Masterarbeit

Entwicklung eines Autotuning Frameworks mittels DVFS zur Steigerung der Energieeffizienz von NVIDIA Grafikkarten am Beispiel von Mining Algorithmen

---

Development of an autotuning framework using DVFS to increase the energy efficiency of NVIDIA graphic cards considering mining algorithms

Alexander Fiebig

Bayreuth, am 7. Januar 2019

Universität Bayreuth
Angewandte Informatik 2

Betreuer: Prof. Dr. Thomas Rauber, Matthias Stachowski, MSc.

Zusammenfassung

Energieeffizientes Rechnen ist insbesondere im Bereich des High-Performance-Computing auf Supercomputern von Bedeutung. Deshalb ist die automatische Optimierung der Energieeffizienz zur Laufzeit von rechenintensiven Programmen wünschenswert.

In dieser Arbeit wird ein Framework zur automatischen Steigerung der Energieeffizienz von NVIDIA GPUs mithilfe von DVFS vorgestellt. Als Anwendung wird das Mining von Kryptowährungen gewählt, da hierbei die Energieeffizienz von besonderer Bedeutung ist. Beim Mining erhält derjenige, der ein rechenintensives mathematisches Puzzle zuerst löst eine Belohnung in Form von Coins, welche in Echtgeld umtauschbar sind. Dabei ist die Wahrscheinlichkeit für einen Gewinn proportional zur verwendeten Rechenleistung und die Kosten entsprechen dem Energieverbrauch der Hardware.

Das Framework ermittelt automatisch die energieoptimalen Frequenzen für jede verfügbare Währung auf allen GPUs eines Rechners. Im Anschluss wird das Mining gestartet und in einer Überwachungsphase sichergestellt, dass immer die beim Energieoptimum profitabelste der verfügbaren Währungen auf jeder GPU gemint wird.

Tests mit verschieden GPUs zeigen, dass die Energieeffizienz, je nach GPU und Währung, um bis zu 84% im Vergleich zu den Standardfrequenzen gesteigert werden kann, was den Profit beim Minen entsprechend fast verdoppelt.

Abstract

Energy efficient computing is especially important in the field of High-Performance-Computing on supercomputers. Therefore, automated optimization of energy efficiency during the runtime of a compute intensive program is desireable.

In this thesis a framework for the automatic increase of energy efficiency on NVIDIA GPUs using DVFS is presented. As an application the mining of cryptocurrencies is chosen, because here, energy effieciency is of particular importance. When mining cryptocurrencies, the one that solves a mathematical puzzle first earns an reward in coins, that can be exchanged to real money. The probability to win the reward is proportional to the used computing power and the cost correspond to the energy consumption of the hardware.

The framework determines the energy optimal frequencies for each available currency on each GPU of a computer automatically. Following this, the mining is started and during a monitoring phase it is ensured that always the most profitable currency, considering optimal frequencies, is mined on each GPU.

Tests with different GPUs show that the energy efficiency, depending on the GPU and currency, can be increased up to 84% compared to when using default frequencies. This in turn almost doubles the mining profit.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Moderne Supercomputer verfügen über massive Rechenleistung, benötigen aber auch entsprechend viel Energie. Der Stand November 2018 auf Platz eins der TOP500-Liste stehende Summit verfügt bei 4608 Rechenknoten über zwei IBM POWER9 Prozessoren und sechs NVIDIA Tesla V100 GPUs pro Knoten. Er hat eine maximale Leistungsaufnahme von insgesamt 13MW, wovon bei 300W pro GPU 8.29MW auf die GPUs entfallen (siehe [37] und [21]). Das zeigt, dass GPUs im Bereich des High-Performance-Computing von großer Bedeutung sind und solche Supercomputer einen enormen Energiebedarf haben. Daher ist die automatische Optimierung der Energieeffizienz zur Laufzeit rechenintensiver Programme auch auf GPUs wünschenswert. Dabei sollte die Optimierung die Performanz des ausgeführten Programms möglichst wenig beeinflussen.

1.2 Ziel der Arbeit

Ziel der Arbeit ist die Entwicklung eines Frameworks zur automatischen Optimierung der Energieeffizienz von NVIDIA Grafikkarten mittels Dynamic Voltage Frequency Scaling (DVFS). Dabei werden Core- und VRAM-Frequenz einer GPU an die Eigenschaften des ausgeführten Programms so angepasst, dass möglichst viel Energie bei möglichst wenig Performanceeinbußen gespart wird. Je nachdem wie stark ein Programm Memory- oder Compute-Bound ist, kann die Core- bzw. VRAM-Clock ohne großen Performanceverlust reduziert, und somit Energie gespart werden. Für die Optimierung der Frequenzen sollen verschiedene Optimierungsalgorithmen implementiert und miteinander verglichen werden.

Als Anwendung wird das Mining von Kryptowährungen betrachtet, da hier GPUs häufig zum Einsatz kommen und die Energieeffizienz von besonderer Bedeutung ist. Beim Mining ist die Rechenleistung proportional zur Gewinnwahrscheinlichkeit und der Energieverbrauch der Hardware entspricht den Kosten. Eine Erhöhung der Energieeffizienz erhöht hier direkt den erzielten Profit. Das Framework soll für alle verfügbaren Währungen die energieeffizientesten Frequenzen auf einer GPU automatisch finden und anschließend die am Energieoptimum profitabelste Währung minen. Das Ganze soll parallel für alle GPUs eines Rechners erfolgen.

1.3 Verwandte Arbeiten

In [18] werden verschiedene Techniken für DVFS auf GPUs vorgestellt und in einer Studie miteinander verglichen. In [7] wird der Effekt von DVFS auf einer NVIDIA Geforce GTX 560 Ti anhand verschiedener Beispielprogramme untersucht. Dabei werden sowohl Core- und VRAM-Frequenzen als auch Core- und VRAM-Voltages mittels den Tools *NVIDIA Inspector* und *MSI Afterburner* manuell angepasst. In [40] wird der Effekt von DVFS auf

GPU und CPU am Beipiel der Matrixmultiplikation verglichen. In [5] wird eine Technik zur Reduzierung des Energieverbrauchs zur Laufzeit von GPU-Programmen mittels DVFS im Zusammenspiel mit der Überwachung der aktuell genutzten Speicherbandbreite vorgestellt. In [13] werden Modelle zur Vorhersage des Energieverbrauchs einer GPU bei verschiedenen Core- und VRAM-Frequenzen erstellt. Die Modelle werden mittels Machine Learning Techniken durch Messdaten von verschiedenen Anwendungen trainiert. In [22] werden solche Modelle verwendet um die Energieeffizienz von mobilen Videospielen zu steigern. Die trainierten Modelle werden in einem Power-Management-System verwendet, welches die CPU- und GPU-Frequenzen zur Laufzeit einstellt. In [12] ist ein allgemein gehaltener Überblick und eine Kategorisierung verschiedener Autotuning-Techniken zu finden.

Im Bereich des Mining gibt es die kommerzielle Software *Awesome Miner* (siehe [1]). Hier kann für jede GPU und Währung manuell ein Profil mit GPU-Frequenzen, Hashrate und Energieverbrauch erstellt werden. Auf Basis dieser Profile und entsprechenden Coin-Statistiken wird der Mining-Profit berechnet und immer die jeweils profitabelste Währung gemint.

1.4 Aufbau der Arbeit

Zunächst wird in Kapitel 2 ein kurzer Überblick über die Blockchain-Technologie gegeben, welche den Kyrptowährungen zugrunde liegt. Kapitel 3 beschäftigt sich anschließend mit dem Mining, dessen Funktionsweise, den verschiedenen Arten des Mining sowie der zum Mining verwendeten Hardware.

Den Hauptteil der Arbeit bilden Kapitel 4 und 5. Kapitel 4 beschreibt die Implementierung des Autotuning-Frameworks und Kapitel 5 evaluiert dieses anhand von verschiedenen GPUs und Währungen. Abschließend fasst Kapitel 6 die Ergebnisse zusammen und gibt einen Ausblick über mögliche Erweiterungen.

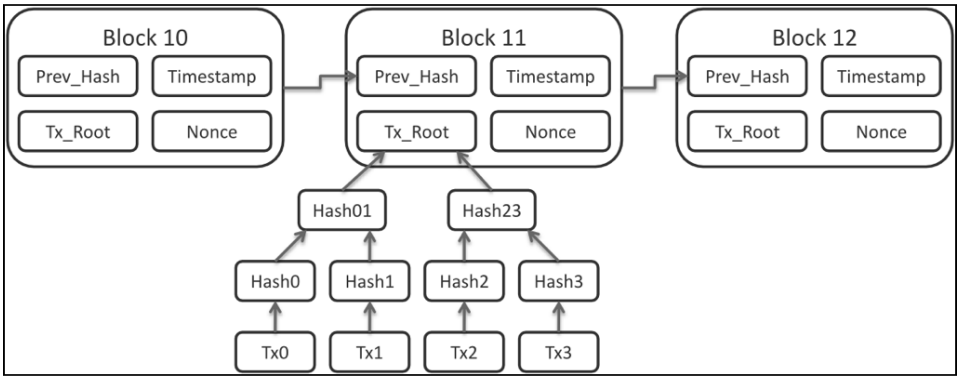

Abbildung 2.1: Vereinfachte Darstellung einer Blockchain mit Hash-Baum der Transaktionen (Quelle: [4])

2 Blockchain und Kryptowährungen

Dieses Kapitel soll einen kurzen Überblick über die Grundlagen der Blockchain-Technologie und der darauf aufbauenden Kryptowährungen geben.

2.1 Aufbau einer Blockchain

Eine Blockchain ist eine Liste von Datensätzen (Blöcke), die mittels kryptographischer Hashes miteinander verkettet sind. Jeder dieser Blöcke besteht aus einem Header und einem Body. Der Body enthält eine bestimmte Anzahl von durchgeführten Transaktionen und deren Hashes, welche in einem Hash-Baum (Merkle-Tree) abgelegt sind. Der Header besteht aus der Wurzel des Hash-Baums (Tx_Root), einem Zeitstempel (Timestamp) und einer Zufallszahl (Nonce).

Zudem enthält der Header den Hash des vorherigen Blocks (Prev_Hash). Dadurch wird sichergestellt, dass eine Transaktion nicht nachträglich modifiziert werden kann, ohne deren Block und alle nachfolgenden Blöcke zu modifizieren. Der Header kann, je nach Blockchain-Technologie, auch noch weitere Informationen enthalten. Auch die verwendete Hashfunktion kann sich je nach Blockchain-Technologie unterscheiden. Abbildung 2.1 zeigt schematisch den Aufbau einer Blockchain mit Hash-Baum der Transaktionen.

2.2 Transaktionen, Wallets und Währungen

Eine Transaktion ist eine Operation, die in der Blockchain festgehalten wird. Bei den meisten Transaktionen handelt es sich um Überweisungen von einem Nutzer zu einem anderen. Mit Ethereum ist es beispielsweise aber auch möglich Programmcode in eine Transaktion zu stecken. Dieser wird dann bei Eintreten einer bestimmten Bedingung ausgeführt. Eine solche Bedingung kann z.B. eine Überweisung eines bestimmten Betrages an einen bestimmten Nutzer sein (Stichwort Smart Contracts).

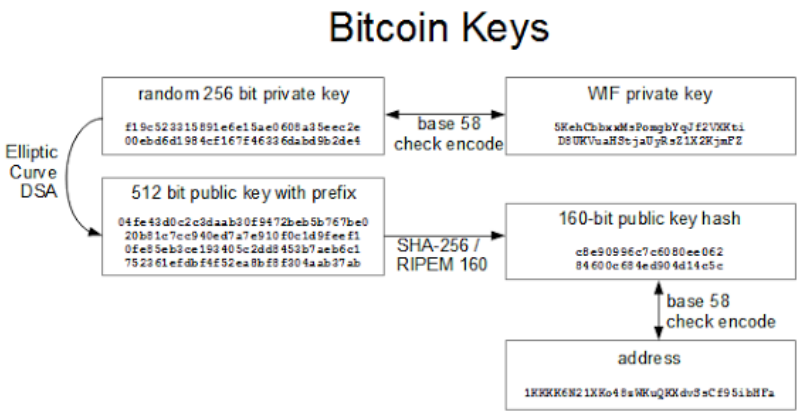

Abbildung 2.2: Zusammenhang zwischen öffentlichen Schlüssel, privatem Schlüssel und Wallet-Adresse am Beispiel von Bitcoin (Quelle: [35])

Die Nutzer der Blockchain werden über sogenannte Wallet-Adressen identifiziert. Wenn man eine neue Wallet-Adresse erzeugt, wird im Hintergrund ein privater und ein öffentlicher Schlüssel für ein asymetrisches Verschlüsselungsverfahren wie RSA erzeugt. Mit dem privaten Schlüssel kann der Nutzer Nachrichten signieren, und mit dem öffentlichen Schlüssel können die signierten Nachrichten auf ihre Gültigkeit überprüft werden. Die Kombination von Nachricht, öffentlicher Schlüssel und Signatur heißt dann Transaktion. Die Wallet-Adresse wird aus dem öffentlichen Schlüssel berechnet (siehe [3] und [35] für mehr Information). Da alle Transaktionen öffentlich in der Blockchain gespeichert werden, kann jeder den Inhalt einer Transaktion anhand von öffentlichen Schlüssel und Signatur verifizieren. Abbildung 2.2 zeigt den Zusammenhang zwischen öffentlichen Schlüssel, privatem Schlüssel und Wallet-Adresse am Beispiel von Bitcoin.

Blockchains haben meist eine eigene digitale Währung, welche der jeweiligen Blockchain-Technologie ihren Namen gibt. Diese Währung wird in den Transaktionen der Blockchain verwendet. Da diese Währungen nur durch Transaktionen innerhalb der Blockchain generiert und überwiesen werden können, ist sichergestellt, dass alle Transaktionen von allen Teilnehmern zurückverfolgbar sind. Niemand hat die Möglichkeit eigene Einheiten der Währung zu generieren oder zu verteilen. Die Währung einer Blockchain kann aber meist in andere digitale Währungen oder Echtgeld umgetauscht werden.

2.3 Peer-to-Peer Netzwerk und Konsens

Wichtigstes Merkmal der Blockchain-Technologie ist, dass eine Blockchain dezentral verwaltet wird. D.h. es gibt keinen zentralen Server auf dem die Blockchain gespeichert ist und an den Clients Anfragen stellen. Stattdessen gibt es ein Peer-to-Peer Rechnernetz, in dem jeder Rechner unabhängig die komplette Blockchain speichert und die enthaltenen Transaktionen verifiziert. Wird ein neuer Block von Transaktionen generiert, verifiziert

jeder Rechner lokal zunächst die Transaktionen des Blocks bevor er ihn speichert und an die anderen Rechner des Netzwerks schickt.

In einem solchen Peer-to-Peer Netzwerk kann es passieren, dass nicht jeder Rechner die gleichen Transaktionen in seiner lokalen Kopie der Blockchain speichert. Zum Beispiel kann ein kompromittierter Rechner versuchen illegale Transaktionen im Netzwerk zu verbreiten. Die Mehrheit der Rechner des Netzwerks muss sich daher auf einen Zustand der Blockchain einigen bzw. Konsens erreichen. Dabei muss davon ausgegangen werden, dass n Rechner bösartig arbeiten und versuchen das Netzwerk maximal zu stören. Anders ausgedrückt, das Netzwerk muss tolerant gegen byzantinische Fehler sein.

In der Blockchain-Technologie gibt es verschiedene Algorithmen um byzantinische Fehlertoleranz zu erreichen. Der am häufigsten genutzte Algorithmus ist Proof of Work (PoW), welcher in Abschnitt 3.1 erläutert wird. Andere Algorithmen sind z.B. Proof of Stake (PoS), Delegated Proof of Stake (DPoS) oder Proof of Elapsed Time (PoET) welche hier nicht weiter beschrieben werden (siehe [20]).

2.4 Verzweigungen

Verzweigungen in einer Blockchain können grundsätzlich auf zwei Arten entstehen:

1. Zwei oder mehr valide Blöcke werden gleichzeitig bzw. fast zur gleichen Zeit generiert.

2. Nach Änderung der Konsens-Regeln folgt ein Teil der Nutzer weiterhin den alten Regeln.

Werden zwei valide Blöcke ungefähr gleichzeitig generiert, entscheidet jeder Knoten des Peer-to-Peer Netzwerks der Blockchain individuell welcher der Blöcke akzeptiert und an die lokale Kopie der Blockchain angehängt wird. In der Regel wird der Block gewählt, welcher zuerst verfügbar ist. Aufgrund der Latenz des Netzwerkes kann sich dieser Block von Knoten zu Knoten unterscheiden. Es gilt die Regel, dass immer der längste gültige Zweig einer Blockchain fortgeführt wird. Daher verwaist eine so entstandene Verzweigung, sobald ein Block generiert wird, der keine Konkurrenz hat (kein zweiter gleichzeitig generierter Block vorhanden).

