	0.152	7.799
binIntffloat_seed	3.205	3.361	0.156	4.867
calculateCorrectionFloat	0.169	0.363	0.194	114.793
fft_key	6.561	6.448	-0.113	-1.722
fft_seed	6.844	6.614	-0.230	-3.361
setFirstElementToZero	0.171	0	-0.171	-100.000
elementWiseProduct	2.555	2.554	-0.001	-0.039
ifft	6.909	6.875	-0.034	-0.492
wait_for_output_buffer	0.004	0.007	0.003	75.000
toBinaryArray	1.877	1.742	-0.135	-7.192
Total	30.687	30.199	-0.488	-1.590
Speed	4373.825	4444.512	70.687	1.616

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/d82cbc2801b7a421bd5099b9964302f7d13628ae>

82. SetFirstElementToZero durch VkFFT Zero Padding

Verbesserte Leistung beim Speichern von Konstanten im globalen GPU-Speicher anstelle des vom Host angehefteten Speichers. Anstelle von memset.spv Kernbel für set_count_key_to_zero und set_count_seed_to_zero verwende ich nun das schnellere cudaMemcpy. Durch Hinzufügen von registerBoost = true und performHalfBandwidthBoost = true zu den VkFFT-Konfigurationen konnte die Performance leicht verbessert werden.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/3a4597937c0da609da9feee2f988cca08936da16>

83. Vuda: cudaMemset implementiert

CudaMemset und cudaMemsetAsync wurden in Vuda implementiert, um die Leistung von set_count_key_to_zero und set_count_seed_to_zero noch weiter zu verbessern.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/40d7778029546dbc5f2901900830e2f5c0ed2ae3>

84. Vuda: cudaMemset optimiert

Ich habe cudaMemset in Vuda verbessert, indem nur geflasht wird, wenn der Zielspeicher gepinnt ist.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/22621755ce5f7d2f403492e870beb504f6409de7>

85. set_count_one_to_zero vereint

Die Leistung von set_count_one_to_zero durch Zuweisen von count_one_of_global_seed und count_one_of_global_seed in benachbarten Speicherbereichen wurde verbessert, sodass wir den Wert von in einem einzigen Aufruf auf Null gesetzt werden kann, wodurch sich die Leistung im Grunde verdoppelt.

Geschwindigkeitsgewinn:

Function	Vulkan Old	Vulkan New			Differnece C2-B2	Differnece % =((C2/B2)*100)- 100
wait_for_input_buffer	0.006	0.006			0.000	0.000
cleaned_memory	0	0			0.000	0
set_count_to_zero	0.123	0.038			-0.085	-69.106
binIntffloat_seed	3.361	3.14			-0.221	-6.575
binIntffloat_key	2.101	2.058			-0.043	-2.047
calculateCorrectionFloat	0.363	0.172			-0.191	-52.617
fft_key	6.448	6.254			-0.194	-3.009
fft_seed	6.614	7.526			0.912	13.789
setFirstElementToZero	0	0			0.000	0
elementWiseProduct	2.554	2.613			0.059	2.310
ifft	6.875	6.552			-0.323	-4.698
wait_for_output_buffer	0.007	0.003			-0.004	-57.143
toBinaryArray	1.742	1.755			0.013	0.746
Total	30.199	30.119			-0.080	-0.265
Speed	4444.512	4456.22			11.708	0.263

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/dad7c874cf1523992d709de3956184c24a0e8976>

86. GLSL: Round anstelle von RoundEven

ToBinaryArray kann nun round anstelle von roundEven verwenden, da wir sowieso abs verwenden müssen und dies daher keine Rolle mehr spielt.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/46573e0d960148344b7d6745563433251cbfb45>

87. VkFFT kann nun 64 bit

Aktualisierte VkFFT für OpenCL- und 64-Bit-Adressierungsunterstützung. Nach meinen Tests sollte dieses Update keine messbaren Auswirkungen auf die Leistung haben.

Nach VkFFT update: Mit 64-Bit support und gleicher Performance.

Geschwindigkeitsvergleich zwischen alter und neuer VkFFt:

	VkFFT Old	VkFFT New
fft_key	6.254	6.504 ms
fft_seed	7.526	6.829 ms
ifft	6.552	6.855 ms

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/f6c612eeb8c222d0058138c502469d860af71701>

88. ElementWiseProduct optimiert

ElementWiseProduct wurde optimierter geschrieben. Aufgrund von Compiler-Optimierungen, die ähnliche Aktionen ausführen, gibt es jedoch keinen messbaren Leistungsunterschied.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/c9aa6910eb91ddd7098a2d0e7c20e163e3606231>

89. Cuda 11.3

Ich habe das Projekt auf Cuda 11.3 upgedatet. Dazu wurde auch die ganze Nvidia Entwicklungsumgebung und die Grafikkartentreiber upgedatet.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/106557f2140c0f1e60ab2cc0312ed56eb9cbce31>

90. Fliesskommazahlen halber Präzision

Ich habe die Ampout-Überprüfung deaktiviert.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/d322db512d4b85783a4a3339123db7f849960ad7>

VkFFT verwendet jetzt Speicher mit halber Genauigkeit:

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/22789f6435b631f6304539f6463b45c36a19c4ca>

BinInt2float GLSL-Shader mit halber Genauigkeit:

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/28e71665025bad847d5df0b9aa23740e16f1d8f8>

CalculateCorrektionFloat GLSL-Shader mit halber Genauigkeit:

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/cf0e2b29f7232f67e7a98c0f8631a83927db94e0>

ElementWiseProduct GLSL-Shader mit halber Präzision:

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/0d49c439c833c59ce6dc43673b1fa8d31c76980d>

ToBinaryArray GLSL-Shader mit halber Genauigkeit:

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/26f95103959815cfa886c5e48678c6da6a8818dc>

Die VkFFT wurde auf das neueste Commit aktualisiert, damit mein Problem mit dem "Shader compilation error when planning multi-upload half precision R2C/C2R FFTs" behoben werden kann.

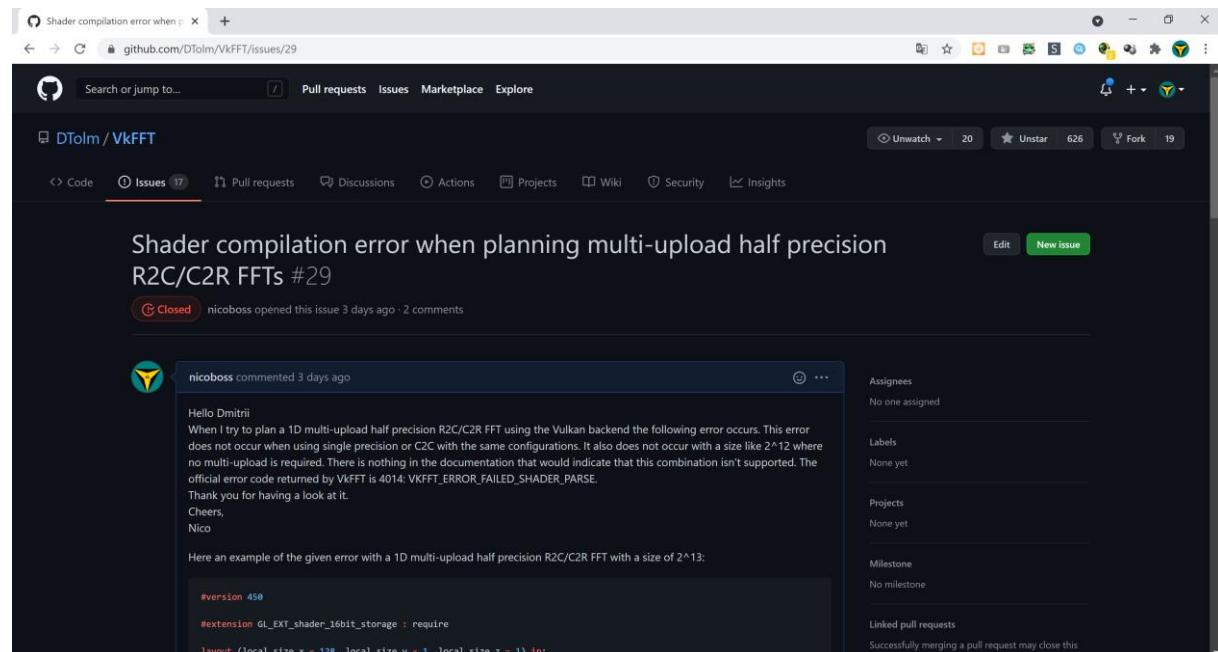


Abbildung 41: Fehlerbeschreibung für VkFFT

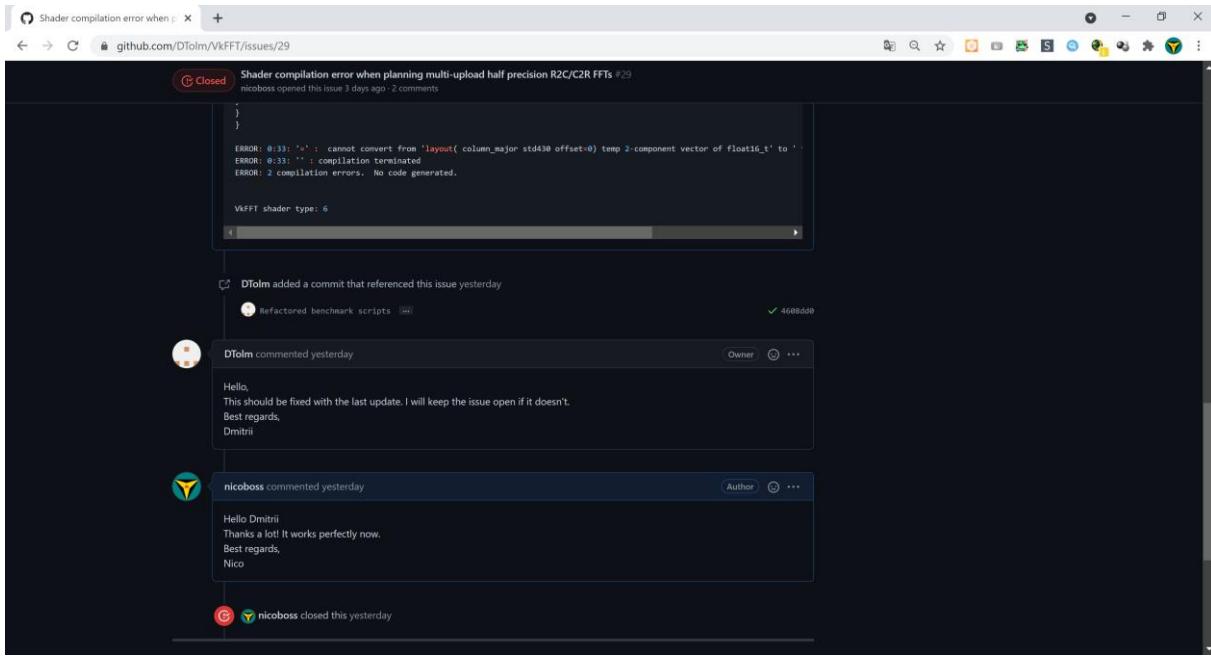


Abbildung 42: Fehlerbehebung durch Dimitrii

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/2a958a691517e964559d891eb2b2adba9f7e15e1>

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/d0c6ea76326f4a07dff730f065f541bb8447c416>

Halbpräzision-Tests:

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/bc9b2cc18f798153bd9315f6adbf1a08f87489aa>

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/2fa80194aaa3a5b4843d8c9d05d8d37c38c68acc>

Der elementWiseProduct GLSL shader wurde zurück auf einfache Präzision geändert.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/303c8970b28951a1521a590a62bb7e6c549d9e7a>

Mehrere Halbpräzision-Tests:

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/249bad7e298d8611c94ae1216f89aaffd0b15629>

91. Test mit grossen Blockgrössen

VkFFT war bis 2^{29} erfolgreich.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplificationvulkan/commit/ec9b02e0927511d6f6278c9c8333e2e15116eaf2>

92. AMD-Probleme behoben

Vor dem Treiberupdate erschien schon nach wenigen Blöcken die folgende Fehlermeldung:

Assertion failed: node != nullptr, file

C:\Users\nico\OneDrive\Öffentlich\Dokumente\GitHub\privacyamplification\PrivacyAmplification\vuda\state\logicaldevice.inl, line 321

Durch das Treiberupdate wurde dieses Problem behoben.

