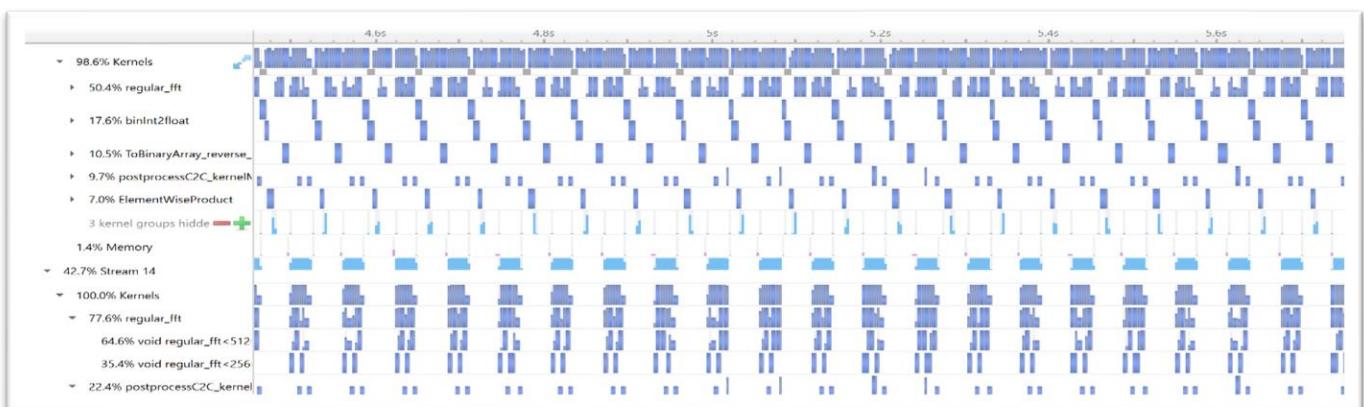


Quantum key distribution fast post-processing

Focus on fast Privacy Amplification on GPUs

Laborjournal

Nico Bosshard, Forschungsmitarbeit HS 2020
 Betreuung: Dr. Esther Hänggi, Christen Roland
 Hochschule Luzern, Informatik, 10. Januar 2021



Inhalt

1.	Zusammenfassung Ferienjob 10. – 21. August 2020	4
2.	Recherchen Ampere Architektur	7
3.	Libtrevisan-py.....	8
4.	Idee für Testing	8
5.	Trevisan - Algorithmus	9
6.	Nintendo Switch.....	10
7.	Grafikkarte GeForce RTX 3080 – Algorithmus ist 60% schneller	10
8.	Testing.....	14
9.	Privacy Amplification auf dem Nintendo Switch	17
10.	Unit Tests.....	19
11.	CalculateCorrectionFloat Unit Test	19
12.	Ein Fehler wurde gefunden	21
13.	SetFirstElementToZero Unit Test	21
14.	ElementWiseProduct Unit Test	22
15.	ReadMe	23
16.	Azure Pipeline.....	23
17.	GitHub	27
18.	Self-Hosted Azure Agent	28
19.	ShortGitHash	28
20.	Azure Artifacts.....	29
21.	AssertEquals	30
22.	BinInt2float Unit Test	30
23.	ToBinaryArray Unit Test	33
24.	Vollautomatischer Performance-Graph.....	36
25.	cuLDPC Separation	39
26.	cufft neu planen	39
27.	GitLab Pipelines	41
28.	Absturz behoben	42
29.	Zusammenfassen mehrerer Blöcke.....	43
30.	Zusammenfassen mehrerer Blöcke.....	45
31.	Testcases für beliebige Blockgrößen.....	47
32.	cuLDPC.....	48
33.	Reparieren der bestehenden Tests	48

34.	GitLab Runner.....	48
35.	Fehlerbehebungen	49
36.	Weitere Fehlerbehebungen	49
37.	Nochmals Fehlerbehebungen	50
38.	Neuorganisation von GitLab.....	50
39.	Recherche zu Referenzarbeiten	50
40.	Abstract ergänzt	50
41.	cuLDPC für Veröffentlichung vorbereitet.....	51
42.	Schnelle Generierung von Testcases für beliebige Blockgrößen	51
	Zusammenfassung der Forschungsmitarbeit.....	52
	Schlusswort	53
	Abbildungsverzeichnis.....	54

1. Zusammenfassung Ferienjob 10. – 21. August 2020

Montag, 10.08.2020

- Linux Unterstützung ohne ZeroMQ
- Linux Unterstützung für ZeroMQ in PrivacyAmplification.cu
- Beispiele Reorganisiert
- Linux Unterstützung für SendKeysExample mit ZeroMQ
- Linux Unterstützung für MatrixSeedServerExample mit ZeroMQ
- Linux Unterstützung für ReceiveAmpOutExample mit ZeroMQ
- Zugriffsverletzung durch Interpretierung von value als pointer behoben
- Parameterfehler beim Kompressionsratioaustausch behoben
- keyfile.bin Dateiunterstützung für SendKeysExample
- Leserlichkeit vom SendKeysExample Code verbessert
- Weitere Fehler beim Kompressionsratioaustausch behoben

Dienstag, 11.08.2020

- Keys empfangen von SendKeysExample erhalten nun durch keySetZeroPadding() das richtige Resultat jedoch Verbesserungswürdig
- keySetZeroPadding() in key2StartRest() integriert
- Es werden nur noch RAM Regionen mit Nullen gefüllt welche nicht schon beim letzten Zugriff auf dieser Speicherregion mit Nullen aufgefüllt wurde.
- Es wird nur noch horizontal_block anstelle von key_blocks des empfangenen Keys kopiert um unnötiger Speicherzugriff zu verhindern welcher ansonsten sogar später noch mit Nullen gefüllt werden müsste.
- toeplitz_seed.bin Dateiunterstützung für MatrixSeedServerExample
- Leserlichkeit vom MatrixSeedServerExample Code verbessert
- Automatisches Testen der Korrektheit der PrivacyAmplification durch das berechnen des SHA3-256 Hashes vom Resultat und vergleichen mit dem des korrekten Resultates. Dazu würden 3 verschiedene SHA3 Bibliotheken (tiny_sha3, SHA3IUF und RHash) getestet wobei sich RHash als schnellste herausstellte.

Mittwoch, 12.08.2020

- Alle Warnungen (von allen Projekten und CUDA 10.2) behoben
- Zeitstempel zu MatrixSeedServerExample und ReceiveAmpOutExample
- Regressionsfehler den der Berechnung von vertical_bytes in ReceiveAmpOutExample behoben
- SHA3-256 Verifikation auf Linux portiert
- Linux PrivacyAmplification auf C++14 geupdated
- Linux Meta Makefile erstellt um alle Projekte auf einmal zu bauen
- Skript up automatisch zu bauen starten und stoppen vom ganzen Privacy Amplification System mit rivacyAmplification, SendKeysExample, MatrixSeedServerExample und ReceiveAmpOutExample. Die Projekte werden je in einem separaten durch tmux gesplitteten Konsolenfenster geöffnet. Durch dieses Skript ist entwickeln und testen auf Linux viel komfortabler.
- Docker Integration mit Dockerfile
- SSH Public Key Authentication zu Docker Container
- Der Private Schlüssel wird nun in denselben Ordner kopiert

Donnerstag, 13.08.2020

- SSH Private Key wird nun automatisch in das PuTTY Private Key Format konvertiert
- CUDA DEVICE ID kann spezifiziert werden
- Privacy Amplification System auf gpu04.res.el.eee.intern zum Laufen gebracht.
- Config mit der mini-yaml Bibliothek begonnen
- Präprozessordefinitionen in Variablen umgewandelt da sie sonst nicht konfigurierbar sind

Freitag, 14.08.2020

- Thread pool crash behoben
- mini-yaml auf Linux portiert
- ZeroMQ socket Initialisierung auf Linux behoben
- Rhash in Docker
- Absolut zufälliger NVIDIA CUDA Compiler-Fehler behoben
- Linux config file in bulds durch Hardlinks verbessert
- Shell skripts haben nun auf Git Ausführungsrechte nach klonen
- Mit Codedokumentation begonnen

Samstag, 15.08.2020

- An der Codedokumentation weitergearbeitet

Sonntag, 16.08.2020

- Config Dokumentation fertiggestellt

Montag, 17.08.2020

- Funktionsdokumentation fertiggestellt
- Aller unbenutzte Code gelöscht
- `readMatrixSeedFromFile` und `readKeyFromFile` in eigene Funktionen gesplittet um Lesbarkeit zu verbessern
- Speicher wiederverwendet um weitere 537 MB Grafikspeicher zu sparen
- Linux Kompilierungsfehler behoben

Dienstag, 18.08.2020

- Code Walkthrough Präsentation

Mittwoch, 19.08.2020

- libzmq ist Bibliothekspfad Benutzerneutral gemacht
- libzmq auf neuste Version aktualisiert
- ZeroMQ Fehlerbehandlung verbessert sodass die Kommunikation immer wiederhergestellt werden kann. Dadurch kann jeder Komponente jederzeit unerwartet neu gestartet werden.

Donnerstag, 20.08.2020

- Alle Kompilierungswarnungen behoben
- Speicherdoppelbenutzung intelligenter gestaltet da es zuvor zu einem Konflikt kam.
- Beispielprogramme dokumentiert

Freitag, 21.08.2020

- Version v1.1.0 intern veröffentlicht und per E-Mail an Genf gesendet

2. Recherchen Ampere Architektur

Meine Recherchen über die neue Ampere Architektur sind mehrheitlich abgeschlossen und ich warte nun auf die bestellte neue Grafikkarte.

Zu Trevisan-Extraktoren habe ich mir auf GitHub schon die drei bestehenden Implementierungen angesehen und mich für libtrevisan-py wegen ihrer Einfachheit als Referenzimplementierung entschieden.

Mit der Linux Installation auf dem Nintendo Switch habe ich auch schon begonnen. Ich warte jedoch noch Version 3.2.0 welche in einigen Tagen kommt, da 3.0.1 nicht bootet. Dafür habe ich aber schon mein selbst gebuilddetes Android Pie auf dem Nintendo Switch zum Laufen gebracht.

3. Libtrevisan-py

Ich habe ein Windows-batch-Scrip geschrieben um libtrevisan-py mit einem Doppelklick ausführen zu können. Das libtrevisan-py ist in italienischer Sprache und ich habe das wichtigste auf Englisch übersetzt. Damit die Parameter für die Trevisan -Extraction nicht bei jedem Programmstart eingegeben werden müssen, habe ich sie im Code fixiert.

Um zu sehen, wieweit die Trevisan -Extraction schon fortgeschritten ist habe ich eine Prozessbar hinzuprogrammiert.

Der Trevisan-Code wurde gesäubert indem unbenutzte Dateien und Codeteile gelöscht wurden. So wurde z.B. die unbenutzten Funktionen numbthy und finitefield entfernt. Ich habe die Number-Theory-Python auf die neuste Version aktualisiert.

Es gab nun ein Performanceproblem mit readconway. Um die Geschwindigkeit von readconway markant zu verbessern habe ich eine ConwayLookupTable generiert mit dem Maximum von p und e sowie der sortierten Menge aller p und e. Die implementiert Funktion findet nun die Conway Polynome in einer Laufzeit von O(1) anstelle von O(n). Der alte Code wurde gelöscht.

Die libtrevisan-py ist ein einfaches Python-Projekt. Um den Entwicklungsprozess zu vereinfachen und die Leistungsmesstools vom Visual Studio verwenden zu können, habe ich das ganze Projekt in ein Visual Studio 2019 Projekt umgewandelt.

<https://gitlab.enterpriselab.ch/qkd/cpu/libtrevisan-py/-/commit/939f03a00144d9ce40c8bac23f065d556efbccfd>

<https://gitlab.enterpriselab.ch/qkd/cpu/libtrevisan-py/-/commit/32915b7f984fb3d840850dc8437f9266887ebc1c>

<https://gitlab.enterpriselab.ch/qkd/cpu/libtrevisan-py/-/commit/0a2fab01eac08f77cb03c14632d25af9888d9f3e>

<https://gitlab.enterpriselab.ch/qkd/cpu/libtrevisan-py/-/commit/ca88f4397b7034bef38cd3f8911bf3efe360b888>

<https://gitlab.enterpriselab.ch/qkd/cpu/libtrevisan-py/-/commit/f513723e324c88e6499c1d5a082dd2d70db08d95>

<https://gitlab.enterpriselab.ch/qkd/cpu/libtrevisan-py/-/commit/b78f0c72a2c2d33a36355104cc300ca47d99aab1>

<https://gitlab.enterpriselab.ch/qkd/cpu/libtrevisan-py/-/commit/4654380b4ebff3c96cf30de924074446caaf965f>

4. Idee für Testing

Idee für das Testing: Um nicht die ganzen Zwischenresultate bei den Tests speichern zu müssen, soll nach jedem relevanten Zwischenschritt der erhaltene SHA3 Hash des aktuellen Zwischenresultats mit dem erwarteten Hash verglichen werden. So müssen keine Zwischenresultate gespeichert, eingelesen und verglichen werden.

5. Trevisan - Algorithmus

Durch das Umschreiben der `reduc_table` in NumPy konnte die Performance verbessert werden.

Das Resultat des Trevisan- Algorithmus kann nun mittels Hash manuell mit dem Referenzhash des nicht optimierten Trevisan -Codes verglichen werden.

Die vorausberechneten Conway Polynomials wurden aus Entwicklungsumgebungsperformancegründen aus dem Visual Studio 2019 Project entfernt.

Sie werden beim Ausführen jedoch dennoch benutzt.

Die `FiniteFieldElt` Initialisierung wurde verbessert. Der nutzlose `trevisanextractor` Code wurde entfernt und die Lesbarkeit des `OneBitExt` Funktion wurde verbessert.

Nun kann das Resultat des Trevisan - Algorithmus automatisch verglichen und überprüft werden. Es ist ein intelligenter aber leider langsamer Algorithmus.

<https://gitlab.enterpriselab.ch/qkd/cpu/libtrevisan-py/-/commit/0f44b5a59135c4b4b6febe4f7850b27df9d1f46e>

<https://gitlab.enterpriselab.ch/qkd/cpu/libtrevisan-py/-/commit/1ffd99339e98e79ea83a6c112e403baca3ef5c4b>

<https://gitlab.enterpriselab.ch/qkd/cpu/libtrevisan-py/-/commit/fa5a34575bad8311c6586181f0d5a5135c7a746a>

<https://gitlab.enterpriselab.ch/qkd/cpu/libtrevisan-py/-/commit/edc016941bd0a9554b9b5e6043630841ad43850c>

<https://gitlab.enterpriselab.ch/qkd/cpu/libtrevisan-py/-/commit/31c3df7da70477c232a72056753fbf51aacacd2c>

<https://gitlab.enterpriselab.ch/qkd/cpu/libtrevisan-py/-/commit/f469e0b12a5b19bc220ca6366846cad85d9a6374>

<https://gitlab.enterpriselab.ch/qkd/cpu/libtrevisan-py/-/commit/d7c29d6c42f582db0f92b9bee2efb00a4b9414d3>

<https://gitlab.enterpriselab.ch/qkd/cpu/libtrevisan-py/-/commit/bb140fb5790f72dc5de8b39af04d1020e3e3d4f>

<https://gitlab.enterpriselab.ch/qkd/cpu/libtrevisan-py/-/commit/a7557458698dd993888a5063e39611a159a35e28>

<https://gitlab.enterpriselab.ch/qkd/cpu/libtrevisan-py/-/commit/824aa5216c29658af0071fc2f63316d2830bacf4>

<https://gitlab.enterpriselab.ch/qkd/cpu/libtrevisan-py/-/commit/890092dbe6d2ee4fa82e5295542907616d6dce0>

<https://gitlab.enterpriselab.ch/qkd/cpu/libtrevisan-py/-/commit/8de5738ec250e92bac771be8ed90d4c414ae7f8c>

<https://gitlab.enterpriselab.ch/qkd/cpu/libtrevisan-py/-/commit/159c575b189c45e912cf20531aa3104993d1cbaf>

<https://gitlab.enterpriselab.ch/qkd/cpu/libtrevisan-py/-/commit/ec2687e5a4e111b98ec43698e8893516cd5c99ea>

<https://gitlab.enterpriselab.ch/qkd/cpu/libtrevisan-py/-/commit/645d46b8c4e21894668012a5648240b686087742>

<https://gitlab.enterpriselab.ch/qkd/cpu/libtrevisan-py/-/commit/c3a15d55fce5d25655a1f2c7c1a8137a56da0323>

<https://gitlab.enterpriselab.ch/qkd/cpu/libtrevisan-py/-/commit/ddad9144fa118945ca6b9cea1258446103120c14>

<https://gitlab.enterpriselab.ch/qkd/cpu/libtrevisan-py/-/commit/dd7063c341862a4bdd7e559060c56e50f978020f>

6. Nintendo Switch

Ich habe nun Ubuntu 3.2 auf dem Nintendo Switch zum Laufen gebracht. Leider hat sich herausgestellt, dass das neustes ARMHF AnyDesk für Raspberry Pi nur noch auf dem Raspberry Pi läuft. Ich hätte noch eine alte Version die gehen würde. Trotzdem habe ich mich zur Alternative mit XRD (Linux-Implementierung von Microsoft Windows Remotedesktop Protokoll) entschieden. Ich kann mich per SSH mit dem Nintendo verbinden. Zudem habe ich ein selbstgebildetes Android Q auf dem Nintendo Switch zum Laufen gebracht.