Durch eine Änderung der Konsens-Regeln, wie z.B. einer Anpassung der Hashfunktion der Blockchain, kann auch eine Verzweigung entstehen, indem ein Teil der Nutzer weiterhin den alten Regeln folgt. Je nachdem wie die Regeln geändert werden, spricht man von einem Hard- oder Softfork. Sind die neuen Regeln zu den alten kompatibel handelt es sich um einen Softfork, andernfalls um einen Hardfork. Details und die Fälle dazwischen sind in [33] beschrieben. Währungen wie Bitcoin Gold oder Ethereum Classic sind beispielsweise durch Hardforks entstanden und verwenden eine ältere Version der Konsens-Regeln von Bitcoin bzw. Ethereum.

Abbildung 2.3 zeigt zwei verschiedene Arten von Verzweigungen einer Blockchain. Oben ist eine verwaiste Verzweigung entstanden durch zeitgleiche Generierung zwei-

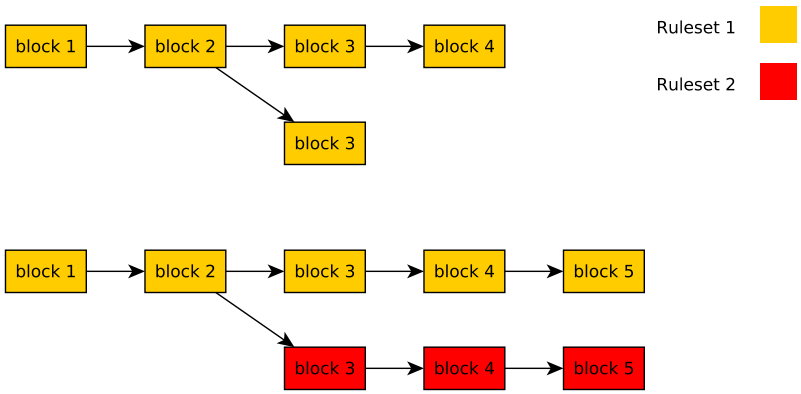

Abbildung 2.3: Verschiedene Verzweigungen einer Blockchain

er Blöcke zu sehen. Unten ist eine Verzweigung entstanden durch eine Änderung der Konsens-Regeln dargestellt.

3 Mining und Energieeffizienz

Folgendes Kapitel gibt einen kurzen Überblick über das Mining, dessen Nutzen und Funktionsweise, die zum Minen verwendete Hardware, sowie die verschiedenen Arten des Mining.

3.1 Funktionsweise PoW

Das meistgenutzte Verfahren um Konsens im Peer-to-Peer Rechnernetz zu erreichen ist Proof of Work (PoW) (siehe Abschnitt 2.3). Die Teilnehmer beim PoW heißen Miner. Es darf der Miner einen neuen Block an Transaktionen der Blockchain hinzufügen (den Block minen), der zuerst eine Zahl X findet, sodass der Hash des Blockes zusammen mit X kleiner ist als eine gegebene Zahl Y:

$$hash(block\_data \,||\, X) < Y \tag{3.1}$$

Die Zahl X wird auch *nonce* genannt (siehe Abbildung 2.1) und die Zahl Y heißt $target$. Das $target$ wird nach gesamter Rechenleistung aller Miner dynamisch angepasst, sodass es immer etwa gleich lang dauert bis ein neuer Block gefunden wird. Die Schwierigkeit einen Block zu minen korrespondiert mit dem $target$, wobei ein niedriges $target$ einer hohen Schwierigkeit entspricht. Das Problem aus Gleichung 3.1 hat zwei wichtige Eigenschaften:

1. Eine Lösung kann nur per Brute-Force gefunden werden.
2. Eine vorhandene Lösung kann leicht verifiziert werden.

Die höchste Wahrscheinlichkeit eine Lösung zu finden hat der Miner mit der größten Rechenleistung. Derjenige Miner, der einen Block mint erhält einen Block Reward und Transaction Fees in Form der Währung der Blockchain (der Block Reward selbst ist eine Transaktion, die in jedem Block enthalten ist). Dadurch werden die Miner angespornt weiter zu minen.

Falls ein Angreifer eine Transaktion nachträglich modifizieren will, muss er aufgrund der Struktur der Blockchain (siehe Abschnitt 2.1) alle nachfolgenden Blöcke neu minen bevor der Rest des Netzwerkes den aktuellen Block gemint hat. Abbildung 3.1 veranschaulicht dies anhand eines Beispiels. Dazu benötigt ein Angreifer die Mehrheit der Rechenleistung des Netzwerks (Stichwort 51% Attacke). Zudem werden Angreifer durch die hohen Energie- und Hardwarekosten vom Versuch abgeschreckt.

Bevor ein neuer Block vom Peer-to-Peer Netzwerk akzeptiert wird, wird geprüft ob der Hash des Blockes kleiner als das aktuelle $target$ ist. Konsens über den Zustand der Blockchain wird erreicht, indem immer der längste valide Zweig der Blockchain als gültig betrachtet wird.

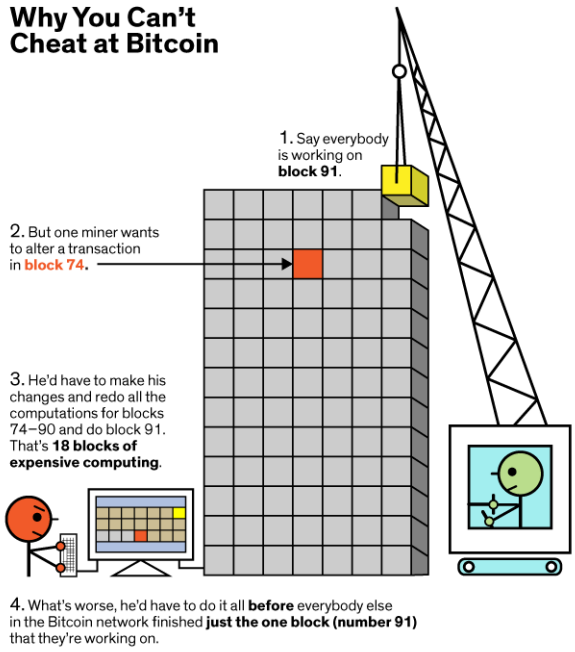

Abbildung 3.1: Veranschaulichung der Funktionsweise von PoW (Quelle: [20])

3.2 Mining-Hardware

Die Suche nach einem passenden Nonce-Wert beim PoW aus Gleichung 3.1 kann massiv parallelisiert werden. Es können beliebig viele Nonce-Werte gleichzeitig geprüft werden. Daher eignet sich häufig spezialisierte Hardware für das Minen, wobei die Effizienz abhängig von der verwendeten Hashfunktion ist. Zu der eingesetzten Hardware gehören CPUs, GPUs, FPGAs und ASICs, welche im Folgenden kurz vorgestellt werden.

Aufgrund ihrer geringen Rechenleistung lohnen sich CPUs für die meisten Währungen bzw. Hashfunktionen nicht. Eine Ausnahme ist die von der Währung Monero verwendete Cryptonight-Hashfunktion. Diese profitiert stark vom schnellen Cache-Speicher der CPUs.

GPUs sind für parallele Berechnungen ausgelegt und finden deutlich häufiger Verwendung beim Mining als CPUs. Insbesondere Hashfunktionen, welche viel Speicher und eine hohe Speicherbandbreite benötigen, lassen sich auf GPUs effizient minen. Am meisten trifft das auf Ethash zu. Diese Hashfunktion wird von Ethereum genutzt, was fast ausschließlich auf GPUs gemint wird.

In jüngerer Vergangenheit stehen für immer mehr Hashfunktionen FPGAs und ASICs zur Verfügung, welche das Minen auf CPUs und GPUs unrentabel machen. Für Bitcoin mit SHA256 etwa, ist das schon relativ lang der Fall. Ein ASIC ist speziell designte Hardware, die nur für eine Aufgabe und für nichts anderes verwendet werden kann. FPGAs sind programmierbare Logikbausteine und damit flexibler einsetzbar. Allerdings

ist die Programmierung aufwendig und sie sind nicht ganz so performant wie ASICs. Durch Anpassung der Hardware an den Algorithmus kann die Performance beim Minen mit FPGAs und ASICs im Vergleich zu CPUs und GPUs stark erhöht werden.

Aufgrund der hohen Hashleistung von FPGAs bzw. ASICs und der damit einhergehenden Zentralisierung der Rechenleistung auf wenige Miner besteht die realistische Gefahr einer 51% Attacke auf die Blockchain einer Währung, wie das bei Bitcoin Gold der Fall war (siehe [11]). Einige Währungen wie Monero versuchen durch regelmäßige Anpassungen der Hashfunktion ASIC-Resistent zu bleiben. Nachteil davon ist, dass bei jeder Änderung der Hashfunktion ein Hardfork der Blockchain nötig ist. Viele Hashfunktionen neuer Währungen wie Luxcoin (PHI2) oder Raven (X16R) wurden speziell entwickelt möglichst ASIC-Resistent zu sein. Generell ist es für speicherintensive Hashfunktionen wie beispielsweise Ethash schwieriger ASICs zu entwickeln. Ein guter Artikel zum Thema ASICs und ASIC-Resistenz beim Mining ist in [38] zu finden.

3.3 Mining-Arten

Es gibt im Wesentlichen zwei verschiedene Arten des Mining, welche in diesem Abschnitt vorgestellt werden.

3.3.1 Solo-Mining

Beim Solo-Mining versucht jeder Miner alleine neue Blöcke zu generieren, d.h. einen Hash kleiner als das aktuelle Target zu finden. Im Erfolgsfall erhält der Miner dann den kompletten Block-Reward und alle Transaction Fees. Allerdings ist die Wahrscheinlichkeit alleine einen Block zu minen sehr gering, weshalb sich das Solo-Mining im Normalfall nicht lohnt.

In Abbildung 3.2 oben ist die Funktionsweise des Solo-Mining am Beispiel von Bitcoin dargestellt. Die Mining-Software erhält eine Liste neuer Transaktionen direkt vom Peer-to-Peer Rechnernetz der Blockchain. Die Kommunikation mit dem Netzwerk erfolgt über eine spezielles Client-Programm (hier bitcoind). Aus den erhaltenen Transaktionen erstellt die Mining-Software den Prototyp eines neuen Blocks. Für dessen Header wird dann durch Permutation des Nonce-Wertes ein Hash unter dem aktuellen Target gesucht. Ist ein entsprechender Nonce-Wert gefunden, wird der neue Block an das Peer-to-Peer Rechnernetz zum Anhängen an die Blockchain gesendet. War ein anderer Miner schneller, erhält die Mining-Software neue Transaktionen vom Netzwerk, bricht die aktuelle Suche ab und generiert einen neuen Block-Prototypen.

3.3.2 Pool-Mining

Beim Pool-Mining schließt sich ein Miner mit anderen Minern in einem Pool zusammen um mehr Rechenleistung zur Verfügung zu haben. Gemeinsam finden die Miner eines Pools so häufiger einen Block. Im Erfolgsfall werden Block-Reward und Transaction Fees

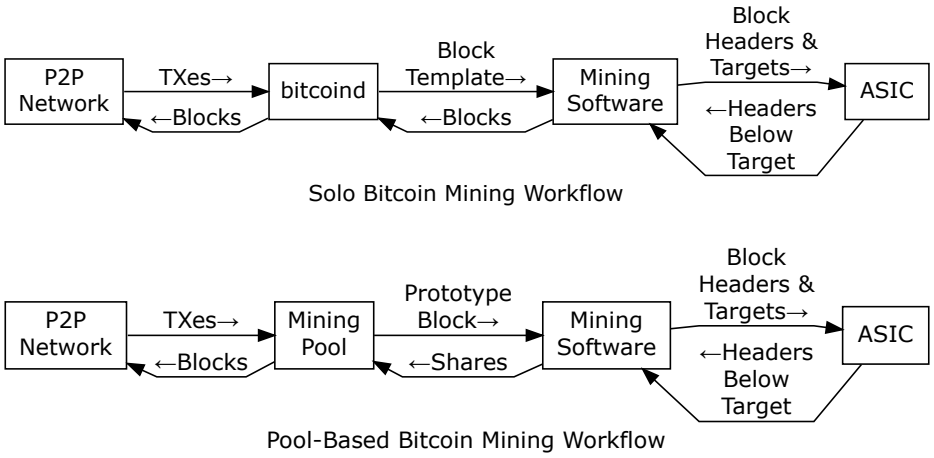

Abbildung 3.2: Veranschaulichung des Solo- und Pool-Mining (Quelle: [9])

unter den Minern eines Pools nach beigetragener Rechenleistung aufgeteilt. Auf diese Weise kann auch ein Miner mit geringer Rechenleistung regelmäßig Gewinne erzielen.

In Abbildung 3.2 unten ist der Ablauf beim Pool-Mining zu sehen. Die Mining-Software kommuniziert beim Pool-Mining nicht direkt mit dem Peer-to-Peer Rechnernetz der Blockchain. Stattdessen übernimmt diese Aufgabe der Mining-Pool, welcher neue Transaktionen vom Blockchain-Netzwerk erhält und daraus einen Block-Prototypen erstellt. Dieser Block-Prototyp wird dann an alle Miner des Pools gesendet. Der Ablauf auf Seiten der Mining-Software ist identisch zum Solo-Mining.

Allerdings ist das Target des Block-Prototypen vom Mining-Pool deutlich größer (die Schwierigkeit ist geringer) als das aktuelle Target der Blockchain. Daher generiert ein Miner deutlich häufiger einen validen Block als beim Solo-Mining, welcher dann an den Mining-Pool gesendet wird. Ein solcher Block wird auch Share genannt. In der Regel erfüllt eine Share die Schwierigkeit der Blockchain nicht, kann also nicht an die Blockchain angehängt werden. Anhand der Anzahl der eingereichten Shares kann der Mining-Pool jedoch die von den einzelnen Minern beigetragene Rechenleistung bemessen. Erhält der Pool eine Share, welche auch die Schwierigkeit der Blockchain erfüllt, sendet er diese an das Peer-to-Peer Rechnernetz zum Anhängen an die Blockchain.

Für die Aufteilung der Gewinne an die Miner des Pools gibt es verschiedene Verfahren. Die am häufigsten genutzten Verfahren sind Pay-Per-Share (PPS) und Pay-Per-Last-N-Shares (PPLNS). Beim PPS-Verfahren werden die Miner direkt nach Anzahl der eingereichten Shares bezahlt, unabhängig davon, ob der Pool einen Block findet. Beim PPLNS-Verfahren wird, immer wenn der Pool einen Block findet, ein Fenster der letzten N eingereichten Shares betrachtet. Die Auszahlung an die Miner erfolgt nach Anteil der Shares in diesem Fenster. Die Zahl N ist abhängig von der aktuellen Schwierigkeit der Blockchain. Eine Beschreibung der genauen Funktionsweise von PPLNS ist in [16] zu finden.

4 Autotuning-Framework für energieoptimales Mining auf NVIDIA GPUs

Im Rahmen dieser Arbeit ist ein Autotuning-Framework für das energieoptimale Mining auf NVIDIA GPUs in C++ implementiert worden, welches im Folgenden vorgestellt wird. Das Framework ermöglicht das energieoptimale Mining auf allen GPUs eines Rechners mithilfe von DVFS. Zudem wird sichergestellt, dass immer die beim Energieoptimum profitabelste der verfügbaren Währungen auf jeder GPU gemint wird.

4.1 Programmaufbau

Dieses Kapitel beschreibt die Implementierung des Framework, welche sich aus folgenden Komponenten mit entsprechender Funktionalität zusammensetzt:

- **nvml/nvapi:** Setzen von Core- und VRAM-Frequenz sowie die Bestimmung des aktuellen Energieverbrauchs einer GPU (siehe Abschnitt 4.2).
- **freq_optimization/freq_exhaustive:** Optimierungsalgorithmen zur Bestimmung der energieoptimalen Frequenzen (siehe Abschnitt 4.3).
- **freq_core:** Hilfsfunktionen zum Starten/Beenden von Benchmark- und Miningprozessen, zum Schreiben/Lesen von Konfigurationsdateien sowie für Netzwerkanfragen.
- **profit_optimization:** Hauptprogramm mit Frequenzoptimierung und Berechnung/Überwachung der Energiekosten und Miningerträge (siehe Abschnitt 4.4 und 4.5).

Eine Übersicht mit den Abhängigkeiten zwischen den Komponenten ist in Abbildung 7.2 zu sehen.

Im Folgenden wird auf den Ablauf des Hauptprogramms (`profit_optimization`) eingegangen, welches alle Komponenten des Framework verwendet. Abbildung 4.1 zeigt schematisch den groben Ablauf des Programms. Der genaue Ablauf und die Umsetzung der einzelnen Punkte wird in diesem und folgenden Unterkapiteln beschrieben.