Geschwindigkeitsvergleich mit dem neuen Treiberupdate:

Funktion	Old Driver	New Driver	Differnece C2-B2	Differnece % =((C2/B2)*100)-100
wait_for_input_buffer	0.006	0.007	0.001	16.667
cleaned_memory	0	0	0	0
set_count_to_zero	0.031	0.035	0.004	12.903
binIntffloat_seed	5.45	4.969	-0.481	-8.826
binIntffloat_key	3.788	3.102	-0.686	-18.110
calculateCorrectionFloat	0.299	0.129	-0.17	-56.856
fft_key	57.212	47.006	-10.206	-17.839
fft_seed	64.522	50.373	-14.149	-21.929
setFirstElementToZero	0	0	0	0
elementWiseProduct	4.773	4.246	-0.527	-11.041
ifft	61.72	50.178	-11.542	-18.701
wait_for_output_buffer	0.003	0.003	0	0.000
toBinaryArray	2.581	1.894	-0.687	-26.618
Total	200.386	161.946	-38.44	-19.183
Speed	669.795	828.782	158.987	23.737

93. Matrix in Blöcke aufteilen

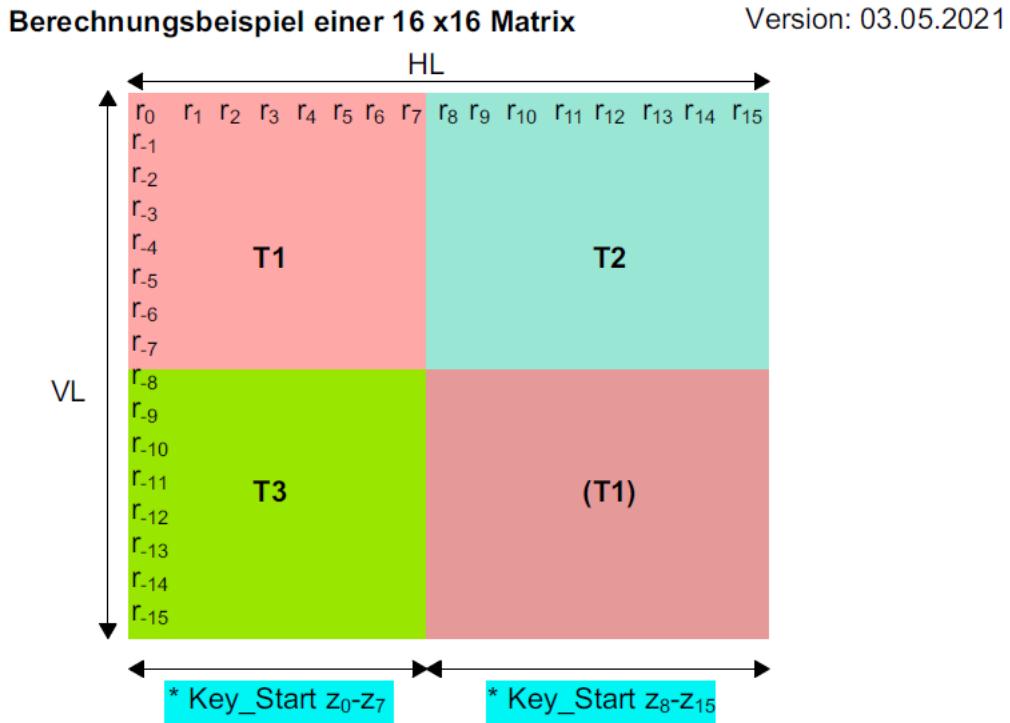
Als nächste grosse Herausforderung werde ich mich mit dem aufteilen der Matrix in Blöcke beschäftigen. Die mathematische Vorlage dazu habe ich von Esther Hänggi erhalten. Um die Mathematischen Formulierungen besser zu verstehen und den Algorithmus abändern zu können, habe ich die Sache grafisch dargestellt.

Als Beispiel nachfolgend die grafische Aufteilung einer 24×40 Matrix in 3 vertikale und 5 horizontale 8×8 er Blöcke.



Abbildung 43: Aufteilung der Matrix in Blöcke

Wie die Umsetzung der Berechnung für ein 8x8er Block mit meinem Algorithmus ablaufen soll:



Unser Algorithmus rechnet für die rote Fläche (Matrix 8x8):

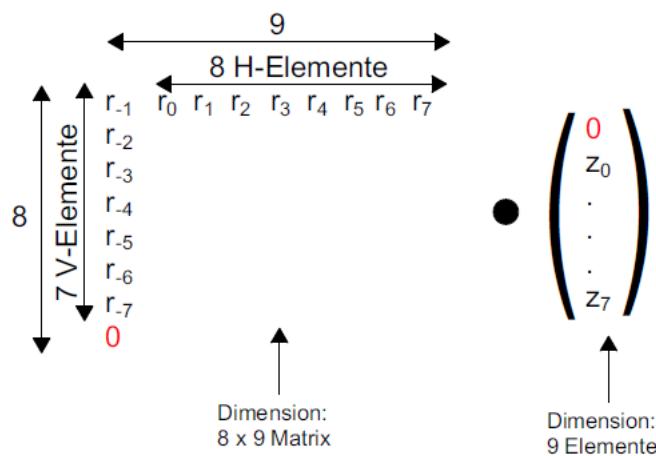


Abbildung 44: Berechnung eines Blocks

Matrix splitten:

Zahlenbeispiel der gesamten Matrix mit unmodifizierter circulant Matrix:

```
[[1. 1. 0. 0. 0. 1. 1. 0. 1. 1. 0. 1. 1. 0. 1.]  
[1. 1. 1. 0. 0. 0. 1. 1. 0. 1. 1. 0. 1. 1. 0.]  
[1. 1. 1. 1. 0. 0. 0. 1. 1. 0. 1. 1. 1. 0. 1. 1.]  
[0. 1. 1. 1. 0. 0. 0. 1. 1. 0. 1. 1. 1. 0. 1.]  
[0. 0. 1. 1. 1. 1. 0. 0. 0. 1. 1. 0. 1. 1. 1. 0.]  
[1. 0. 0. 1. 1. 1. 1. 0. 0. 0. 1. 1. 0. 1. 1. 1.]  
[1. 1. 0. 0. 1. 1. 1. 1. 0. 0. 0. 1. 1. 0. 1. 1.]  
[0. 1. 1. 0. 0. 1. 1. 1. 0. 0. 0. 1. 1. 0. 1.]  
[0. 0. 1. 1. 0. 0. 1. 1. 1. 1. 0. 0. 0. 1. 1. 0.]  
[0. 0. 0. 1. 1. 0. 0. 1. 1. 1. 1. 0. 0. 0. 1. 1.]  
[0. 0. 0. 0. 1. 1. 0. 0. 1. 1. 1. 1. 0. 0. 0. 1.]  
[1. 0. 0. 0. 0. 1. 1. 0. 0. 1. 1. 1. 1. 0. 0.]  
[0. 1. 0. 0. 0. 0. 1. 1. 0. 0. 1. 1. 1. 1. 0. 0.]  
[0. 0. 1. 0. 0. 0. 0. 1. 1. 0. 0. 1. 1. 1. 1. 0.]  
[1. 0. 0. 1. 0. 0. 0. 0. 1. 1. 0. 0. 1. 1. 1. 1.]  
[0. 1. 0. 0. 1. 0. 0. 0. 0. 1. 1. 0. 0. 1. 1. 1.]]
```

key_start: [1 0 1 1 1 1 0 0 0 0 0 0 0 1 1]

key_rest: [0 1 0 1 1 0 1 0 0 0 1 1 1 0 1 1]

preXOR: [1 1 1 0 1 0 1 1 1 0 1 0 0 0 1]

Amplificated Key: [1 0 1 1 0 0 0 1 1 0 0 1 1 0 1 0]

Zahlenbeispiel der gesamten Matrix mit modifizierter circulant Matrix:

```
[[1. 1. 1. 0. 0. 0. 1. 1. 0. 1. 1. 0. 1. 1. 0. 1.]  
[1. 1. 1. 1. 0. 0. 0. 1. 1. 0. 1. 1. 1. 0. 1. 1. 0.]  
[0. 1. 1. 1. 1. 0. 0. 0. 1. 1. 0. 1. 1. 1. 0. 1. 1.]  
[0. 0. 1. 1. 1. 1. 0. 0. 0. 1. 1. 0. 1. 1. 1. 0. 1.]  
[1. 0. 0. 1. 1. 1. 1. 0. 0. 0. 1. 1. 0. 1. 1. 1. 0.]  
[1. 1. 0. 0. 1. 1. 1. 1. 0. 0. 0. 1. 1. 0. 1. 1. 1.]  
[0. 1. 1. 0. 0. 1. 1. 1. 1. 0. 0. 0. 1. 1. 0. 1. 1.]  
[0. 0. 1. 1. 0. 0. 1. 1. 1. 1. 0. 0. 0. 1. 1. 0. 1.]  
[0. 0. 0. 1. 1. 0. 0. 1. 1. 1. 1. 0. 0. 0. 1. 1. 0.]  
[0. 0. 0. 0. 1. 1. 0. 0. 1. 1. 1. 1. 0. 0. 0. 1. 1.]  
[1. 0. 0. 0. 0. 1. 1. 0. 0. 1. 1. 1. 1. 0. 0. 0. 1.]  
[0. 1. 0. 0. 0. 0. 1. 1. 0. 0. 1. 1. 1. 1. 0. 0. 0.]  
[0. 0. 1. 0. 0. 0. 0. 1. 1. 0. 0. 1. 1. 1. 1. 0. 0.]  
[1. 0. 0. 1. 0. 0. 0. 0. 1. 1. 0. 0. 1. 1. 1. 1. 0.]  
[0. 1. 0. 0. 1. 0. 0. 0. 0. 1. 1. 0. 0. 1. 1. 1. 1.]  
[0. 0. 1. 0. 0. 1. 0. 0. 0. 0. 1. 1. 0. 0. 1. 1. 1.]]
```

```

key_start: [0 1 0 1 1 1 1 0 0 0 0 0 0 0 0 1 1]
key_rest: [0 1 0 1 1 0 1 0 0 0 1 1 1 0 1 1]
preXOR: [1 1 1 0 1 0 1 1 1 0 1 0 0 0 0 1]

Amplified Key: [1 0 1 1 0 0 0 1 1 0 0 1 1 0 1 0]

```

Erster Block mit unmodifizierter circulant Matrix:

```

[[1. 1. 0. 0. 0. 1. 1. 0.]
 [1. 1. 1. 0. 0. 0. 1. 1.]
 [1. 1. 1. 1. 0. 0. 0. 1.]
 [0. 1. 1. 1. 1. 0. 0. 0.]
 [0. 0. 1. 1. 1. 1. 0. 0.]
 [1. 0. 0. 1. 1. 1. 1. 0.]
 [1. 1. 0. 0. 1. 1. 1. 1.]
 [0. 1. 1. 0. 0. 1. 1. 1.]]

```

[1 0 1 1 1 1 0 0]
[2. 2. 3. 3. 4. 4. 3. 2.]
[0. 0. 1. 1. 0. 0. 1. 0.]

Zweiter Block mit unmodifizierter circulant Matrix:

```

[[1. 1. 1. 0. 1. 1. 0. 1.]
 [0. 1. 1. 1. 0. 1. 1. 0.]
 [1. 0. 1. 1. 1. 0. 1. 1.]
 [1. 1. 0. 1. 1. 1. 0. 1.]
 [0. 1. 1. 0. 1. 1. 1. 0.]
 [0. 0. 1. 1. 0. 1. 1. 1.]
 [0. 0. 0. 1. 1. 0. 1. 1.]
 [1. 0. 0. 0. 1. 1. 0. 1.]]

```

[0 0 0 0 0 0 1 1]
[1. 1. 2. 1. 1. 2. 2. 1.]
[1. 1. 0. 1. 1. 0. 0. 1.]

Dritter Block mit unmodifizierter circulant Matrix:

```

[[0. 0. 1. 1. 0. 0. 1. 1.]
 [0. 0. 0. 1. 1. 0. 0. 1.]
 [0. 0. 0. 0. 1. 1. 0. 0.]
 [1. 0. 0. 0. 0. 1. 1. 0.]
 [0. 1. 0. 0. 0. 0. 1. 1.]

```

[0. 0. 1. 0. 0. 0. 0. 1.]
[1. 0. 0. 1. 0. 0. 0. 0.]
[0. 1. 0. 0. 1. 0. 0. 0.]]