7. Grafikkarte GeForce RTX 3080 – Algorithmus ist 60% schneller

Nach wochenlangem Warten habe ich meine Grafikkarte MSI GeForce RTX 3080 VENTUS 3X 10G OC (10GB) endlich erhalten. Ich hatte enormes Glück da sie auch jetzt am 12.10.2020 noch nirgends erhältlich ist. Der einzige Grund, dass ich überhaupt eine RTX 3080 erhielt war meine sofortige Bestellung zum Releasezeitpunkt. Durch das geringe Angebot und die hohe Nachfrage sind mittlerweile auch die Preise enorm gestiegen. Trotz meines schnellen Bestellens habe ich glücklicherweise eine Grafikkarte der 2-ten Hardwarerevision erhalten, sodass das Kondensatordesign 10 Keramikkondensatoren enthält. Dies führt dazu, dass die Grafikkarte auch beim schnellen Wechseln auf die Boost-Frequenz stabil ist und nicht mehr wegen instabiler Stromversorgung abstürzt.



Abbildung 1: Vergleich Hardwarerevision 1 + 2

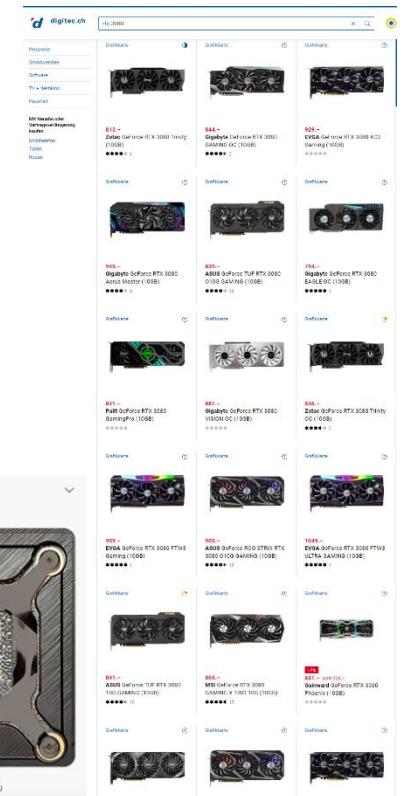


Abbildung 2: «Erhältlich» 3080 GPUs auf Digitac

Auch habe ich mich mit dem Übertakten der Grafikkarte beschäftigt. Ampere GPUs werden bei der Herstellung in 3 Bins aufgeteilt. Der beste Bin nimmt Nvidia, Gerüchten zufolge, für ihre Founders Edition. Die im besten Bin noch übriggebliebenen GPUs werden an die

Boardpartner weitergeben. Diese nutzen die dann für High-end-Modelle. Mit der Ventus habe ich die billigste Ampere GPU von MSI gekauft und somit eine GPU aus dem schlechten oder vielleicht mittleren Bin erwartet. Glücklicherweise erhielt ich jedoch eine der Besten vom allerbesten Bin, was mich sehr freut. Dadurch war mir ein mehr oder wenig stabiles Übertakten auf 1860 MHz mit 2100 MHz peak und über 20 Gbit/s memory speed möglich. Alle Messungen in diesem Tagebuch werden, falls nicht anders spezifiziert, ohne Übertaktung durchgeführt. Jedoch werde ich später auf diesen Vorteil meiner Grafikkarte zurückgreifen und vergleichen wie viel Performanceunterschied ein Übertakten für mein Privacy Amplification Algorithmus ausmacht.

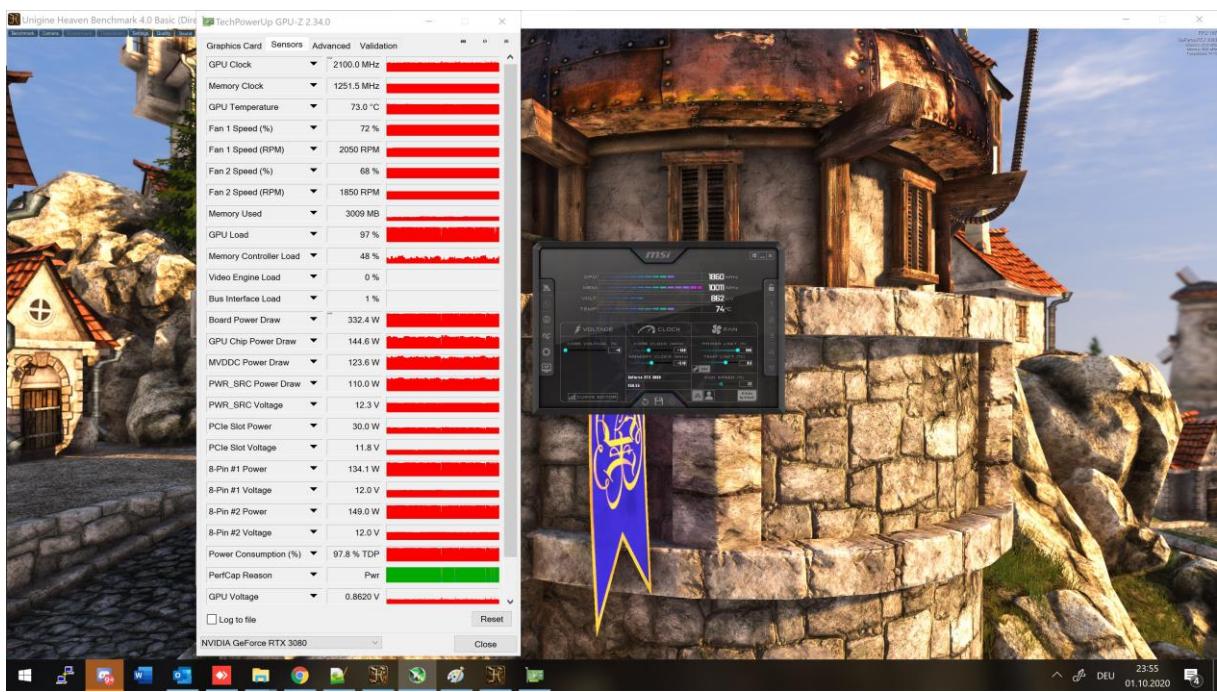


Abbildung 3: Übertakten der Grafikkarte

Über das Wochenende habe ich die neue Grafikkarte schliesslich in meinem PC eingebaut. Da in meinem PC schon die AMD Radeon 5700 XT und die Nvidia RTX 2070 super befindet musste ich die RTX 2070 super ausserhalb des PC-Gehäuses platzieren und über ein 30 cm langes PCIe extension Kabel anschliessen. Auch musste ich die RTX 2070 super an einen PCIe 8x anstatt an einen PCIe 16x Slot anschliessen, da mein Mainboard nur zwei 16x PCIe Slots besitzt. Glücklicherweise hat sich herausgestellt, dass dies die Cuda-Leisung der GPU nicht messbar beeinträchtigt. Der Grund dass ich die RTX 2070 super ausserhalb des Gehäuses plaziert habe ist, weil sie von den drei Grafikkarten die einzige GPU ohne PCIe 4.0 Support ist und bei einem PCIe extension Kabel ein PCIe 4.0 nicht mehr möglich ist.



Abbildung 4: Externe RTX 2070 super



Abbildung 5: PC mit den 3 Grafikkarten

Als ich versucht habe mein Programm auf meiner neuen Grafikkarte auszuführen blieb der Algorithmus einfach stecken – es stellte sich jedoch später heraus, dass durch etwas Warten der Algorithmus dennoch korrekt ausgeführt würde. Nach einigen Researchen habe ich herausgefunden, was der Grund für mein Problem war. Mein Programm ist nicht mit CUDA Toolkit 11.1 kompiliert wurden und somit offiziell nicht mit Ampere kompatibel. Nachdem ich es auf das neuste Cuda toolkit portiert habe, ging alles bestens. Auch ist CUDA Toolkit 11.1 rückwärtskompatibel zu der RTX 2070 super und anderen älteren Grafikkarten. Zudem hat das Update im Allgemeinen die Performance von etwa 64 auf 59 ms auf der RTX 2070 super verbessert. Dies durch bessere Kompileroptimierungen und verbesserter cuFFT Bibliothek.

Interessanterweise hat die CUDA Toolkit 10.2 Version auf meiner RTX 3080 genau die gleiche Performance wie die RTX 2070 Super. Mit dem neuen CUDA Toolkit 11.1 bringe ich auf der neuen Grafikkarte den Wert jedoch von 60 ms auf unter 38 ms!- Also fast 60 % schneller!!!

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/-/commit/f46064d78d892e7dca9acde885dfe831e6c58689>

8. Testing

Funktionstests durch Pipelinetests.

Unitests sind für Funktionen welche auf der Grafikkarte laufen ungeeignet. Dies ist so, weil bei gewissen Funktionen über 1 GB an Eingabedaten vorausgesetzt werden müssen. Das Resultat ist auch schwer überprüfbar, da auch die Ausgabe über 1GB an Daten erzeugt. Bei Unitest würde zudem das ineinanderschachteln und das Wiederverwerten von Speicher nicht getestet werden.

Mein Testkonzept ist es, die Testdaten, die schon für den Korrektheitstest verwendet werden zu verwenden und vor der Ausführung jedes Kernels alle Eingabewerte zu verifizieren.

Da ich immer noch das Problem mit den riesigen Eingabedaten habe, habe ich mich entschieden nur den Hash SHA3-256 vom erwarteten Input mit dem Hash des tatsächlichen Inputs zu vergleichen.

Leider stellte sich heraus, dass bei CUDA je nach Kompileroptimierung die Ausgabedaten eines Algorithmus, welcher mit Fliesskommazahlen rechnet, sich geringfügig unterscheiden. Somit gibt es unterschiedliche Werte je nachdem ob mein Programm im Debug- oder im Release-Mode ausgeführt wird. Der Grund sind Optimierungen wie vertauschen von Instruktionen und fmad. Auch gibt es GPU abhängige Unterschiede.

Nach langem Überlegen ist mir eine Prüfsumme eingefallen, die ich einmal im Fach Mikrokontroller kennengelernt haben: Fletcher's Checksum. Durch einen an diesen Algorithmus angelehnten, modulolosen und mit Fliesskomazahlen arbeitenden Algorithmus gelingt es mir auf eine ungefähre Genauigkeit hin zu prüfen. Fletcher's Checksum stellt übrigens auch die Reihenfolge der Elemente sicher.

Als Beispiel nachfolgend die eigentlich fast identischen aber nicht gleichen Werte bei einer Ausführung auf Debug- und auf Release-Mode:

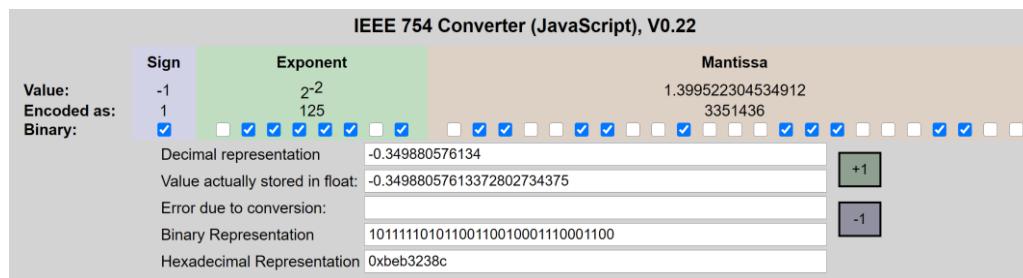


Abbildung 6: Wert im Debug-Mode

Abbildung 7: Wert im Release-Mode

OFFSET	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00000000	00	00	00	00	00	00	00	00	7F	90	09	C0	E7	8D	64	C1
00000010	C8	10	4E	3F	11	3E	D3	BF	8C	23	B3	BE	B2	9D	7A	C0
00000020	9B	31	60	40	1A	73	51	C0	28	5A	32	3F	9E	A7	10	3F
00000030	48	51	65	BE	66	4D	3E	C0	72	E2	E9	BF	DC	FB	03	C0
00000040	00	A0	17	B9	FE	F2	20	3B	3E	73	9B	BF	31	08	2E	BF
00000050	C0	AF	DB	3F	20	A3	E3	BD	AF	0A	97	BF	BC	EB	1C	BF
00000060	AC	34	E3	3E	9F	CD	91	3F	10	E8	DB	BD	FA	D5	DE	BE
00000070	EC	FA	BF	BF	50	E5	6B	3F	20	7C	78	BF	AD	A9	35	BE
00000080	4F	3D	6B	3B	86	A9	45	3B	9E	7B	8A	BF	30	8C	9D	3F
00000090	D1	4E	C3	3E	B2	7F	1B	BF	D9	90	06	BF	FC	D3	4B	BE
000000A0	D6	92	05	C0	42	FB	D6	3E	B0	6A	A9	BE	EB	F2	D2	3D
000000B0	F6	D6	9F	BE	C8	D4	07	3D	7E	7C	D3	3E	51	2A	83	BF
000000C0	BB	D1	62	39	59	C5	8A	BA	77	39	56	3F	BF	7A	0C	BE
000000D0	4A	CE	63	3E	00	F6	ED	3E	5C	51	59	3E	65	2C	C0	3E
000000E0	8E	93	92	BF	8B	35	4B	BF	10	4F	30	BE	46	31	87	BD
000000F0	BB	EB	9C	BE	94	B9	38	BE	BB	A0	98	3F	A8	7F	23	BE
00000100	F0	50	E3	3B	61	3D	98	3A	C2	22	A0	BE	90	4C	0A	3F
00000110	C6	15	DD	3E	44	CC	2A	3F	33	B6	72	3E	EE	8D	16	3D
00000120	59	1E	EA	3E	9E	2B	DF	BD	D5	70	BE	BD	20	A6	D1	3D
00000130	25	DF	AA	3E	8A	B7	73	BE	B6	B6	8D	3E	F4	6D	6A	BD
00000140	31	BC	C3	3A	4E	34	DF	B9	B6	11	8C	BF	D8	79	04	BE
00000150	11	57	39	BE	0D	F4	28	BF	EB	5B	9A	BD	EF	F5	E0	3E
00000160	CF	D6	09	3C	55	0A	A5	BE	A7	0F	70	3E	84	E4	87	BD
00000170	89	C1	80	3E	56	CB	FA	3D	21	82	CC	3E	0F	17	56	BE

OFFSET	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00000000	00	00	00	00	00	00	00	00	7F	90	09	C0	E7	8D	64	C1
00000010	C8	10	4E	3F	11	3E	D3	BF	8B	23	B3	BE	B2	9D	7A	C0
00000020	9B	31	60	40	19	73	51	C0	29	5A	32	3F	9E	A7	10	3F
00000030	4B	51	65	BE	65	4D	3E	C0	72	E2	E9	BF	DD	FB	03	C0
00000040	03	A0	17	B9	FE	F2	20	3B	3E	73	9B	BF	30	08	2E	BF
00000050	C0	AF	DB	3F	21	A3	E3	BD	AE	0A	97	BF	BC	EB	1C	BF
00000060	AC	34	E3	3E	9F	CD	91	3F	10	E8	DB	BD	FA	D5	DE	BE
00000070	EC	FA	B0	BF	50	E5	6B	3F	20	7C	78	BF	AD	A9	35	BE
00000080	50	3D	6B	3B	86	A9	45	3B	9E	7B	8A	BF	30	8C	9D	3F
00000090	D1	4E	C3	3E	B2	7F	1B	BF	D8	90	06	BF	FC	D3	4B	BE
000000A0	D6	92	05	C0	42	FB	D6	3E	B0	6A	A9	BE	EC	F2	D2	3D
000000B0	F7	D6	9F	BE	C8	D4	07	3D	7E	7C	D3	3E	51	2A	83	BF
000000C0	BB	D1	62	39	59	C5	8A	BA	77	39	56	3F	BF	7A	0C	BE
000000D0	4A	CE	63	3E	FF	F5	ED	3E	5C	51	59	3E	65	2C	C0	3E
000000E0	8E	93	92	BF	8B	35	4B	BF	11	4F	30	BE	47	31	87	BD
000000F0	BB	EB	9C	BE	95	B9	38	BE	BB	A0	98	3F	A6	7F	23	BE
00000100	F0	50	E3	3B	61	3D	98	3A	C3	22	A0	BE	90	4C	0A	3F
00000110	C6	15	DD	3E	44	CC	2A	3F	33	B6	72	3E	ED	8D	16	3D
00000120	59	1E	EA	3E	9E	2B	DF	BD	D5	70	BE	BD	21	A6	D1	3D
00000130	25	DF	AA	3E	8A	B7	73	BE	B6	B6	8D	3E	F5	6D	6A	BD
00000140	31	BC	C3	3A	4E	34	DF	B9	B6	11	8C	BF	D8	79	04	BE
00000150	11	57	39	BE	0D	F4	28	BF	EB	5B	9A	BD	EF	F5	E0	3E
00000160	CF	D6	09	3C	54	0A	A5	BE	A6	0F	70	3E	84	E4	87	BD
00000170	89	C1	80	3E	55	CB	FA	3D	21	82	CC	3E	0F	17	56	BE

Abbildung 8: Vergleich Debug-memdump und Release-memdump

<https://stackoverflow.com/questions/18739191/cuda-precision-difference-in-release-version-between-with-and-without-debug-in>

https://de.wikipedia.org/wiki/Fletcher%20%99s_Checksum

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/-/commit/cad6e7a2fe812934924980545a2403daa34625f4>

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/-/commit/f18c1f392e232c5fd2025fe4b7e646fa4d04590e>

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/-/commit/200dbc3d27f450b79218ae884502d5de37ee5e70>

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/-/commit/200dbc3d27f450b79218ae884502d5de37ee5e70>

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/-/commit/3f855355b948d9c4b8de4213ce390cdb8c5ada22>

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/-/commit/9a7a1fc880187322bb540c9936926e6aa95a531c>

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/-/commit/c977b5d41ca2697e3f71bd5c400dc9553b6fb7b>

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/-/commit/0665380b2f482e77ce04bf75181038577ccab066>

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/-/commit/01bb48cca835e80c5c9824e4a050eaa58508a1f1>

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/-/commit/161d01f094a109f3c33ceb042f51c7b169d008c2>

9. Privacy Amplification auf dem Nintendo Switch

Ein Ziel dieser Arbeit ist es den Privacy Amplification – Algorithmus auf einem langsamen und billigen Grafikprozessor zum Laufen zu bringen.