4.1.1 Initialisierung

Bei Programmstart stehen folgende Kommandozeilenoptionen zur Verfügung:

```
--currency_config=<filename>        Zu nutzende Währungskonfiguration.
--user_config=<filename>            Zu nutzende Miningkonfiguration.
--opt_result=<filename>             Zu nutzendes Optimierungsresultat.
--help                              Anzeige eines Hilfetexts.
```

```
struct device_clock_info{
  int device_id_;
  bool nvml_supported_, nvapi_supported_;//nvapi == nv-control x under linux
  int default_mem_clock_, default_graph_clock_;
  std::vector<int> available_graph_clocks_;
  int min_mem_oc_, max_mem_oc_;
  int min_graph_oc_, max_graph_oc_;
};
```

Listing 4.1: device_clock_info struct

Alle Parameter sind optional. Falls keine *Währungskonfiguration* angegeben ist, wird eine Standardkonfiguration verwendet. Bei fehlender *Miningkonfiguration* wird eine über einen Nutzerdialog erstellt und bei fehlendem Optimierungsresultat kann die *Frequenzoptimierungsphase* (siehe Abschnitt 4.4) nicht übersprungen werden. Der Aufbau und Inhalt der Konfigurationsdateien ist in Abschnitt 4.6 beschrieben.

Nach dem Lesen der Kommandozeilenparameter werden zunächst die *NVML* und *NVAPI* (Windows) bzw. *NV-Control X* (Linux) Bibliotheken initialisiert. Anschließend wird für jede in der *Miningkonfiguration* angegebenen GPU geprüft, ob das setzen der Frequenzen mittels *NVML* und *NVAPI* bzw. *NV-Control X* möglich ist. Dabei wird die *NVML* nur zum Einstellen der Core-Frequenz, die *NVAPI*/*NV-Control X* zum Einstellen von beiden, Core- und VRAM-Frequenz, verwendet. Der Grund liegt darin, dass die verschiedenen APIs unterschiedliche Frequenzbereiche unterstützen (siehe Abschnitt 4.2). Ist das Setzen der Frequenzen möglich, wird der einstellbare Frequenzbereich abgefragt bzw., falls angegeben, ein benutzerdefinierter aus der *Miningkonfiguration* übernommen. Der einstellbare Frequenzbereich wird zusammen mit den Default-Frequenzen in einer Struktur vom Typ `device_clock_info` (siehe Listing 4.1) gespeichert. In dieser Struktur wird auch vermerkt, falls das Setzen der Frequenzen mit einer der APIs nicht möglich ist. In diesem Fall ist das Optimieren der Frequenzen auf dieser GPU gar nicht oder nur eingeschränkt möglich.

4.1.2 Hauptprogramm

Nachdem für alle GPUs eine `device_clock_info` erstellt wurde, werden die GPUs für die *Frequenzoptimierungsphase* in Gruppen eingeteilt. Dabei werden GPUs mit gleichem Namen derselben Gruppe zugeordnet. Jede dieser GPU-Gruppen muss die Frequenzen für alle verfügbaren Währungen optimieren. Falls bei Programmstart ein Optimierungsresultat angegeben wurde, werden alle Werte für Währungen, die es für die GPUs einer Gruppe enthält übernommen. Dadurch verringert sich die Anzahl der zu optimierenden Währungen für diese Gruppe. Enthält das Optimierungsresultat für eine Gruppe Werte für alle Währungen, wird die komplette *Frequenzoptimierungsphase* für diese Gruppe übersprungen. Innerhalb einer Gruppe werden die Währungen dann auf die einzelnen GPUs der Gruppe gleichmäßig aufgeteilt (siehe Listing 4.2).

```
map<int, set<currency>> get_currencies_to_optimize(list<gpu_group> all_groups,
  set<currency> all_currencies, map<int, map<currency, measurement>> opt_results)
{
  map<int, set<currency>> res;
  for (gpu_group group : all_groups){
    //index of gpu within group to add next currency
    int next_insert_idx = 0;
    for (currency ct : all_currencies) {
      for (int i = next_insert_idx; i < group.members_.size(); i++) {
        int gpu = group.members_.at(i);
        //skip optimization if entry in opt_result for gpu and currency is available
        if (opt_results.contains(gpu) && opt_results.at(gpu).contains(ct))
          continue;
        //add currency to gpu for optimization
        res.at(gpu).insert(ct);
        next_insert_idx = (i + 1) % group.members_.size();
        break;
      }
    }
  }
  return res;
}
```

Listing 4.2: Aufteilung der zu optimierenden Währungen auf die GPUs

Der Grund für die Aufteilung der GPUs in Gruppen ist, dass bei gleichen GPUs die optimalen Frequenzen für einzelne Währungen identisch sind. Somit kann das Optimieren der Währungen innerhalb einer Gruppe parallel erfolgen, was die *Frequenzoptimierungsphase* beschleunigt. Anschließend wird für jede GPU ein Thread gestartet. Diese Threads laufen bis zum Ende des Programms, d.h. die komplette *Frequenzoptimierungsphase* und *Profitüberwachungsphase* erfolgen separat für jede GPU innerhalb eines Threads. Der genaue Ablauf von *Frequenzoptimierungsphase* und *Profitüberwachungsphase* wird in Abschnitt 4.4 bzw. 4.5 näher beschrieben.

Am Ende der *Frequenzoptimierungsphase* werden die Threads einer Gruppe synchronisiert und die Optimierungsergebnisse ausgetauscht. Die gruppenbasierte Synchronisation ist mittels Condition-Variablen implementiert. Ist ein Thread mit der *Frequenzoptimierungsphase* fertig, setzt dieser unter Mutex-Schutz eine Boolean-Flag und prüft ob die Flags der anderen Threads der Gruppe gesetzt sind. Falls ja, ist dieser Thread der letzte der Gruppe und benachrichtigt die anderen per Condition-Variable. Falls nicht, wartet der Thread auf der Condition-Variable (siehe Listing 4.3). Das Austauschen der Optimierungsergebnisse erfolgt über eine geteilte Datenstruktur, welche ebenfalls per Mutex geschützt wird. Im Anschluss sind für alle GPUs einer Gruppe optimale Frequenzen für alle Währungen vorhanden, und die *Profitüberwachungsphase* wird für diese Gruppe gestartet. Während der *Profitüberwachungsphase* ist keine Synchronisation zwischen den Threads notwendig. Die Threads laufen in der *Profitüberwachungsphase* in einer Endlosschleife.

Währenddessen blockiert der Mainthread und wartet auf Beendigung des Programms per Nutzereingabe. Eine weitere Möglichkeit das Programm normal zu beenden, ist per

```
map<int, map<currency, measurement>> main_prog(list<gpu_group> all_groups,
  set<currency> all_currencies, map<int, map<currency, measurement>> opt_results)
{
  //assign optimization work to each gpu
  map<int, set<currency_type>> currencies_to_optimize =
    get_currencies_to_optimize(all_groups, all_currencies, opt_results);
  //parallel region
  parallel foreach gpu g{
    if(!currencies_to_optimize.at(g).empty()){
      //optimize frequencies for assigned currencies of gpu
      map<currency, measurement> optimized_currencies =
          frequency_optimization(currencies_to_optimize.at(g));
      //synchronize threads that share the same group
      gpu_group group = all_groups.find_group_of(g)
      group.mutex.lock()
      group.set_member_completed(g);
      if(group.has_completed())
        group.cond_var.notify_all();
      else
        group.cond_var.wait(group.mutex);
      //add result to other gpus of group
      for(gpu g2 : group.members){
        opt_results.at(g2).add(optimized_currencies)
      }
      group.mutex.unlock()
    }
    //start profit monitoring using optimization results; returns updated values
    opt_results.at(g) = profit_monitoring(opt_results.at(g));
  }
  //return updated optimization results
  return opt_results;
}
```

Listing 4.3: Hauptprogramm mit Synchronisation der GPU-Gruppen

SIGTERM-Signal, für welches ein entsprechender Signalhandler implementiert ist. Das ist insbesondere nützlich, falls das Programm beispielsweise per *nohup* gestartet wird und keine Standardeingabe zur Verfügung steht. Sobald das Signal zum Beenden des eintrifft, wird eine von allen Threads geteilte atomare Boolean-Flag gesetzt, welche die Endlosschleife in der *Profitüberwachungsphase* beendet. Ist ein Thread in der Wartephase zwischen zwei Iterationen der Endlosschleife, wird dieser über eine ebenfalls von allen Threads geteilte Condition-Variable notifiziert. Auf diese Weise werden die Threads ohne lange Wartephase sauber beendet. Der Mainthread sammelt alle Threads auf und speichert anschließend das während der *Profitüberwachungsphase* aktualisierte Optimierungsresultat. Abschließend werden noch die GPU-Frequenzen zurückgesetzt und die *NVML* und *NVAPI/NV-Control X* Bibliotheken deinitialisiert.

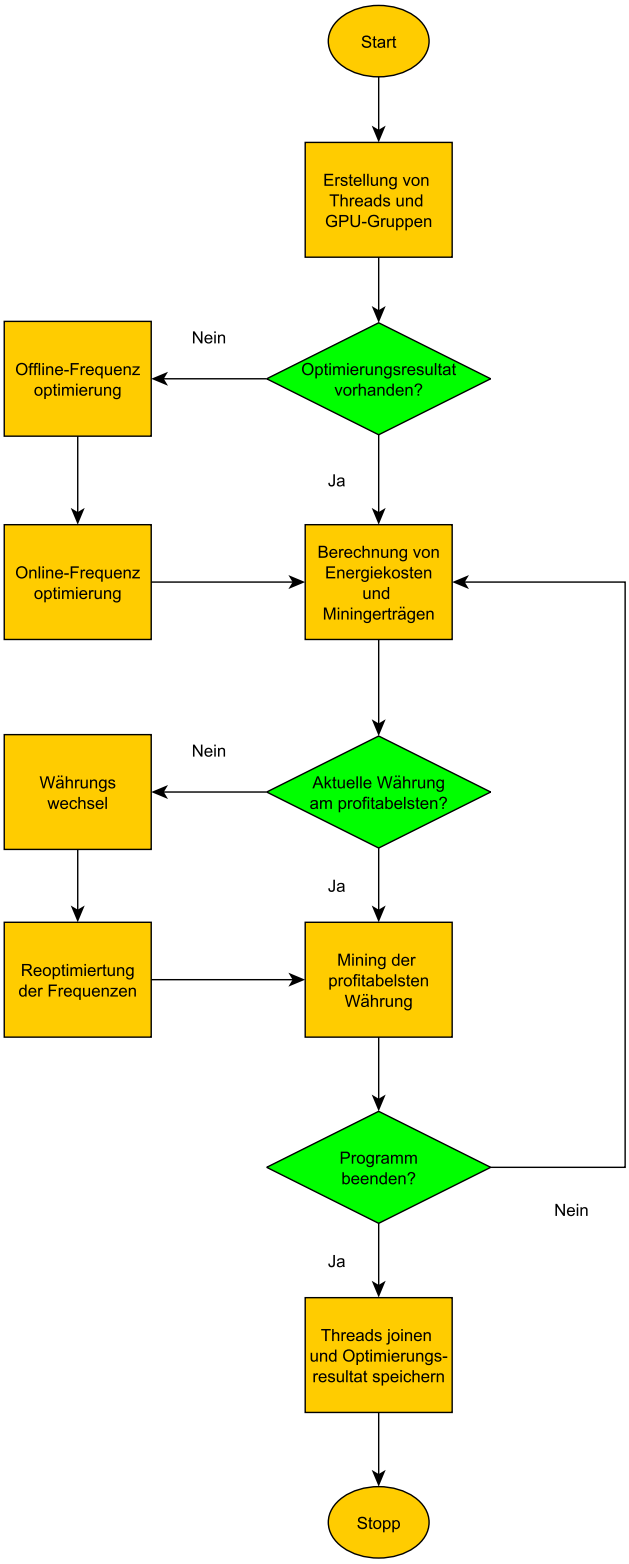

Abbildung 4.1: Schematischer Ablauf des in der Arbeit erstellten Programms

4.2 Frequenzanpassung/Energiemessung bei NVIDIA GPUs

In diesem Abschnitt werden die in der Arbeit verwendeten Bibliotheken zum Ändern der Frequenzen und zum Messen des Energieverbrauchs von NVIDIA-GPUs vorgestellt. Zudem wird deren Verwendung im Framework erläutert.

4.2.1 NVML (Windows und Linux)

Die *NVIDIA Management Library (NVML)* steht sowohl auf Windows- als auch Linux-Systemen zur Verfügung, und wird mit dem NVIDIA-Treiber mitgeliefert (siehe [27]). Sie ermöglicht die Abfrage und Änderung verschiedener GPU-Zustände, z.B.:

- Aktuelle GPU-Auslastung mit Speicherverbrauch
- Aktive GPU-Prozesse mit PID und Speicherbedarf
- GPU-Temperatur und Geschwindigkeit des Lüfters
- Aktuelle und maximale Frequenzen verschiedener Clocks
- Aktueller und maximaler Energieverbrauch

Eine genaue Dokumentation der Funktionen ist in [30] zu finden. Außerdem kann über das Kommandozeilenprogramm *nvidia-smi* (siehe [28]) die Funktionalität der *NVML* direkt verwendet werden.

Im Framework wird die *NVML* zum Setzen der Core-Frequenz und zum Messen des aktuellen Energieverbrauchs verwendet. Das geschieht mit den Funktionen *nvmlDeviceSetApplicationsClocks()* bzw. *nvmlDeviceGetPowerUsage()*. Die einstellbare Frequenzbereich ist hierbei nicht kontinuierlich. Stattdessen wird mit der Funktion *nvmlDeviceGetSupportedGraphicsClocks()* eine Liste aller validen Frequenzen ermittelt. Bei der Verwendung der *NVML* haben sich während der Entwicklung des Framework folgende Probleme ergeben:

- Das Setzen der VRAM-Frequenz ist laut Dokumentation möglich, hat aber bei keiner getesteten GPU funktioniert.
- Das Setzen der Core-Frequenz ist nur bei GPUs mit Architektur $\geq$ Pascal möglich, und auch dann nur bei GPUs der Geforce-TITAN-, Quadro- oder Tesla-Reihe.
- Bei GPUs mit Architektur $<$ Pascal funktioniert Setzen der Core-Frequenz überhaupt nicht. Zudem gibt die Bibliothek bei diesen GPUs keine Fehlermeldung, sondern fälschlicherweise eine Erfolgsmeldung zurück.

Daher wird die *NVML* im Framework nur zum Anpassen der Core-Frequenz verwendet. Ob dies unterstützt wird, kann zur Laufzeit nur für GPUs mit Architektur $\geq$ Pascal geprüft werden. Für alle anderen GPUs wird das Setzen der Frequenzen mithilfe der *NVML* komplett deaktiviert.

4.2.2 NVAPI (Windows)

Die *NVAPI* ist nur auf Windows-Systemen verfügbar und erlaubt vollen Zugriff auf die Features des NVIDIA-Treiber (siehe [25]). Die öffentlich verfügbare Version der *NVAPI* ist von der Funktionalität jedoch stark eingeschränkt und ermöglicht nur das Abfragen, aber nicht das Setzen von Werten. Eine Dokumentation der öffentlich verfügbaren Version ist in [26] zu finden.

Für die volle Funktionalität wird die NDA-Edition der *NVAPI* benötigt, welche in dieser Arbeit jedoch nicht zur Verfügung stand. Um dennoch das Ändern der Frequenzen zu ermöglichen, wird der Code aus [32] verwendet. In diesem Artikel wird der zum Ändern der Frequenzen benötigte Code aus dem GPU-Übertaktungstool *MSI Afterburner* durch Reverse Engineering extrahiert. Dieses Tool verwendet dafür die in der öffentlichen Version der *NVAPI* nicht verfügbare Funktion *NvAPI_GPU_SetPstates20()*. Der einstellbare Frequenzbereich wird mithilfe der Funktion *NvAPI_GPU_GetPstates20()* ermittelt. Damit wird im Framework unter Windows insbesondere das Anpassen der VRAM-Frequenz ermöglicht, was mit der *NVML* nicht funktioniert (siehe Abschnitt 4.2.1).

4.2.3 NV-Control X (Linux)

Die *NV-Control X* API ist nur auf Linux-Systemen verfügbar und ermöglicht X-Clients die Einstellung und Abfrage verschiedener Attribute des NVIDIA X Treibers (siehe [24]). Voraussetzung für die Verwendung ist allerdings eine laufende X-Session, d.h. der Nutzer muss den X-Server auf dem Zielrechner starten. Die Funktionalität der *NV-Control X* API kann auch direkt über das Programm *nvidia-settings* verwendet werden. Zum Anpassen der GPU-Frequenzen muss in der X-Konfiguration (/etc/X11/xorg.conf) die Coolbits-Option für die jeweilige GPU gesetzt sein (siehe [36]).