[1 0 1 1 1 1 0 0]
[2. 2. 2. 2. 0. 1. 2. 1.]
[0. 0. 0. 0. 0. 1. 0. 1.]

Vierter Block mit unmodifizierter circulant Matrix:

[[1. 1. 0. 0. 0. 1. 1. 0.]
[1. 1. 1. 0. 0. 0. 1. 1.]
[1. 1. 1. 1. 0. 0. 0. 1.]
[0. 1. 1. 1. 1. 0. 0. 0.]
[0. 0. 1. 1. 1. 1. 0. 0.]
[1. 0. 0. 1. 1. 1. 1. 0.]
[1. 1. 0. 0. 1. 1. 1. 1.]
[0. 1. 1. 0. 0. 1. 1. 1.]]

[0 0 0 0 0 1 1]
[1. 2. 1. 0. 0. 1. 2. 2.]
[1. 0. 1. 0. 0. 1. 0. 0.]

Erster Block mit modifizierter circulant Matrix:

[[1. 1. 1. 0. 0. 0. 1. 1. 0.]
[1. 1. 1. 1. 0. 0. 0. 1. 1.]
[0. 1. 1. 1. 1. 0. 0. 0. 1.]
[0. 0. 1. 1. 1. 1. 0. 0. 0.]
[1. 0. 0. 1. 1. 1. 1. 0. 0.]
[1. 1. 0. 0. 1. 1. 1. 1. 0.]
[0. 1. 1. 0. 0. 1. 1. 1. 1.]
[0. 0. 1. 1. 0. 0. 1. 1. 1.]]

key_start: [0 1 0 1 1 1 1 0 0]
key_rest: [0 0 0 0 0 0 0]
preXOR: [0 0 1 1 0 0 1 0]

Zweiter Block mit modifizierter circulant Matrix:

[[0. 1. 1. 1. 0. 1. 1. 0. 1.]
[1. 0. 1. 1. 1. 0. 1. 1. 0.]
[1. 1. 0. 1. 1. 1. 0. 1. 1.]]

```
[0. 1. 1. 0. 1. 1. 1. 0. 1.]  
[0. 0. 1. 1. 0. 1. 1. 1. 0.]  
[0. 0. 0. 1. 1. 0. 1. 1. 1.]  
[1. 0. 0. 0. 1. 1. 0. 1. 1.]  
[0. 1. 0. 0. 0. 1. 1. 0. 1.]]
```

```
key_start: [0 0 0 0 0 0 0 1 1]  
key_rest: [0 0 0 0 0 0 0]  
preXOR: [1 1 0 1 1 0 0 1]
```

Dritter Block mit modifizierter circulant Matrix:

```
[[0. 0. 0. 1. 1. 0. 0. 1. 1.]  
[0. 0. 0. 0. 1. 1. 0. 0. 1.]  
[1. 0. 0. 0. 0. 1. 1. 0. 0.]  
[0. 1. 0. 0. 0. 0. 1. 1. 0.]  
[0. 0. 1. 0. 0. 0. 0. 1. 1.]  
[1. 0. 0. 1. 0. 0. 0. 0. 1.]  
[0. 1. 0. 0. 1. 0. 0. 0. 0.]  
[0. 0. 1. 0. 0. 1. 0. 0. 0.]]
```

```
key_start: [0 1 0 1 1 1 1 0 0]  
key_rest: [0 0 0 0 0 0 0]  
preXOR: [0 0 0 0 0 1 0 1]
```

Vierter Block mit modifizierter circulant Matrix:

```
[[1. 1. 1. 0. 0. 0. 1. 1. 0.]  
[1. 1. 1. 1. 0. 0. 0. 1. 1.]  
[0. 1. 1. 1. 1. 0. 0. 0. 1.]  
[0. 0. 1. 1. 1. 1. 0. 0. 0.]  
[1. 0. 0. 1. 1. 1. 1. 0. 0.]  
[1. 1. 0. 0. 1. 1. 1. 1. 0.]  
[0. 1. 1. 0. 0. 1. 1. 1. 1.]  
[0. 0. 1. 1. 0. 0. 1. 1. 1.]]
```

```
key_start: [0 0 0 0 0 0 0 1 1]  
key_rest: [0 0 0 0 0 0 0]  
preXOR: [1 0 1 0 0 1 0 0]
```

T1 XOR T2: [0. 0. 1. 1. 0. 0. 1. 0.] XOR [1. 1. 0. 1. 1. 0. 0. 1.] = [1 1 1 0 1 0 1 1]

T3 XOR T4: [0. 0. 0. 0. 0. 1. 0. 1.] XOR [1. 0. 1. 0. 0. 1. 0. 0.] = [1 0 1 0 0 0 0 1]

preXor aus den vier 8x8-er Blöcken zusammengesetzt: **[1 1 1 0 1 0 1 1 1 0 1 0 1 0 0 0 0 1]**

preXOR der 16x16 Beispielmatrix ohne Blockaufteilung: [1 1 1 0 1 0 1 1 1 0 1 0 0 0 0 1]

Mid-Range GPU und ohne Cache:

2⁴⁸ mit 2²⁷ Blöcken:

$((2^{**}19/2+2^{**}19/8)*(2^{**}19/4+2^{**}19/8))*0.05)/(3600*24*365.25)$

102 Jahre

2⁴⁸ mit 2³² Blöcken (geschätzt):

$((2^{**}14/2+2^{**}14/8)*(2^{**}14/4+2^{**}14/8))*(0.05*2^{**}5*5))/(3600*24*365.25)$

16 Jahre

2⁴⁰ mit 2²⁷ Blöcken:

$((2^{**}13/2+2^{**}13/8)*(2^{**}13/4+2^{**}13/8))*0.05)/(3600*24)$

9.1 Tage

2⁴⁰ mit 2³² Blöcken (geschätzt):

$((2^{**}8/2+2^{**}8/8)*(2^{**}8/4+2^{**}8/8))*(0.05*2^{**}5*5))/3600$

34.1 Stunden

2³⁸ mit 2²⁷ Blöcken:

$((2^{**}11/2+2^{**}11/8)*(2^{**}11/4+2^{**}11/8))*0.05)/(3600*24)$

13.7 Stunden

2³⁸ mit 2³² Blöcken:

$((2^{**}6/2+2^{**}6/8)*(2^{**}6/4+2^{**}6/8))*(0.05*2^{**}5*5))/3600$

2.1 Stunden

2³⁶ mit 2²⁷ Blöcken:

$((2^{**}9/2+2^{**}9/8)*(2^{**}9/4+2^{**}9/8))*0.05)/3600$

51.2 Minuten

2³⁶ mit 2³² Blöcken:

$((2^{**}4/2+2^{**}4/8)*(2^{**}4/4+2^{**}4/8))*(0.05*2^{**}5*5))/60$

8 Minuten

2³⁴ mit 2²⁷ Blöcken:

$((2^{**}7/2+2^{**}7/8)*(2^{**}7/4+2^{**}7/8))*0.05)/60$

3.2 Minuten

2³⁴ mit 2³² Blöcken:

$((2^{**}2/2+2^{**}2/8)*(2^{**}2/4+2^{**}2/8))*(0.05*2^{**}5*5)$

30 Sekunden

2³² geschätzt:

$0.05*2^{**}5*5$

8 Sekunden

2³² mit 2²⁷ Bläcken:

$((2^{**}5/2+2^{**}5/8)*(2^{**}5/4+2^{**}5/8))*0.05$

12 Sekunden

Initialisierung der PrivacyAmplification splitted big blocks.py als Kopie der PrivacyAmplification_with_real_data_1times_1024.py

<https://gitlab.enterpriselab.ch/qkd/cpu/privacyamplificationpython/commit/e2c2bb2f769849fd1be9a30416351750b7a6b02a>

Planung des PrivacyAmplification splitted big blocks algorithm

<https://gitlab.enterpriselab.ch/qkd/cpu/privacyamplificationpython/commit/b53816fb06f5a74dcf207f7ada18d6aa3c0466b2>

Behebung der numpy depreciation warnings

<https://gitlab.enterpriselab.ch/qkd/cpu/privacyamplificationpython/commit/5e01efc3370be45b34aaa2b087191228178ebcd>

Modified Toepliz Matrixmultiplikation Beispiel

<https://gitlab.enterpriselab.ch/qkd/cpu/privacyamplificationpython/commit/ac7a3a597f16c9a0f125e48dc4fbf8bfa4884bc8>

Das Gleiche mit zero

<https://gitlab.enterpriselab.ch/qkd/cpu/privacyamplificationpython/commit/596fef7e355815f229dc3abe470aa1350d13a114>

Zero -Test mit FFT

<https://gitlab.enterpriselab.ch/qkd/cpu/privacyamplificationpython/commit/fbf4dd22ddb37768eac2c174af08d9ee29ef13b1>

Zero-Test mit 16x16 Block

<https://gitlab.enterpriselab.ch/qkd/cpu/privacyamplificationpython/commit/1aa164fe61af7e8c92fdf4b7a5729281f0c4d6e9>

PrivacyAmplification_Test_Modified überarbeitet, damit PrivacyAmplification als Funktion aufgerufen werden kann.

<https://gitlab.enterpriselab.ch/qkd/cpu/privacyamplificationpython/commit/e872b30021085edda01241ef4270f5d14727a96a>

Failed test

<https://gitlab.enterpriselab.ch/qkd/cpu/privacyamplificationpython/commit/4fe401f3cc813b24f27945422e7305df396d83ae>

Fehler mit h3 repariert.

<https://gitlab.enterpriselab.ch/qkd/cpu/privacyamplificationpython/commit/8da1c730521971b185928f18120c3c5bb92862ba>

Erster Block

<https://gitlab.enterpriselab.ch/qkd/cpu/privacyamplificationpython/commit/4ada51e7dae836d1889beeb6e762f84a69afa6f5>

Big fail

<https://gitlab.enterpriselab.ch/qkd/cpu/privacyamplificationpython/commit/76a429de048e69306bb44cb01deb26db93923d04>

Zweiter Block

<https://gitlab.enterpriselab.ch/qkd/cpu/privacyamplificationpython/commit/cb03c4384e8e10db4491489928b5c5d9da1e5db2>

Dritter Block

<https://gitlab.enterpriselab.ch/qkd/cpu/privacyamplificationpython/commit/3aa50e3b916392110197c65a995e96c82b5b5752>

Vierter Block

<https://gitlab.enterpriselab.ch/qkd/cpu/privacyamplificationpython/commit/7fca8f3a18f24a5f54a06aaad465f94997c04be2>

$$(2^{29})/(1024*1024*1024) = 0.5 \text{ GB}$$

RAM zu GPU ist 10 GB/s => 50 ms

FFT geht aber weniger als 7 ms => Cache unnötig wenn nicht auf GPU memory.

$$((2^{27}+2^{29})*(2^{27}*4))/(1024*1024*1024)$$

512 GB Seed Cache würde nötig sein

$$(2^{27}/2+2^{27}/8)*(2^{27}/4+2^{27}/8)$$

Cachen von seed vs key:

Messung (von AMD): Seed: 55.342 ms, Key: 50.108 ms (binIntToFloat + fft)

Matrix: 3r * 5r

Seed Cachen: $8r[-1]*55.342 + 15*r^2*50.108 = r(442+751r)$

Key Cachen: $5r*50.108 + 15r^2*55.342 = r(250+830r)$

$r(442+751r) = r(250+830r) \Rightarrow 2.43 \Rightarrow$ ab einer 9x15 Matrix ist seed caching schneller.