Ich habe dies nun mit einem Nintendo Switch ausgetestet. Der Nintendo Switch beinhaltet ein Tegra X1 SoC.

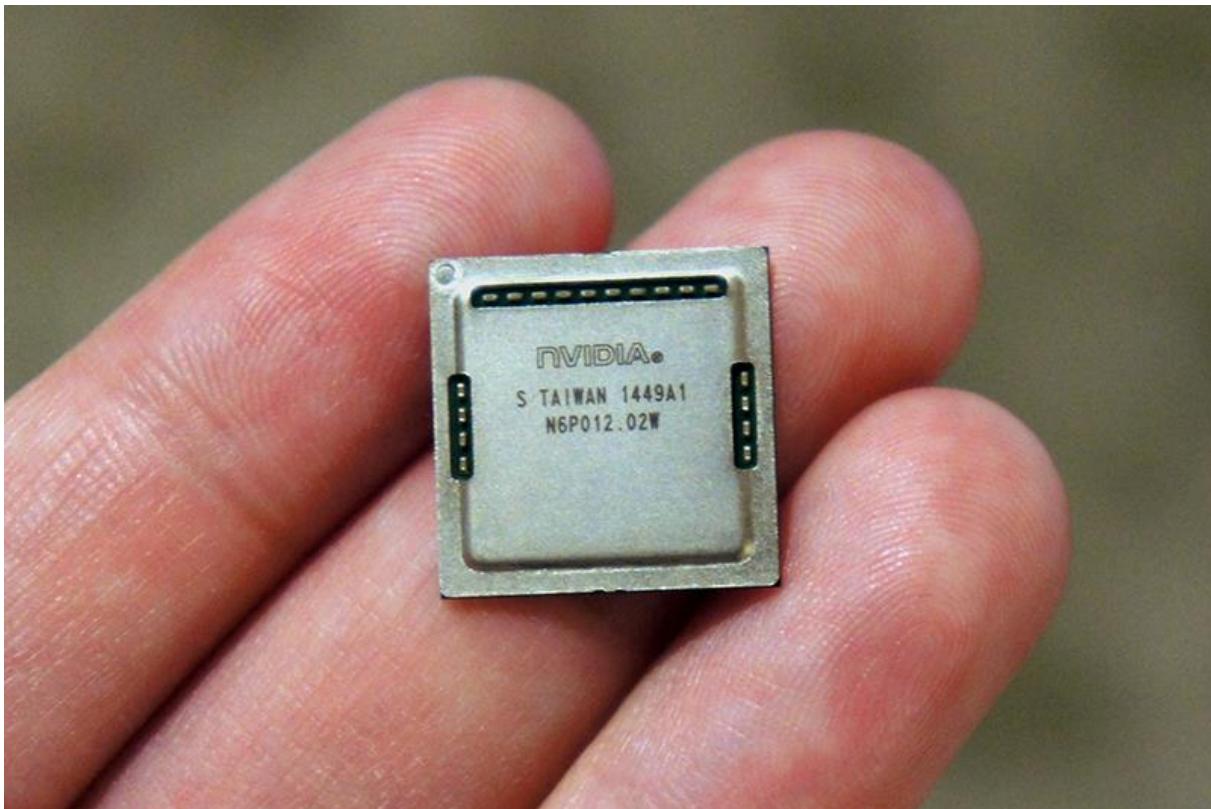


Abbildung 9: Tegra X1 SoC

Bildquelle: <https://www.hardwarezone.com.sg/feature-preview-nvidia-tegra-x1-benchmark-results>

Zuerst musste ich auf dem Nintendo Switch Linux installieren. Dies ist nur inoffiziell möglich und geht mit Hilfe eines Bufferüberlaufs im Recoverymodus. Der verwendete Linuxkern und das auf Ubuntu 18.04 (Bionic Beaver) basierende Betriebssystem wurde so modifiziert, dass es mit dem Nintendo Switch kompatibel ist. Es wird ein CUDA, welches von NVIDIA für den Tegra SoC entwickelt wurde und auf CUDA Toolkit 10.0 basiert, verwendet.

Mein Algorithmus funktioniert bestens und er ist am schnellsten mit 2^{21} Blockgrößen.

Die Geschwindigkeit auf der neuen PC-Grafikkarte war erstaunlicherweise nur ca. 20-mal schneller als auf dem Nintendo Switch!

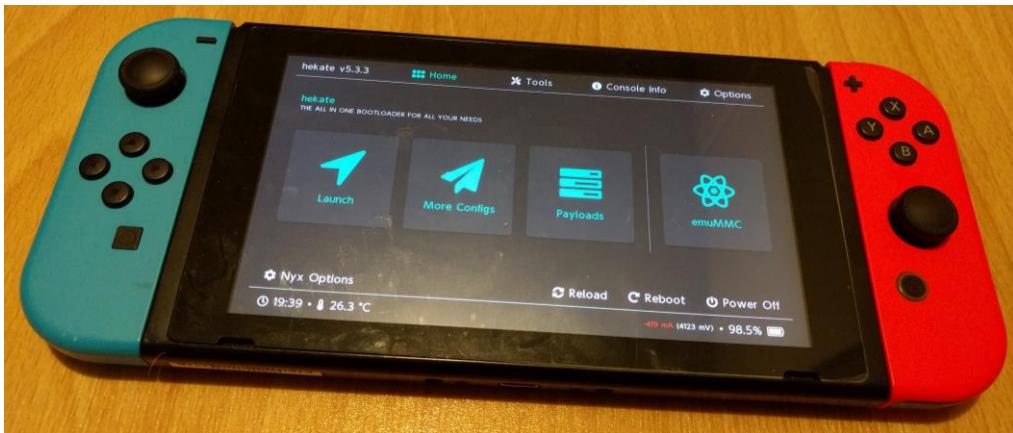


Abbildung 10: Durch bootloaderexploit injektiertes Bootmenü



Abbildung 11: Laufender Privacy Amplification-Algorithmus

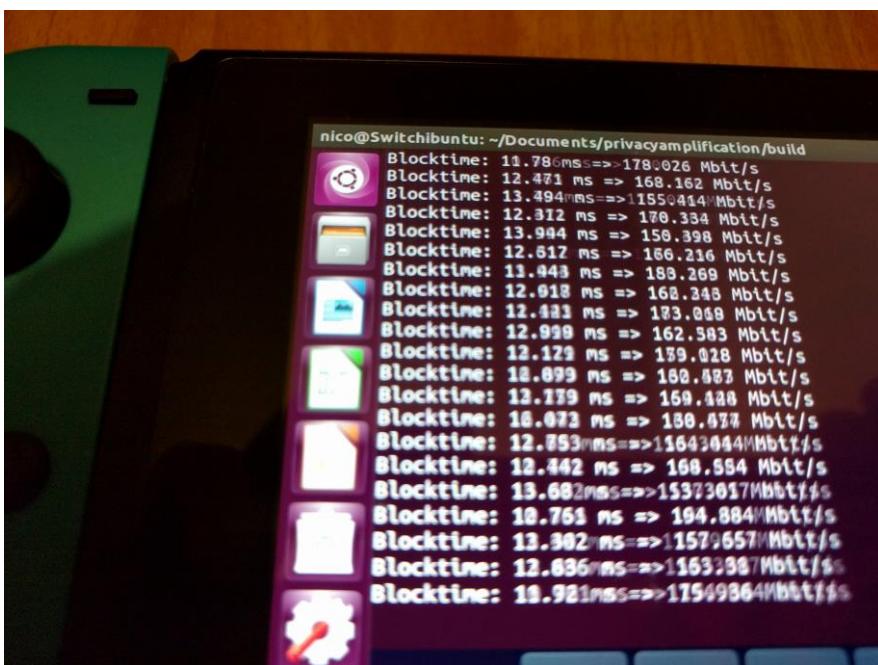


Abbildung 12: Ausschnitt aus laufendem Programm

10. Unit Tests

Wir haben uns entschieden, dass Integration- und Pipelinetests, also das Testen von Zwischenresultaten, nicht ausreichend für dieses Programm sind, weil die Testunabhängigkeit zwischen einzelnen Methoden nicht gegeben ist. Pipelinetests reichen um einen Fehler im Code zu lokalisieren, jedoch kann eine einzelne Funktion unabhängig getestet werden. Das Implementieren von Unit-Tests sollte Contributors einen grösseren Anreiz geben sich an diesem Projekt zu beteiligen. Dies hilft das Projekt populär zu machen. Da für die Privacy Amplification sehr grosse Inputdatenmengen benötigt werden, wurde die Blockgrösse auf 2^{10} beschränkt und die Daten automatisch erzeugt. Auch die erwarteten Werte werden ebenfalls automatisch (auf dem CPU) berechnet. Dies wird mit einem alternativen Algorithmus auf dem CPU gemacht. Diesen Algorithmus musste ich komplett neu schreiben, da der Code nicht auf der GPU ausgeführt wird. Nicht die Performance, sondern die Korrektheit waren dabei massgebend.

11. CalculateCorrectionFloat Unit Test

Berechnet um wieviel die IFFT-Ausgabe verschoben werden muss.

Weil wir zufällige 0 und 1 als Eingabe des Durchschnitts verwenden (ohne Reduktion) wird der Durchschnitt immer bei 0,5 liegen. Dies summiert sich, da wir bis zu 2^{27} als Eingabe haben. Das würde dazu führen, dass das erste Element der FFT (0 Hz) ungefähr 2^{26} ist. Nach der elementweisen Multiplikation wäre das 2^{56} , was zu Ungenauigkeiten in der Gleitkommapräzision führt. Aus diesem Grund setzen wir das erste Element nach der FFT einfach auf Null. Die Korrektur erfolgt mit korrigierter_float_dev nach der normalisierten IFFT.

Vereinfachte Berechnung: $\text{arg3} = ((\text{arg1} * \text{arg2}) / \text{sample_size}) \% 2$

Getestet werden müssen folgende Parameter:

Eingabeparameter: Hamming-Gewicht (Menge von 1 Bit) des Toeplitz-Matrix-Seeds

Eingabeparameter: Hamming-Gewicht (Anzahl von 1 Bit) des Schlüssels

Ausgabeparameter: Die Kernel-Ausgabe wird in `inrection_float_dev` gespeichert

Beschreibung der Umsetzung:

Für den Testfall werden alle für die Funktion notwendigen Variablen und CUDA-Streams lokal erstellt. Der vom CUDA-Kernel benötigte Grafikspeicher wird als pinned memory über cudaMallocHost so alloziert, dass sowohl CPU und GPU ohne cudaMemcpy auf diese Speicherregion zugreifen können. Die im konstanten Speicher gesetzten Variablen werden durch Testwerte überschrieben und nach dem Test wiederhergestellt. Alle mit einer 2^{10} Blockgrösse möglichen 2^{20} Kombinationen werden sowohl auf der CPU und der GPU ausgeführt und dann verglichen. Für den Vergleich wird eine Toleranz von 0,0001 verwendet um der Ungenauigkeit von Fliesskommazahlen entgegen zu wirken. Schlägt ein Testcase fehl wird eine Meldung in stderr ausgegeben, aber die anderen Tests laufen weiter. Schlägt ein Test fehl wird am Ende des Test 100 zurückgegeben, was dem Exit-Code eines falschen Testcases entspricht.

```
bool unitTestCalculateCorrectionFloat() {
    println("Started CalculateCorrectionFloat Unit Test...");
    bool unitTestsFailedLocal = false;
    cudaStream_t CalculateCorrectionFloatTestStream;
    cudaStreamCreate(&CalculateCorrectionFloatTestStream);
    uint32_t sample_size_test = pow(2, 10);
    uint32_t* count_one_of_global_seed_test;
    uint32_t* count_one_of_global_key_test;
    float* correction_float_dev_test;
    cudaMallocHost((void**)&count_one_of_global_seed_test, sizeof(uint32_t));
    cudaMallocHost((void**)&count_one_of_global_key_test, sizeof(uint32_t));
    cudaMallocHost((void**)&correction_float_dev_test, sizeof(float));
    cudaMemcpyToSymbol(sample_size_dev, &sample_size_test, sizeof(uint32_t));
    for (uint32_t i = 0; i < sample_size_test; ++i) {
        for (uint32_t j = 0; j < sample_size_test; ++j) {
            *count_one_of_global_seed_test = i;
            *count_one_of_global_key_test = j;
            calculateCorrectionFloat KERNEL_ARG4(1, 1, 0, CalculateCorrectionFloatTestStream)
                (count_one_of_global_seed_test, count_one_of_global_key_test, correction_float_dev_test);
            cudaStreamSynchronize(CalculateCorrectionFloatTestStream);
            uint64_t cpu_count_multiplied = *count_one_of_global_seed_test * *count_one_of_global_key_test;
            double cpu_count_multiplied_normalized = cpu_count_multiplied / (double)sample_size_test;
            double count_multiplied_normalized_modulo = fmod(cpu_count_multiplied_normalized, 2.0);
            assertThreshold(*correction_float_dev_test - count_multiplied_normalized_modulo, 0.0001, i * sample_size_test + j);
        }
    }
    cudaMemcpyToSymbol(sample_size_dev, &sample_size, sizeof(uint32_t));
    println("Completed CalculateCorrectionFloat Unit Test");
    return unitTestsFailedLocal ? 100 : 0;
}
```

Abbildung 13: Code des Unit-Tests

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/-/commit/013397b7c9fa6dbec7f7983e3f339101f3449083>

12. Ein Fehler wurde gefunden

Während dem unter Kapitel 11 beschriebenen Prüfverfahren wurde ein Fehler in der aktuellen Implementierung festgestellt. Anstelle des Befehls fmod wurde irrtümlich modf verwendet. Dies geschah, da das Verwenden von modf in der Python-Referenzimplementierung korrekt war. Der Befehl führte genau das aus, was ich ausführen wollte. Beim Übersetzen des Python-Codes in CUDA wurde dieser modf-Pythonbefehl in ein modf-Cuda-Befehl umgeschrieben. Doch hier bedeutet dieser Befehl leider etwas ganz anderes. Anstelle des Fliesskomma-Modulos wird von der Fliesskommazahl der Ganzzahlteil gelöscht. Syntaktisch sind die Befehle unterschiedlich aber wenn ein Output-Parameter als Input verwendet wird stimmt der Syntax zufälligerweise überein. Da modf eigentlich einem Modulo 1 entspricht stimmen viele Ergebnisse überein. Wenn es eine Abweichung gab, dann wurden alle Bits geflippt, was in der ursprünglichen Testmethode nicht gut erkannt werden konnte.

```
206 | - Real count_multiplied_normalized_modulo = (float)modf(count_multiplied_normalized, &two);  
206 | + Real count_multiplied_normalized_modulo = (float)fmod(count_multiplied_normalized, two);
```

Abbildung 14: So wurde der Fehler behoben

Die Links zur CUDA-Dokumentation:

<https://developer.download.nvidia.com/cg/modf.html>

<https://developer.download.nvidia.com/cg/fmod.html>

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/-/commit/0a171c267c3af6c9ced88ea22b693af0f8869f56>

13. SetFirstElementToZero Unit Test

Die Funktion setFirstElementToZero setzt das erste Element auf float2(0.0f, 0.0f), wodurch sowohl der Real- als auch der Imaginärteil auf Null gesetzt werden.

Dies geschieht aus Gründen der Präzision. Weitere Informationen finden Sie unter dem Kapitel «CalculateCorrectionFloat Unit».

Die Funktion erwartet die FFT-Ausgabe des Keys und Seeds und setzt deren erstes Element auf Null.

Testbeschrieb:

Es werden 2^{10} grosse Listen aus komplexen Fliesskommazahlen mit einer arithmetischen Folge gefüllt. Die Grafikkarte wird dann aufgefordert das erste Element auf Null zu setzen. Es

wird geprüft ob sowohl der Real-, wie auch der Komplexeil des ersten Elements auf Null gesetzt wurde und ob der Rest der Liste unverändert geblieben ist.

```
bool unitTestSetFirstElementToZero() {
    cout << "Started SetFirstElementToZero Unit Test...";
    bool unitTestsFailedLocal = false;
    cudaStream_t SetFirstElementToZeroStreamTest;
    cudaStreamCreate(&SetFirstElementToZeroStreamTest);
    float* do1_test;
    float* do2_test;
    cudaMallocHost((void**)&do1_test, pow(2, 10) * 2 * sizeof(float));
    cudaMallocHost((void**)&do2_test, pow(2, 10) * 2 * sizeof(float));
    for (int i = 0; i < pow(2, 10) * 2; ++i) {
        do1_test[i] = i + 0.77;
        do2_test[i] = i + 0.88;
    }
    setFirstElementToZero KERNEL_ARG4(1, 2, 0, SetFirstElementToZeroStreamTest)
        (reinterpret_cast<Complex*>(do1_test), reinterpret_cast<Complex*>(do2_test));
    cudaStreamSynchronize(SetFirstElementToZeroStreamTest);
    assertThreshold(do1_test[0], 0.00001, 0);
    assertThreshold(do1_test[1], 0.00001, 1);
    assertThreshold(do2_test[0], 0.00001, 2);
    assertThreshold(do2_test[1], 0.00001, 3);
    for (int i = 2; i < pow(2, 10) * 2; ++i) {
        assertThreshold(do1_test[i] - (i + 0.77), 0.0001, i * 2);
        assertThreshold(do2_test[i] - (i + 0.88), 0.0001, i * 2 + 1);
    }
    cout << "Completed SetFirstElementToZero Unit Test";
    return unitTestsFailedLocal ? 100 : 0;
}
```

Abbildung 15: Bild des Testcodes

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/-/commit/e65ec50be6d65e4238d759abc25121aaa67273d2>

14. ElementWiseProduct Unit Test

Diese Funktion berechnet die komplexe, elemetweise Multiplikation zweier Eingabelisten mit komplexen Zahlen und überschreibt die erste Liste mit dem Resultat.