Die Attribute NV_CTRL_GPU_MEM_TRANSFER_RATE_OFFSET bzw. NV_CTRL_GPU_NVCLOCK_OFFSET ermöglichen die Steuerung der Frequenzen. Mit der Funktion *XNVCTRLQueryValidTargetAttributeValues()* wird der einstellbare Wertebereich abgefragt und mit der Funktion *XNVCTRLSetTargetAttribute()* wird ein Wert gesetzt. Die *NV-Control X* ist im Framework unter Linux, ähnlich wie die *NVAPI* unter Windows, für das Anpassen der VRAM-Frequenz notwendig.

4.2.4 Umsetzung der Energiemessung

Die Energiemessung ist im Framework als eigenständiges Programm umgesetzt, das vom Hauptprogramm für jede GPU einzeln als Hintergrundprozess gestartet wird. Im Energiemessungs-Programm wird in einer Endlosschleife periodisch der aktuelle Energieverbrauch auf der spezifizierten GPU mithilfe der *NVML* (siehe Abschnitt 4.2.1) ermittelt und zusammen mit der aktuellen Systemzeit in eine Datei geschrieben. Im Hauptprogramm wird der Energieverbrauch für einen angegebenen Zeitraum ermittelt, indem die Werte aus der Datei für den Zeitraum ausgelesen und der Mittelwert gebildet wird.

```
void changeGPUClocks(device_clock_info dci, int mem_oc, int graph_clock_idx)
{
  //graph_clock == core_clock
  //nvapi == nv-control x under linux
  if (dci.nvapi_supported_) {
    int graph_clock = dci.available_graph_clocks_[graph_clock_idx];
    //dci.default_graph_clock_ equals maximum through nvml configurable core-freq
    if (!dci.nvml_supported_ || graph_clock > dci.default_graph_clock_) {
      int graph_oc = graph_clock - dci.default_graph_clock_;
      //set core and vram-freqs through nvapi/nv-control x
      nvapiOC(dci.device_id_, graph_oc, mem_oc);
    } else {
      //set vram-freq through nvapi/nv-control x
      nvapiOC(dci.device_id_, 0, mem_oc);
      //set core-freq through nvml (resetting of vram-freq has no effect here)
      nvmlOC(dci.device_id_nvml_, graph_clock, 0);
    }
  } else if (dci.nvml_supported_) {
    int graph_clock = dci.available_graph_clocks_[graph_clock_idx];
    //set core-freq through nvml
    nvmlOC(dci.device_id_nvml_, graph_clock, 0);
  }
}
```

Listing 4.4: Funktion zum Ändern der Frequenzen

4.2.5 Umsetzung der Frequenzanpassung

Die Frequenzanpassung erfolgt im Framework mithilfe der Funktion *changeGPUClocks()* (siehe Listing 4.4). Diese verwendet die *NVML* und, je nach Betriebssystem, die *NVAPI* (Windows) bzw. die *NV-Control X* (Linux). Als Parameter bekommt die Funktion eine `device_clock_info` (siehe Listing 4.1), die einzustellende VRAM-Frequenz als Differenz zur Standardfrequenz sowie die Core-Frequenz als Index eines Vektors verfügbarer Frequenzen. Für das Setzen der Frequenzen gibt es drei Fälle, je nachdem welche APIs auf der jeweiligen GPU unterstützt werden:

- **NVML und NVAPI/NV-Control X:** VRAM- und Core-Frequenz können in vollem Wertebereich gesetzt werden.
- **Nur NVML:** VRAM-Frequenz kann gar nicht und Core-Frequenz in leicht eingeschränktem Wertebereich gesetzt werden.
- **Nur NVAPI/NV-Control X:** VRAM-Frequenz kann in vollem Wertebereich und Core-Frequenz in stark eingeschränktem Wertebereich gesetzt werden.

4.3 Optimierungsverfahren

Für das Finden der energieoptimalen Frequenzen sind drei verschiedene Optimierungsverfahren implementiert worden, welche im Folgenden vorgestellt werden. Der Nutzer

Algorithm 1: Hill Climbing

Input: Objective function f, $startNode$, $stepSize$, $iterations$
Output: Values that maximize f

```
currentNode ← startNode;
bestEval ← f(currentNode);
bestNode ← currentNode;
for i ← 1 to iterations do
    L ← Neighbors(currentNode, stepSize);
    tempBestEval ← −INF;
    lastNode ← currentNode;
    foreach Node n in L do
        if f(n) > tempBestEval then
            currentNode ← n;
            tempBestEval ← f(n);
    bestDirection ← currentNode − lastNode;
    while f''(currentNode) > 0 do
        currentNode ← currentNode + bestDirection;
        tempBestEval ← f(currentNode);
    if tempBestEval > bestEval then
        bestNode ← currentNode;
        bestEval ← tempBestEval;
return bestNode;
```

kann in der *Miningkonfiguration* für jede Währung das zu verwendende Verfahren und dessen Parameter einstellen.

4.3.1 Zielfunktion

Die zu optimierende Funktion $f : [a, b] \times [c, d] \longmapsto \mathbb{R}_0^+$ mit $a, b, c, d \in \mathbb{N}$ bildet den auf einer GPU einstellbaren Frequenzbereich auf die Anzahl der Hashes pro Joule ab, die beim Minen einer Währung erzielt werden:

$$f(vram\_freq, core\_freq) = \frac{hashrate(vram\_freq, core\_freq)\ [\frac{\text{Hashes}}{\text{s}}]}{energy\_consumption(vram\_freq, core\_freq)\ [\frac{\text{Joule}}{\text{s}}]} \quad (4.1)$$

Ziel der Optimierungsverfahren ist es, diesen Wert im einstellbaren Frequenzbereich zu maximieren. Der einstellbare Wertebereich wird zu Programmstart für jede GPU ermittelt und gespeichert (siehe Abschnitt 4.1.1). Er ist von den auf der GPU unterstützten APIs zur Frequenzanpassung (siehe Abschnitt 4.2) abhängig. Je nach unterstützten APIs

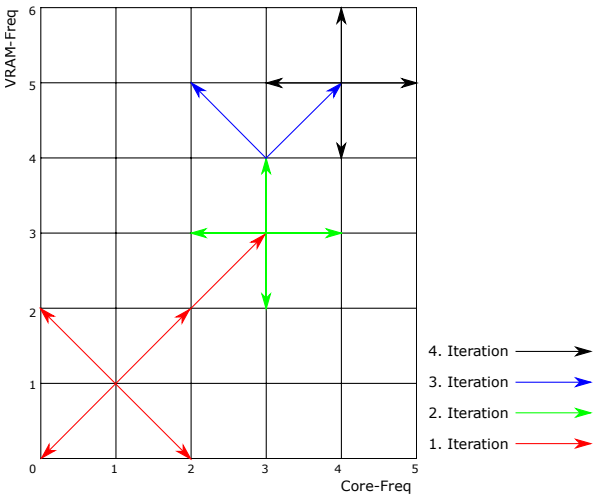

Abbildung 4.2: Veranschaulichung des Hill Climbing

ist die Zielfunktion aus Gleichung 4.1 nulldimensional (keine Optimierung möglich), eindimensional oder zweidimensional.

4.3.2 Hill Climbing

Das Hill Climbing ist ein einfaches, heuristisches Optimierungsverfahren, das die Zielfunktion Schritt für Schritt verbessert. Dabei wird jeweils eine lokale Veränderung durchgeführt und nur übernommen, wenn der entstandene Lösungskandidat besser geeignet ist. Das Verfahren endet, wenn vom aktuellen Punkt aus keine Verbesserung mehr möglich ist oder die maximale Anzahl an Iterationen erreicht ist. Problem des Hill Climbing ist, dass es in lokalen Optima hängenbleibt.

In der Implementierung im Framework wird für die lokale Veränderung in jeder Dimension eine Schrittweite spezifiziert. Vom aktuellen Punkt aus werden dann, in jeder Iteration abwechselnd, die vier diagonalen bzw. horizontalen Nachbarn betrachtet. In der Richtung des besten Nachbarpunktes wird dann solange weitergesucht, wie die zweite Ableitung (Steigung der Steigung) positiv ist. Erst dann beginnt die nächste Iteration (siehe Algorithmus 1). Dadurch wird schneller ein Maximum gefunden. Eine Veranschaulichung des Algorithmus ist in Abbildung 4.2 zu sehen.

4.3.3 Simulated Annealing

Simulated Annealing ist ein Optimierungsverfahren ähnlich dem Hill Climbing. Im Gegensatz zum Hill Climbing kann das Verfahren jedoch ein lokales Optimum wieder verlassen und ein besseres finden, indem ungünstigere Zwischenlösungen mit einer gewissen Wahrscheinlichkeit akzeptiert werden. Dadurch ist es unwahrscheinlicher in einem lokalem Optimum hängenzubleiben.

Algorithm 2: Simulated Annealing

Input: Objective function f, $startNode$, $stepSize$, $iterations$
Output: Values that maximize f

```
1  currentNode ← startNode;
2  bestEval ← f(currentNode);
3  bestNode ← currentNode;
4  for k ← 1 to iterations do
       // Find neighbor with one iteration of hill climbing
5      newNode ← HillClimbing(f, currentNode, stepSize, 1);
6      if f(newNode) ≥ f(currentNode) or
         exp(−(f(currentNode)−f(newNode))/T(k)) > Rand(0,1)) then
7          currentNode ← newNode;
8      if f(currentNode) > bestEval then
9          bestNode ← currentNode;
10         bestEval ← f(currentNode);
11 return bestNode;
```

Die Wahrscheinlichkeit ungünstigere Zwischenlösungen (hier a) zu akzeptieren ist:

$$P(accept\ a\ as\ solution) = exp\left(-\frac{f(b) - f(a)}{T(k)}\right),\ f(b) > f(a) \tag{4.2}$$

Dabei ist $T(k)$ die Temperatur in Iteration k, welche mit jeder Iteration des Algorithmus sinkt und a bzw. b sind der neue bzw. alte Lösungskandidat. Je größer die Temperatur ist, desto größer ist die Wahrscheinlichkeit schlechtere Zwischenlösungen zu akzeptieren. Die Starttemperatur sollte größer als die Differenz von Maximum und Minimum von f sein:

$$T(1) > max_x(f(x)) - min_x(f(x)), \quad \forall k \in \mathbb{N} : T(k+1) < T(k) \tag{4.3}$$

Der Rest des Algorithmus entspricht dem Hill Climbing, weswegen sich das Verfahren mit abnehmendem T immer mehr wie dieses verhält. Die Exploration neuer Lösungskandidaten erfolgt im Framework genauso wie beim Hill Climbing (siehe Algorithmus 2).

4.3.4 Nelder Mead

Das Nelder-Mead Verfahren ist ein Optimierungsverfahren, dessen Idee darin besteht, eine Folge von Simplexen zu erzeugen, die immer kleiner werdenden Durchmesser aufweisen und sich schließlich um das gesuchte Optimum herumscharen. Aus diesem Grund wird das Nelder–Mead Verfahren oft auch als Simplex–Verfahren bezeichnet. Ein Simplex

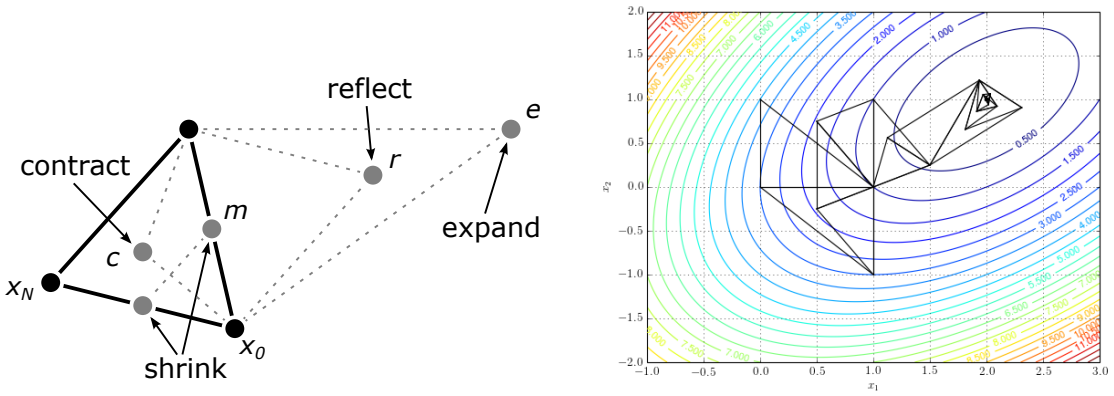

Abbildung 4.3: Änderung des Simplex in einer Nelder-Mead Iteration (links) und Iterationsverlauf bei der Optimierung einer 2D-Funktion (rechts) (Quellen: [6] und [17])

ist dabei das einfachste Volumen im N-Dimensionalen Raum, das aus $N+1$ Punkten aufgespannt wird. In 1D ist das eine Linie, in 2D ein Dreieck und in 3D ein Tetraeder.

Das Verfahren startet mit einem initialen Simplex $x_0, x_1, \ldots x_N$ sowie den Parametern $\alpha > 0$, $\gamma > 1$, $0 < \beta < 1$ und $0 < \sigma < 1$. In jeder Iteration werden die $N+1$ Punkte des Simplex nach Funktionswert $f(x_0) \geq f(x_1) \geq \cdots \geq f(x_N)$ geordnet. Dann wird der Mittelpunkt $m = \frac{1}{N}\sum_{i=0}^{N-1} x_i$ der N besten Punkte berechnet und der Punkt $r = (1+\alpha)m - \alpha x_N$ bestimmt. Anschließend wird der schlechteste Punkt x_N entweder durch

1. den besseren der Punkte $e = (1+\gamma)m - \gamma x_N$ und r ersetzt, falls $f(r) > f(x_0)$.
2. r ersetzt, falls $f(r) > f(x_{N-1})$.
3. $c = \beta m + (1-\beta)h$ ersetzt, falls $f(c) > f(x_N)$. Dabei ist h der bessere der Punkte x_N und r.

Die Punkte r, e und c heißen reflektierter, expandierter und kontrahierter Punkt, da diese jeweils durch Reflexion/Expansion/Kontraktion von x_N am Mittelpunkt m entstehen. Falls keiner der Fälle zutrifft wird der Simplex in Richtung des besten Punktes x_0 komprimiert: $\forall\, i \in [1, N] : x_i = \sigma x_0 + (1-\sigma)x_i$ (siehe Abbildung 4.3 links). Auf diese Weise wird der Simplex in jeder Iteration in Richtung des Optimums verlagert (siehe Abbildung 4.3 rechts). Der Algorithmus terminiert, wenn die maximale Anzahl an Iterationen erreicht, der Simplex eine bestimmte Größe unterschreitet ($max_{0 \leq i < j \leq N}(||x_i - x_j||_2) < \epsilon$) oder die Funktionswerte des Simplex nah genug aneinander liegen ($f(x_0) - f(x_N) < \epsilon$).

Im Framework wurde das Nelder-Mead Verfahren im Gegensatz zum Hill Climbing und Simulated Annealing nicht selbst implementiert. Stattdessen wird die Implementierung in [2] verwendet, welche die Eigen-Bibliothek nutzt.

4.4 Frequenzoptimierungsphase

Im Folgenden Abschnitt wird die Umsetzung der *Frequenzoptimierungsphase* beschrieben. Die *Frequenzoptimierungsphase* läuft für jede GPU separat im einem Thread und ermittelt die energieoptimalen Frequenzen für jede der GPU zugewiesenen Währungen (siehe Abschnitt 4.1.2). Das erfolgt mithilfe der in Abschnitt 4.3 vorgestellten Optimierungsverfahren.

Die *Frequenzoptimierungsphase* selbst ist in eine Offline- und Onlinephase unterteilt. Offline- und Onlinephase unterscheiden sich durch die Auswertung der Zielfunktion aus Gleichung 4.1, d.h. durch die Ermittlung der Hashrate und des Energieverbrauchs bei gegebenen Frequenzen.

Die *Frequenzoptimierungsphase* kann für jede Währung mit folgenden Parametern konfiguriert werden:

- Zu verwendendes Optimierungsverfahren.
- Minimale Hashrate die nicht unterschritten werden darf als Nebenbedingung der Optimierung. Angabe ist relativ zur maximal erzielbaren Hashrate.
- Schrittweite des Optimierungsverfahrens relativ zum verfügbaren Frequenzbereich.
- Maximale Anzahl der Iterationen des Optimierungsverfahrens.
- Dauer einer Messung während der Onlinephase.