VI Woche 11 + 12

94. VkFFT aktualisiert

Die VkFFT wurde auf das neueste Commit aktualisiert, damit mein Problem "Shader compilation error when planning multi-upload half precision R2C/C2R FFTs" behoben werden kann.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/2a958a691517e964559d891eb2b2adba9f7e15e1>

95. Aufteilen des reciveData-Thread

Aufteilen des reciveData-Thread in einen reciveDataSeed- und einen reciveDataKey-Thread, um die ZeroMQ-Kommunikationsleistung mithilfe von Multithreading zu verbessern.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/3b1801d44802490110be8b8a8d660660025fc873>

96. Keine ZeroMQ-Statusmeldungen

Es werden keine ZeroMQ-Statusmeldungen mehr angezeigt, da dies erhebliche Auswirkungen auf die Leistung hat.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/193e34299534184bd़fa26dcc618c746df1dc6117>

97. Send_alice und send_bob zu sendData

MatrixSeedServerBeispiel: Kombinierte send_alice und send_bob zu sendData, um Code-Redundanz zu vermeiden.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/a39ed39cc61eaa6004c89821ba050f6c018bebde>

98. Kommunikationsfehler behoben

Es wurde ein Fehler behoben, der dazu führte, dass der Hauptthread nicht auf reciveDataKey wartete, was zu einem Überprüfungsfehler führte, wenn der Schlüsselserver nicht aktiviert war.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/c10f4848363e0e0cce3d669dd918fb434695edb>

99. Eingabe für mehrfache Verwendung des Seeds

Reuse_seed_amount wurde implementiert, sodass dem Matrix-Seed-Server jetzt angeben kann, wie oft ein Seed wiederverwendet werden soll.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/59d62943194a7f4b9ac6b6fe670458629f3b404d>

100. LargeBlockSizeExample hinzugefügt

LargeBlockSizeExample hinzugefügt und einige projektspezifische IDs bereinigt, die ich durch einfaches Kopieren und Ändern vorhandener Beispiele durcheinandergebracht hatte.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/760a9a5cb275c0fa4adde644b08428dc4011b345>

101. LargeBlockSizeExample mit Multithreading

Kombiniertes MatrixSeedServerExample und SendKeysExample in LargeBlockSizeExample mit sauberem Multithreading

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/562068db127a1a5aedf28d37682b2958837812ef>

102. ReceiveAmpOutExample integriert

Es wurde ein ReceiveAmpOutExample in LargeBlockSizeExample mit sauberem Multithreading integriert.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/134e682d43ff98475c517b94ae26060aa2a9b3fa>

103. ReceiveAmpOut

ReceiveAmpOut muss nichts zurückgeben.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/471c56676f954f72a8854ca8cfb6973392aa8229>

104. ToBinaryArrayNoXOR

Implementierung des ToBinaryArrayNoXOR in Vulkan und Cuda Kernel. Ebenfalls auch in Unit Tests integrierte.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/7b3f1864dd5f36201b230687c3a99cc4e51cc80d>

105. Do_xor_key_rest

Die Nachricht do_xor_key_rest wurde implementiert

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/53d32c137f98200e2efc5e8c961bb8b7830a1122>

106. Codeverbesserungen durch templates

Ich habe viel kopierten Code durch templates ersetzt, um die Lesbarkeit und Wartbarkeit zu verbessern.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/098618cd74edf7f68faead3bac0be1526c2c9807>

107. VkFFT für Mehrfach-Upload aktualisiert

VkFFT wurde aktualisiert, um R2C / C2R-FFT mit Mehrfach-Upload in einer Richtung zu unterstützen. Der C2R-FFT-Fix mit nur einer umgekehrten Einzelrichtung und mehreren Uploads stammt tatsächlich aus meiner eigenen Pull-Anfrage an VkFFT, da er nach dem offiziellen Fix immer noch fehlerhaft war.

bugfix to single direction inverse only multi-upload C2R FFT #33

Merged DTolm merged 1 commit into `DTolm:master` from `nicoboss:inverse_only_multi-upload_C2R` 10 days ago

Conversation 1 Commits 1 Checks 1 Files changed 1

nicoboss commented 10 days ago

Hello Dmitrii

I saw that you fixed single direction multi-upload R2C FFT in [79327f5](#)

When trying it out it worked perfectly fine for forward R2C multi-upload FFTs (`makeForwardPlanOnly = true`). However, when I tried it with inverse C2R multi-upload FFTs (`makeInversePlanOnly = true`), `performVulkanFFT` crashed on the following line due to `app->localFFTPlan` being a nullptr:

```
if (app->localFFTPlan->multiUploadR2C) app->configuration.size[0] *= 2;
```

I was able to fix it by using `app->localFFTPlan_inverse->multiUploadR2C` instead:

```
if (app->localFFTPlan_inverse->multiUploadR2C) app->configuration.size[0] *= 2;
```

Best regards,
Nico

bugfix to single direction inverse only multi-upload C2R FFT Verified ✓ f46934c

DTolm commented 10 days ago

Dear Nico,
I have also found this issue earlier today and was about to submit a fix with a few other modifications. Thanks for the attention!
Best regards,
Dmitrii

DTolm merged commit [67a89df](#) into `DTolm:master` 10 days ago
1 check passed

[View details](#) [Revert](#)

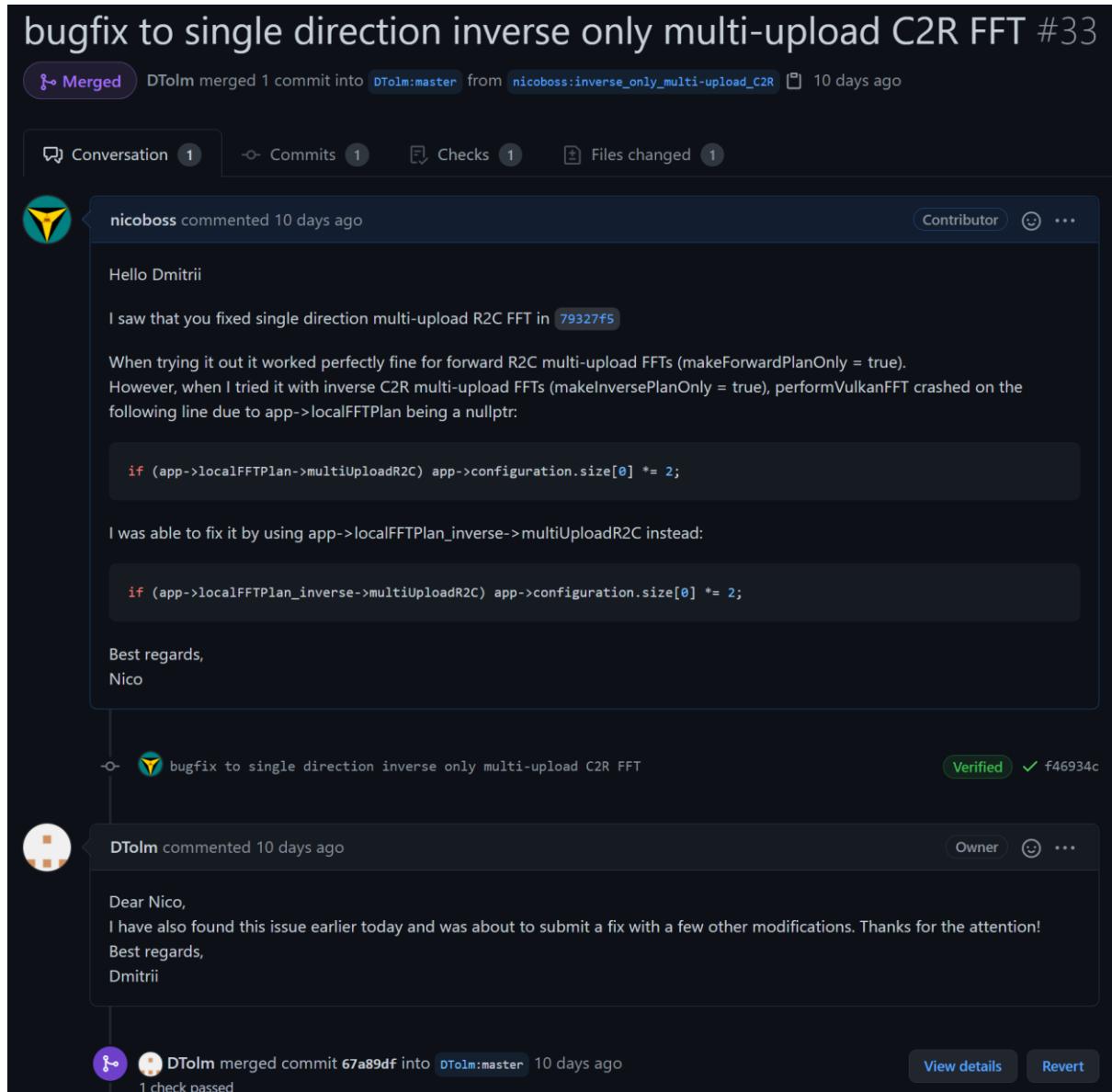


Abbildung 45: Mein pull request

<https://github.com/DTolm/VkFFT/pull/33>

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/cb61beedb6d4de16f55799a4b7f900be68735da>

108. Problem bei Cuda behoben

Ich habe das Problem, bei dem Cuda aufgrund von `reuse_seed_amount` und `ToBinaryArrayNoXOR` nicht mehr kompiliert wurde, behoben.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/5f1d400a055083656fc14e1c5308134c8c32b11>

109. Speedtest

Ich habe jetzt den speedtest.py aus azure-pipelines.yml extrahiert und den Speedtest überprüft.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/e1a993f85898d80fb101d8b16ade6cf00100001c>

110. PrivacyAmplification_Matrix_Splitten

PrivacyAmplification_Matrix_Splitten.py: Erster commit

<https://gitlab.enterpriselab.ch/qkd/cpu/privacyamplificationpython/commit/7408f6adfc378d3883bff4ea645601d2a15dfd6c>

111. Keine horizontale und vertikale Aufteilung mehr

Die Aufteilung in eine horizontale und vertikale Komponente wurde aufgehoben.

<https://gitlab.enterpriselab.ch/qkd/cpu/privacyamplificationpython/commit/14540685805cb81563cb9b6a5c2a259859c1d3b6>

112. Block-Split-Algorithmus vereinfacht

Ich habe versucht, den Block-Split-Algorithmus zu vereinfachen und zurückzusetzen

<https://gitlab.enterpriselab.ch/qkd/cpu/privacyamplificationpython/commit/d2ea25496a6a499b03bd8bd028960c049f1d7107>

<https://gitlab.enterpriselab.ch/qkd/cpu/privacyamplificationpython/commit/3bb75eaa3afc431a88caf5c49331ce60309c88f8>

113. Zweiter Block automatisiert

Die Berechnung des zweiten Blocks ist jetzt automatisiert.

<https://gitlab.enterpriselab.ch/qkd/cpu/privacyamplificationpython/commit/f5ba70fa59c2af5b1b93cb94c2c85ee604eb1db0>

114. Erster Block automatisiert

Die Berechnung des ersten Blocks ist jetzt automatisiert.

<https://gitlab.enterpriselab.ch/qkd/cpu/privacyamplificationpython/commit/41a26cfa4adc33c54a01039eade98d5526ae5604>

115. Dritter Block automatisiert

Die Berechnung des dritten Blocks ist jetzt automatisiert.

<https://gitlab.enterpriselab.ch/qkd/cpu/privacyamplificationpython/commit/a3189668d592fae9fc1fd9cfb4a34e3b016a0c5>

116. Alle vier Blöcke automatisiert!

Die Berechnung aller vier Blöcke ist jetzt automatisiert.

<https://gitlab.enterpriselab.ch/qkd/cpu/privacyamplificationpython/commit/5df5f43e72a4d9a34d847fa8992a15742c0d69fe>

117. Blocktracker

Es wurde ein aktueller Blocktracker implementiert.

<https://gitlab.enterpriselab.ch/qkd/cpu/privacyamplificationpython/commit/9093bb329cd4cfadd5379846c33e37662d4178f>

118. Zählen des aktuellen Schlüssels

Das Zählen des aktuellen Schlüssels funktioniert!

<https://gitlab.enterpriselab.ch/qkd/cpu/privacyamplificationpython/commit/935f3dc6a45901004bd64217e283be2808e08072>

119. Die Seed-Berechnung funktioniert

Ich habe es endlich geschafft, dass die Seed-Berechnung korrekt und erfolgreich ist!

<https://gitlab.enterpriselab.ch/qkd/cpu/privacyamplificationpython/commit/f502f243ef2ca024852a7c02b5764a5a3d63d971>

120. CurrentRowNr

Dem Algorithmus wurde eine CurrentRowNr hinzugefügt.

<https://gitlab.enterpriselab.ch/qkd/cpu/privacyamplificationpython/commit/a62d9eeba6720d40447eb82e38a130dfa52a9a94>

121. Amp_out automatisiert

Die amp_out-Funktion wurde automatisiert.

<https://gitlab.enterpriselab.ch/qkd/cpu/privacyamplificationpython/commit/756373f2626b9a643c61f065c7bcab30174df92c>

122. Horizontal XOR automatisiert

Die horizontale XOR-Funktion wurde automatisiert.