Testbeschrieb:

Es werden 2^{10} grosse Listen aus komplexen Fliesskommazahlen mit einer arithmetischen Folge gefüllt. Die Grafikkarte wird dann zur Ausführung der komplexen elemetweisen Multiplikation aufgerufen und die Ergebnisse werden in der ersten Liste gespeichert. Der CPU berechnet dasselbe direkt mit der arithmetischen Folge und vergleicht die erhaltenen Resultate mit den Werten der GPU-Berechnung.

```

bool unitTestElementWiseProduct() {
    println("Started ElementWiseProduct Unit Test...");
    bool unitTestsFailedLocal = false;
    cudaStream_t ElementWiseProductStreamTest;
    cudaStreamCreate(&ElementWiseProductStreamTest);
    uint32_t r = pow(2, 5);
    float* do1_test;
    float* do2_test;
    cudaMemcpyToSymbol(pre_mul_reduction_dev, &r, sizeof(uint32_t));
    cudaMallocHost((void**)&do1_test, pow(2, 10) * 2 * sizeof(float));
    cudaMallocHost((void**)&do2_test, pow(2, 10) * 2 * sizeof(float));
    for (int i = 0; i < pow(2, 10) * 2; ++i) {
        do1_test[i] = i + 0.77;
        do2_test[i] = i + 0.88;
    }
    ElementWiseProduct KERNEL_ARG4((int)((pow(2, 10) + 1023) / 1024),
        min((int)pow(2, 10), 1024), 0, ElementWiseProductStreamTest)
        (reinterpret_cast<Complex*>(do1_test), reinterpret_cast<Complex*>(do2_test));
    cudaStreamSynchronize(ElementWiseProductStreamTest);
    for (int i = 0; i < pow(2, 10) * 2; i+=2) {
        float real = ((i + 0.77) / r) * (((i + 0.88) / r) - (((i + 1) + 0.77) / r)) * (((i + 1) + 0.88) / r);
        float imag = ((i + 0.77) / r) * (((i + 1) + 0.88) / r) + (((i + 1) + 0.77) / r) * (((i + 1) + 0.88) / r);
        assertThreshold(do1_test[i] - real, 0.001, i);
        assertThreshold(do1_test[i + 1] - imag, 0.001, i + 1);
    }
    cudaMemcpyToSymbol(pre_mul_reduction_dev, &pre_mul_reduction, sizeof(uint32_t));
    println("Completed ElementWiseProduct Unit Test");
    return unitTestsFailedLocal ? 100 : 0;
}

```

Abbildung 16: Code des Testcase

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/-/commit/7957e055facc0fed9eeb4678a763cdcb6f26696>

15. ReadMe

Die ReadMe Datei wurde nun ausführlich mit dem Textdokument, welches ich an Genf gesendet habe, erweitert. Zur Konvertierung der pdf-Datei in ein Markdown-Dokument wurde folgender Konverter verwendet:

<https://pdf2md.morethan.io/>

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/-/commit/b6e4d8d3e9dd23a6379c5ec722eb225451d61c0e>

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/-/commit/0dc8fef1f493e8212b24640e9c825b47f7372c23>

16. Azure Pipeline

Vollautomatisierte Tests helfen Entwicklern fehlerfreien Code zu schreiben. Bei jedem Commit beginnt die Azure-Pipline zu laufen. Das Repository wird in einen leeren Ordner geklont. Dort wird es von VisualStudio-Bildtools in eine ausführbare Datei umgewandelt.

Nach der Umwandlung werden alle Unitests ausgeführt und die Testresultate geloggt. Waren alle Tests erfolgreich wird der Build als Pipeline-Artefakte auf Azure DevOps hochgeladen. Dieser kann als Canary release, Snapshot build des master branches, heruntergeladen werden. So können interessierte Nutzer immer die neuste funktionierende Entwicklungsversion, ohne sie selbst builden zu müssen, benutzen.

Die Pipeline wird in YAML geschrieben.

```

1  variables:CRLF
2    - ShortGitHash: ''CRLF
3    CRLF
4    jobs:CRLF
5    CRLF
6    - job: - Build_PrivacyAmplification_On_Self_Hosted_AgentCRLF
7      - timeoutInMinutes: 30CRLF
8    pool:CRLF
9      - name: PrivacyAmplificationPoolCRLF
10     CRLF
11     steps:CRLF
12     CRLF
13     - script: |CRLF
14       echo $(Build.SourceVersion)CRLF
15       set gitHash=$(Build.SourceVersion)CRLF
16       set shortHash=%gitHash:~0,7%CRLF
17       echo %shortHash%CRLF
18       echo ##vso[task.setvariable variable=ShortGitHash;]%shortHash%CRLF
19       displayName: 'Get ShortGitHash Script'CRLF
20     CRLF
21     - task: VSBuild@1CRLF
22       displayName: 'Build PrivacyAmplification'CRLF
23     inputs:CRLF
24       solution: 'PrivacyAmplification/PrivacyAmplification.sln'CRLF
25       platform: 'x64'CRLF
26       configuration: 'Release'CRLF
27     CRLF
28     - task: CmdLine@2CRLF
29     inputs:CRLF
30     - script: |CRLF
31       cd "PrivacyAmplification/bin/Release/"CRLF
32       PrivacyAmplification.exe unitTestCalculateCorrectionFloatCRLF
33       failOnStderr: trueCRLF
34       displayName: 'CalculateCorrectionFloat Unit Test'CRLF
35     CRLF
36     - task: CmdLine@2CRLF
37     inputs:CRLF
38     - script: |CRLF
39       cd "PrivacyAmplification/bin/Release/"CRLF
40       PrivacyAmplification.exe unitTestSetFirstElementToZeroCRLF
41       failOnStderr: trueCRLF
42       displayName: 'SetFirstElementToZero Unit Test'CRLF
43     CRLF
44     - task: CmdLine@2CRLF
45     inputs:CRLF
46     - script: |CRLF
47       cd "PrivacyAmplification/bin/Release/"CRLF
48       PrivacyAmplification.exe unitTestElementWiseProductCRLF
49       failOnStderr: trueCRLF
50       displayName: 'ElementWiseProduct Unit Test'CRLF
51     CRLF
52     - task: PublishPipelineArtifact@1CRLF
53     inputs:CRLF
54       targetPath: 'PrivacyAmplification/bin/Release'CRLF
55       artifact: 'PrivacyAmplification_$(ShortGitHash)'CRLF
56       publishLocation: 'pipeline'CRLF
57   CRLF

```

Abbildung 17: Meine Azure Pipeline (azure-pipelines.yml)

```

1 Starting: PublishPipelineArtifact
2
3 Task : Publish Pipeline Artifacts
4 Description : Publish (upload) a file or directory as a named artifact for the current run
5 Version : 1.2.3
6 Author : Microsoft Corporation
7 Help : https://docs.microsoft.com/azure/devops/pipelines/tasks/utility/publish-pipeline-artifact
8
9 Artifact Name: PrivacyAmplification_artifact
10 Uploading pipeline artifact at C:\Users\Nicolas\Documents\Azure\agent_PrivacyAmplification\_work\1\PrivacyAmplification\bin\Release for build #158
11 ApplicationInsightsTelemetrySender will correlate events with X-TFS-Session a9f8365a-ed06-4d4f-925a-31bfa74a8af7
12 DebugManifestForArtifact will correlate http requests with X-TFS-Session a9f8365a-ed06-4d4f-925a-31bfa74a8af7
13 Using .artifactignore file located at: C:\Users\Nicolas\Documents\Azure\agent_PrivacyAmplification\_work\1\PrivacyAmplification\bin\Release\.artifactignore for globbing
14 Processing .artifactignore file surfaced 6 files. Total files under source directory: 12
15 6 files processed.
16 Processed 6 files from C:\Users\Azure\Documents\Azure\agent_PrivacyAmplification\_work\1\PrivacyAmplification\bin\Release successfully.
17 Uploading 6 files from directory C:\Users\Azure\Documents\Azure\agent_PrivacyAmplification\_work\1\PrivacyAmplification\bin\Release.
18 Uploaded 0 out of 17,806,564 bytes.
19 Uploading 6 files from directory C:\Users\Azure\Documents\Azure\agent_PrivacyAmplification\_work\1\PrivacyAmplification\bin\Release.
20 Uploaded 0 out of 17,806,564 bytes.
21 Uploading 6 files from directory C:\Users\Azure\Documents\Azure\agent_PrivacyAmplification\_work\1\PrivacyAmplification\bin\Release.
22 Uploaded 1,876,924 out of 17,807,480 bytes.
23 Uploaded 1,876,924 out of 17,807,480 bytes.
24 Content upload is done!
25
26 Content upload statistics:
27 Total Content: 35.4 MB
28 Physical Content Uploaded: 0.6 MB
29 Logical Content Uploaded: 1.9 MB
30 Compression Ratio: 5.9 MB
31 Decompression Saved: 33.7 MB
32 Number of Chunks Uploaded: 21
33 Total Number of Chunks: 638
34
35 Associated artifact B with build 158
36 ApplicationInsightsTelemetrySender correlated 2 events with X-TFS-Session a9f8365a-ed06-4d4f-925a-31bfa74a8af7
37 Uploading pipeline artifact finished.
38 Finishing: PublishPipelineArtifact

```

Abbildung 18: Ausgeführt Pipeline

Pipeline	Last run	
nicoboss.PrivacyAmplification	#20201026.6 • Moved .artifactignore into the proper directory ⌚ Individual CI for master	⌚ 3h ago ⌚ 2m 40s

Abbildung 19: PrivacyAmplification Pipeline

Name	Size
PrivacyAmplification_f3a6110	17 MB
PrivacyAmplification.exe	3 MB
PrivacyAmplification.exp	819 B
PrivacyAmplification.lib	2 KB
PrivacyAmplification.pdb	15 MB
config.yaml	6 KB
downloadAssets.cmd	2 KB

Abbildung 20: Pipeline Artefacts

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/-/commit/bd6901b185917331d34d0af9a6c8cc705991160f>
<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/-/commit/c159f4d7e1ea43a59bc60d679b3c56c348a8da8f>

17. GitHub

Um das Repository mit Azure zu verbinden musste ich das ganze Projekt auf Github klonen.

Um mit Git auf beide Remotes zu pushen habe ich bei Origin beide Url's eingefügt. Das Repository ist auf privat gestellt, was dank meinem Studentenstatus gratis möglich ist.

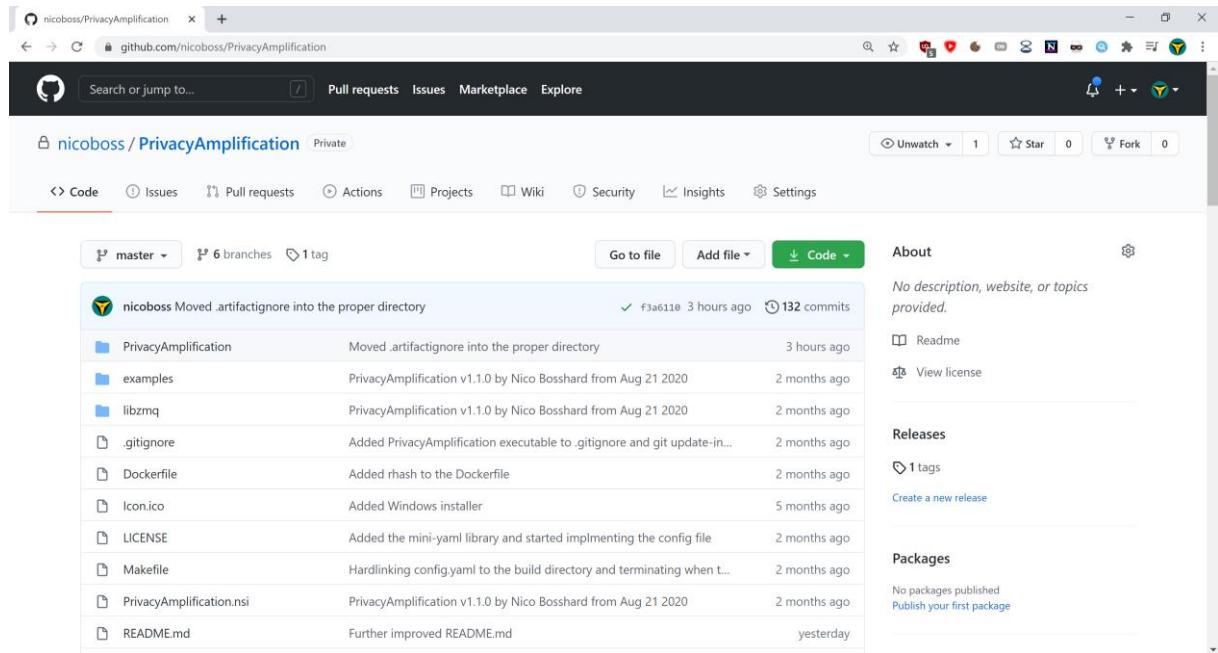


Abbildung 21: Repository auf GitHub pro

Installed GitHub Apps

GitHub Apps augment and extend your workflows on GitHub with commercial, open source, and homegrown tools.

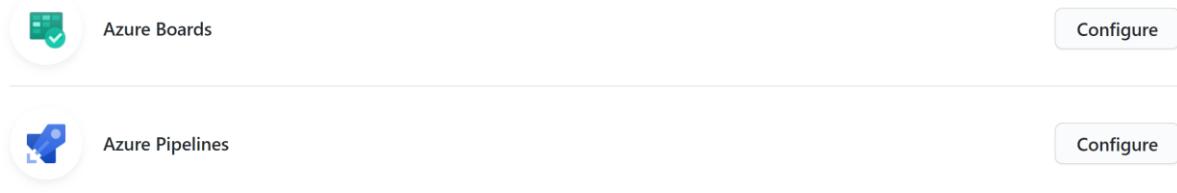


Abbildung 22: GitHub erfolgreich mit Azure verbunden

18. Self-Hosted Azure Agent

Da zum Ausführen der Testcases eine Grafikkarte benötigt wird und die Azure Hosted Agent dies nicht anbieten und Azure Server mit einer Grafikkarte zu mieten viel zu teuer wäre, habe ich mich entschieden selber einen Azure Agent auf meinem Homeserver zu hosten. Dazu musste ich ein Zugangstoken generieren um den als Service gestarteten Azure Agent Client mit dem von Azure gehosteten DevOps-Server zu verbinden. Es dauerte lange, bis ich diese Konstellation zum Laufen brachte.

Personal Access Tokens			
These can be used instead of a password for applications like Git or can be passed in the authorization header to access REST APIs			
Token name	Status	Organization	Expires on
PrivacyAmplification Download Artifacts Packaging (Read)	Active	nicoboss	25.10.2021
PrivacyAmplification Upload Artifacts Packaging (Read & write)	Active	nicoboss	25.10.2021
PrivacyAmplification Agent CASTLEPEAK Agent Pools (Read & manage); Auditing (Read Audit Log)	Active	nicoboss	24.10.2021

Abbildung 23: Meine Tokens

PrivacyAmplificationPool				
Jobs	Agents	Details	Security	
				<button>Update all agents</button> <button>New agent</button> <button>⋮</button>
Name	Last run	Current status	Agent version	Enabled
CASTLEPEAK ● Online	2h ago	Idle	2.175.2	<input checked="" type="checkbox"/> On

Abbildung 24: Mein Server Castlepeak im Pool

19. ShortGitHash

Die Pipelines Artefakte sollten sinnvoll benannt werden. Dazu wird ein auf dem Commit hash basierender gekürzter Hash verwendet. Um diesen zu ermitteln wird untenstehendes batch-script benutzt:

```
- script: |CRLF
  echo $(Build.SourceVersion)CRLF
  set gitHash=$(Build.SourceVersion)CRLF
  set shortHash=%gitHash:~0,7%CRLF
  echo %shortHash%CRLF
  echo ##vso[task.setvariable variable=ShortGitHash;] %shortHash%CRLF
  displayName: 'Get ShortGitHash Script'CRLF
```

Abbildung 25: Script shortGitHash

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/-/commit/905fd53ab6899ca897332202dd11eea8b4414011>

20. Azure Artefacts

Ein signifikanter Größenanteil jedes Pipelines-Artefacts besteht aus Bibliotheken und Testdateien. Mein Build ist komprimiert über 200 MB gross. Jedes Mal über 200 MB hochzuladen ist eine Ressourcenverschwendung. Azure bietet zwar unlimitierten Pipeline-Artefacts-Speicherplatz an, diesen möchte ich aber nicht einfach verschwenden. Darum entschied ich mich die Bibliotheken und Testdaten, welche sich nie verändern, als Azure Artefakte hochzuladen. Im Buildordner befindet sich nun ein Download-script. Dies habe ich selber geschrieben und es ermöglicht dem Endnutzer durch einen Klick alle Azure-Artefakte in den Canary release reinzuladen.