4.4.1 Offlinephase

In der Offlinephase werden die Frequenzen auf Basis kurzer Offline-Benchmarks optimiert. Offline bedeutet hier, dass keine Verbindung zum Mining-Pool aufgebaut wird und somit keine Netzwerkkommunikation stattfindet. Die Performance (Anzahl der Hashes pro Sekunde) des Hashalgorithmus wird dadurch unter Idealbedingungen bestimmt, da Netzwerkstörungen keinen Flaschenhals darstellen.

Bei jeder Auswertung der Zielfunktion aus Gleichung 4.1 wird der Miner für die zu optimierende Währung im Benchmark-Modus gestartet. Die erzielte Hashrate wird zusammen mit dem maximalen Energieverbrauch, der während des Benchmarks gemessen wurde in einer Struktur vom Typ `measurement` gespeichert (siehe Listing 4.5). In dieser Struktur werden außerdem Informationen zu den für die Messungen verwendeten Frequenzen, sowie die Dauer des Benchmarks gespeichert. Ein `measurement` entspricht dem Ergebnis einer Auswertung der Zielfunktion.

Die Dauer der Offlinephase ist von der Dauer einer Messung und dem verwendeten Optimierungsverfahren abhängig. Bei den getesteten Währungen beträgt die Messdauer zwischen zwei und 30 Sekunden. Die Optimierungsverfahren benötigen, je nach Startpunkt, typischerweise zehn bis 15 Funktionsauswertungen.

```
struct measurement{
  int mem_clock_, graph_clock_;
  double power_, hashrate_, energy_hash_;
  int nvml_graph_clock_idx_;
  int mem_oc_, graph_oc_;
  int power_measure_dur_ms_, hashrate_measure_dur_ms_;
};
```

Listing 4.5: measurement struct

4.4.2 Onlinephase

In der Onlinephase werden die Frequenzen unter realen Bedingungen optimiert. Dazu wird zu Beginn der Onlinephase das normale Minen mit Mining-Pool als Hintergrundprozess gestartet. Dieser Hintergrundprozess schreibt die momentan erzielte Hashrate zusammen mit der Systemzeit kontinuierlich in eine Logdatei.

Für die Auswertung der Zielfunktion werden zunächst die gewünschten Frequenzen gesetzt und die aktuelle Systemzeit gespeichert. Dann wird eine einstellbare Zeit gewartet (typischerweise zwei bis drei Minuten) und die mittlere erzielte Hashrate während dieser Zeit durch Auslesen der Logdatei bestimmt. Zudem wird der mittlere Energieverbrauch während dieser Zeit wie in Abschnitt 4.2.4 beschrieben bestimmt. Hashrate, Energieverbrauch, Frequenzinformation und Messdauer werden wie in der Offlinephase in einer `measurement` Struktur gespeichert.

Als Startpunkt wird in der Onlinephase das Ergebnis der Offlinephase verwendet. Die Erstellung eines Messpunktes (= Funktionsauswertung) dauert während der Onlinephase deutlich länger als in der Offlinephase. Dafür werden weniger Funktionsauswertungen benötigt, da der Startpunkt i.d.R. bereits nahe am Optimum ist. Zudem werden während der Onlinephase bereits Miningerträge erzielt, da das normale Minen im Hintergrund läuft.

4.5 Profitüberwachungsphase

Dieser Abschnitt erläutert die Umsetzung der *Profitüberwachungsphase*. Diese läuft, genau wie die *Frequenzoptimierungsphase*, für jede GPU in einem separaten Thread. Sie wird gestartet, sobald die optimalen Frequenzen für alle Währungen auf der entsprechenden GPU zur Verfügung stehen. Das ist der Fall, wenn die *Frequenzoptimierungsphase* für alle GPUs der Gruppe beendet ist (siehe Abschnitt 4.1.2).

Die *Profitüberwachungsphase* läuft in einer Endlosschleife, in der periodisch Folgendes ausgeführt wird (siehe Listing 4.6):

1. Mining der profitabelsten Währung im Hintergrund. Warten für die Dauer einer Periode.

2. Update von Hashrate und Energieverbrauch für die aktuell geminte Währung.

```
map<currency, measurement> profit_monitoring(map<currency, measurement> opt_results,
device_clock_info dci, int monitoring_period)
{
  //calc best currency based on results of frequency optimization phase; start mining
  currency best_currency = profit_calculation(opt_results);
  change_frequencies(dci, opt_results.at(best_currency));
  start_mining(dci, best_currency);
  //monitoring loop
  while(!terminate){
    //wait; mining is performed in background process
    sleep(monitoring_period);
    //update hashrate and energy_consumption
    opt_results.at(best_currency).hashrate_local = average_local_hashrate(dci);
    opt_results.at(best_currency).hashrate_pool = average_pool_hashrate(dci);
    opt_results.at(best_currency).power = average_energy_consumption(dci);
    //recalculate best currency
    currency new_best_currency = profit_calculation(opt_results);
    if(new_best_currency != best_currency){
      stop_mining(dci);
      best_currency = new_best_currency;
      //reoptimize frequencies using best known frequencies as start point
      start_mining(dci, best_currency);
      opt_results.at(best_currency) = frequency_optimization_online(
        dci, opt_results.at(best_currency));
      change_frequencies(dci, opt_results.at(best_currency));
    }
  }
  stop_mining(dci);
  return opt_results;
}
```

Listing 4.6: Ablauf der Profitüberwachungsphase

3. Berechnung von Energiekosten und Miningerträgen bei optimalen Frequenzen auf Basis aktueller Werte für alle Währungen.

4. Bei Bedarf, Wechsel der aktuell geminten Währung und anschließende Reoptimierung der Frequenzen.

Vor Eintritt in die Endlosschleife wird Schritt 3 mit den Werten aus der gerade beendeten *Frequenzoptimierungsphase* ausgeführt. Die Periodendauer und die Energiekosten pro kWh können in der *Miningkonfiguration* eingestellt werden. Auf diese Weise wird in der *Profitüberwachungsphase* dafür gesorgt, dass immer die am Energieoptimum profitabelste der verfügbaren Währungen gemint wird.

4.5.1 Update von Hashrate und Energieverbrauch

In diesem Schritt wird die Hashrate und der Energieverbrauch der aktuell geminten Währung aktualisiert. Der aktuelle Energieverbrauch wird wie in Abschnitt 4.2.4 erläutert bestimmt. Für die Hashrate werden zwei Werte ermittelt:

1. **Lokale Hashrate:** Lokale, vom Miner ausgegebene Hashrate.

2. **Pool-Hashrate:** Hashrate auf Basis der beim Mining-Pool eingereichten Shares.

Die Pool-Hashrate ist nur verfügbar, falls in der *Währungskonfiguration* ein REST-API Endpunkt vom Mining-Pool dafür angegeben ist. Da man beim Pool-Mining anhand der eingereichten Shares ausgezahlt wird, ist dieser Wert für die Berechnung der Miningerträge besser geeignet als die lokale Hashrate. Die lokale Hashrate ist dafür stabiler und unterliegt weniger Schwankungen.

Nachfolgend wird erklärt wie die Pool-Hashrate mithilfe der Anzahl der eingereichten Shares bestimmt werden kann (siehe [8]). Die durchschnittliche Anzahl an Hashes, die für das Minen einer Share benötigt werden ist:

$$\#hashes\ per\ share = \frac{2^L}{cur\_target}, \quad cur\_target = \frac{max\_target}{pool\_diff} \tag{4.4}$$

Dabei ist max_target der Hash, dessen Wert unterschritten werden muss um eine Share bei Schwierigkeit 1 zu minen, $pool\_diff$ ist die aktuelle Schwierigkeit des Mining-Pool und L ist Anzahl der Bits eines Hashes, typischerweise $L = 256$.

Damit ist die durchschnittliche Hashrate bei gegebener Anzahl an geminten Shares während eines Zeitraumes:

$$hashrate = \frac{\#shares\ mined \cdot \#hashes\ per\ share}{mining\_duration\ [s]} \tag{4.5}$$

Normalerweise erfolgt diese Berechnung auf Seiten der REST-API des Mining-Pool und es kann direkt die erzielte Hashrate abgefragt werden. Dies wird vom Framework auch erwartet. Beim Solo-Mining ist die Berechnung identisch, statt Shares werden hier direkt Blöcke der Blockchain gemint.

4.5.2 Profitberechnung

Dieser Schritt ist für die Berechnung von Energiekosten und Miningerträgen zuständig. Die Energiekosten pro Sekunde berechnen sich folgendermaßen:

$$energy\_cost = energy\_consumption\ [W] \cdot \frac{energy\_cost\ [\frac{euro}{kWh}]}{1000 \cdot 3600} \tag{4.6}$$

Für die Berechnung der Miningerträge pro Sekunde wird folgende Formel verwendet (siehe [15]):

$$mining\_reward = \frac{user\_hashrate}{net\_hashrate} \cdot \frac{1}{block\_time} \cdot block\_reward \cdot stock\_price \tag{4.7}$$

Aus Energiekosten und Miningerträgen ergibt sich dann der Profit:

$$mining\_profit = mining\_reward - energy\_cost \tag{4.8}$$

Hierbei ist *user_hashrate* die vom Miner erzielte Hashrate und *net_hashrate* ist die insgesamte Hashrate aller Miner der Währung. Weiterhin ist *block_time* die durchschnittliche Zeit bis ein neuer Block gemint wird, *block_reward* ist die Belohnung in der geminten Währung, die der Miner eines Blockes erhält und *stock_price* ist der Börsenpreis für eine Einheit der Währung in Euro.

Die durchschnittliche Blockzeit ist von der Währung vorgegeben (z.B. bei Ethereum 15 Sekunden) und soll unabhängig von den Minern bzw. der Hashrate des Netzwerkes konstant bleiben. Daher wird die Blockschwierigkeit je nach aktueller Hashrate des Netzwerkes angepasst. Steigt bzw. sinkt diese, wird auch die Blockschwierigkeit erhöht bzw. gesenkt. Daher gilt:

$$net\_hashrate = \frac{block\_diff}{block\_time} \tag{4.9}$$

Im Framework wird *stock_price* von *CryptoCompare* (siehe [10]), und *net_hashrate*, *block_time* sowie *block_reward* von *WhatToMine* (siehe [39]) über deren REST-APIs abgefragt.

Für Hashrate bzw. Energieverbrauch, welche für die Berechnung von Miningerträgen bzw. Energiekosten genutzt werden, gibt es ein Zeitfenster der Länge *window_size* innerhalb dessen Werte berücksichtigt werden. Beim Energieverbrauch wird einfach das Mittel für das Zeitfenster genommen. Für die Hashrate der aktuell geminten Währung wird

$$user\_hashrate = \alpha \cdot user\_hashrate\_local + (1 - \alpha) \cdot user\_hashrate\_pool \tag{4.10}$$

bzw. für alle anderen Währungen

$$user\_hashrate = (1 - \alpha) \cdot user\_hashrate\_local + \alpha \cdot user\_hashrate\_pool \tag{4.11}$$

verwendet. Dabei ist *user_hashrate_local* die lokale, vom Miner bestimmte Hashrate und *user_hashrate_pool* die Hashrate, welche über die REST-API vom Mining-Pool ermittelt wird (siehe Abschnitt 4.5.1). Diese Werte sind immer über die aktuelle bzw. letzte Miningperiode der entsprechenden Währung gemittelt. Der Parameter $0 \leq \alpha \leq 1$ bestimmt mit welchem Anteil diese jeweils bei der Berechnung eingehen. Er ist von der Länge des Zeitfensters *window_size* und einer Zeitspanne Δt abhängig, welche angibt, wie lange die entsprechende Währung bereits gemint wird bzw. nicht mehr gemint wurde. Es gilt:

$$\alpha = \frac{min(\Delta t, window\_size)}{window\_size} \tag{4.12}$$

In Abbildung 4.4 ist die Berechnung der verwendeten Hashrate am Beispiel von drei Währungen veranschaulicht. Währung 1 wird von Zeitpunkt t_0 bis t_2 gemint, Währung 2 von t_2 bis t_3 und Währung 3 ab t_3. Die Zeiträume von t_0 bis t_1, t_2 bis t_4 und t_3 bis t_6 entsprechen jeweils der Länge eines Zeitfensters.

Der Grund warum das so berechnet wird ist, dass die auf Basis der eingereichten Shares

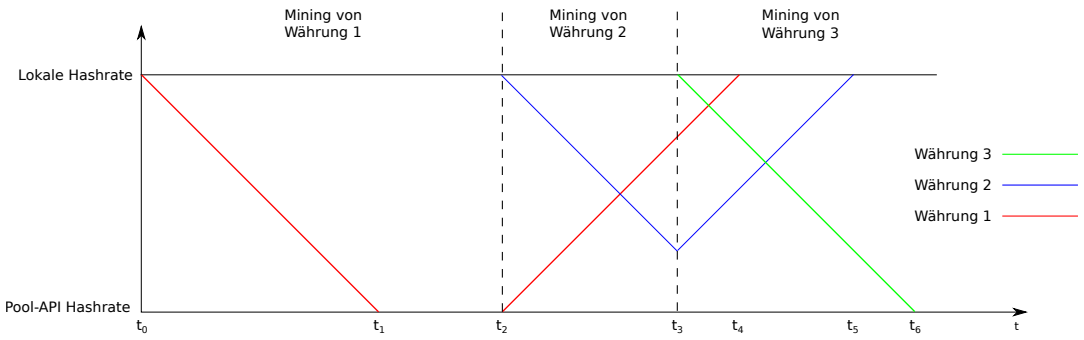

Abbildung 4.4: Verwendete Hashrate bei der Berechnung der Miningerträge am Beispiel von drei Währungen

ermittelte Hashrate der Pool-API starken Schwankungen unterliegt, da das Minen einer Share auch Glückssache ist (siehe Abschnitt 3). Über einen längeren Zeitraum gemittelt relativiert sich dies jedoch. Dieser Zeitraum entspricht der Länge eines Zeitfensters *window_size* und sollte in der Größenordnung von zwei bis sechs Stunden liegen.

4.5.3 Währungswechsel und Frequenzreoptimierung

Dieser Schritt wird ausgeführt, falls die aktuell geminte Währung nach Neuberechnung von Energiekosten und Miningerträgen (siehe Abschnitt 4.5.2) nicht mehr die Profitabelste ist. Es wird das im Hintergrund laufende Mining der aktuellen Währung beendet und das Mining der neuen profitabelsten Währung gestartet.

Anschließend wird für die neu geminte Währung nochmals nach energieeffizienteren Frequenzen gesucht. Das geschieht wie im Online-Teil der *Frequenzoptimierungsphase* (siehe 4.4.2). Startpunkt der Suche sind die zuletzt verwendeten Frequenzen. Mit dem Ergebnis dieser erneuten Optimierung wird das bisherige Optimierungsresultat aktualisiert.

4.6 Konfigurationsdateien

In diesem Abschnitt wird der Aufbau der Konfigurationsdateien beschrieben, welche bei Programmstart angegeben werden können (siehe Abschnitt 4.1.1). Es gibt drei verschiedene Konfigurationsdateien, welche alle das JSON-Format verwenden:

1. **Währungskonfiguration:** Einstellung der verfügbaren Währungen.
2. **Miningkonfiguration:** Einstellung der zu nutzenden GPUs, Währungen und Optimierungsverfahren. Die Währungen hier müssen eine Teilmenge der in der *Währungskonfiguration* spezifizierten sein.
3. **Optimierungsresultat:** Zu verwendendes Ergebnis einer vorherigen Optimierung.

4.6.1 Währungskonfiguration

Die *Währungskonfiguration* enthält eine Liste von verfügbaren Währungen. Jede Währung hat folgende notwendige und optionale Attribute:

- **currency_name:** Name der Währung.
- **use_ccminer:** Boolean der angibt, ob die Währung mit dem ccminer gemint wird.
- **ccminer_algo:** ccminer-Bezeichnung des Hashalgorithmus der Währung. Nicht notwendig, falls use_ccminer=false.
- **bench_script_path:** Relativer Pfad zum Benchmark-Skript der Währung. Optional, falls use_ccminer=true.
- **mining_script_path:** Relativer Pfad zum Mining-Skript der Währung. Optional, falls use_ccminer=true.
- **pool_adresses:** Liste mit Server-Adressen (IP+Port) des Mining-Pools, wobei alle Adressen nach der ersten als Fallback verwendet werden.
- **pool_pass:** Passwort des Mining-Pool.
- **whattomine_coin_id:** ID der Währung auf *WhatToMine* (siehe [39]).
- **cryptocompare_fsym:**: Kürzel der Währung auf *CryptoCompare* (siehe [10]). Entspricht dem Standardkürzel.