<https://gitlab.enterpriselab.ch/qkd/cpu/privacyamplificationpython/commit/5ae0bc9e72f7f0a1c0e44ab3be5a8c2edbb293f6>

123. Amplified_key automatisiert

Die amplified_key -Funktion wurde automatisiert.

<https://gitlab.enterpriselab.ch/qkd/cpu/privacyamplificationpython/commit/22a3de15d56258ccd731786fafb661637d6259d4>

124. Test mit sample_size 64

Ich habe einige Fehler behoben und mit einer sample_size von 64 getestet.

<https://gitlab.enterpriselab.ch/qkd/cpu/privacyamplificationpython/commit/4730e0373eb9e3cf4f1c6b03461013750a5777e7>

125. Gewünschte Länge

Der Algorithmus berechnet nun die gewünschte, respektive geforderte Länge.

<https://gitlab.enterpriselab.ch/qkd/cpu/privacyamplificationpython/commit/8d9bd78925ceac193f14741c403f1d6cb1ca06f8>

126. Test mit sample_size 1024 und 3840 Blöcken

Ich habe den Test mit einer sample_size von 1024 und mit 3840 Blöcken durchgeführt.

<https://gitlab.enterpriselab.ch/qkd/cpu/privacyamplificationpython/commit/73e556daa7adf840ca5e2683b90fdb2967f55ebf>

127. Test mit sample_size 1024 und 16x16 Unterblöcke

Ich habe den Test mit einer sample_size von 1024 und mit 16x16-Unterblöcken durchgeführt.

<https://gitlab.enterpriselab.ch/qkd/cpu/privacyamplificationpython/commit/85db17c6802c1dd6687dac25c0f43fd7f3d78808>

128. Blockgrößenkontrolle

Ich habe eine Blockgrößenkontrolle (chunk_size check) implementiert.

<https://gitlab.enterpriselab.ch/qkd/cpu/privacyamplificationpython/commit/ed86800a678f0b53e67b28a2c4cfef816d8b56b5>

129. 2^{14} -Testmatrix mit 2^{10} Blöcken

Ich habe den Test mit einer 2^{14} Testmatrix mit 2^{10} Blöcke durchgeführt.

<https://gitlab.enterpriselab.ch/qkd/cpu/privacyamplificationpython/commit/45621b7980d0c5d623e5d562ba17357cf6fb4eed>

130. 2^{14} -Testmatrix mit 2^{11} Blöcken

Ich habe den Test mit einer 2^{14} Testmatrix mit 2^{11} Blöcke durchgeführt und zurückgesetzt.

<https://gitlab.enterpriselab.ch/qkd/cpu/privacyamplificationpython/commit/9ea61133ffbfdd624c76355abf9be9364c6e97c6>

<https://gitlab.enterpriselab.ch/qkd/cpu/privacyamplificationpython/commit/152e5c7cbd05621f75e7210115a4dd7924f4ddf9>

131. Endianness bei Seed und Key

Seed und Key wurden korrekt in 4-Byte-Endianness umgewandelt. Ich kontrollierte den Speicher mit dem Memory Viewer.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/786889461434810744ed8dbe6b06afb5b4649191>

132. Gleiches Ergebnis bei Matrix mit Blockaufteilung

Ich habe das gleiche Ergebnis wie bei der Python-Referenzimplementierung mit der Testmatrix für die Blockaufteilung erhalten.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/0cf5592268b89958dc25011a88a3edcb6f16287d>

<https://gitlab.enterpriselab.ch/qkd/cpu/privacyamplificationpython/commit/612e3c98874c870ff30ef228a26ee7bb3973b886>

133. Variablen anstelle von Arrays

Ich verwende jetzt Variablen anstelle von Arrays, um Variablen an GLSL-Shader zu übergeben.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/e7787716c923a9b37c1f41577fcc776ff879c29f>

134. Fixed Pipeline Tests

Fixed Pipeline-Tests

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/ead1abf7642635401d1e544211e8c05ff60aca0e>

135. 2^{14} Pipeline-Tests

Es wurden 2^{14} Pipeline-Tests hinzugefügt, was sehr viel Aufwand erforderte.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/6fbef06e858c7d5a459ddcafff81431a064cfb7e>

136. Modifiziertes Toeplitz-Hashing mit sample size of 16383

Modifiziertes Toeplitz-Hashing mit FFT und einer sample size von 16383 funktioniert.

<https://gitlab.enterpriselab.ch/qkd/cpu/privacyamplificationpython/commit/27ad3d4c09e23476abee38fc62b34ab41335b80e>

137. Modifiziertes Toeplitz-Hashing mit sample size of 16384

Modifiziertes Toeplitz-Hashing mit FFT und einer sample size von 16384 funktioniert

<https://gitlab.enterpriselab.ch/qkd/cpu/privacyamplificationpython/commit/2ecdaffc03683b6f0b3bc7efbf3bbf845b39dec2>

138. Simulierter Python-Algorithmus von Cuda funktioniert

Der simulierte Python-Algorithmus von Cuda funktioniert mit 16384, jedoch erst, nachdem das letzte key_rest-Element zur Laufzeit entfernt wurde.

<https://gitlab.enterpriselab.ch/qkd/cpu/privacyamplificationpython/commit/4a9b2c6662ea2f2f94a783a26ffe9109705ea10b>

139. Sichtbare Eingabelänge

Die Eingabelänge ist jetzt sichtbar.

<https://gitlab.enterpriselab.ch/qkd/cpu/privacyamplificationpython/commit/f19c3580c6f09df8424536ed72f921088c776e1d>

140. Gleiches Ergebnis wie mit Python!

Ich habe jetzt das gleiche Ergebnis wie mit dem Python-Algorithmus erhalten.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/ed8509c3eee155c6d2c9e5147fd659bf927f06d0>

141. First Split Block auf GPU funktioniert

First Split Block funktioniert korrekt mit einer echten GPU.

<https://gitlab.enterpriselab.ch/qkd/cpu/privacyamplificationpython/commit/f6e779496e3f3193ddabfd851dd7d5bf0ba70e33>

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/35ba23203ede0cf6ddb9ceb4353facb3a802ffe8>

142. Block-Splitting in C ++

Block-Splitting in C ++ implementiert.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/e57b1d34a2e23846b8c113fe884d5cc0e2f46165>

143. Multiple Split Block Algorithmus auf GPU

Mehrere Fehler beim Multiple Split Block Algorithmus auf einer realen GPU behoben. Der Test mit den Split Blocks auf einer realen GPU funktioniert jetzt.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/e090630a966d860eff405bcc1f6f332cf86d3ec6>

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/4b3df0615e064ee3fb5a8132e966a9c05fddb0a6>

<https://gitlab.enterpriselab.ch/qkd/cpu/privacyamplificationpython/commit/05a43470e4902b5b1aee2c37f079c3cddde58611>

144. Chunk_size Kompatibilität

Chunk_size ist jetzt zwischen dem Split-Algorithmus und dem Privacy Amplification-Algorithmus kompatibel.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/afd46e772ce0edb995ac431257a54364a69b2a39>

<https://gitlab.enterpriselab.ch/qkd/cpu/privacyamplificationpython/commit/a56f645111e4dcf41cc25a97d7f53f4385f82af2>

145. Korrekter permuted_key_raw

Der Berechnete permuted_key_raw stimmt beim Split-Algorithmus und dem Privacy Amplification-Algorithmus überein!

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/26972acc9751f664b1cf4aa35bc93ddc254f29ad>

146. BinOut

Ich verwenden binOut anstelle von testMemoryHost, da sonst race conditions auftreten.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/ca7bda5a4db4946d0a79ef3b082b686cd9a525ea>

147. XOR mit key_rest

Ich habe nun dem Algorithmus die Funktion XOR mit key_rest implementiert.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/94341a26f73dc468ead07e82fef94c36320e4b4>

148. Blockreihenfolge

Ich habe die falsche Reihenfolge der Blöcke geflickt.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/61791ea86088c1431b1549d8c3ae63cf999c2226>

149. ZeroMQ

Das ZeroMQ-Kommunikationsprotokoll wurde stark verbessert.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/dc3d1fce7ff64f11f651d06879f3b9edc22d79e9>

150. Entfernung der Komprimierung

Die Komprimierung im privacy amplification chunk Algorithmus wurde entfernt.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/334b585e4e93099a97d64c6353b97842d1f030ed>

151. Debugging zur Überprüfung von C ++

Dem Python-Skript wurde ein Debugging hinzugefügt, um die C ++ - Version zu überprüfen.

<https://gitlab.enterpriselab.ch/qkd/cpu/privacyamplificationpython/commit/74e648db3403c08e06645e3e44e7dc5d3ad5ff69>

152. GitLab-URL korrigiert

Die GitLab-URL im Shell-Skript Pretty_git_log wurde korrigiert.

<https://gitlab.enterpriselab.ch/qkd/cpu/privacyamplificationpython/commit/6eb2c009be398e95a0f80a2ed22ca4006447b64f>

153. Chunk-Splitting-Algorithmus funktioniert!

Der Chunk-Splitting-Algorithmus funktioniert vollständig!

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/ae45e94b09aa3e08f39e3c0725d46f8f9fee40f5>

VII Woche 13 - 16

154. Aktualisierung des Zugriffstoken für GitLab

Der Zugangstoken zur Privacy Amplification wurde aktualisiert.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/e0d258dc15dee987f694070978f103b37e450a3d>

155. Dockerfile mit neuester Cuda-Version und Accessss Token

Ich habe das Dockerfile mit der neuesten Cuda-Version aktualisiert und einen neuen Accessss Token erstellt.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/3468b09199b5d281876ac52c90472e91e507229f>

156. Downloadort des Dockers behoben

Ich habe den Downloadort vom Docker-Container *nvidia/cuda:11.3.0-devel-ubuntu20.04* behoben.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/3b61a9adb38acf4aa79f6a645b83d298345a903>

157. Keine VkFFT beim Kompilieren mit Cuda

Die VkFFT wird beim Kompilieren mit Cuda nicht benötigt.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/5af32c40ae567127ed7991cc580fa155345907a8>

158. Linux-Kompilierungsfehler behoben

Den Mainloop habe ich in eine eigene Funktion verschoben, damit der Speedtest ohne Goto möglich ist. Das Goto löste bei der Linux-Kompilierung den Fehler "transfer of control bypasses initialization" aus. Auf diese Weise konnte das Problem behoben werden.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/3bfddfa86c29f55b4e837d036ba073f7fb0bd531>

159. Atomics ist Linux kompatibel

Ich habe Atomics Linux kompatibel gemacht.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/a76612a91c4b830f34d5ff859caf5462618893a4>

160. Zeitmessung neu mit high_resolution_clock

Neu wird high_resolution_clock anstelle von steady_clock zum Speichern von Zeitmessung verwendet.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/e6c9d0b9ef40b942e73ebefd181071019a042f10>

161. Kommunikationsprotokoll aktualisiert

Das Kommunikationsprotokoll wurde auf MatrixSeedServerExample, SendKeysExample und ReceiveAmpOutExample aktualisiert.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/5ae040746f4652ec69e98a7cbb05595947b45c3c>

162. SendKeysExample

SendKeysExample kann jetzt entscheiden, ob der PrivacyAmplification-Algorithmus das komprimierte oder unkomprimierte amp_out an das ReceiveAmpOutExample senden soll.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/fa2c9e7695221b288f60900b2d2fa7a5b68bddae>

163. PrivacyAmplification-Algorithmus Test

Testen des PrivacyAmplification-Algorithmus.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/242b565b94a7f174947ba16893fc6b20e4c3ebd>

164. Kommunikationsprotokoll der LargeBlocksizeExample

Ich habe das Kommunikationsprotokoll der LargeBlocksizeExample aktualisiert.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/d7b1495909c8a1eefc9fca21afda06daa207ca52>

165. ToBinaryArray-Ergebnispipelinetest geflickt

Ich habe den toBinaryArray-Ergebnispipelinetest repariert.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/85de9c55f673828ee7d73110d60e93f58673b8a2>

166. VkFFT aktualisiert

Ich habe die VkFFT geupdated.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/332bd8afbe7e8a292ff1c8f5de6aa23a57d349e1>

167. VkFFT-Test mit Grösse von 2^{14}

Durchführen des VkFFT-Tests mit Größen von 2^{14} .

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/9b15e6815e7dce186569822615cb80dee483e6b3>

168. Problembehebung mit Blockgrösse von 2^{14}

Durch das Allozieren des Grafikspeichers für eine Grösse von 2^{27} funktioniert der Algorithmus mit einer Blockgrösse von 2^{14} .