Package	Views	Last pushed	Description	Downloads	Users
cufft64_10.dll Version 1.1.0		7h ago	CUDA Fast Fourier Transform library	↓ 9	jR 1
libzmq-v142-mt-4_3_3.dll Version 1.1.0		7h ago	ZeroMQ core engine in C++	↓ 2	jR 1

Abbildung 26: Azure-Artefakte Bibliotheken

Package	Views	Last pushed	Description	Downloads	Users
keyfile.bin Version 1.0.0		6h ago	keyfile.bin		
toeplitz_seed.bin Version 1.0.0		6h ago	toeplitz_seed.bin		

Abbildung 27: Azure-Artefakte Testdaten

```
1 WHERE azCRLF
2 IF %ERRORLEVEL% NEQ 0 powershell -command "Invoke-WebRequest
-Uri https://aka.ms/installazurecliwindows -OutFile
.\AzureCLI.msi; Start-Process msieexec.exe -Wait -ArgumentList
'/I AzureCLI.msi /quiet'; rm .\AzureCLI.msi"azCRLF
3 echo 73fvfjbljqa7h715asob5rxbggay5lgva6osjinomk7hwlbosifa | az
devops login --organization https://dev.azure.com/nicoboss/azCRLF
4 az artifacts universal download ^azCRLF
5 --organization "https://dev.azure.com/nicoboss/" ^azCRLF
6 --project "d47efac3-15b4-4aae-9e4d-6a4c0a44a420" ^azCRLF
7 --scope project ^azCRLF
8 --feed "Libs" ^azCRLF
9 --name "cufft64_10.dll" ^azCRLF
10 --version "1.1.0" ^azCRLF
11 --path . ^azCRLF
```

Abbildung 28: Ausschnitt des Downloadskripts

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/-/commit/914df2c1b7b0186f85f0ff107f8298ee5f6d098b>
<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/-/commit/5f2b796ec39558fc4370538175e3b3925e8471bb>
<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/-/commit/a8acb37dfddf570f212eafdbcb2ccfecde1263fd>
<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/-/commit/1b99ef52c3742fbff453c95088991c82975ecc8a>
<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/-/commit/cec3f498d4425e35408f874faae4a25e012ccb9>

21. AssertEquals

In diesem neuen, funktionsähnlichen Makro vergleicht man den aktuellen mit dem erwarteten Wert. Falls er ungleich ist, wird ein Fehler ausgegeben. Dieser beinhaltet den erwarteten und den erhaltenen Wert. Zudem werden die Datei und die Codezeilennummer, auf der der Fehler auftrat ausgegeben.

```
#define assertEquals(actual, expected, testCaseNr) \
if (actual != expected) { \
    std::cerr << "assertEqualsError in function " << __func__ << " in " << __FILENAME__ << ":" << __LINE__ << " on test case " << testCaseNr \
        << ": Expected " << expected << " but it was " << actual << endl; \
    unitTestsFailed = true; \
    unitTestsFailedLocal = true; \
    BREAK \
}
```

Abbildung 29: Code vom assertEquals

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/-/commit/820c2a0df2d2cd6ddc33b0f8038a940518cba8d0>
<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/-/commit/736c48141565739f9c035d0f45c117c942c0bc39>
<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/-/commit/b9d81db8f4f1aecfb89b6f87328780bc9141e966>
<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/-/commit/195de2bb9b498f80c3ed7211ba65309a79b3e363>
<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/-/commit/4a6ca14bb43c990afa1bf1ab740a87bcdcc7dc68>
<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/-/commit/f3a61106d4c5a0f5fc61f0860814b74dbf1818b4>
<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/-/commit/d8627a6a9e0543098bdc7c28a05154f094997ea8>

22. BinInt2float Unit Test

Die Funktion BinInt2float konvertiert Binärdaten zu einem Fliesskommaarray. Dabei repräsentiert jedes Bit ein Element in diesem Fliesskommaarray.

Eingabeparameter: Binärdaten

Ausgabeparameter: Fliesskommaarray, Hamming-Gewicht (Anzahl Einer in den Binärdaten)

Beschreibung der Umsetzung:

Für den Testfall werden alle für die Funktion notwendigen Variablen und CUDA-Streams lokal erstellt. Der vom CUDA-Kernel benötigte Grafikspeicher wird als pinned memory über cudaMallocHost so alloziert, dass sowohl CPU und GPU ohne cudaMemcpy auf diese Speicherregion zugreifen können. Die im konstanten Speicher gesetzten Variablen werden durch Testwerte überschrieben und nach dem Test wiederhergestellt.

Da dieser Test sehr viel Prozessorleistung benötigt, wird er auf alle verfügbaren Prozessorkernen aufgeteilt. Dazu wird ein thread pool erzeugt. Für jede 1 Mio. Testfälle wird eine neue Arbeitseinheit erstellt. Die Anzahl Testfälle beträgt $\sum_{i=10}^{27} 2^i \approx 264,5 \text{ Mio.}$. Schlägt ein Testcase fehl, wird eine Meldung in stderr ausgegeben, aber die anderen Tests laufen weiter. Schlägt ein Test fehl, wird am Ende des Test 100 zurückgegeben, was dem Exit-Code eines falschen Testcases entspricht.

Beim Test wird der Eingabearray mit Zahlen je 32-bit von 0 bis 2^{22} aufgefüllt. Für jeden Exponent von 10 bis 27 wird nun folgendes ausgeführt:

Die erwartete Anzahl Einer wird durch den Code von David W. Wilson (oeis.org, A000788) berechnet.

```
//David W. Wilson: https://oeis.org/A000788/a000788.txt
unsigned A000788(unsigned n)
{
    unsigned v = 0;
    for (unsigned bit = 1; bit <= n; bit <= 1)
        v += ((n >> 1) & ~(bit - 1)) + ((n & bit) ? (n & ((bit << 1) - 1)) - (bit - 1) : 0);
    return v;
}
```

Abbildung 30: Code von David W. Wilson

Die Ausgabespeicherregion wird mit 0xFF gefüllt um so nichtbeschriebenen Speicher zu erkennen. Der BinInt2float Algorithmus wird auf der Grafikkarte ausgeführt.

Als Erstes wird geprüft ob das erhaltene Hamming-Gewicht mit dem von der Berechnung mit dem Wilson-Code übereinstimmt. Im Thread pool werden die eigentlichen Fliesskommawerte im Output-array geprüft. Durch die Bedingung « $((i/32) & (1 << (31 - (i \% 32)))) == 0$ » wird geprüft ob an der Stelle i- des Ausgabearrays eine 0.0f oder 1.0f / reduction sein muss.

```

int unitTestBinInt2float() {
    println("Started TestBinInt2float Unit Test...");
    atomic<bool> unitTestsFailedLocal = false;
    cudaStream_t BinInt2floatStreamTest;
    cudaStreamCreate(&BinInt2floatStreamTest);
    uint32_t* binInTest;
    float* floatOutTest;
    cudaMallocHost((void**)&binInTest, (pow(2, 27) / 32) * sizeof(uint32_t));
    cudaMallocHost((void**)&floatOutTest, pow(2, 27) * sizeof(float));
    uint32_t* count_one_test;
    cudaMallocHost(&count_one_test, sizeof(uint32_t));

    const auto processor_count = std::thread::hardware_concurrency();
    for (int i = 0; i < pow(2, 27) / 32; ++i) {
        binInTest[i] = i;
    }
    unitTestBinInt2floatVerifyResultThreadFailed = false;
    for (uint32_t sample_size_test_exponent = 10; sample_size_test_exponent <= 27; ++sample_size_test_exponent)
    {
        int elementsToCheck = pow(2, sample_size_test_exponent);
        println("TestBinInt2float Unit Test with 2^" <> sample_size_test_exponent <> " samples...");
        uint32_t sample_size_test = elementsToCheck;
        uint32_t count_one_expected = A000788((sample_size_test/32)-1);
        *count_one_test = 0;
        memset(floatOutTest, 0xFF, pow(2, 27) * sizeof(float));
        binInt2float KERNEL_ARG4((int)((int)(sample_size_test)+1023) / 1024), min_template(sample_size_test, 1024), 0,
            BinInt2floatStreamTest) (binInTest, floatOutTest, count_one_test);
        cudaStreamSynchronize(BinInt2floatStreamTest);
        assertEquals(*count_one_test, count_one_expected, -1);
        int requiredTotalTasks = elementsToCheck % 1000000 == 0 ? elementsToCheck / 1000000 : (elementsToCheck / 1000000) + 1;
        ThreadPool* unitTestBinInt2floatVerifyResultPool = new ThreadPool(min(max(processor_count, 1), requiredTotalTasks));
        for (int i = 0; i < elementsToCheck; i += 1000000) {
            unitTestBinInt2floatVerifyResultPool->enqueue(unitTestBinInt2floatVerifyResultThread, floatOutTest, i, min(i + 1000000, elementsToCheck));
        }
        unitTestBinInt2floatVerifyResultPool->~ThreadPool();
    }
    if (unitTestBinInt2floatVerifyResultThreadFailed) {
        unitTestsFailedLocal = true;
    }
    println("Completed TestBinInt2float Unit Test");
    return unitTestsFailedLocal ? 100 : 0;
}

```

Abbildung 31: Unitest des BinInt2float

```

void unitTestBinInt2floatVerifyResultThread(float* floatOutTest, int i, int i_max)
{
    bool unitTestsFailedLocal = false;
    register const Real float0 = 0.0f;
    register const Real float1_reduced = 1.0f / reduction;
    for (; i < i_max; ++i) {
        if (((i / 32) & (1 << (31 - (i % 32)))) == 0) {
            assertEquals(floatOutTest[i], float0, i)
        }
        else
        {
            assertEquals(floatOutTest[i], float1_reduced, i)
        }
    }
    if (unitTestsFailedLocal) {
        unitTestBinInt2floatVerifyResultThreadFailed = true;
    }
}

```

Abbildung 32: Validierung der Werte

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/-/commit/0deb0581d2bf86072833ffc28f0251f8c9a1b255>

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/-/commit/215d0bf41ac9c98f173e2640a518ae8b7492ec88>

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/-/commit/a6ac017748d8fa32bd2063f937a0ee10c1d89281>

23. ToBinaryArray Unit Test

Dieser Algorithmus generiert das PrivacyAmplification Resultat vom IFFT-Resultat und vom Key-Rest.

Eingabeparameter:

- Rohe IFFT-Ausgabe
- Key-Rest
- Resultat vom kalkulierten CorrectionFloat

Ausgabeparameter: PrivacyAmplification Resultat im gewünschten Endianness.

Diese Funktion normalisiert die IFFT-Fliesskomma-Ausgabe und konvertiert diese in das Binärsystem. Dabei wird auch mit dem key-Rest Xor gerechnet.

Beschreibung der Umsetzung:

Es werden die gleichen Vorbereitungen und Terminierungsbedingungen wie unter Kapitel 22 (BinInt2float Unit Test) beschrieben gemacht.

Auch hier wird ein Thread -Pool verwendet um die Validierung der Testresultate zu beschleunigen. Der 2^{27} Output der IFFT wird wie folgt gefüllt:

`«invOutTest[i]=(((i/32)&(1<<(31-(i%32)))) == 0)? 0.0f : 1.0f».`

Für jeden Exponent von 10 bis 27 wird nun folgendes ausgeführt:

Die vertikale Länge wird berechnet, der Ausgabespeicherplatz wird mit 0xCC gefüllt um ein nichtbeschriebenen Speicher zu erkennen. ToBinaryArray wird auf der GPU ausgeführt. Im Thread -Pool wird nun folgendes validiert:

Daten und der Key-Rest werden zuerst in die korrekte Endianess umgewandelt.

Das tatsächliche, das erwartete und das Xor-Bit werden an der Stelle i ermittelt und wenn notwendig miteinander verrechnet. Danach wird das finale erwartete mit dem tatsächlichen Bit verglichen.

```

int unitTestToBinaryArray() {
    cout.println("Started ToBinaryArray Unit Test...");
    atomic<bool> unitTestsFailedLocal = false;
    cudaStream_t ToBinaryArrayStreamTest;
    cudaStreamCreate(&ToBinaryArrayStreamTest);
    register const Real float0 = 0.0f;
    register const Real float1 = 1.0f;
    float* inoutTest;
    uint32_t* binOutTest;
    uint32_t* key_rest_test;
    Real* correction_float_dev_test;
    cudaMallocHost((void**)&binOutTest, pow(2, 27) * sizeof(float));
    cudaMallocHost((void**)&binOutTest, (pow(2, 27) / 32) * sizeof(uint32_t));
    cudaMallocHost((void**)&key_rest_test, (pow(2, 27) / 32) * sizeof(uint32_t));
    cudaMallocHost((void**)&correction_float_dev_test, sizeof(Real));
    memset(key_rest_test, 0b10101010, (pow(2, 27) / 32) * sizeof(uint32_t));
    *correction_float_dev_test = 1.9f;
    uint32_t normalisation_float_test = 1.0f;
    cudaMemcpyToSymbol(normalisation_float_dev, &normalisation_float_test, sizeof(uint32_t));
    const auto processor_count = std::thread::hardware_concurrency();
    for (int i = 0; i < pow(2, 27); ++i) {
        inoutTest[i] = ((i / 32) & (1 << (31 - (i % 32)))) == 0 ? float0 : float1;
    }
    unitTestToBinaryArrayVerifyResultThreadFailed = false;
    for (uint32_t sample_size_test_exponent = 10; sample_size_test_exponent <= 27; ++sample_size_test_exponent) {
        uint32_t sample_size_test = pow(2, sample_size_test_exponent);
        uint32_t vertical_len_test = sample_size_test / 4 + sample_size_test / 8;
        uint32_t elementsToCheck = vertical_len_test;
        uint32_t vertical_block_test = vertical_len_test / 32;
        cout.println("ToBinaryArray Unit Test with 2^" << sample_size_test_exponent << " samples...");
        memset(binOutTest, 0xC0, (pow(2, 27) / 32) * sizeof(uint32_t));
        ToBinaryArray KERNEL_ARG4((int)((vertical_block_test) / 31) + 1, 1023, 0, ToBinaryArrayStreamTest) (inoutTest, binOutTest, key_rest_test, correction_float_dev_test);
        cudaStreamSynchronize(ToBinaryArrayStreamTest);
        int requiredTotalTasks = elementsToCheck % 1000000 == 0 ? elementsToCheck / 1000000 : (elementsToCheck / 1000000) + 1;
        ThreadPool* unitTestToBinaryArrayVerifyResultPool = new ThreadPool(min(max(processor_count, 1), requiredTotalTasks));
        for (int i = 0; i < elementsToCheck; i += 1000000) {
            unitTestToBinaryArrayVerifyResultPool->enqueue(unitTestToBinaryArrayVerifyResultThread, binOutTest, key_rest_test, i, min(i + 1000000, elementsToCheck));
        }
        unitTestToBinaryArrayVerifyResultPool->~ThreadPool();
    }
    if (unitTestToBinaryArrayVerifyResultThreadFailed) {
        unitTestsFailedLocal = true;
    }
    cudaMemcpyToSymbol(normalisation_float_dev, &normalisation_float, sizeof(uint32_t));
    cout.println("Completed ToBinaryArray Unit Test");
    return unitTestsFailedLocal ? 100 : 0;
}

```

Abbildung 33: Algorithmus ToBinaryArray Unit Test

```

void unitTestToBinaryArrayVerifyResultThread(uint32_t* binOutTest, uint32_t* key_rest_test, int i, int i_max)
{
    bool unitTestsFailedLocal = false;
    uint32_t mask;
    uint32_t data;
    uint32_t key_rest_little;
    uint32_t key_rest_xor;
    uint32_t actualBit;
    uint32_t expectedBit;
    uint32_t xorBit;
    for (; i < i_max; ++i) {
        mask = 1 << (31 - (i % 32));
        data = binOutTest[i / 32];
        #if AMPOUT_REVERSE_ENDIAN == TRUE
        data = (((data) & 0xff000000) >> 24) |
            (((data) & 0x00ff0000) >> 8) |
            (((data) & 0x0000ff00) << 8) |
            (((data) & 0x000000ff) << 24));
        #endif
        #if XOR_WITH_KEY_REST == TRUE
        #if AMPOUT_REVERSE_ENDIAN == TRUE
        key_rest_little = key_rest_test[i / 32];
        key_rest_xor = (((key_rest_little) & 0xff000000) >> 24) |
            (((key_rest_little) & 0x00ff0000) >> 8) |
            (((key_rest_little) & 0x0000ff00) << 8) |
            (((key_rest_little) & 0x000000ff) << 24));
        #else
        uint32_t key_rest_xor = key_rest_test[i / 32];
        #endif
        #endif
        actualBit = (data & mask) > 0;
        expectedBit = ((i / 32) & mask) > 0;
        xorBit = (key_rest_xor & mask) > 0;
        #if XOR_WITH_KEY_REST
        expectedBit ^= xorBit;
        #endif
        assertEquals(actualBit, expectedBit, i)
    }
    if (unitTestsFailedLocal) {
        unitTestToBinaryArrayVerifyResultThreadFailed = true;
    }
}

```

Abbildung 34: Validierung der Werte

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/-/commit/0045759d33a40da551b236791f589dfd1f3e14b3>

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/-/commit/6746daf77eb35506a82023993e7444792eb5c5ab>

24. Vollautomatischer Performance-Graph

Um die Performance des Programms zu messen habe ich ein neues command line argument namens Speedtest hinzugefügt. Mit diesem Argument wird die Kontrolle über das Programm kurz vor dem Erreichen des mainloops erlangt. Dort werden verschiedene Variablen und Einstellungen mit dem für einen Speedtest notwendigen Werten überschrieben. Danach wird eine Art gosub für Pseudofunktionsausführung innerhalb desselben Stackframes verwendet um eine Iteration des mainloops aufzurufen und danach die Kontrolle wieder zu erlangen. Dort werden einige Parameter verändert und eine weitere Iteration aufgerufen. Auch werden dort dem Python-Skript über stdout Messwerte mitgeteilt. Das Skript zeichnet mittels matplotlib ein Performancediagramm. Diese Bild wird in ein separates GitHubgist hochgeladen und im ReadMe verlinkt. Dadurch bleibt die History der Performance-Statistik erhalten ohne das mainrepository zu füllen. Dieses Skript wird bei jedem Azure Pipeline Durchlauf ausgeführt.

```
if (strcmp(*arg, "speedtest") == 0) {
    speedtest = true;
    doTest = false;
    verify_ampout = false;
    host_ampout_server = false;
    auto start = chrono::high_resolution_clock::now();
    auto stop = chrono::high_resolution_clock::now();

    for (int i = 0; i < 2; ++i) {
        switch (i) {
            case 0: dynamic_toeplitz_matrix_seed = true; break;
            case 1: dynamic_toeplitz_matrix_seed = false; break;
        }
        for (int j = 10; j < 28; ++j) {
            sample_size = pow(2, j);
            vertical_len = sample_size / 4 + sample_size / 8;
            horizontal_len = sample_size / 2 + sample_size / 8;
            vertical_block = vertical_len / 32;
            horizontal_block = horizontal_len / 32;
            desired_block = sample_size / 32;
            key_blocks = desired_block + 1;
            normalisation_float = ((float)sample_size) / ((float)total_reduction) / ((float)total_reduction);
            cudaMemcpyToSymbol(normalisation_float_dev, &normalisation_float, sizeof(float));
            cudaMemcpyToSymbol(sample_size_dev, &sample_size, sizeof(uint32_t));
            dist_freq = sample_size / 2 + 1;
            for (int k = 0; k < 10; ++k) {
                //GOSUB reimplementation - Function call in same stackframe
                goto mainloop;
                return_speedtest;

                stop = chrono::high_resolution_clock::now();
                auto duration = chrono::duration_cast<chrono::microseconds>(stop - start).count();
                start = chrono::high_resolution_clock::now();
                println("d[" << i << "," << j << "," << k << "]=" << (1000000.0 / duration) * (sample_size / 1000000.0));
            }
        }
    }
    exit(0);
}
```