Nachfolgende Attribute konfigurieren die REST-API Optionen des Mining-Pool. Diese Attribute sind alle optional. Von der API wird immer ein String im JSON-Format als Antwort erwartet. Es gibt drei API Optionen und es müssen immer alle Attribute einer Option spezifiziert sein, damit diese verwendet werden kann:

1. Aktuelle Hashrate:
    - **api_address:** API-Endpunkt Adresse mit einem Platzhalter (%s) für die Wallet-Adresse auf der gemint wird.
    - **json_path_worker_array:** JSON-Pfad zum Worker-Array. Dieses enthält Informationen zu den einzelnen Workern (Minern), welche unter der gegebenen Wallet-Adresse auf dem Pool minen.
    - **json_path_worker_name:** JSON-Pfad innerhalb eines Elements des Worker-Arrays, der zur ID eines einzelnen Workers führt.
    - **json_path_hashrate:** JSON-Pfad innerhalb eines Elements des Worker-Arrays, der zur Hashrate eines einzelnen Workers führt.
    - **api_unit_factor_hashrate:** Umrechnungsfaktor, der die im JSON spezifizierte Hashrate für einen Worker in Hashes/s umrechnet.

2. Durchschnittliche Hashrate:
   - **api_address:** Siehe 1) mit einem zweiten Platzhalter für die Zeitspanne der Durchschnittsberechnung.
   - **api_unit_factor_period:** Umrechnungsfaktor für den Zeitspannenparameter, der die von der API erwartete Einheit in Sekunden umrechnet.
   - **json_path_worker_array:** Siehe 1).
   - **json_path_worker_name:** Siehe 1).
   - **json_path_hashrate:** Siehe 1).
   - **api_unit_factor_hashrate:** Siehe 1).
3. Geschätzte Miningerträge:
   - **api_address:** API-Endpunkt Adresse mit einem Platzhalter (%s) für die Hashrate für welche die Miningerträge berechnet werden.
   - **api_unit_factor_hashrate:** Umrechnungsfaktor für den Hashrateparameter, der die von der API erwartete Einheit in Hashes/s umrechnet.
   - **json_path_earnings:** JSON-Pfad zu den geschätzten Miningerträgen in der entsprechenden Währung für eine Zeiteinheit.
   - **api_unit_factor_period:** Umrechnungsfaktor, der die im JSON spezifizierten Minigerträge für einen Worker in Erträge/h umrechnet.

Die Attribute wurden so gewählt, dass die API von möglichst vielen Mining-Pools genutzt werden kann. Von den API-Optionen für die durchschnittliche und aktuelle Hashrate kann immer nur eine verwendet werden, wobei die für die durchschnittliche Hashrate bevorzugt wird. D.h. falls beim Mining-Pool beide API-Optionen vorhanden sind, sollte die für die durchschnittliche Hashrate in der *Währungskonfiguration* angegeben werden. Falls die API-Option für die geschätzten Miningerträge angegeben ist, ersetzt diese die eigene Berechnung der Erträge mit Gleichung 4.7. Eine beispielhafte *Währungskonfiguration* mit zwei Währungen ist in Listing 4.7 zu sehen.

4.6.2 Miningkonfiguration

Die *Miningkonfiguration* enthält alle konfigurierbaren Parameter für einen Programmdurchlauf. Die globalen Attribute sind:

- **energy_cost:** Energiekosten für eine kWh in Euro.
- **monitoring_interval:** Periodendauer während der *Profitüberwachungsphase* in Sekunden.
- **skip_offline_phase:** Boolean der angibt, ob die Offline-Frequenzoptimierung übersprungen wird.

```
{
  "available_currencies": [
    {
      "currency_name": "ETH",
      "use_ccminer": "false",
      "bench_script_path": ".\/scripts\/run_benchmark_eth_ethminer.sh",
      "mining_script_path": ".\/scripts\/start_mining_eth_ethminer.sh",
      "pool_addresses": [
        "eth-eu1.nanopool.org:9999",
        "eth-eu2.nanopool.org:9999"
      ],
      "pool_pass": "x",
      "whattomine_coin_id": "151",
      "cryptocompare_fsym": "ETH",
      #attributes specifying avg_hashrate and approximated_earnings REST-API options
      "pool_api_options": {
        "avg_hashrate": {
          "api_address":
              "https:\/\/api.nanopool.org\/v1\/eth\/avghashrateworkers\/%s\/%s",
          "json_path_worker_array": "data",
          "json_path_worker_name": "worker",
          "json_path_hashrate": "hashrate",
          "api_unit_factor_hashrate": "1000000",
          "api_unit_factor_period": "3600000"
        },
        "approximated_earnings": {
          "api_address":
              "https:\/\/api.nanopool.org\/v1\/eth\/approximated_earnings\/%s",
          "json_path_earnings": "data.hour.coins",
          "api_unit_factor_hashrate": "1000000",
          "api_unit_factor_period": "1"
        }
      }
    },
    {
      "currency_name": "LUX",
      "use_ccminer": "true",
      "ccminer_algo": "phi2",
      "pool_addresses": [
        "phi2.mine.zpool.ca:8332"
      ],
      "pool_pass": "c=LUX,mc=LUX",
      "whattomine_coin_id": "212",
      "cryptocompare_fsym": "LUX",
      #attributes specifying current_hashrate REST-API option
      "pool_api_options": {
        "current_hashrate": {
          "api_address": "http:\/\/www.zpool.ca\/api\/walletEx?address=%s",
          "json_path_worker_array": "miners",
          "json_path_worker_name": "ID",
          "json_path_hashrate": "accepted",
          "api_unit_factor_hashrate": "1"
        }
      }
    }
  ]
}
```

Listing 4.7: Beispiel einer Währungskonfiguraton mit zwei Währungen

- **online_bench_duration:** Dauer eines Benchmarks der Online-Frequenzoptimierung in Sekunden.
- **email:** Email-Adresse für Benachrichtigungen der Mining-Pools. Optionales Attribut.
- **devices_to_use:** Liste der zu nutzenden GPUs.
- **currencies_to_use:** Liste der zu nutzenden Währungen. Jede dieser Währungen muss in der *Währungskonfiguration* vorhanden sein.

Jede der zu nutzenden GPUs wird mit folgenden Attributen spezifiziert bzw. konfiguriert:

- **index:** ID der GPU.
- **name:** Name der GPU. Optionales Attribut.
- **min_mem_clock:** Minimale zu verwendende VRAM-Clock. Optionales Attribut.
- **max_mem_clock:** Maximale zu verwendende VRAM-Clock. Optionales Attribut.
- **min_graph_clock:** Minimale zu verwendende Core-Clock. Optionales Attribut.
- **max_graph_clock:** Maximale zu verwendende Core-Clock. Optionales Attribut.
- **worker_name:** Bezeichnung des Miners auf der GPU am Mining-Pool.

Jede der zu nutzenden Währungen wird mit folgenden Attributen spezifiziert bzw. konfiguriert:

- Key-Attribut mit dem Namen der Währung.
- **wallet_address:** Wallet mit der die Währung gemint wird.
- **method:** Optimierungsverfahren, das für die Währung verwendet wird.
- **min_hashrate:** Minimale Hashrate, die nicht unterschritten werden darf als Nebenbedingung der Optimierung. Angabe ist relativ zur maximal erzielbaren Hashrate. Optionales Attribut.
- **max_iterations:** Maximale Anzahl der Iterationen des Optimierungsverfahrens.
- **mem_step:** Schrittweite des Optimierungsverfahrens in der VRAM-Clock-Dimension relativ zum verfügbaren Frequenzbereich.
- **graph_idx_step:** Schrittweite des Optimierungsverfahrens in der Core-Clock-Dimension relativ zum verfügbaren Frequenzbereich.

Listing 4.8 zeigt eine beispielhafte *Miningkonfiguration* mit zwei GPUs und zwei Währungen.

```
{
  "energy_cost": "0.1",
  "monitoring_interval": "3601",
  "skip_offline_phase": "false",
  "online_bench_duration": "120",
  "email": "alexander.fiebig@uni-bayreuth.de",
  "devices_to_use": [
    {
      "index": "0",
      "name": "TITAN X (Pascal)",
      "min_mem_clock": "-1",
      "min_graph_clock": "-1",
      "max_mem_clock": "-1",
      "max_graph_clock": "-1",
      "worker_name": "DESKTOP-6817PNB_gpu0"
    },
    {
      "index": "1",
      "name": "Quadro P4000",
      "min_mem_clock": "-1",
      "min_graph_clock": "500",
      "max_mem_clock": "-1",
      "max_graph_clock": "-1",
      "worker_name": "DESKTOP-6817PNB_gpu1"
    }
  ],
  "currencies_to_use": {
    "ETH": {
      "wallet_address": "0x8291ca623a1f1a877fa189b594f6098c74aad0b3",
      "opt_method_params": {
        "method": "HC",
        "min_hashrate": "-1",
        "max_iterations": "6",
        "mem_step": "0.15",
        "graph_idx_step": "0.15"
      }
    },
    "ZEC": {
      "wallet_address": "t1Z8gLLGyxGRkjRFbNnJ2n6yvHb1Vo3pXKH",
      "opt_method_params": {
        "method": "SA",
        "min_hashrate": "0.85",
        "max_iterations": "6",
        "mem_step": "0.15",
        "graph_idx_step": "0.15"
      }
    }
  }
}
```

Listing 4.8: Beispiel einer Miningkonfiguration mit zwei GPUs und zwei Währungen

4.6.3 Optimierungsresultat

Ein Optimierungsresultat enthält Ergebnisse einer vorherigen Optimierung. Dabei werden für jede verwendete GPU Optimierungsresultate für jede auf dieser GPU verwendeten Währung gespeichert. Ein solches Optimierungsresultat einer Währung auf einer GPU ist in drei Teile gegliedert:

1. **offline:** Enthält Optimierungsresultat der Offline-Frequenzoptimierung.
2. **online:** Enthält Optimierungsresultat der Online-Frequenzoptimierung.
3. **profit:** Enthält aktualisiertes Optimierungsresultat nach der *Profitüberwachungsphase*.

Für jeden dieser Teile gibt es folgende Attribute:

- **power:** Gemessener Energieverbrauch in Watt.
- **hashrate:** Gemessene Hashrate in Hashes/s.
- **energy_hash:** Gemessene Anzahl an Hashes pro Joule.
- **nvml_graph_clock_idx:** Verwendete Core-Frequenz als Index im Vektor der verfügbaren Core-Frequenzen.
- **graph_oc:** Verwendete Core-Frequenz als Offset zur Standard-Core-Frequenz.
- **graph_clock:** Verwendete Core-Frequenz als absoluter Wert.
- **mem_oc:** Verwendete VRAM-Frequenz als Offset zur Standard-VRAM-Frequenz.
- **mem_clock:** Verwendete VRAM-Frequenz als absoluter Wert.
- **power_measure_dur_ms:** Dauer über die der gemessene Energieverbrauch gemittelt ist.
- **hashrate_measure_dur_ms:** Dauer über welche die gemessene Hashrate gemittelt ist.

Diese Attribute speichern die Werte einer `measurement` Struktur (siehe Listing 4.5). Beim Laden eines Optimierungsresultates wird aus diesen Attributen eine solche `measurement` Struktur rekonstruiert. In Listing 4.9 ist ein beispielhaftes Optimierungsresultat mit zwei GPUs und je zwei Währungen dargestellt.

```
{
  "0": {
    "device_name": "TITAN X (Pascal)",
    "currency_opt_result": {
      "XMR": {
        "offline": {
          "power": "84.341999999999999",
          "hashrate": "675.20000000000005",
          "energy_hash": "8.0055014109221982",
          "nvml_graph_clock_idx": "69",
          "mem_oc": "-195",
          "graph_oc": "-798",
          "graph_clock": "1113",
          "mem_clock": "4810",
          "power_measure_dur_ms": "16131",
          "hashrate_measure_dur_ms": "16131"
        },
        "online": {
          "power": "93.974749571183679",
          "hashrate": "778.44583333333321",
          "energy_hash": "8.2835637965034277",
          "nvml_graph_clock_idx": "60",
          "mem_oc": "366",
          "graph_oc": "-684",
          "graph_clock": "1227",
          "mem_clock": "5371",
          "power_measure_dur_ms": "120001",
          "hashrate_measure_dur_ms": "120001"
        },
        "profit": {
          "power": "95.810740448740802",
          "hashrate": "807.19671702454718",
          "energy_hash": "8.4249084522669069",
          "nvml_graph_clock_idx": "60",
          "mem_oc": "366",
          "graph_oc": "-684",
          "graph_clock": "1227",
          "mem_clock": "5371",
          "power_measure_dur_ms": "32660844",
          "hashrate_measure_dur_ms": "32659966"
        }
      },
      "ETH": {
        ...
      }
    }
  },
  "1": {
    "device_name": "Quadro P4000",
    "currency_opt_result": {
      ...
    }
  }
}
```

Listing 4.9: Beispiel eines Optimierungsresultates mit zwei GPUs und zwei Währungen

5 Evaluation

In diesem Kapitel wird das in Kapitel 4 vorgestellte Framework evaluiert. Das Framework wird dazu mit verschiedenen GPUs und verschiedenen Währungen getestet. Anschließend werden die Ergebnisse von *Frequenzoptimierungs-* und *Profitüberwachungsphase* dargestellt.

5.1 Verwendete Hardware

Im Folgenden werden die zum Testen des Frameworks verwendeten GPUs vorgestellt. Tabelle 1 zeigt alle genutzten GPUs mit Informationen zur Hardware. Zudem ist angegeben, welche der APIs zum Übertakten aus Abschnitt 4.2 zur Verfügung stehen, sowie der einstellbare Frequenzbereich.

	TITAN X	**TITAN V**	**GTX 1080**	**Quadro P4000**
Architecture	Pascal	Volta	Pascal	Pascal
CUDA-Cores	3584	5120	2560	1792
Memory type	12GB GDDR5X	12GB HBM2	8GB GDDR5X	8GB GDDR5
Memory bandwidth	480GB/s	652GB/s	320GB/s	243GB/s
Power-Cap	250W	250W	180W	105W
NVML	Ja	Ja	Nein	Ja
NVAPI	Ja	Ja	Ja	Nein
NV-Control X	Ja	Nein	Ja	Nein
VRAM-Clock Range [MHz]	4005-6005	850-1050	4005-6005	3802
Core-Clock Range [MHz]	139-2011	139-2012	1711-2011	139-1708

Tabelle 1: Zur Evaluation verwendete GPUs

Während der Entwicklung wurden auch einige ältere GPUs ausprobiert. Bei diesen funktioniert jedoch das Setzen der Frequenzen gar nicht oder nur stark eingeschränkt:

- TITAN X (Maxwell)
- TITAN Black
- Geforce GTX 980 Ti
- Geforce GTX 960

5.2 Verwendete Währungen

Als Währungen werden in der Evaluation Ethereum (ETH), Monero (XMR), ZCash (ZEC) und einige neue, weniger verbreitete Währungen (LUX, RVN, BTX und VTC),

sogenannte Altcoins, verwendet. Für das Minen von ETH wird der *ethminer* [31] in Version 0.15.0.dev11, für XMR der *xmr-stak* [14] in Version 2.4.5, für ZEC der *excavator* [23] in Version 1.1.0a und für die Altcoins der *ccminer* [34] in Version 2.3 genutzt. ETH, XMR und ZEC werden in folgendem Abschnitt genauer untersucht. Die Messungen werden unter Windows 10 mit dem NVIDIA-Treiber in Version 391.24 und dem CUDA-Toolkit in Version 9.1 durchgeführt.

Zunächst wird das Mining der verschiedenen Währungen mit dem *NVIDIA Visual Profiler* (siehe [29]) geprofilt. Ausschlaggebend für die Performance ist die in der Blockchain der entsprechenden Währung verwendete Hashfunktion. ETH verwendet als Hashfunktion Ethash, XMR Cryptonight und ZEC Equihash. Zum Zeitpunkt der Arbeit liegt Cryptonight in Version 7 vor (Monero ändert die Hashfunktion regelmäßig, siehe Abschnitt 3.2). Anschließend wird die Hashrate und der Energieverbrauch bei allen auf der jeweiligen GPU einstellbaren Frequenzen gemessen und ein Vergleich der Energieeffizienz (Hashes pro Joule) bei optimalen und normalen Frequenzen gemacht.

5.2.1 Profiling

Abbildung 5.1 zeigt die Ergebnisse des Profiling von ETH, ZEC und XMR auf der Titan X (Pascal). Es wurde die genutzte Speicherbandbreite für Lese- und Schreibzugriffe, sowie die Anzahl der Instruktionen pro Taktzyklus (IPC) gemessen. Die verwendete Speicherbandbreite ist ein Maß dafür, wie stark ein Programm Memory-Bound ist und die ausgeführten IPC sind ein Maß dafür, wie stark es Compute-Bound ist. Da die Miner jeweils mehrere CUDA-Kernel verwenden, ist das nach der GPU-Zeit gewichtete Mittel der einzelnen Kernel dargestellt.