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/b8228cc97357afdc6c9fd43909338157410ebb9>

169. Ursache für das 2^{14} - Problem gefunden

Ich habe die Ursache für das 2^{14} -Problems gefunden. Das Problem liegt beim Vuda-Memorymanagement und kann darum nicht behoben werden!

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/9c164bbc15638865000155a750007be8a3d030bf>

170. Standard wieder auf 2^{27} -Pipeline-Test gesetzt

Von den 2^{14} wieder zurück zu den 2^{27} -Pipeline-Test gewechselt.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/92e4c5bec5a6d5cdc8dc1c48d79b4d0516ee09b9>

171. Vorbereitung Linux Docker-Test

Vorbereiten des Codes für den Linux Docker-Tests.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/569bf87648a4717fd0ca821fd23042a76c82ed8a>

172. Keine Wiederverwendung des Toeplitz Matrix Seeds

Keine Wiederverwendung des Toeplitz Matrix Seeds.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/045f5448270c09359f4c5124ab930cd7290b0447>

173. Azure Pipeline: GLSL-shaders

Das Kompilieren der GLSL-shaders wurde zu der Azure Pipeline hinzugefügt.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/d19d819837eeb65e5c139558b8c790771ae594b1>

174. Azure Pipeline: Code-Einrückungen

Azure-pipelines.yml: Code-Einrückungen übersichtlicher gestaltet.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/0723f52d6fb70e8f908d97dc2acf4b07e5e2d9be>

175. Azure Pipeline: cmake als einzelner Task

In der Azure Pipeline ist der Prozess *cmake for glslang* jetzt ein einzelner Task.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/115c1837c7e20427587cb2c1f7944f69f71847a7>

176. Azure Pipeline: cmake in zwei Tasks aufgeteilt

Ich habe in der Azure Pipeline den Prozess *cmake for glslang* neu in zwei Tasks aufgeteilt und die Einrückungen korrekt gestaltet.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/8f8a1a4112eb779a2861f352eb9f664260eac688>

177. UnitTestToBinaryArray repariert

Der UnitTestToBinaryArray wurde repariert.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/ed8e8a4073a7b818f3463a31d9a1c6e735af35e9>

178. Memoryproblem bei Vuda

Bei Vuda erhalte ich beim calculateCorrectionFloat ein Memoryproblem.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/a3b8db6a02fcfd0416fac7079dd815c4599a585f>

179. Fehler beim Speichermanagement behoben

Wird der correction_float_dev gesetzt, dann wird auch der count_one_of_global_key auf den gleichen Wert gesetzt, wegen eines defekten Speichermanagements. Ich konnte den Fehler beheben.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/e9a3238f393f3b867b39913477715b4a4792ceab>

180. UnitTestBinInt2float repariert

Der unitTestBinInt2float wurde repariert.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/bf75136a32ddfe6f616912248ea295dfb158ec62>

181. Azure Pipeline: Kompilieren der GLSL-shaders

Die Azure-pipelines.yml muss die GLSL-Shaders kompilieren.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/0be85e116d8b292fe243cba738ca10a5c93b9c2>

182. Azure Pipeline: Unit Tests for Cuda

Ich habe der Azure Pipeline den Unit Tests for Cuda implementiert.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/6e13406ca9382ccb302e741b4cf4b47b05708f6b>

183. Vulkan: Limite für unitTestBinInt2float

Der unitTestBinInt2float für Vulkan musste limitiert werden. Der Testbereich liegt zwischen 2^{10} und 2^{26} .

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/3c0952eeb7143627f96d22cc8b7f536e397c5b53>

184. Speedtest für Vulkan und Cuda

Der Speedtest läuft jetzt auf Vulkan und Cuda erfolgreich.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/6f9c06a331d413ce78a26777dfdab7feb5fcbe65>

185. Terminierung des Programms verbessert.

Die verzögerte Beendigung des Programms brachte Dinge durcheinander. Dies wurde behoben.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/a3ad5bc94dfa93ce1b2b491cd677d9dff9daa9>

186. Problem mit zu kurzen Unit-Tests behoben

Das Problem mit verwaisten Threads wurde behoben. Diese verursachten Probleme bei zu kurzen Unit-Tests.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/22923de86caaee5409c6df9f6a0d89a749527e6>

187. Speedtest Zeitmessung optimiert

Die Zeitmessung des Speedtests wurde verbessert. Es muss nicht mehr auf den Datenprovider gewartet werden.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/3948cd80788394d22c0e39c9e7b39c4d146a5a46>

188. Speedtest mit 10 Runden pro Test

Für die Speedtests wurde eine Limite von 10 Durchläufen pro Test eingeführt.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/626a1d6696b09c46b28ef4aff1ab22dec65ba676>

189. Azure-Artefakte geupdatet

Ich habe die Azure-Artefakte geupdatet.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/b294bd7c997a5f099d2e54fc6289b8a99913438b>

190. Speedtest mit Aufwärmung

Vor der Durchführung des Speedtest werden zuerst Aufwärmrunden durchgeführt.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/9f22e9d73be141c80102a17985e799c8bab5ff40>

191. Negativer Speedtest-Index

Ich habe neu int32_t anstelle von uint32_t für den Speedtest-Index eingeführt, weil dieser in der Aufwärmphase negativ sein kann.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/22ea3a81157b9f6581b82d3ae3a7ca4c56500606>

192. Beispiele für Azure-Pipelines

azure-pipelines.yml: Build examples

azure-pipelines.yml: Copy examples to Release debugging

azure-pipelines.yml: More copy examples to Release debugging

azure-pipelines.yml: Fixed "Copy examples to Release"

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/5b06bf417ad41c995d8f927b3defc2c9bd6eef40>

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/1608c975e9ea9f8ae839c8158643a42226fe3ab4>

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/26721e8e84dc8adb60435e716826513055209e17>

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/256a9e594c25f08b8a1925a4e798108973ec998f>

193. Array von pointer anstelle Zeigerarithmetik

Neu werden Array von pointer anstelle von Zeigerarithmetik verwendet. Damit Probleme bei der Vuda-Speicherverwaltung vermeiden werden können, da die zu undefiniertem Verhalten führten. Dadurch wurde die Lesbarkeit des Codes stark erhöht.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/61e8e3c860d9962c30a32b9cbdb28efc4fe07515>

194. Tests für LargeBlocksizeExample

Ich habe Tests für LargeBlocksizeExample hinzugefügt.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/e92e11e10c0ead54821cd58a7e5e226d275d9bd5>

195. Problembehebung beim LargeBlocksizeExample-Test

Das Problem mit den LargeBlocksizeExample-Tests wurde behoben

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/ba9a1a33dfea4c7c2d37b51309d598bd785ba79a>

196. Standard für gpu_device_id_to_use

Den Standard für die gpu_device_id_to_use habe ich neu mit 0 festgelegt.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/d8991b910cfe603aa224621c0ea9e948d6d94e7b>

197. Taskkill

Ttaskkill: /F gibt an, dass Prozesse zwangsweise beendet wurden sollen. Dies wird im LargeBlocksizeTest verwendet um die PA zu schliessen.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/7fe015def9826cebb85df7bcb7370ffbbea11ba5>

198. Problembehebung in stderr

„wmic Process call create“ verursacht eine leere Zeile in stderr, die dazu führte, dass die Azure-Pipeline aufgrund von failOnStderr fehlgeschlug. Dies wurde nun behoben.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/94b43b35967554cf6cd4c22f4129f1abc8ba2335>

199. Speedtest mit sample_size von 2^{27}

Der Speedtest sollte immer eine sample_size von 2^{27} verwenden.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/881146af9a2ae9f2984ca1e0f803723aba138d23>

200. Fehlerbehebung beim Large Blocksize Test

Das command line argument --factor exp wurde implementiert. Dieses überschreibt, respektive hat höhere Priorität, sodass der Large Blocksize Test funktioniert, auch wenn factor_exp in der Konfigurationsdatei auf 27 gesetzt ist.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/c6c9f88c2f6eb693a8a54d802ef94f2a450ac41e>

201. LargeBlocksizeExample Debug-Ausgabe

Ich habe die LargeBlocksizeExample Debug-Ausgabe verbessert

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/f1a01a00472bf3471484e08b6c9c3adbb103ae31>

202. Reparatur des Vulkan ToBinaryArray

Der Fehler verursachte ein stundenlanges Suchen und Debuggen. Es war ein falsch platziertes abs (Runden), welches nur in gewissen Randfällen zu falschen Resultaten führte. Die Behebung war dann natürlich simpel.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/8627e71a773c1e519070c1bb002e457e56ad640a>

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/c0043ae519e6e3d98739800f15ddafc7d440232>

203. Fehlerbehebung

Beim Wechsel von der Zeigerarithmetik auf Array of pointer tauchten nun beim readMatrixSeedFromFile und beim readKeyFromFile Fehler auf. Diese habe ich jetzt behoben.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/8c8f1c331c8dc3c85f44ec451cf82c5b2c751f1>

204. VkFFT 2^{14} -Test

Ich habe eine Testbasis für Vulkan VkFFT mit einer Blockgrösse von 2^{14} erstellt.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/f4a3d1f49f3922c9ad2a7aa8ccf0365e42f6b149>

205. VkFFT 2^{14} -Test mit erstem Block

Der erste Block beim Test der Vulkan VkFFT mit 2^{14} Blöcken funktioniert, wenn das zero padding entfernt wird.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/600c5d1ece3b7fbc36000def6750af240b579ce5>

206. VkFFT 2^{14} - zero padding Fehler behoben

Da beim Test der Vulkan VkFFT mit 2^{14} Blöcken bei Randfällen mit dem zero padding Fehler auftraten, diese aber bei grösseren Blöcken nicht auftraten, war die Ursachensuche nicht ganz einfach. Doch ich konnte das Problem erfolgreich beheben.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/712a820315caa512507f94c148b5f6c2004e430a>

207. Pipelinetests Präzisionskorrektur

Ich habe vermutet, dass die Präzisionsanforderung die Ursache für das Fehlschlagen des Tests war. Nach der Behebung des zero padding Fehler (siehe vorherigen Eintrag) habe ich festgestellt, dass die Präzisionsanforderung keinen Fehler verursachte und ich habe sie wieder auf die ursprüngliche Einstellung zurückgestellt.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/eddef39f3f649dc0acc5566163202cdb57d1807c>

208. Neuplanung bei Komprimierungsfaktoränderung bei Vulkan

Wenn der Key-Server einen anderen Komprimierungsfaktor an den PA-Algorithmus sendet, dann muss bei Vulkan die FFT neu geplant werden.

Ich habe versucht den Berechnungsunterschieds bei fft_zeropad_left zwischen dem LargeBlocksizeExample-Algorithmus und dem normalen Algorithmus zu finden.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/5c3e1c64e6ee2b4ca0631db5690faa55ab4bc6f0>

209. Cuda Speicherzuweisung

Cuda hat sich mehr Memory als erforderlich zugewiesen. Dies habe ich jetzt behoben.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/d3b0d62de4705eb0fc3635261805d5765605507a>

Fehlende Klammer bei cudaCalloc hinzugefügt.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/733fd2c5a17593759d7d2d8d979a3220cbbae617>

210. Vereinheitlichung der Komprimierungsfaktorberechnung

Der Unterschied in der Berechnung zwischen dem LargeBlockSizeExample- Algorithmus und der Berechnung mit dem normalen Algorithmus wurde behoben.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/a466e4a917ea48d17952a805a61f4e999f651641>

211. Seed caching

Der Toeplitz-Seed wird für die ganze Diagonale gecached (siehe Abbildung 43). Dadurch wird die Performance massiv erhöht.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/996fb540ea9e389171eafa9cdbba88b0db7abc74>

212. Static Seed

Reuse_seed_amount = -1 für einen static seed funktioniert wieder.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/fcd3a42d4eab7c5436000aa29ecb9096994ec4e4>

213. Seed caching bei Cuda

Behoben, dass Cuda mit Seed-Caching nicht richtig funktionierte.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/1e2faa6d9d596a77830f505d016d9a13cb95c3cf>

214. Verify_ampout

Verify_ampout deaktiviert, da sonst der „Large Blocksize Test“ fehlschlägt.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/13ea5a886264e143e1f3defb56a0fea6723a4eed>

215. Azure-pipeline: LargeBlocksizeTest

Der LargeBlocksizeTest funktioniert jetzt in der Azure-Pipelines sowohl für Cuda und für Vuda!