Abbildung 35: Code zur Performance-Messung

Verlinkung im ReadMe:

```

## Performance
![Privacy Ampification - RTX 3080 with dynamic Toeplitz
seed](https://gist.github.com/nicoboss/dfaef7685e20cf3f418559f796
0e33cfe/raw/PrivacyAmpification_RTX_3080_dynamic_seed.svg)
![Privacy Ampification - RTX 3080 with static Toeplitz
seed](https://gist.github.com/nicoboss/dfaef7685e20cf3f418559f796
0e33cfe/raw/PrivacyAmpification_RTX_3080_static_seed.svg)

- task: PythonScript@0
  inputs:
    scriptSource: 'inline'
  script: |
    import os
    os.chdir("PrivacyAmplification/bin/Release/")
    import matplotlib.pyplot as plt
    import numpy as np
    from subprocess import Popen, PIPE
    d = np.zeros(shape=(2,28,10))
    process = Popen(["PrivacyAmpification.exe", "speedtest"], stdout=PIPE, universal_newlines=True)
    (output, err) = process.communicate()
    exit_code = process.wait()
    exec(output)
    fig, ax = plt.subplots()
    ax.plot(range(11, 28), d[0, 11:], 'o')
    ax.set(xlabel='log2(Blocksize)', ylabel='Speed [Mbit/s]', title='Privacy Ampification - RTX 3080 with dynamic Toeplitz seed')
    ax.grid()
    fig.savefig("PrivacyAmpification_RTX_3080_dynamic_seed.svg", format='svg', dpi=1200)
    fig, ax = plt.subplots()
    ax.plot(range(11, 28), d[1, 11:], 'o')
    ax.set(xlabel='log2(Blocksize)', ylabel='Speed [Mbit/s]', title='Privacy Ampification - RTX 3080 with static Toeplitz seed')
    ax.grid()
    fig.savefig("PrivacyAmpification_RTX_3080_static_seed.svg", format='svg', dpi=1200)
  displayName: 'Generate Stats'

```

Abbildung 36: Skript zur Generierung der Statistik

```

- task: CmdLine@2
  inputs:
    script: |
      cd "PrivacyAmplification/bin/Release/"
      gists update dfaef7685e20cf3f418559f7960e33cfe ? PrivacyAmpification_RTX_3080_dynamic_seed.svg
      gists update dfaef7685e20cf3f418559f7960e33cfe ? PrivacyAmpification_RTX_3080_static_seed.svg
  failOnstderr: true
  displayName: 'Upload Stats'

```

Abbildung 37: Skript zum Hochladen der Statistik auf GitHub gist

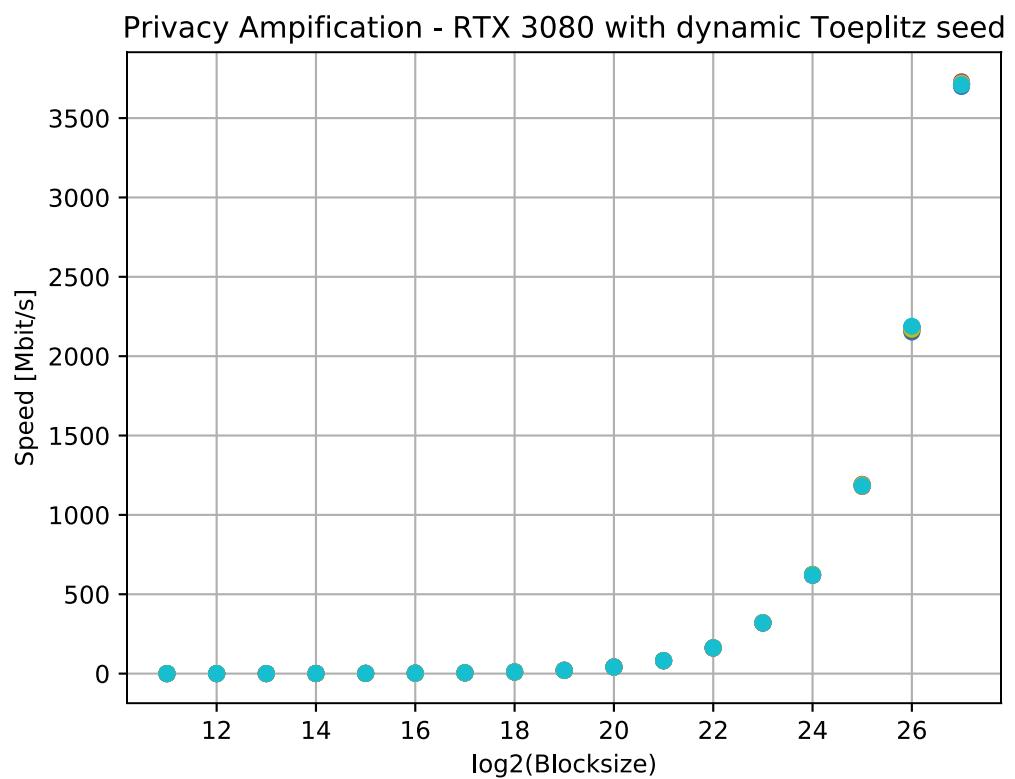


Abbildung 38: PA mit RTX 3080 dynamic seed

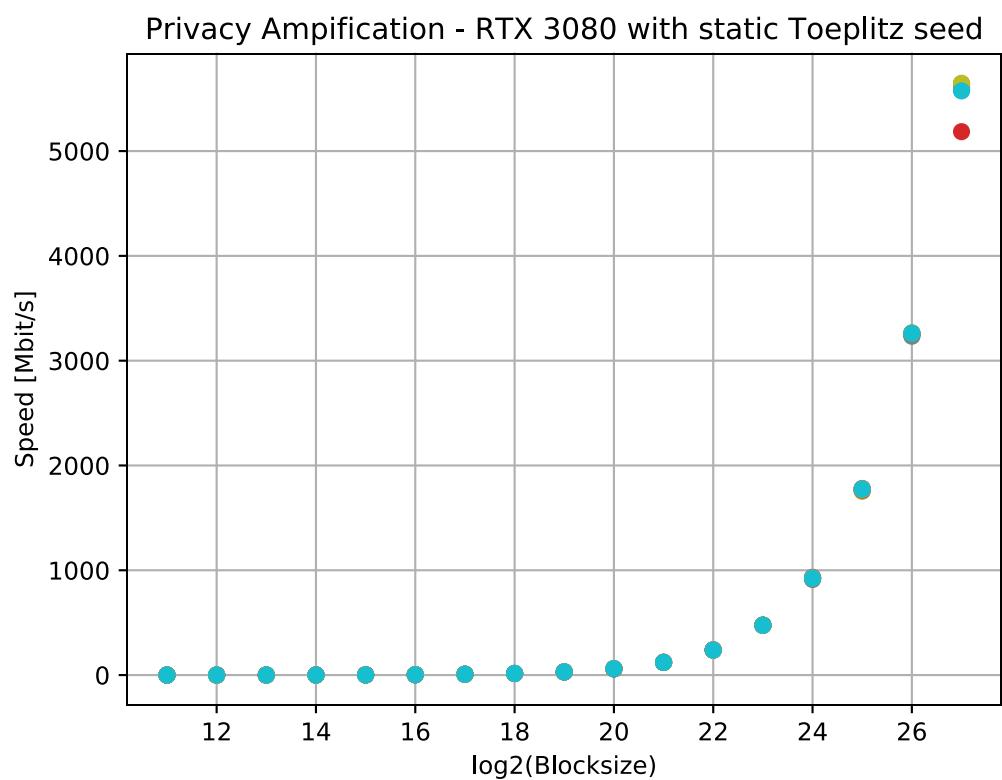
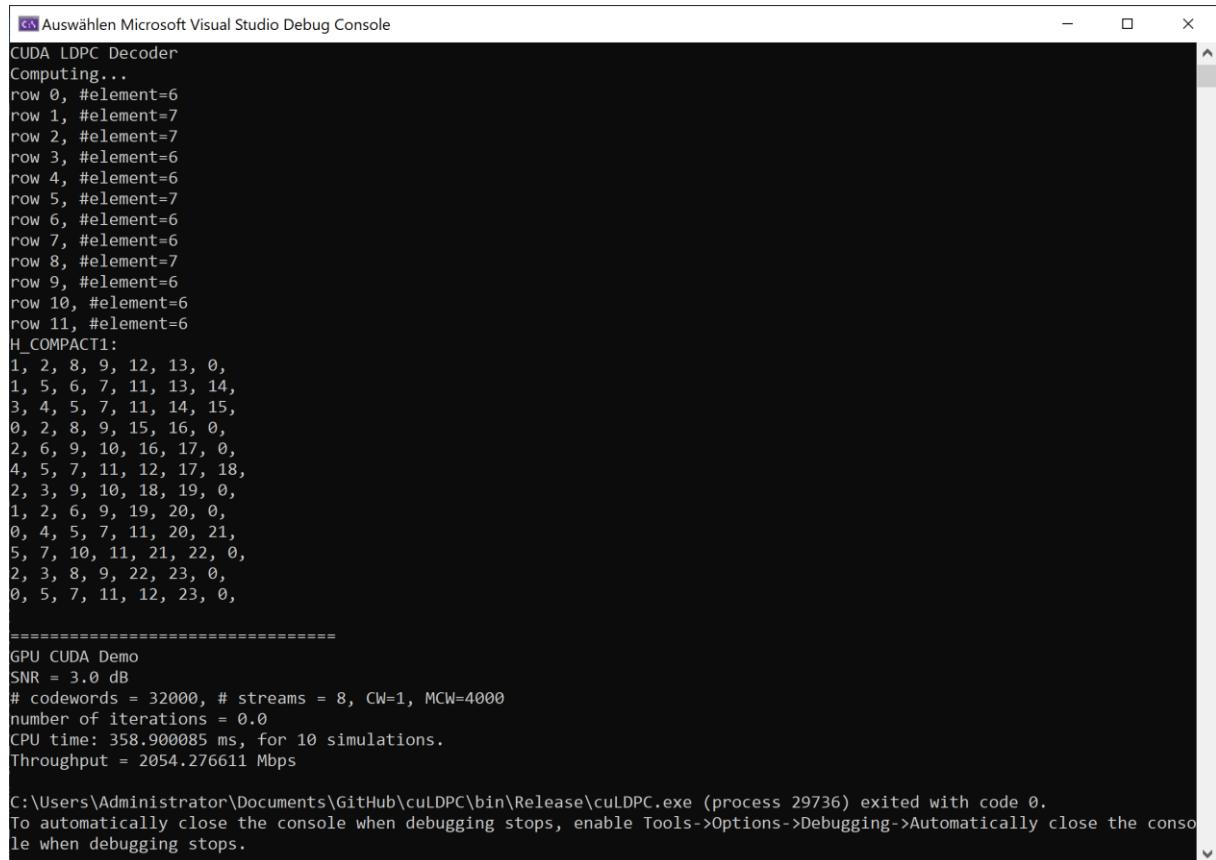


Abbildung 39: PA mit RTX 3080 static seed

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/-/commit/46fb00703bb8a881b11f39feab19d9713eb34f>
<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/-/commit/89a0cb9750ec32648aa4de29e3106316da288c42>

25. cuLDPC Separation

Der cuLDPC-Code wurde vom PrivacyAmplification-Code separiert und ist nun ein eigenständiges Projekt.



The screenshot shows the Microsoft Visual Studio Debug Console window titled "Auswählen Microsoft Visual Studio Debug Console". The console displays the output of a CUDA LDPC Decoder. It starts with "Computing..." followed by a series of row and column indices. A section labeled "H_COMPACT1:" contains a dense matrix of indices. Below this, a separator line "======" is followed by "GPU CUDA Demo" and performance statistics: SNR = 3.0 dB, codewords = 32000, streams = 8, CW=1, MCW=4000, number of iterations = 0.0, CPU time: 358.900085 ms, for 10 simulations, and Throughput = 2054.276611 Mbps. The final line shows the command C:\Users\Administrator\Documents\GitHub\cuLDPC\bin\Release\cuLDPC.exe (process 29736) exited with code 0.

Abbildung 40: Eigenständiges cuLDPC

26. cuFFT neu planen

Durch erneutes und intelligenteres Planen der cuFFT nach dem Ändern der Blockgrösse während der Laufzeit, gelang es mir die Performance der kleineren Blockgrössen zu verbessern. PLAN_FFT kann als Macro nun überall eingesetzt werden, wo eine Neuplanung der FFT sinnvoll ist wie beispielsweise nach dem Ändern der Blockgrösse. Das Zerstören durch cufftDestroy des alten Plans, falls vorhanden, wird auch von diesem Macro erledigt.

Resultate nach Neuplanung:

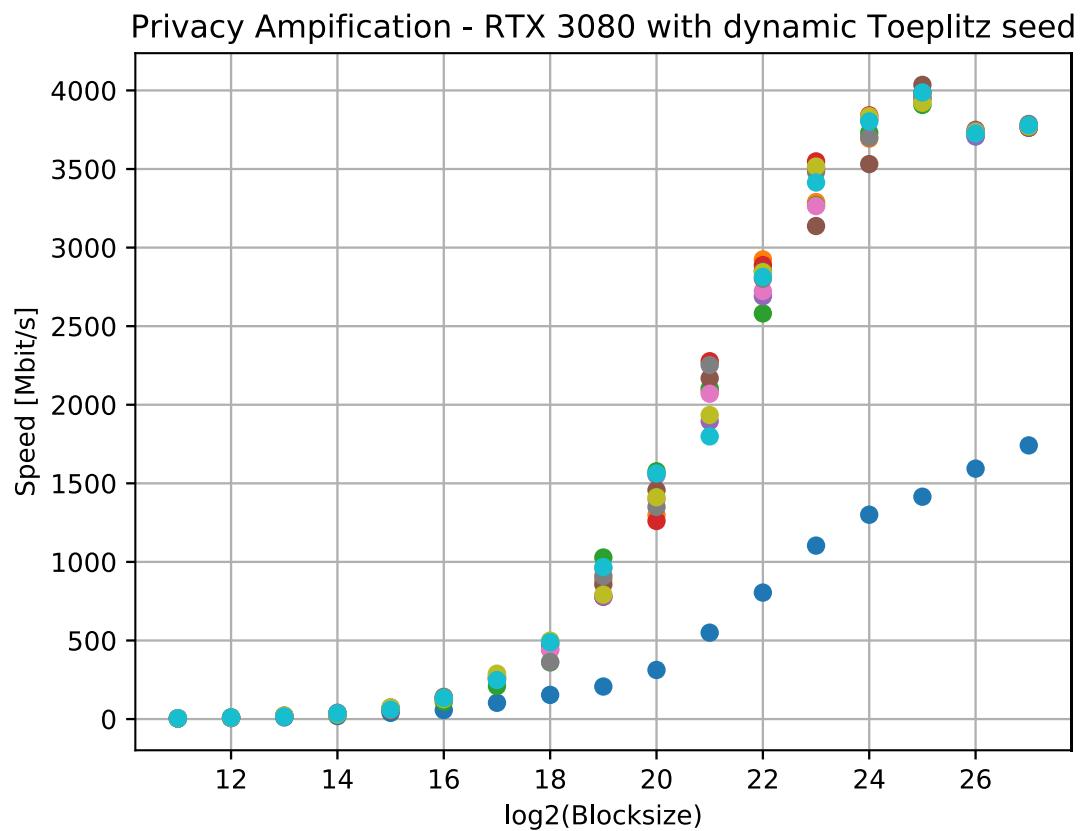


Abbildung 41: Performance mit dynamischem seed

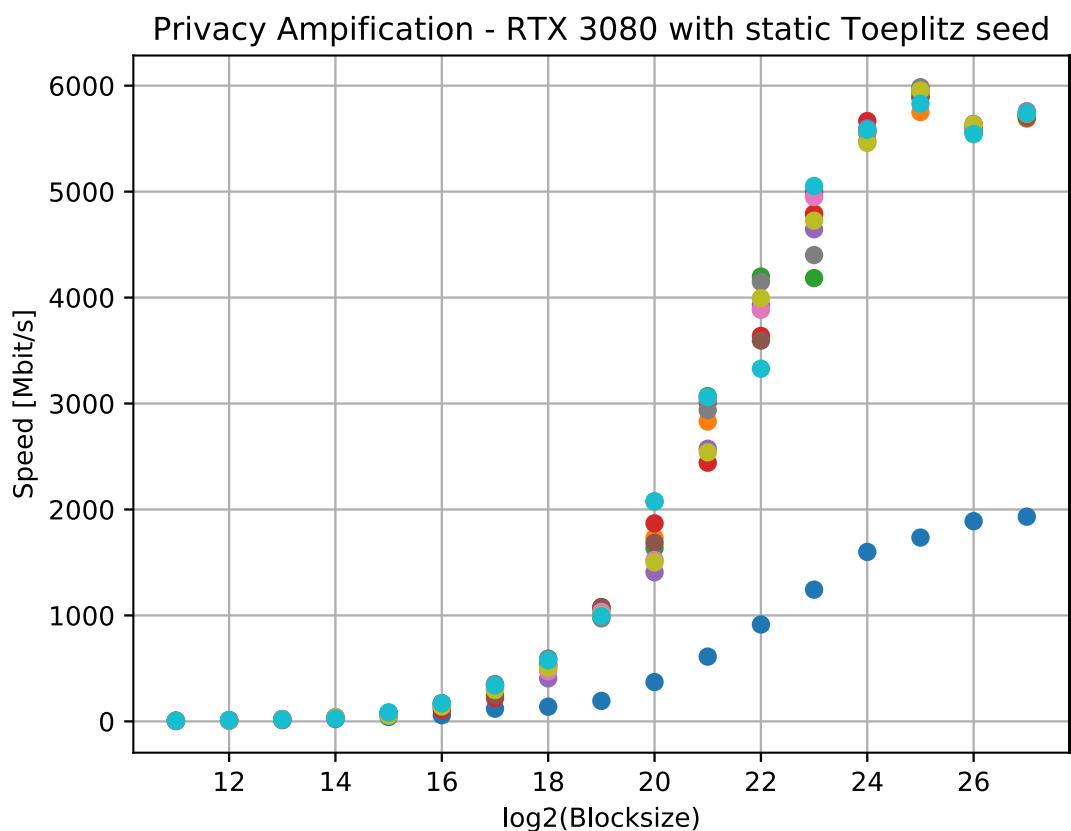


Abbildung 42: Performance mit static seed

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/-/commit/4b1f851e05f6eb32a215cd9c13f9c28b5a92b77d>