An der genutzten Speicherbandbreite ist zu erkennen, dass ETH von den untersuchten Währungen am stärksten Memory-Bound ist, gefolgt von XMR und ZEC. An den ausgeführten IPC sieht man, dass ZEC am stärksten Compute-Bound ist, gefolgt von ETH und XMR. Je stärker eine Währung Memory-Bound ist, desto schneller und länger steigt die Hashrate beim Hochstellen der VRAM-Frequenz. Ebenso steigt die Hashrate beim Hochstellen der Core-Frequenz schneller und länger, je stärker eine Währung Compute-Bound ist (siehe Abbildungen 5.2 - 5.8).

5.2.2 Energieoptimum ETH-Ethash

Zur Ermittlung der Energieoptima (maximale Anzahl an Hashes pro Joule) wird für alle einstellbaren Frequenzen auf der jeweiligen GPU Hashrate und Energieverbrauch gemessen.

Die Abbildungen 5.2 und 5.3 zeigen das Ergebnis der Messungen auf den verfügbaren GPUs (siehe Tabelle 1) für ETH. Hier ist u.a. zu erkennen, dass ETH im Vergleich zu den anderen Währungen am meisten von höheren VRAM-Frequenzen profitiert, was das Ergebnis des Profiling aus Abschnitt 5.2.1 bestätigt. Das Energieoptimum ist zumeist bei mittleren Core- und mittleren bis hohen VRAM-Frequenzen zu finden.

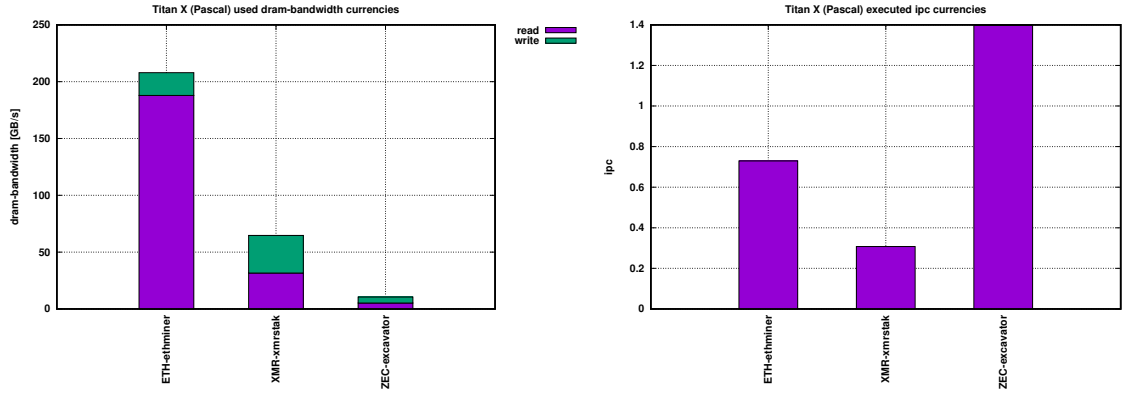

Abbildung 5.1: Genutzte VRAM-Speicherbandbreite (links) und ausgeführte IPC (rechts) beim Minen von ETH (ethminer), XMR (xmrstak) und ZEC (excavator) auf der Titan X (Pascal) mit Standardfrequenzen

Zur Ermittelung der Effizienzsteigerung werden die Werte bei optimalen Frequenzen mit denen bei normalen Frequenzen verglichen (siehe Tabelle 2). Die normalen Frequenzen werden ermittelt, indem der Miner auf der jeweiligen GPU gestartet wird und die vom NVIDIA-Treiber eingestellten Frequenzen beobachtet werden.

Currency: ETH		**TITAN X**	**TITAN V**	**GTX 1080**	**Quadro P4000**
Optimal frequencies	VRAM-Clock	4955	1010	5805	3802
	Core-Clock	1151	1035	1711	1088
	Hashrate	31119928	82603192	25339050	26496646
	Power	113.095	155.207	156.29	77.036
	Hashes/Joule	275166	532213	162128	343951
Default frequencies	VRAM-Clock	5005	850	5005	3802
	Core-Clock	1822	1335	1885	1695
	Hashrate	31532634	67375607	21387776	26630501
	Power	161.008	147.519	135.96	104.775
	Hashes/Joule	195845	456725	157309	254168
Efficiency gain		40.5%	16.53%	3.06%	35.32%

Tabelle 2: ETH: Optimale vs. normale Frequenzen auf den verschiedenen GPUs

5.2.3 Energieoptimum XMR-Cryptonight

XMR verhält sich ähnlich zu ETH, wie in den Abbildungen 5.4 und 5.5 zu sehen ist. Da es aber sowohl weniger Compute-, als auch weniger Memory-Bound wie ETH ist (siehe Abbildung 5.1), ist der Anstieg der Hashrate bei Erhöhung der Frequenzen etwas flacher.

Auffallend ist auch, dass XMR relativ wenig Energie benötigt. Die Effizienzsteigerung durch das Optimieren der Frequenzen ist in Tabelle 3) zu sehen.

Das Minen von XMR lohnt sich im Gegensatz zu den meisten anderen Währungen auch auf CPUs. Ein möglicher Grund dafür ist, dass es die höhere Anzahl an Recheneinheiten einer GPU kaum auslastet, wie der niedrige IPC-Wert zeigt. Dies ist auch der Grund für den niedrigen Energiebedarf.

Abbildung 5.6 zeigt zum Vergleich Messungen mit CPUs auf folgenden Systemen:

1. **node06:** Clusterknoten mit zwei Intel(R) Xeon(R) CPU E5-2630 v3 (8 Threads + HT pro CPU, Haswell-Architektur)

2. **pc-skylake:** Rechner mit einer Intel(R) Core(TM) i7-6700 CPU (4 Threads + HT, Skylake-Architektur)

Diese Messungen entstammen einer Masterarbeit, welche die Energieoptimierung verschiedener Applikationen auf CPUs mit OpenTuner zum Thema hat (siehe [19]). Auf CPUs kann Core- und Cachefrequenz (uncore) eingestellt werden, wofür in genannter Arbeit likwid verwendet wird. Die Frequenz vom Hauptspeicher ist nicht veränderbar.

Die Messungen in Abbildung 5.6 zeigen, dass insbesondere das Minen auf pc-skylake mit der Skylake-CPU sehr effizient ist, da diese sehr wenig Energie verbraucht. Von der Energieeffizienz bei optimalen Frequenzen (14.88 H/J) ist diese CPU im Bereich der Titan V, wobei noch etwa 10W zusätzlicher Energiebedarf vom Hauptspeicher mit einzurechnen ist. Der Clusterknoten node06 erreicht eine deutlich höhere Hashrate, benötigt aber auch entsprechend mehr Energie. Von der Effizienz beim Energieoptimum (9.17 H/J) ist er im Bereich der Titan X. Auch hier muss noch ca. 20W (10W pro Socket) zusätzlicher Energiebedarf für den Hauptspeicher dazugerechnet werden.

Currency: XMR		**TITAN X**	**TITAN V**	**GTX 1080**	**Quadro P4000**
Optimal frequencies	VRAM-Clock	5155	1010	5502	3802
	Core-Clock	1202	1035	1811	999
	Hashrate	744.5	1277.6	581.0	554.1
	Power	86.42	88.115	85.51	45.589
	Hashes/Joule	8.6149	14.4992	6.79453	12.1542
Default frequencies	VRAM-Clock	5005	850	5005	3802
	Core-Clock	1847	1335	1911	1708
	Hashrate	753.5	1178.2	523.8	593.9
	Power	161.02	106.495	126.34	73.179
	Hashes/Joule	4.67954	11.0634	4.14596	8.11572
Efficiency gain		84.1%	31.06%	63.88%	49.76%

Tabelle 3: XMR: Optimale vs. normale Frequenzen auf den verschiedenen GPUs

5.2.4 Energieoptimum ZEC-Equihash

ZEC ist im Gegensatz zu ETH und XMR eher Compute-Bound, wie in Abblildung 5.1 zu sehen. Daher ist das Energieoptimum meistens bei niedrigen VRAM- und etwas höheren Core-Clocks zu finden (siehe Abbildungen 5.7 und 5.8). Aus diesem Grund ist das Minen von ZEC auf der Titan V auch nicht effizient, da diese sich durch den schnellen HBM2-Speicher auszeichnet. Zudem hat ZEC aufgrund des hohen IPC-Wertes einen relativ hohen Energiebedarf. Tabelle 4) zeigt die Effizienzsteigerung für ZEC bei Verwendung der optimalen Frequenzen.

Currency: ZEC		**TITAN X**	**TITAN V**	**GTX 1080**	**Quadro P4000**
Optimal frequencies	VRAM-Clock	4155	850	5555	3802
	Core-Clock	1151	1185	1861	1025
	Hashrate	433.525144	525.599705	432.495997	191.152922
	Power	104.826	109.25	129.22	57.176
	Hashes/Joule	4.13566	4.81098	3.34697	3.34324
Default frequencies	VRAM-Clock	5005	850	5005	3802
	Core-Clock	1784	1335	1885	1708
	Hashrate	538.831065	579.105294	395.216173	219.175340
	Power	159.64	128.46	156.30	100.79
	Hashes/Joule	3.37529	4.50806	2.52857	2.17457
Efficiency gain		22.53%	6.72%	32.37%	53.74%

Tabelle 4: ZEC: Optimale vs. normale Frequenzen auf den verschiedenen GPUs

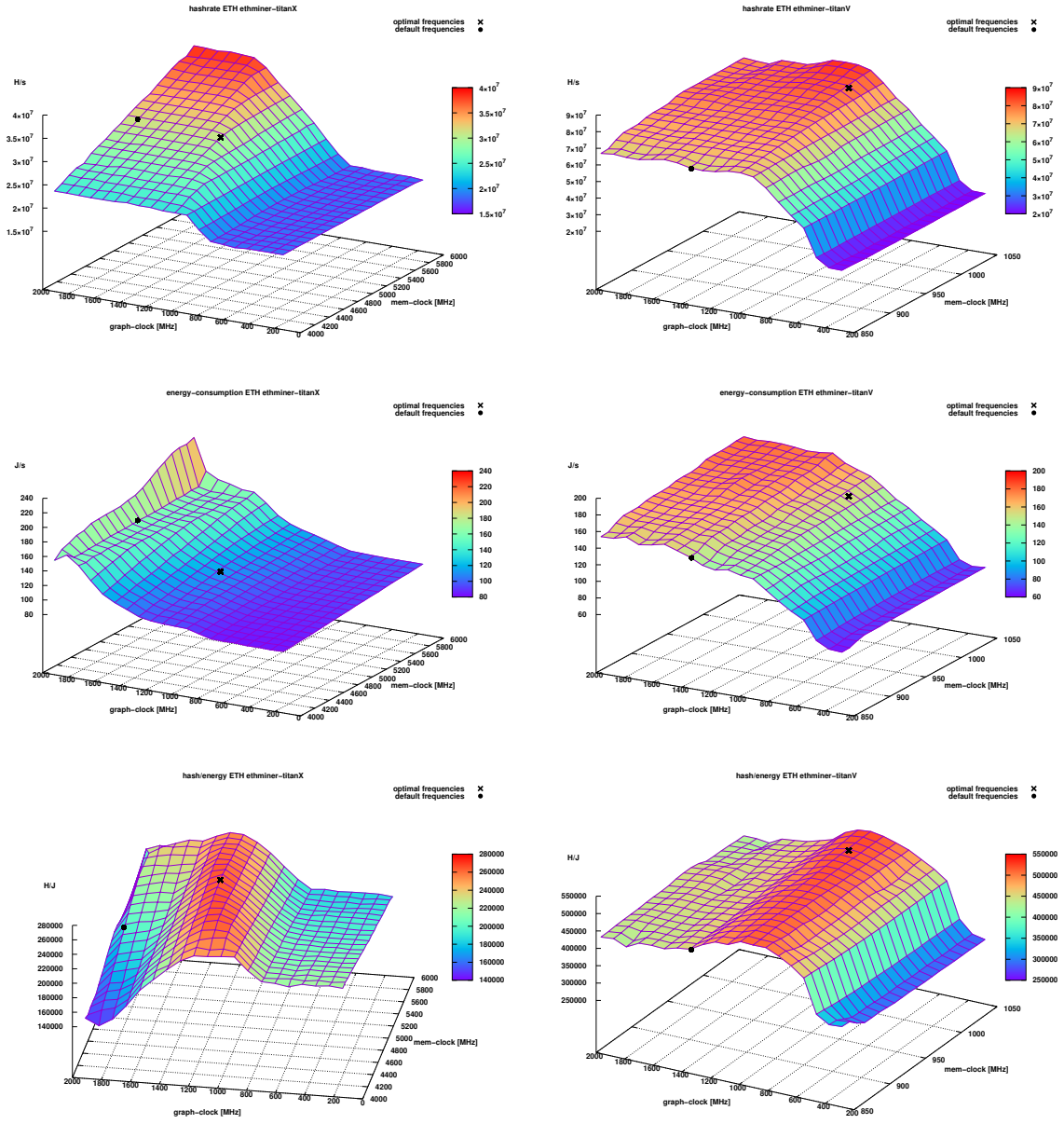

Abbildung 5.2: ETH: Hashrate (oben), Energieverbrauch (mitte) und Hashes pro Joule (unten) bei allen einstellbaren Frequenzen auf der Titan X (Pascal) (linke Spalte) und der Titan V (rechte Spalte)

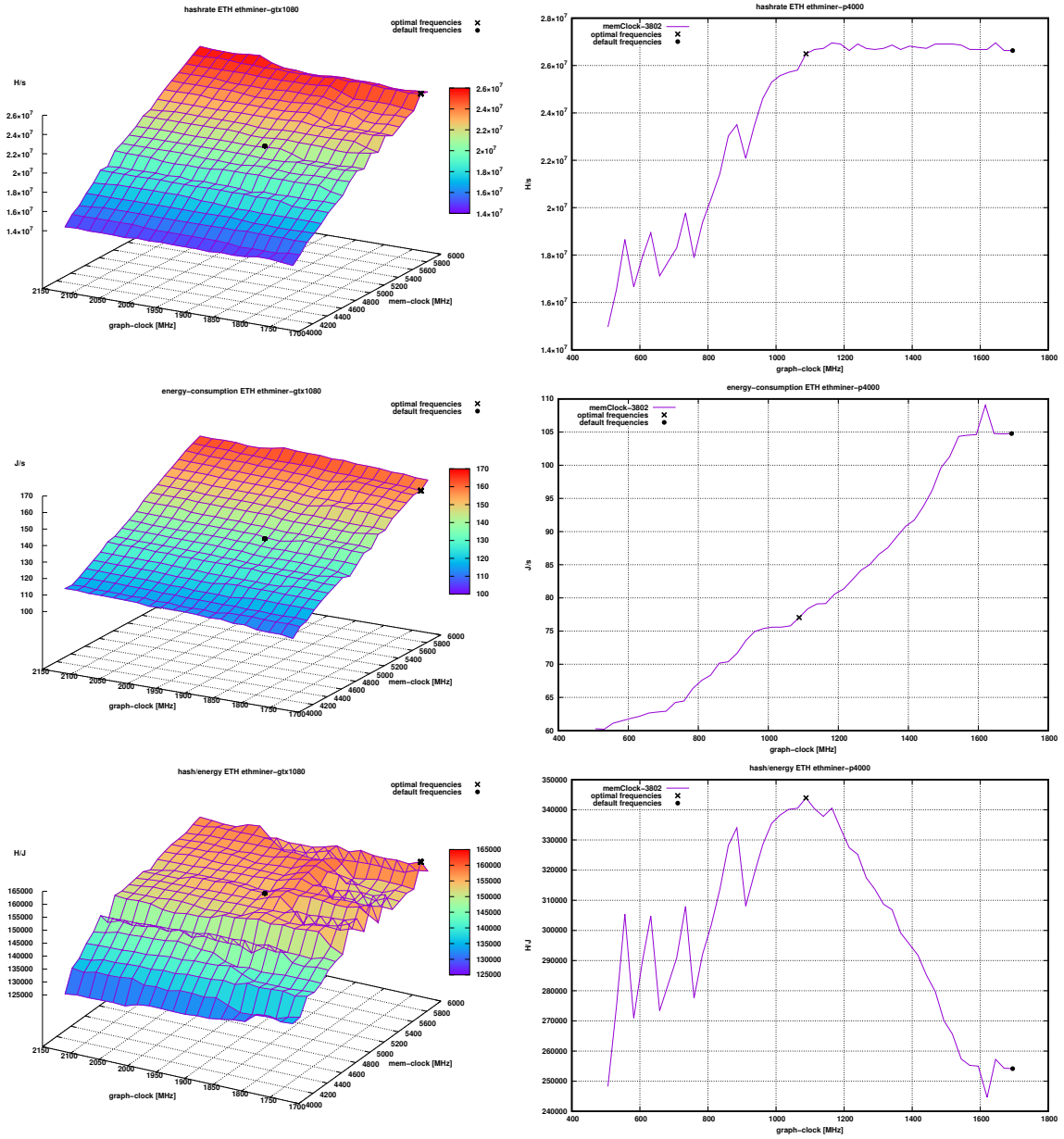

Abbildung 5.3: ETH: Hashrate (oben), Energieverbrauch (mitte) und Hashes pro Joule (unten) bei allen einstellbaren Frequenzen auf der Geforce GTX 1080 (linke Spalte) und der Quadro P4000 (rechte Spalte)