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/c688befa73a196a1e10e4f67d7b832684060d515>

216. Pipeline-Tests wieder mit 2^{27}

Die Pipeline-Tests habe ich wieder auf 2^{27} zurückgesetzt.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/bd685da77a2ede6f994bec0f17c37cd3aa7550f6>

217. Arbeitsanzeige

Vor und nach den beiden langen Pipeline-Tests soll etwas geschrieben werden, damit der Benutzer nicht denkt, dass das Programm feststeckt.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/410eb30245eabe5333fde7d44d3641aab4483a27>

218. VkFFT_2_pow_14_vulkan_testing zusammengefügt

'vkFFT_2_pow_14_vulkan_testing' Branch mit master zusammengefügt.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/3dcc3c8cd814d8cb8ca79cb10fd67d4155bcefab>

219. Speicherberechnung Vulkan

Das Problem bei der Berechnung des erforderlichen Speichers von Vulkan di1 und di2, dass durch die Verwendung von min anstelle von max verursacht wurde, wurde behoben.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/e8a5854768664aed717da575bcd185cb10fcfb7f>

220. Anpassungen beim Config

Folgende Anpassungen am Config wurden vorgenommen: Server einschalten und verify_ampout ausschalten, da sonst der "Large Blocksize Test" fehlschlägt.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/ae29fe317df32938163540314d4f806e543971b0>

221. AzureCLI und azure-devops Installation

Ich habe die downloadAssets.cmd verbessert. Es wurden die AzureCLI und die azure-devops Installation repariert. Dies war sehr schwierig und aufwändig wegen den Windowsberechtigungen.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/ba9e1caf99bc14e8bcc3a18a9305cc7c5edc36be>

222. VkFFT aktualisiert für bessere Fehlerbehandlung

Die VkFFT wurde aktualisiert, damit die Fehler- und Problembehandlung verbessert werden konnten. Die meisten Kompilierungswarnungen wurden auch behoben.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/636197bb60e80870f0a5c7f3482e085b047f00c7>

223. Kompatibilitätsüberprüfung und Grafikkartenanzeigen

Ich habe eine Prüfung beim Programmstart eingebaut, welche verifiziert ob der installierte Grafiktreiber mit dem installierten Cuda-toolkit übereinstimmt. Ist dies nicht der Fall, dann wird eine Fehlermeldung ausgegeben. Bei jedem Programmstart werden Informationen über die installierten Grafiktreiber, das installierte Cuda-toolkit und alle mit dem PC verbundenen Grafikkarte, sowie die für den Algorithmus verwendete Grafikkarte angezeigt.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/19e4789530e73879ea4ff8c81fd35db2a08f348d>

224. Speedtest repariert

Speedtest-unabhängige stdoutput muss ein #-Präfix haben, damit es den Speedtest nicht durcheinanderbringt.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/2e1ea3d6d188976c2715cc1777102aa838020553>

225. PLAN_CUFFT template entfernt

Einige Warnungen behoben und die Lesbarkeit verbessert, indem die PLAN_CUFFT template entfernt wurde.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/ed1c3283eb0cad712cb8f61ded715eef54f325b>

226. Linux Makefile für LargeBlocksizeExample

Damit das LargeBlocksizeExample auch auf Linux ausführen zu können, habe ich ein Makefile geschrieben.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/dcb9b6a1cc90b73f29425c3bfd126b21d4b03b88>

227. LargeBlocksizeExample-Warnungen behoben

Ich habe alle LargeBlocksizeExample-Warnungen behoben.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/07335a0c762a4af6bbfac80ed5002a0c478f6cfa>

228. PrivacyAmplification Vulkan- Algorithmus auf Linux!

Nun kann ich einen grossen Erfolg feiern, weil jetzt der ganze PrivacyAmplification Vulkan-Algorithmus auf Linux fehlerfrei läuft!!! Das Linken der statischen Bibliotheken dauerte lange, da die Reihenfolge korrekt sein musste.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/daf299a94a0d08a75b4b4f01eec0f943c0fd0724>

229. Ausführbare Dateien als ausführbar markiert

Ausführbare Dateien im Git-repository als ausführbar markiert.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/46e229d950c3b21dd006d9d852f5defd19f97588>

230. Speedtests auf Linux

Beide Speedtests, also sowohl der Cuda- wie auch der Vulkan-Speedtest laufen nun auf Linux.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/e2347cd79fd4670edecc682b8bb9944cb1b6d1a6>

231. Run.sh führt Cuda aus

Run.sh sollte die Cuda-Version ausführen, da sie die Einzige ist, die automatisch mit dem Docker-Container erstellt wird.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/bf1e8add936599bc7e4a9d27cb47ebc8978acb20>

232. Statischer Speedtest

Ich habe den statischen Speedtest wieder repariert. Irrtümlich wurde zweimal der dynamische Speedtest ausgeführt.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/995ae0c4ee403f52b65b273d4fd688fe8287982e>

233. CudaSetDevice erstellt

Durch das neue vudaChunkSize Argument kompilierte die Cuda-Version meines Algorithmus nicht mehr. Dies wurde behoben indem ich speziell für Cuda eine cudaSetDevice erstellt habe.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/eb13c114c877228689397ae4443e12c2d1bab7d1>

234. Vuda-Memorymanagement überarbeitet

Die Vuda-Speicherverwaltung wurde korrigiert, indem Ausrichtungsanforderungen und eine angebbare Chunk-Größen-Funktion implementiert wurden. Di1 und di2 können nun endlich unter 256 MB groß sein und es wird immer noch den korrekten, notwendigen Speicher-Chunk für die VkFFT alloziert. Dieses Problem versuchte ich, seit ich mit Vulkan angefangen habe, zu lösen. Bis zu diesem Commit leider erfolglos. Ich bin sehr froh, dass ich das Problem nun doch noch lösen konnte.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/commit/cbf9c5e483eb135d81e0c2d438e70e9e9b413e20>

Zusammenfassung der Forschungsmitarbeit

Grafikkartenunabhängigkeit - Vulkan:

Das Hauptziel dieses Semesters war meinen Algorithmus NVidia unabhängig zu machen. Der Grund dafür ist, weil für billige GPUs AMD und Intel führend sind. Vor allem APUs wären für eine kommerzielle Nutzung wegen ihres Preises sehr interessant. Um dies zu erreichen, musste ein Ersatz für Cuda und cuFFT gefunden werden. Die erste Idee war ROCm und rocFFT- die AMD Alternative zu Cuda und cuFFT. Durch Hipify war es möglich mein Programm in wenigen Tagen auf diese Plattform zu portieren. Leider stellte sich heraus, dass ROCm nur auf wenigen highend AMD GPUs korrekt unterstützt wird und somit nicht meinen Anforderungen entsprach. Somit blieb nur noch Vulkan mit VkFFT. Dieser Schritt war sehr riskant und es bestand eine grosse Wahrscheinlichkeit, dass ich auf dem Weg einem unlösabaren Problem begegnen werde, welches es verunmöglicht, dass mein Algorithmus jemals auf Vulkan laufen würde. Da Vulkan zu lowlevel ist, um mit sinnvollem Aufwand meinen Algorithmus zu portieren, entschied ich mich für Vuda, welches eine teilweise Cuda Reimplementierung in Vulkan bietet. Das Kombinieren von VkFFT und Vuda hätte die Vulkan Portierung beinahe verunmöglicht und war nur durch grössere Vuda Modifikationen möglich. Vuda vereinfacht nur die Portierung des CPU Codes von CUDA zu Vulkan. Der gesamte GPU Code musste neu geschrieben werden. Vuda akzeptiert nur SPIR-V Shaders. SPIR-V ist eine intermediate Shader Language. Verschiedenste Shadersprachen wie unter anderem OpenCL und GLSL können nach SPIR-V kompiliert werden. Obwohl OpenCL einfacher gewesen wäre, entschied ich mich für GLSL um das Maximum an Performance herausholen zu können. Glücklicherweise hatte ich vom letzten Semester eine beinahe 100%-ige Unitestabdeckung des GPU Codes. Dies erleichterte eine Neuprogrammierung durch Test first enorm, da wie schon oft erwähnt, keine Möglichkeiten bestehen GPU Code zu debuggen. VkFFT ermöglichte viele neue Optimierungen wie native zero Padding. Zu Beginn fehlte VkFFT multi upload R2C/C2R FFT und ich hätte C2C FFT nutzen müssen. Der Autor von VkFFT war jedoch enorm hilfreich und implementierte dies für mich und behob viele weitere Fehler, welche ich in VkFFT fand. Schlussendlich brachte ich meinen Privacy Amplification Algorithmus erfolgreich auf Vulkan zum Laufen. Somit läuft mein Programm nun mit jeder modernen Grafikkarte, unabhängig vom Grafikkartenhersteller. Ich habe den Algorithmus erfolgreich auf einer AMD Radeon RX 5700XT und einem AMD Radeon 4700U APU getestet. Trotz den grossen Codeänderungen,

welche für Vulkan notwendig waren, gelang es mir, beide Versionen in der gleichen Codebase zu schreiben. Mit Preprocessordefinitionen wird zu Kompilierzeit vulkanspezifischer von cudaspezifischer Code separiert. Also habe ich jetzt einen einzigen Algorithmus der auf allen modernen Grafikkarten läuft!

Grosse Blöcke

Ein weiteres Hauptziel dieses Semesters war die Unterstützung beliebig grosser Blockgrössen. Durch die Limitierung von cuFFT auf 2^{27} waren bis anhin nur Blockgrössen bis 2^{27} möglich. Durch die Portierung meines Algorithmus auf Vulkan und VkFFT ist es möglich, bei GPUs mit viel Grafikspeicher native Blockgrössen von bis zu 2^{32} zu nutzen. Das Ziel war es jedoch einen Algorithmus zu entwickeln welcher unabhängig von der unterstützten nativen Blockgrösse beliebig grosse Blockgrössen unterstützt. Dazu entwickelte ich einen Algorithmus welcher die zu berechnende Toeplitz Matrix in Blöcke zerteilt. Die Blöcke werden einzeln berechnet und wieder so zusammensetzt, dass man das gleiche Ergebnis erhält, wie wenn man den Algorithmus mit der grossen Matrix berechnet hätte. Dies war mathematisch eine Herausforderung, da eine schelle FFT nur Zweierpotenzen als Eingabe haben darf. Erst programmierte ich diesen Algorithmus in Python und portierte ihn danach nach C++. Ich entschied mich diesen Algorithmus komplett von meinem eigentlichen Privacy Amplification Algorithmus zu trennen. Somit erstellte ich ein separates Projekt und die Kommunikation erfolgt ausschliesslich über ZeroMQ. Durch Caching gelang es mir die Performance trotz ineffizienterem Algorithmus einigermassen hoch zu behalten. So ist es nun möglich mit riesigen Toeplitz Matrizen zu rechnen.

«Open source»-Umgebung mit Testinfrastruktur

Eine Open source Umgebung mit guter Testinfrastruktur bereitzustellen war ein weiterer Fokus. Nur so ist es für andere Programmierer in Zukunft möglich dieses Projekt weiterzuentwickeln.

- **Azure Pipeline**

In diesem Semester wurde die Azure Pipeline massiv verbessert. Es werden sowohl die Vulkan wie auch Cuda Versionen des Privacy Amplification Algorithmus und alle anderen Komponenten und Abhängigkeiten automatisch gebaut. Zudem wurde das automatisierte Testen erweitert. Der Speedtest wird nicht nur für Cuda sondern auch für

Vulkan ausgeführt. Die durch die Azure Pipeline erhaltenen Artefakte können heruntergeladen und auf jedem PC unabhängig von der Grafikkarte portabel genutzt werden. Releases zu erstellen wurde grösstenteils automatisiert. Der NSIS Installer wird nicht mehr benötigt.

- **Unit Tests**

- Mein GPU Code hat eine an 100% grenzende Unit Testabdeckung.
- Ein Unit Test besteht oft aus tausenden, wenn nicht Millionen von assert statements.
- Der GPU Algorithmus wird mit einer langsamen aber sicher korrekten CPU-Implementierung verglichen.
- Alle Unit Tests laufen sowohl auf Cuda wie auch auf Vulkan.
- Continuous Testing: Alle Unit Tests sowohl für Cuda wie auch für Vulkan werden durch die Azure Pipeline ausgeführt.