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/-/commit/599ed4b20182955b7b8af82de2013ea791d68e2a>

27. GitLab Pipelines

Ich habe in einem LXC Container ein selbst gehostetes GitLab aufgesetzt, um das mit den GitLab pipelines zu testen. Zudem habe ich mich in die Sache eingelese. GitLab in LXC Container zu installieren ist schwieriger als ich dachte. Während der Installation versucht GitLab Änderungen am Kernel vorzunehmen, um Kerneleigenschaften wie Prozesslimit zu erhöhen. Zudem wird eine systemd ähnliches System vorausgesetzt, welches üblicherweise in LXC Containern nicht existiert. Ich habe mich in der Vergangenheit schon einmal damit befasst als ich auf meinem Router GitLab installieren wollte und wusste daher, dass diese Probleme nur mit sehr viel Aufwand zu beseitigen sind. Nach einigen Recherchen fand ich jedoch glücklicherweise wonach ich suchte: Ein LXC image auf welchem GitLab schon vorinstalliert ist. Dieses ging dann auch ziemlich einfach zum Installieren. Ich habe jetzt die

perfekte Umgebung um das mit den GitLab pipelines auszuprobieren. Zudem habe ich auch schon über GitLab agents gelesen jedoch noch keine eingerichtet.

28. Absturz behoben

In der zweiten Runde crashte mein Programm seit dem Implementieren der Performancestatistik. Durch Binärsuche der letztwöchigen git commits fand ich den Verursacher. Durch eine erneute Binärsuche suche ich die fehlerhafte Zeile dieses commits. Nach stundenlanger Suche stellte sich heraus, dass ich versehentlich beim «Einifen» der Entscheidung, ob man die Codeflusskontrolle zurück am Performancetest geben oder den Memory bank switchen soll, die Zeile zum Switchen des memory banks nach dem if Statement vergessen habe zu löschen. Dadurch wurde 2-mal den bank geswitched was zu einem Absturz führte.

Ich habe den Fehler wie folgt behoben:

```
@@ -1406,7 +1406,7 @@ int main(int argc, char* argv[])
1406     1406     {
1407     1407         output_cache_write_pos = (output_cache_write_pos + 1) % output_blocks_to_cache;
1408     1408     }
1409 -    output_cache_write_pos = (output_cache_write_pos + 1) % output_blocks_to_cache;
1409 +    +
1410     1410 }
```

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/-/commit/a03a3d4f31efea7dbd43846a38ca9cd626a30284>

29. Zusammenfassen mehrerer Blöcke

Ich habe mehrere Blöcke in einer Memory Bank zusammengefasst, um den Performance-Unterschied zwischen grossen und kleinen Blöcken zu eliminieren. Der Grund das der Datentthroughput bei kleinen Blockgrössen mit dem aktuellen Algorithmus so langsam ist, ist auf den Overhead, der durch das tausendfache Ausführen des Algorithmus pro Sekunde entsteht, zurückzuführen. Bei einer Blockgrösse von 2^{10} wären dies 2^{17} Algorithmen-Durchläufe um dieselbe Datenmenge eines einzigen 2^{27} grossen Blocks zu erreichen. Dies negiert nicht nur den Vorteil der effizienteren FFT bei kleinen Blockgrössen, sondern führt zu einem Algorithmus, welcher für kleine Blockgrössen langsamer ist, als einer, der auf dem CPU läuft. Bis anhin wurde der Algorithmus auch speziell auf 2^{27} bit Blockgrösse optimiert, da dies die für Genf die interessante Blockgrösse ist. Nun wollen wir aber den Algorithmus genereller gestalten um für eine Vielzahl von Anwendungen nützlich zu sein.

Nach langem Überlegen hatte ich folgende Idee um die Geschwindigkeit bei kleinen Blockgrössen zu verbessern: Durch das Zusammenfassen vieler kleiner Blöcke (bis zu 10^{17} im Falle einer Blockgrösse von 2^{10}) in eine Memory Bank der Grösse 2^{27} und das Durchführen des Algorithmus mit diesem gesamten Memory Bank besteht kein Performanceunterschied mehr zwischen den verschiedenen Blockgrössen. Ausser vielleicht, dass ein Overhead beim Auffüllen mit Nullen entsteht. Dafür kann man aber voll und ganz von der schnelleren FFT profitieren.

Leider stellte sich das Anpassen meines Algorithmus an solche sub Blöcke als ziemlich schwierig heraus, da alles ohne Subblöcke geplant wurde. Vor allem das Memory Management musste ziemlich angepasst werden.

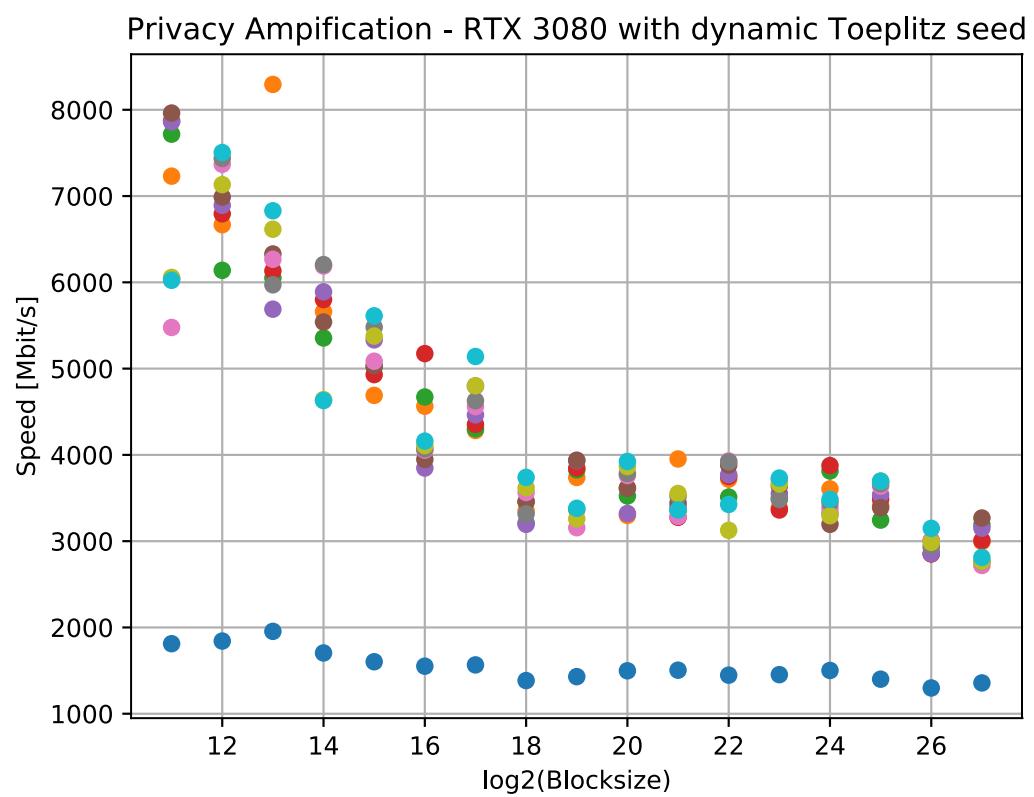


Abbildung 43: Performance mit kleinen Blöcken, dynamisch (provisorisch!!!)

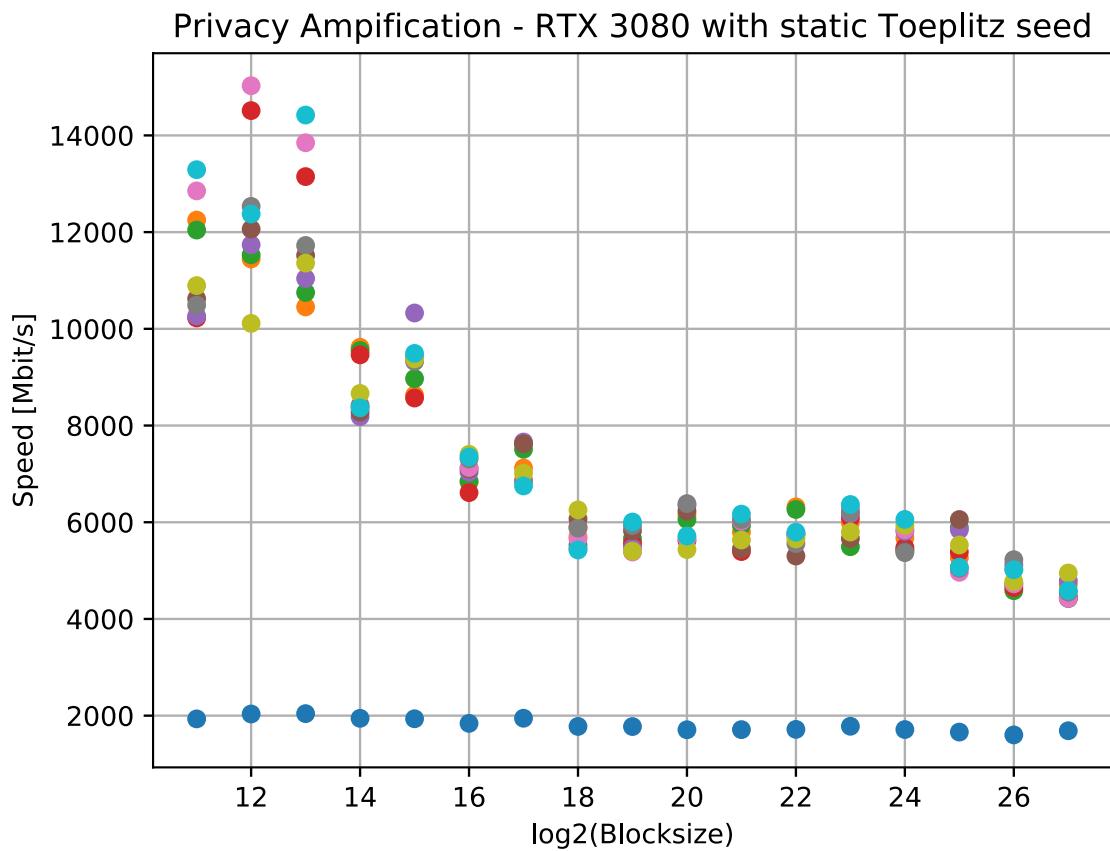


Abbildung 44: Performance mit kleinen Blöcken, statisch (provisorisch!!!)

30. Zusammenfassen mehrerer Blöcke

Beim letzten Meeting war mein Algorithmus nicht funktionsfähig und crashte. Durch den auf letztes Mal hinzugefügten debugging Code war schnell klar, dass das Problem an dieser cudaMemcpy Zeile liegt. Was jedoch alles andere als klar ist, ist was an dieser Zeile falsch sein sollte.

```
if (relevant_keyBlocks_old > relevant_keyBlocks) {
    /*Fill dirty memory regions parts with zeros*/
    uint32_t amount_of_zeros_to_fill = (relevant_keyBlocks_old - relevant_keyBlocks);
    uint32_t key_block_size = relevant_keyBlocks + amount_of_zeros_to_fill;
    for (int i = 0; i < blocks_in_bank; ++i) {
        checkCudaErrors(cudaMemcpy(di1 + i*key_block_size + relevant_keyBlocks, 0b00000000, amount_of_zeros_to_fill * sizeof(Real)));
    }
}
```

Abbildung 45: Fehlerhafter Programmausschnitt

Durch Probieren fand ich heraus, dass durch das Löschen von `i*key_block_size+relevant_keyBlocks` der crash nicht mehr auftrat. Jedoch brauchte ich dies um zu bestimmen für welchen Block die Nullen gesetzt werden sollten.

Um den Fehler zu finden skizzierte ich wie mein momentaner Programmaufbau und dessen Memorymanagement vereinfacht aussieht:

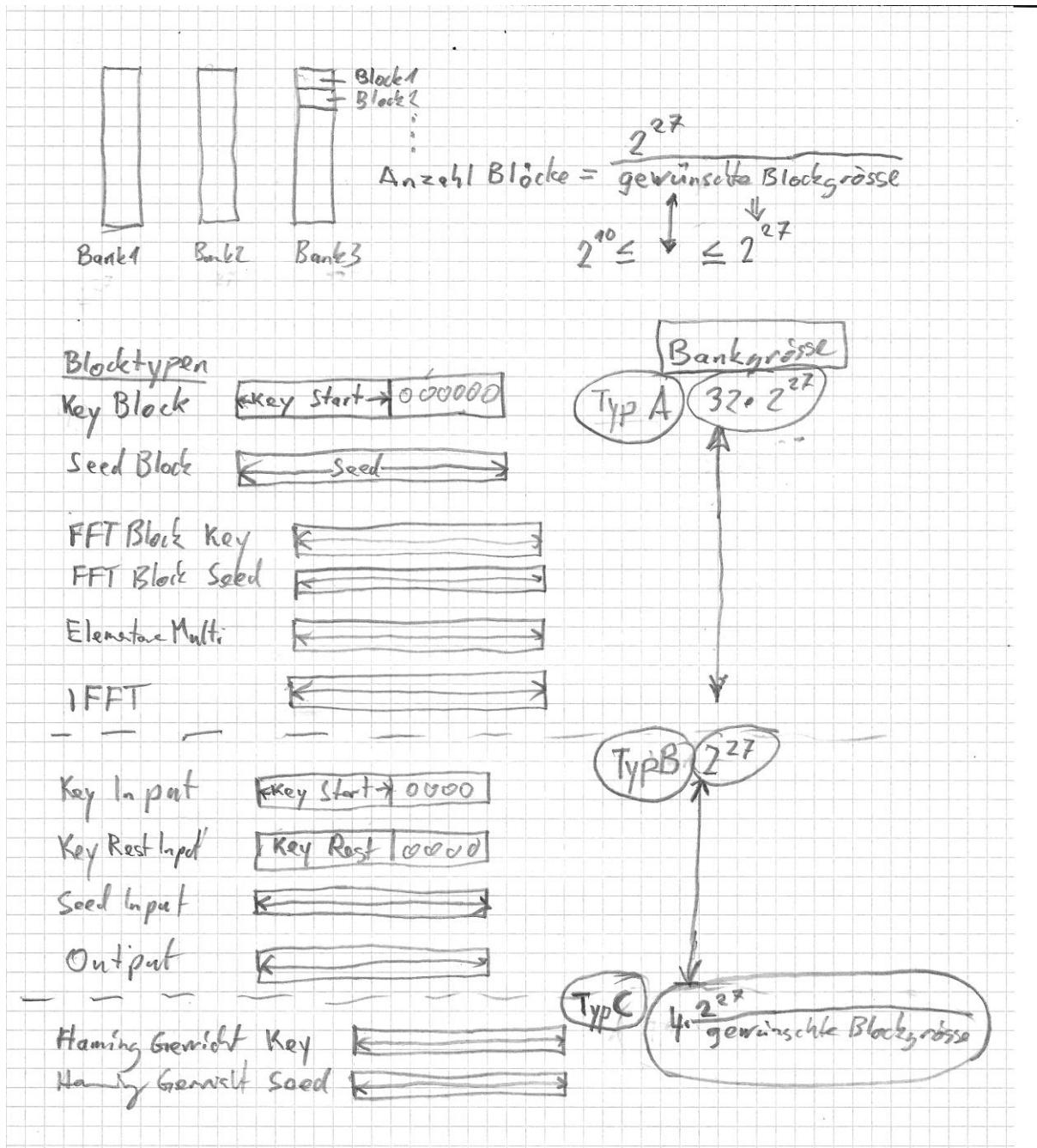


Abbildung 46: Skizze von Blocks in banks

Durch langes Überlegen und viel debuggen stellte sich schlussendlich heraus, dass relevant_keyBlocks nicht dem erwarteten, sondern einem viel zu hohen Wert entsprach. Es stellte sich heraus, dass relevant_keyBlocks durch die speedtest controllroutine nicht upgedatet wurde und sich somit immer auf dem viel zu hohen Standardwert befand. Dadurch werden Speicherregionen ausserhalb des memory banks überschrieben was zu

Fehlverhalten meines Programms im Release mode und sofortigen Crash meines Programms im debug mode führte.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/-/commit/a8ee8502bd7a19785dedbbf35155a7e32e6190ff>
<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/-/commit/11bc91731b867a607ae9d452685248d2a26c0b58>
<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/-/commit/06e7056154022d14217490ecb0a949d7565df1c3>
<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/-/commit/e3d521a394e1f8d42ed1dbf4d95940afce2422a2>
<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/-/commit/13e6b0d08f356d4498387125410e3e8aab48c197>
<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/-/commit/e85d7332c6a274353883a480195019ed545f0c4f>
<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/-/commit/0de79fa11f818dbda3e2884c848643f90d5e046a>

31. Testcases für beliebige Blockgrößen

Bis anhin habe ich nur Blöcke der Grösse 2^{27} per Integrationstests abgedeckt, da zusammen mit den Unit Tests welche jeden Cuda Kernel auf ihre Funktionalität prüfen, dies ausreichend war. Mittlerweile hat sich der Fokus des Projektes jedoch auf kleinere Blöcke geschiftet. Diese erlauben es durch schnellere FFT mit $O(n \log n)$ einen höheren Throughput rauszuholen. Durch diesen Focusschift müssen auch kleinere Blockgrößen nun durch Integrationstests abgedeckt werden. Durch das Zusammenführen der Blöcke in memory banks entsteht ein weiterer Abstraktionslayer. Ein weiterer Abstraktionslayer durch dessen Implementierung viele Fehler möglich sind. Diese treten nur auf falls ein memory bank mehrere Blöcke enthält. Somit muss dies getestet werden.

Um dies umzusetzen musste ich zurück zu der Python Referenzimplementierung, welche wiederum im letzten Semester durch eine Matrixmultiplikation validiert wurde. Ich modifizierte meine Referenzimplementierung so, dass diese nun auch aneinandergehängte Blöcke mit beliebigen Blockgrößen unterstützt. Modifikationen an der Referenzimplementierung sind immer besonders schwierig da sichergestellt werden muss, dass diese auf keinen Fall fehlerhaft ist, weil ich sonst stundenlang nach nichtexistierenden Fehlern suchen würde oder noch schlimmer – bei beiden denselben Fehler mache, was zu einem fehlerhaften Algorithmus führen würde. Dazu validierte ich meine Referenzimplementierungsmodifikationen immer noch mit anderen Python scripts und überlegte mir, ob das was ich mache mathematisch einen Sinn ergibt. Auch versuchte ich die Modifikationen so klein und simpel wie möglich zu halten.

Ein grosser Nachteil vom Python script ist seine Laufzeit. Da die Referenzimplementierung nicht auf Geschwindigkeit optimiert wurde dauert alles immer sehr lange. Während das Generieren der eigentlichen FFT Resultate in tolerierbarer Zeit von wenigen Minuten berechnet wird, dauert das Umwandeln des boolean array Resultats in Binärdaten Stunden! `out_arr = np.packbits(amp_key)` braucht einfach ewig. Eventuell sollte ich mir eine schnellere Implementierung dazu schreiben.

Es dauerte 7 Stunden und 17 Minuten um die Testcases der Blockgrösse 2^{10} zu generieren.

<https://gitlab.enterpriselab.ch/qkd/cpu/privacyamplificationpython/-/commit/86e4d421287785b2330df45a27435d34ba89ce27>

<https://gitlab.enterpriselab.ch/qkd/cpu/privacyamplificationpython/-/commit/c4fcdb5f7e04e4f3b676574e0ff1ae9ad222889e>

32. cuLDPC

Ich habe cuLDPC für einen Release nächste Woche vorbereitet. Mein Code ist gut verständlich. Das Einzige was noch fehlt sind einige Kommentare, das Portieren in ein neues Visual Studio Projekt und ein Linux Makefile. Dies muss ich selber schreiben, weil es sonst unter der NVidia Cuda Toolkit Lizenz wäre und ich es aber unter Apache lizenzieren möchte.

https://gitlab.enterpriselab.ch/qkd/gpu/culdpc_optimized/-/commit/84da11ebb655d43a9e5e1c5e0a51262be534b507

33. Reparieren der bestehenden Tests

Durch das Zusammenfassen mehrerer Blöcke funktionierten die bestehenden Testcases nicht mehr, da sie nicht dem neuen Code angepasst wurden. Dies habe ich nun erledigt. Bei 2^{27} waren alle erfolgreich.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/-/commit/52ece68cf72b03327e8ea1b1d332927ae2b400ff>

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/-/commit/0ce72eaeaffd86b66e9e950accad6c178ca1907>

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/-/commit/25db7e7fff1f156e12ecf6614886cea21672e112>

34. GitLab Runner

Ich habe einen GitLab shell runner erstellt:

```

Administrator: Windows PowerShell
Copyright (C) Microsoft Corporation. Alle Rechte vorbehalten.

PS C:\Windows\system32> cd ..
PS C:\Windows> cd ..
PS C:\> cd \Users\Azure
PS C:\Users\Azure> cd \Documents
PS C:\Users\Azure\Documents> ls

Verzeichnis: C:\Users\Azure\Documents

Mode LastWriteTime      Name
---- -----          ----
d----- 28.06.2020 11:14  agent_Hot
d----- 25.10.2020 21:27  Azure
d----- 08.12.2020 12:29  GitLab
d----- 30.06.2020 13:17  Visual Studio 2017
d----- 28.06.2020 12:10  Visual Studio 2019
-->--- 28.05.2020 19:13  94 stefanbulianczuk_Azure_access_token

PS C:\Users\Azure\Documents> cd \GitLab
PS C:\Users\Azure\Documents\GitLab> ls

Verzeichnis: C:\Users\Azure\Documents\GitLab

Mode LastWriteTime      Name
---- -----          ----
-->--- 08.11.2020 12:29  77 config.toml
-->--- 08.12.2020 12:28  400900000 gitlab-runner.exe.exe

PS C:\Users\Azure\Documents\GitLab> .\gitlab-runner.exe install
Runtime platform           arch=amd64 os=windows pid=34188 revision=8fa89735 version=13.6.0
PS C:\Users\Azure\Documents\GitLab> .\gitlab-runner.exe start
Runtime platform           arch=amd64 os=windows pid=32796 revision=8fa89735 version=13.6.0
PS C:\Users\Azure\Documents\GitLab> .\gitlab-runner.exe register
Runtime platform           arch=amd64 os=windows pid=33460 revision=8fa89735 version=13.6.0
Enter the instance URL (for example, https://gitlab.com/):
https://gitlab.nico.re
Enter the registration token:
PS C:\Users\Azure\Documents\GitLab> .\gitlab-runner.exe register
Runtime platform           arch=amd64 os=windows pid=24496 revision=8fa89735 version=13.6.0
Enter the instance URL (for example, https://gitlab.com/):
http://192.168.2.184
Enter the registration token:
PS C:\Users\Azure\Documents\GitLab>
Enter a description for the runner:
(CastlePeak)
Enter the executor for the runner (comma separated):
SSH-20H5
Registering runner... success
runner=QKdByz2
[runner] executor: docker-in-machine, custom, docker-windows, virtualbox, shell, ssh, docker-machine, kubernetes, docker, docker-ssh, parallels:
runner registered successfully. Feel free to start it, but if it's running already the config should be automatically reloaded!
PS C:\Users\Azure\Documents\GitLab>

```

Abbildung 47: Mein GitLab runner

35. Fehlerbehebungen

Ich habe etliche Abstürze behoben. Somit läuft nun der Code wieder stabil.

Die meisten Abstürze wurden durch die Verwendung von `input_blocks_to_cache` / `output_blocks_to_cache` anstelle von `input_banks_to_cache` / `output_banks_to_cache` verursacht. Dadurch waren die Modulos falsch und `output_cache_write_pos` wurde zu weit hochgezählt. Dadurch entstand ein Speicherüberlauf.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/-/commit/64e2780ecce45eaad65146a4eeacc52cc6e92a43>

36. Weitere Fehlerbehebungen

Ich habe nochmals etliche Fehler behoben. Somit läuft nun der Code noch stabiler.

Es wurden irrtümlicherweise an einigen Orten `BANK_SIZE_UINT32` anstelle von `BANK_SIZE_BYTES` verwendet. Durch diesen Fehler wurden viermal weniger Memory verarbeitet als gewollt.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/-/commit/1f6ec54bf3d5416b0a6c32fc7e8c01543a63c770>

37. Nochmals Fehlerbehebungen

An einigen Orten wurde desired_block anstelle von BANK_SIZE_BYTES verwendet. Dies führte dazu, dass zu wenig Daten verarbeitet wurden. Es ist mir dabei aufgefallen, dass es noch ein Problem bei der Aufteilung beim Keyrest gibt. Dies muss später noch erledigt werden.

<https://gitlab.enterpriselab.ch/qkd/gpu/privacyamplification/-/commit/c61b9615d1db691f64defefcd90ac54c35632a63>

38. Neuorganisation von GitLab

Gemäss der Anregung von Roland Christen habe ich das gesamte GitLab neu organisiert. Die Gruppe wurde nun in QKD umbenannt. Es wurden die Untergruppen DOC, LIB, CPU und GPU erstellt.

<https://gitlab.enterpriselab.ch/qkd>

39. Recherche zu Referenzarbeiten

Ich habe im Internet nach Fachartikeln zum Thema schnelle PA-Applikationen mit QKD gesucht. Die Suche war noch schwierig, da es dazu nicht viele neue Publikationen gab. Oft waren die Dokumente gesperrt und auch nicht so einfach zu interpretieren. Die Werte zu den Geschwindigkeiten wurden uneinheitlich angegeben und die Randbedingungen waren mir nicht immer klar.

40. Abstract ergänzt

Ich habe die Vorlage mit meinen Daten erweitert. Ich habe nicht viel hinzugefügt, da ich denke, dass er möglichst kurz und prägnant sein soll. Die genaueren Erläuterungen wären ja dann in den anderen Dokumenten abzuliefern.

41. cuLDPC für Veröffentlichung vorbereitet

Ich habe allen nicht durch Apache Lizenzierten Codes entfernt, zu gitignore hinzugefügt und README angepasst.

https://gitlab.enterpriselab.ch/qkd/gpu/culdpc_optimized/-/commit/441994080e422138a9e6340f82cb0d7aafce6af2

42. Schnelle Generierung von Testcases für beliebige Blockgrößen

Die Generierung von Testcases dauerte 8 Stunden. Ich habe einen 500-mal schnelleren Algorithmus entwickelt. Alles lag an einem print und der alte Algorithmus wäre perfekt gewesen. Durch weitere Verbesserung wurde er nun 1000-mal schneller.

<https://gitlab.enterpriselab.ch/qkd/cpu/privacyamplificationpython/-/commit/63e745a5e97cf044e16cda2b9a7995af1add1a43>

<https://gitlab.enterpriselab.ch/qkd/cpu/privacyamplificationpython/-/commit/9c3aa4f16026cc716ad7b673d693b5bc2be053a0>

<https://gitlab.enterpriselab.ch/qkd/cpu/privacyamplificationpython/-/commit/89066a36b13ded94c6dae2ea75591cf4ae6b7a72>

Zusammenfassung der Forschungsmitarbeit

Dass ich in den Sommerferien zwei Wochen als Ferienjob an dem Projekt weiterarbeiten durfte, war natürlich super. In dieser Zeit habe ich für Genf das Programm noch benutzerfreundlicher gemacht, den Code korrekt kommentiert und eine Dokumentation erstellt. Die Arbeit wurde dann zum Forschungsteam nach Genf geschickt.

Bei der Forschungsmittarbeit während des Semesters habe ich folgende Themen bearbeitet respektive Ziele erreicht:

- **Linux + Docker**

Der Algorithmus läuft jetzt auch auf Linux und in einem Docker-Container.

- **libtrevisan-py / Trevisan-algorithmus**

Der Versuch mit dem Trevisan-Algorithmus hat gezeigt, dass die Berechnungen mit der Toeplitz-Matrix wesentlich schneller sind. Der Trevisan ist ein intelligenter aber leider langsamer Algorithmus.

- **PA auf billigen, leistungsschwachen Grafikkarten**

Ein Ziel dieser Forschungsarbeit war, dass ich meinen Privacy Amplification – Algorithmus auf einem langsamem und billigen Grafikprozessor zum Laufen bringe. Dazu benutzte ich den Nintendo Switch und installierte darauf ein Linux-Betriebssystem. Mein Algorithmus funktionierte bestens und er ist am schnellsten mit 2^{21} Blockgrößen. Die Geschwindigkeit auf der PC-Grafikkarte war erstaunlicherweise nur ca. 20-mal schneller als auf dem Nintendo Switch!

- **Ampere Architektur – neue Grafikkartengeneration**

Ich konnte erfolgreich die neue Grafikkarte GeForce RTX 3080 kaufen, die im Oktober neu auf den Markt gekommen ist. Diese Grafikkarte hat die neue Ampere-Architektur. Mit der neuen Grafikkarte bringe die Rechenzeit für meinen Privacy Amplification – Algorithmus von 60 ms auf unter 38 ms! Also fast 60 % schneller!!!

- **Pipeline-, Unit- und Integrationstests**

Um die Korrektheit des Algorithmus jederzeit zu gewährleisten habe ich die Pipeline-, Unit- und Integrationstests in meinen Code integriert.

- **Azure Pipeline**

Der Vorteil der Integration meines Projekts in die Azure Pipeline ist der vollautomatisierte Testlauf nach jedem Commit (CICD). Nach einigen Schwierigkeiten gelang die Integration.

- **Blocks in banks / beliebige Blockgrößen**

Ich habe mehrere Blöcke in einer Memory Bank zusammengefasst, um den Performance-Unterschied zwischen grossen und kleinen Blöcken zu eliminieren. Durch das Zusammenfassen vieler kleiner Blöcke (bis zu 10^{17} im Falle einer Blockgröße von 2^{10}) in eine Memory Bank der Grösse 2^{27} und das Durchführen des Algorithmus mit dieser gesamten Memory Bank besteht keinen Performanceunterschied mehr zwischen den verschiedenen Blockgrößen. Er ist sogar schneller.

Schlusswort

Ich habe mich auch in diesem Semester wieder sehr über die spannende Forschungsmitarbeit gefreut. Toll fand ich natürlich, dass ich die Möglichkeit für einen Ferienjob im Sommer hatte und dort die Arbeit für die Genfer Forscher abschliessen durfte. Dass wir unsere Arbeit für die Poster-Session an der 24th Annual Conference on Quantum Information Processing (QIP) einreichen durften war sehr cool. Der Hammer war dann, dass unsere Arbeit akzeptiert wurde und wir nun an der Konferenz unser Poster präsentieren dürfen. Leider ist der Zeitpunkt zur Einreichung des Posters genau in meiner stressigsten Prüfungszeit. Ich freue sehr auf diese Teilnahme und bin gespannt was da auf uns zukommt.

Abbildungsverzeichnis

Abbildung 1: Vergleich Hardwarerevision 1 + 2	10
Abbildung 2: «Erhältlich» 3080 GPUs auf Digitec	10
Abbildung 3: Übertakten der Grafikkarte	11
Abbildung 4: Externe RTX 2070 super	12
Abbildung 5: PC mit den 3 Grafikkarten	13
Abbildung 6: Wert im Debug-Mode	14
Abbildung 7: Wert im Release-Mode	15
Abbildung 8: Vergleich Debug-memdumb und Release-memdumb	15
Abbildung 9: Tegra X1 SoC	17
Abbildung 10: Durch bootloaderexploit injektiertes Bootmenü	18
Abbildung 11: Laufender Privacy Amplification-Algorithmus	18
Abbildung 12: Ausschnitt aus laufendem Programm	18
Abbildung 13: Code des Unit-Tests	20
Abbildung 14: So wurde der Fehler behoben	21
Abbildung 15: Bild des Testcodes	22
Abbildung 16: Code des Testcase	23
Abbildung 17: Meine Azure Pipeline (azure-pipelines.yml)	25
Abbildung 18: Ausgeführte Pipeline	26
Abbildung 19: PrivacyAmplification Pipeline	26
Abbildung 20: Pipeline Artefacts	26
Abbildung 21: Repository auf GitHub pro	27
Abbildung 22: GitHub erfolgreich mit Azure verbunden	27
Abbildung 23: Meine Tokens	28
Abbildung 24: Mein Server Castlepeak im Pool	28
Abbildung 25: Script shortGitHash	28
Abbildung 26: Azure-Artefacte Bibliotheken	29
Abbildung 27: Azure-Artefacte Testdaten	29
Abbildung 28: Ausschnitt des Downloadskripts	29
Abbildung 29: Code vom AssertEquals	30
Abbildung 30: Code von David W. Wilson	31

Abbildung 31: Unitest des BinInt2float.....	32
Abbildung 32: Validierung der Werte	32
Abbildung 33: Algorithmus ToBinaryArray Unit Test	34
Abbildung 34: Validierung der Werte	35
Abbildung 35: Code zur Performance-Messung	36
Abbildung 36: Skript zur Generierung der Statistik	37
Abbildung 37: Skript zum Hochladen der Statistik auf GitHub gist	37
Abbildung 38: PA mit RTX 3080 dynamic seed	38
Abbildung 39: PA mit RTX 3080 statik seed	38
Abbildung 40: Eigenständiges cuLDPC	39
Abbildung 41: Performance mit dynamischem seed	40
Abbildung 42: Performance mit static seed.....	41
Abbildung 43: Performance mit kleinen Blöcken, dynamisch (provisorisch!!!)	44
Abbildung 44: Performance mit kleinen Blöcken, statisch (provisorisch!!!)	45
Abbildung 45: Fehlerhafter Programmausschnitt.....	45
Abbildung 46: Skizze von Blocks in banks	46
Abbildung 47: Mein GitLab runner.....	49