- **Pipeline Tests**

- Es existieren Pipeline Tests für den ganzen Privacy Amplification Algorithmus
 - 12 Pipeline Tests.
- Es existieren Pipeline Tests für den ganzen Large Blocksize Example Algorithmus.
 - 130 Pipeline Tests
- Die Ungenauigkeit von Fliesskommaberechnungen wird berücksichtigt.
- Pipeline Tests lassen sich einfach ergänzen.
- Durch ein Macro wird die Zeilennummer des fehlgeschlagenen Tests ausgegeben.
- Pipeline Tests lassen sich einfach an neue Testdaten anpassen.
 - Während dem Semester wurden beispielsweise Pipeline Tests für eine Blockgrösse von 2^{14} geschrieben um etwas zu testen.
- Durch Pipeline Tests wird jeder Schritt des Algorithmus getestet was eine genaue Lokalisierung des Fehlers ermöglicht.
- Ist ein Pipeline Test erfolgreich, so ist aller Code vor dem Pipeline Test korrekt. Anders als Unit Tests wird hier gleich mit realen Daten getestet. So geht keine Codezeile vergessen und auf das Testergebnis kann man sich verlassen.

- Continuous Testing: Die Large Blocksize Example Pipeline Tests werden durch die Azure Pipeline ausgeführt.
- **System Tests**
 - Das Ergebnis des Privacy Amplification Algorithmus wird durch ein SH3-256 Hash validiert.
 - Key Start, Key Rest und das Ergebnis des Large Blocksize Example Algorithmus wird durch ein SH3-256 Hash validiert.
 - Continuous Testing: Das Ergebnis des Large Blocksize Example Algorithmus wird durch Azure Pipeline validiert.
- **Clean Code**

Im Vergleich zum letzten Semester wurde die Codequalität enorm verbessert.

 - Die meiste Pointerarithmetik wurde durch Arrays mit Pointers auf andere Arrays ersetzt. Dadurch erhöhte sich die Lesbarkeit und Verständlichkeit enorm.
 - Der Mainloop ist neu eine Funktion. Zur Ausführung des Speedtests ist kein goto mehr notwendig.
 - Templates werden nur noch falls notwendig eingesetzt und anstelle inline Functions verwendet.
 - Der GPU Code hat keine Nebeneffekte
 - Es wird nur in durch Argumente übergebene Speicherregionen geschrieben.
 - Vulkan nutzt keine globalen Konstanten und jedes übergebene Argument wird durch readonly/writeonly markiert.
 - Das ZeroMQ Protokoll wurde so verändert, dass es dem Aktor Modell entspricht. Jede Komponente kann zur Laufzeit neu gestartet und sogar aktualisiert werden. Durch flow control wird sichergestellt, dass langsamere Konsumenten nicht von schnellen Produzenten überflutet werden.
 - Durch die Separierung vom Privacy Amplification Algorithmus mit dem MatrixSeedServerExample, SendKeysExample und ReceiveAmpOutExample wurde eine schöne Trennung zwischen Komponenten umgesetzt. Diese wurde mit dem Algorithmus für grosse Blöcke (LargeBlocksizeExample) beibehalten.

- Alle wichtigen Funktionen sind im Header file sowie im Config file gut dokumentiert.
- Ein **Guide** wurde geschrieben.
- **Linux + Docker**
 - Es existiert ein Dockerfile um meinen Algorithmus unter Linux + Docker auszuführen, siehe Guide.

Benchmarking-Toolkits

- Um die Performance meines Algorithmus zu testen habe ich den im letzten Semester entwickelten Speedtest weiter verbessert. Durch Aufwärmen entstehen konsistenter Resultate. Auch wurde das Speedtest Python Script von der Azure Pipeline separiert, sodass es unabhängig von der Azure Pipeline gestartet werden kann. Zudem existiert nun eine Vulkan sowie eine Cuda Version des Speedtests. Beide werden während der Azure Pipeline automatisch ausgeführt. Der Speedtest funktioniert sogar falls die Blockgrösse nicht auf 2^{27} gesetzt wurde.
- Zusätzlich zum Speedtest existiert nun ein integriertes Benchmarking Tool welches für jeden Schritt des Algorithmus die benötigte Zeit anzeigt. So können verschiedene Implementierungen verglichen werden. Dadurch ist es möglich Flaschenhälse im Algorithmus zu erkennen und durch Optimierung zu eliminieren. Durch dieses Benchmarking Tool war es möglich die Vulkan Implementierung so zu optimieren, dass diese mittlerweile schneller als die Cuda Implementierung auf derselben Grafikkarte ist.

Ergebnis des Benchmarking-Toolkits mit der finalen Programmversion:

Function	Cuda	Vulkan	Differnece	Differnece %
			Vulkan - Cuda	((Vu/Cu)*100)-100
wait_for_input_buffer [ms]	0.006	0.006	0	0
cleaned_memory [ms]	0.003	0	-0.003	-100 %
set_count_to_zero [ms]	0.007	0.172	0.165	2'357 %
binIntffloat_seed [ms]	2.782	3.043	0.261	9 %
binIntffloat_key [ms]	1.762	1.991	0.229	13 %
calculateCorrectionFloat [ms]	0.068	0.226	0.158	232 %
fft_key [ms]	7.225	6.283	-0.942	-13 %
fft_seed [ms]	7.425	6.638	-0.787	-11 %
setFirstElementToZero [ms]	0.094	0	-0.094	-100 %
elementWiseProduct [ms]	2.397	2.467	0.07	3 %
ifft [ms]	7.352	6.65	-0.702	-10 %
wait_for_output_buffer [ms]	0.005	0.003	-0.002	-40 %
toBinaryArray [ms]	1.725	1.612	-0.113	-7 %
Total [ms]	30.868	29.094	-1.774	-6 %
Speed [MBit/s]	4348.055	4613.213	265.158	6 %

Ergebnisse des Speedtest mit der finalen Version:

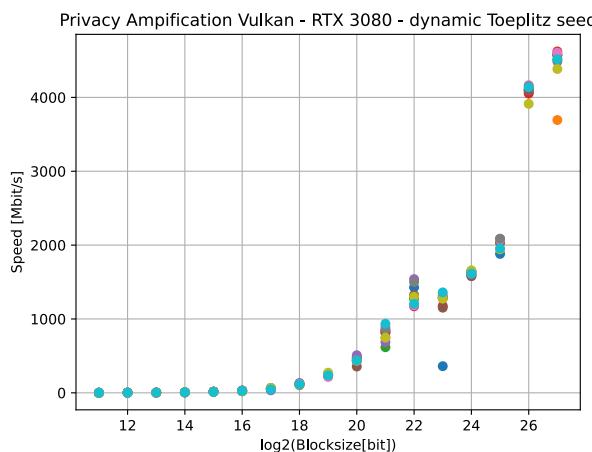


Abbildung 46: Speedtest mit Vulkan

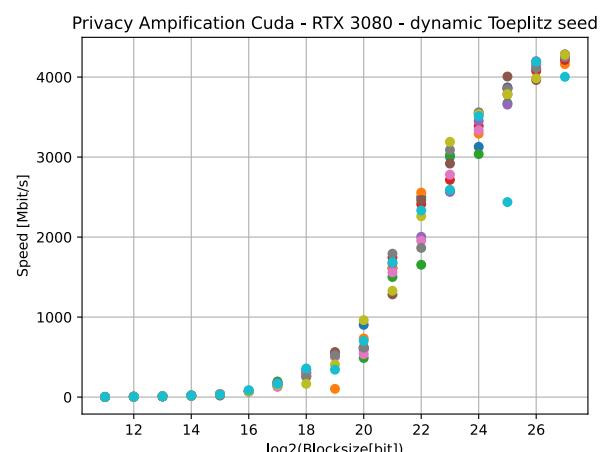


Abbildung 47: Speedtest mit Cuda

Schlusswort

Es sind nun 1 ½ Jahre vergangen, seit ich von der HSLU die Möglichkeit der Forschungsmitarbeit zum Thema *Fast Privacy Amplification on GPUs* erhalten habe. Für mich war diese Mitarbeit ein Glücksfall und ich bin der HSLU und vor allem Herrn Diethelm sehr dankbar, dass ich mich drei Semester lang mit diesem aktuellen und interessanten Thema auseinandersetzen durfte. Äusserst dankbar bin ich natürlich Esther Hänggi, dass sie mich 1 ½ Jahre an ihrer Forschung teilnehmen liess und mir die theoretisch-mathematischen Inputs so lieferte, dass ich daraus einen entsprechenden Algorithmus entwickeln konnte. Roland Christen war mir mit seinen kreativen Inputs und seinem fundierten Wissen über Docker und Testing eine grosse Hilfe. Neben den spannenden Aufgaben waren die Kontakte zu anderen Forschern und Forschergruppen für mich eine grosse Bereicherung.

Das Programm läuft nun auf allen modernen Grafikkarten und ist sogar Betriebssystem unabhängig. Es werden zwei verschiedene Backends, Cuda und Vulkan, unterstützt. Der Algorithmus kann für beliebig grosse Blockgrössen ausgeführt werden. Der Grafikkartencode beinhaltet eine nahezu 100% Testabdeckung. Durch die Azure Pipeline werden automatisch sämtliche Test- und Performancemessungen vorgenommen und ein portables Entwicklungs-bild erstellt.

Ich bin schon ein wenig stolz, dass meine Forschungsmitarbeit mit einem voll funktions-fähigen und recht komplexen Programm abgeschlossen werden konnte. Trotzdem hoffe ich auf eine Weiterentwicklung und sinnvolle Verwendung des Programms.

Abbildungsverzeichnis

Abbildung 1: PA erfolgreich auf ROCm kompiliert.....	5
Abbildung 2: hipFFT erfolgreich kompiliert.	6
Abbildung 3: ROCm Nvidia CentOS 8	8
Abbildung 4:PA auf AMD RX 5700 XT ausgeführt.	8
Abbildung 5: Vuda-Fehler	9
Abbildung 6: Vuda Issue	10
Abbildung 7: Vuda Pull-Request.....	10
Abbildung 8: vkFFT build Vulkan	11
Abbildung 9: vkFFT build CUDA.....	11
Abbildung 10: vkFFT update	11
Abbildung 11: vkFFT ausgeführt.....	12
Abbildung 12: SPIR-V Umgebung	13
Abbildung 13: Skript für GLSL-Shader zu kompilieren	15
Abbildung 14: Neuer unitTest	16
Abbildung 15: CUDA-Code für calculateCorrectionFloat	17
Abbildung 16: GLSL-Code für calculateCorrectionFloat	17
Abbildung 17: CUDA-Code für setFirstElementToZero	18
Abbildung 18:GLSL-Code für setFirstElementToZe	18
Abbildung 19: CUDA-Code für ElementWiseProduct.....	19
Abbildung 20: GLSL-Code für ElementWiseProduct	19
Abbildung 21: CUDA-Code für binInt2float	20
Abbildung 22: GLSL-Code für binInt2float	21
Abbildung 23: CUDA-Code für ToBinaryArray	22
Abbildung 24: GLSL-Code für ToBinaryArray	23
Abbildung 25: Erfolgreicher vkFFT	24
Abbildung 26: Alternativtests für R2C mit C2C	25
Abbildung 27: Antwort des Programmierers auf Limitation	26
Abbildung 28: Erfolgreiche Tests.....	27
Abbildung 29: PrivacyAmplificationVulkan.....	28
Abbildung 30: Rückmeldung von Tolmachev.....	30

Abbildung 31: Gefixtes Issue.....	31
Abbildung 32: Ausschnitt aus Dokumentation über Kompilierungsflags.....	33
Abbildung 33: VkBuffer aus Vuda-Suchbaum extrahieren	34
Abbildung 34: Fehlermeldung VK_DEVICE_LOST.....	35
Abbildung 35: Ereignisprotokoll.....	37
Abbildung 36: Video_Sheduler_Internal_Fehlerbehebung	38
Abbildung 37: Hardware der Testing VM	38
Abbildung 38: Snapshots der Testing VM.....	39
Abbildung 39: Voll funktionsfähiger PA-Algorithmus auf Vulkan.....	42
Abbildung 40: Skizze der Planung der Blockaufteilung	43
Abbildung 41: Fehlerbeschreibung für VkFFT	50
Abbildung 42: Fehlerbehebung durch Dimitrii	51
Abbildung 43: Aufteilung der Matrix in Blöcke	53
Abbildung 44: Berechnung eines Blocks.....	54
Abbildung 45: Mein pull request.....	65
Abbildung 46: Speedtest mit Vulkan	94
Abbildung 47: Speedtest mit Cuda.....	94