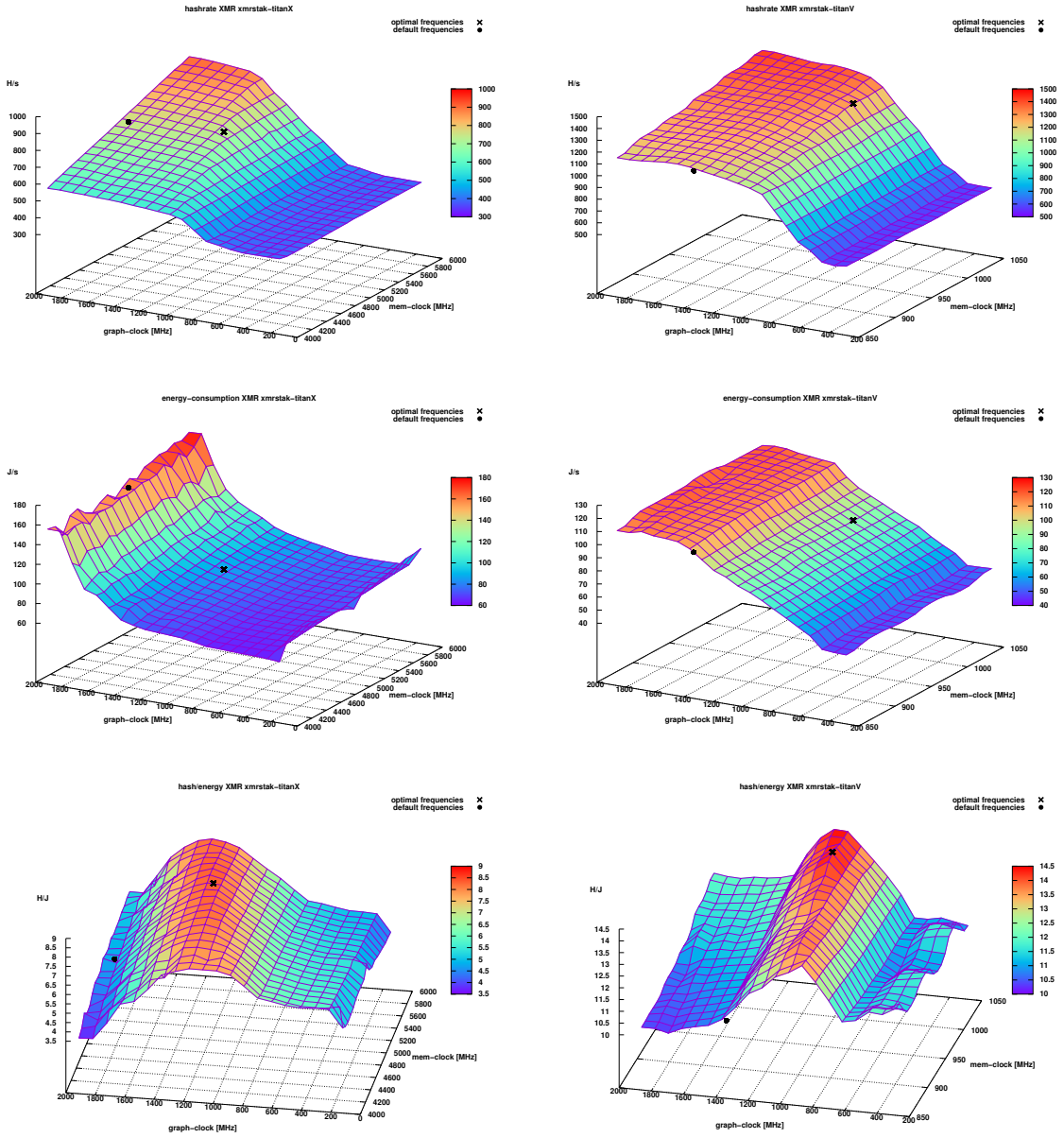

Abbildung 5.4: XMR: Hashrate (oben), Energieverbrauch (mitte) und Hashes pro Joule (unten) bei allen einstellbaren Frequenzen auf der Titan X (Pascal) (linke Spalte) und der Titan V (rechte Spalte)

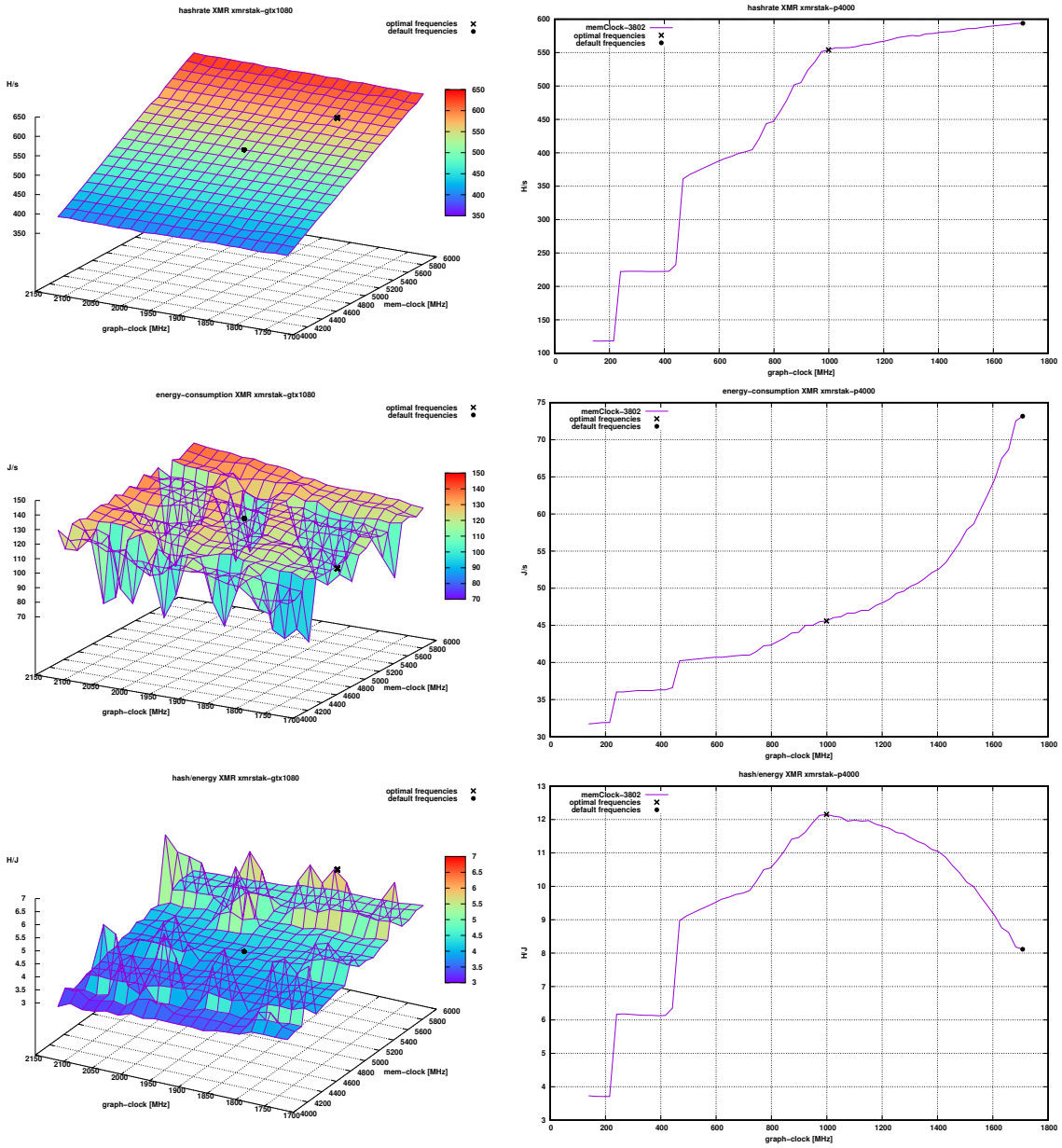

Abbildung 5.5: XMR: Hashrate (oben), Energieverbrauch (mitte) und Hashes pro Joule (unten) bei allen einstellbaren Frequenzen auf der Geforce GTX 1080 (linke Spalte) und der Quadro P4000 (rechte Spalte)

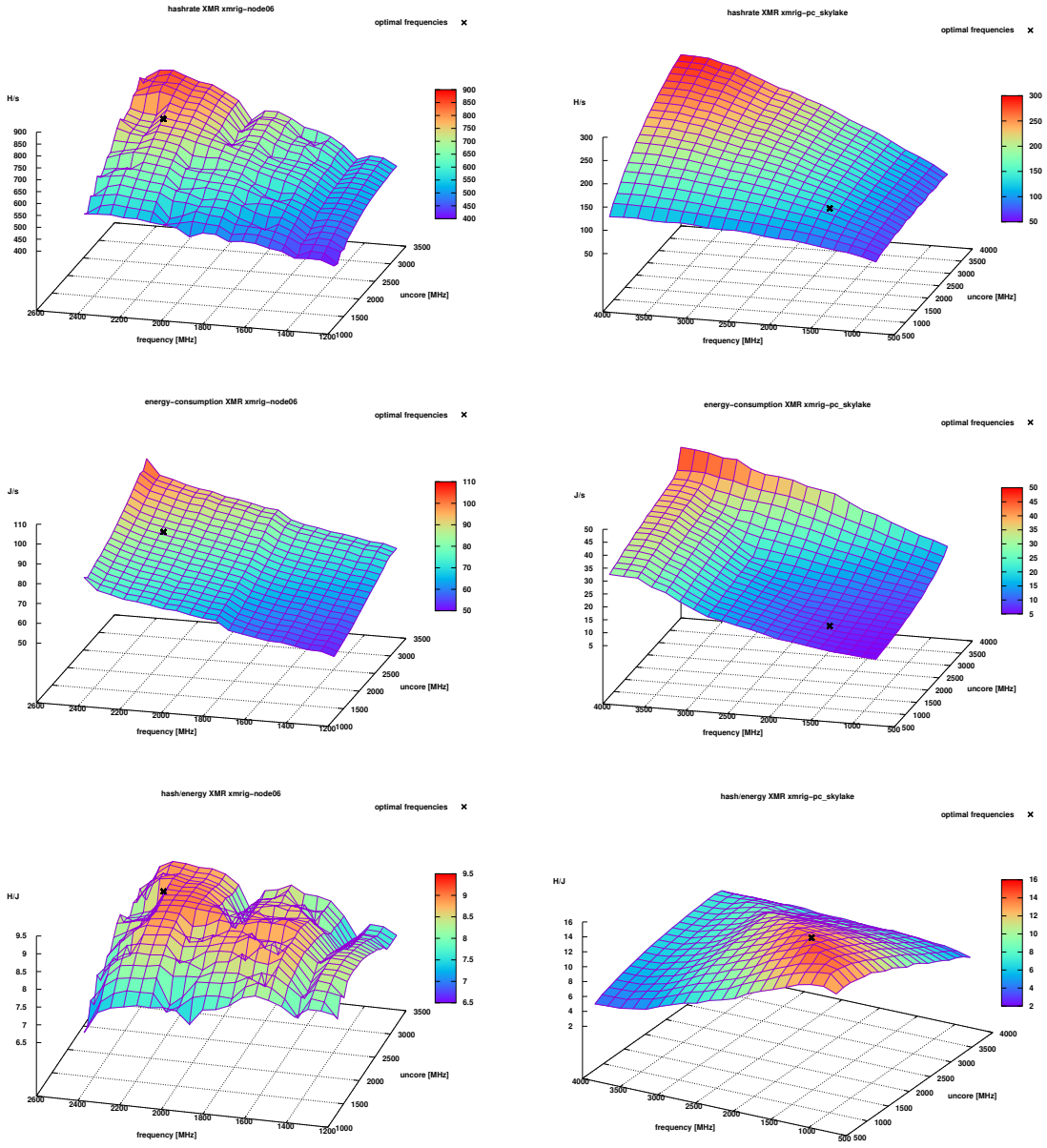

Abbildung 5.6: XMR auf CPUs: Hashrate (oben), Energieverbrauch (mitte) und Hashes pro Joule (unten) bei allen einstellbaren Frequenzen auf node06 (linke Spalte) und pc-skylake (rechte Spalte) (Quelle der Daten: [19])

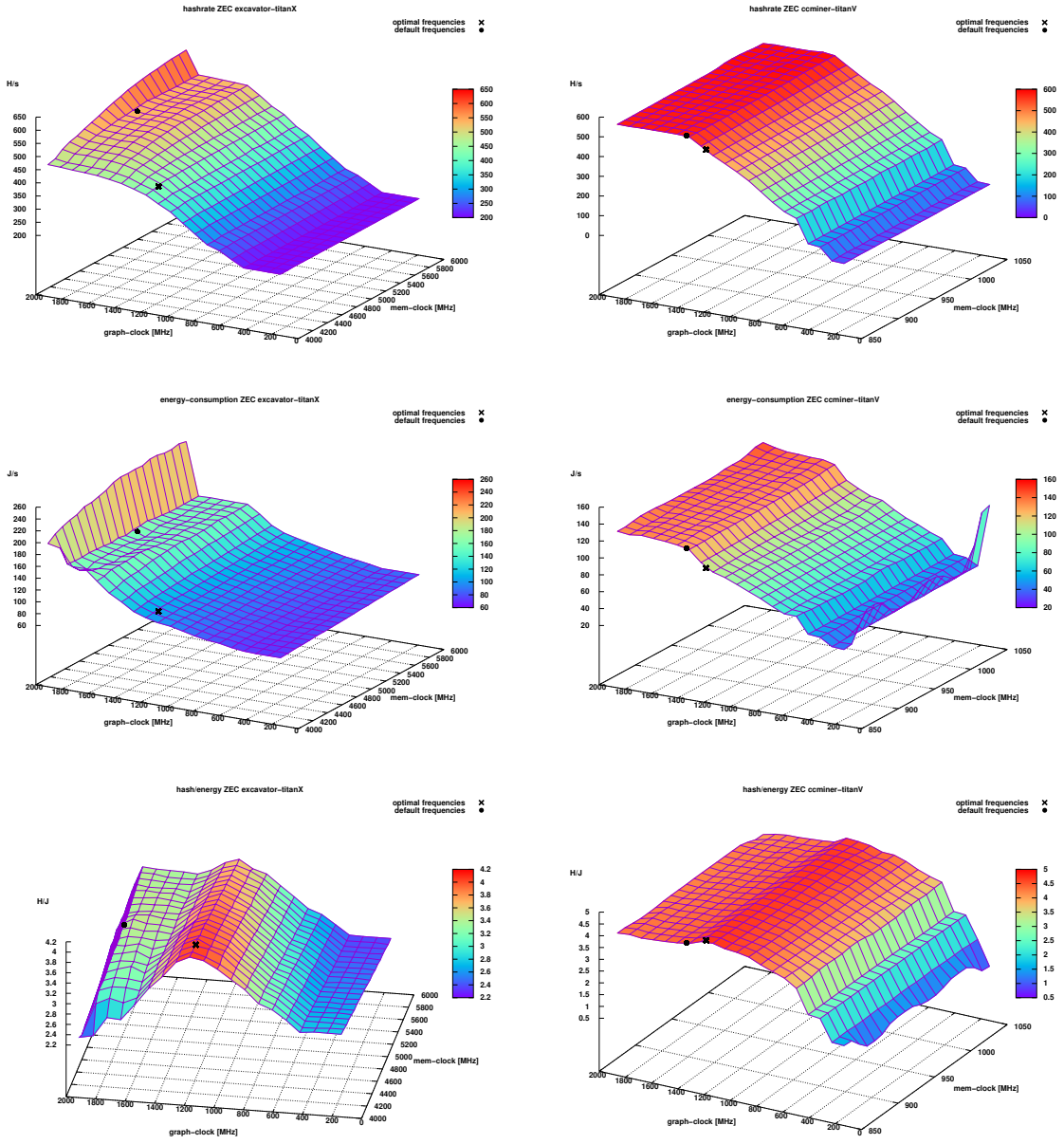

Abbildung 5.7: ZEC: Hashrate (oben), Energieverbrauch (mitte) und Hashes pro Joule (unten) bei allen einstellbaren Frequenzen auf der Titan X (Pascal) (linke Spalte) und der Titan V (rechte Spalte)

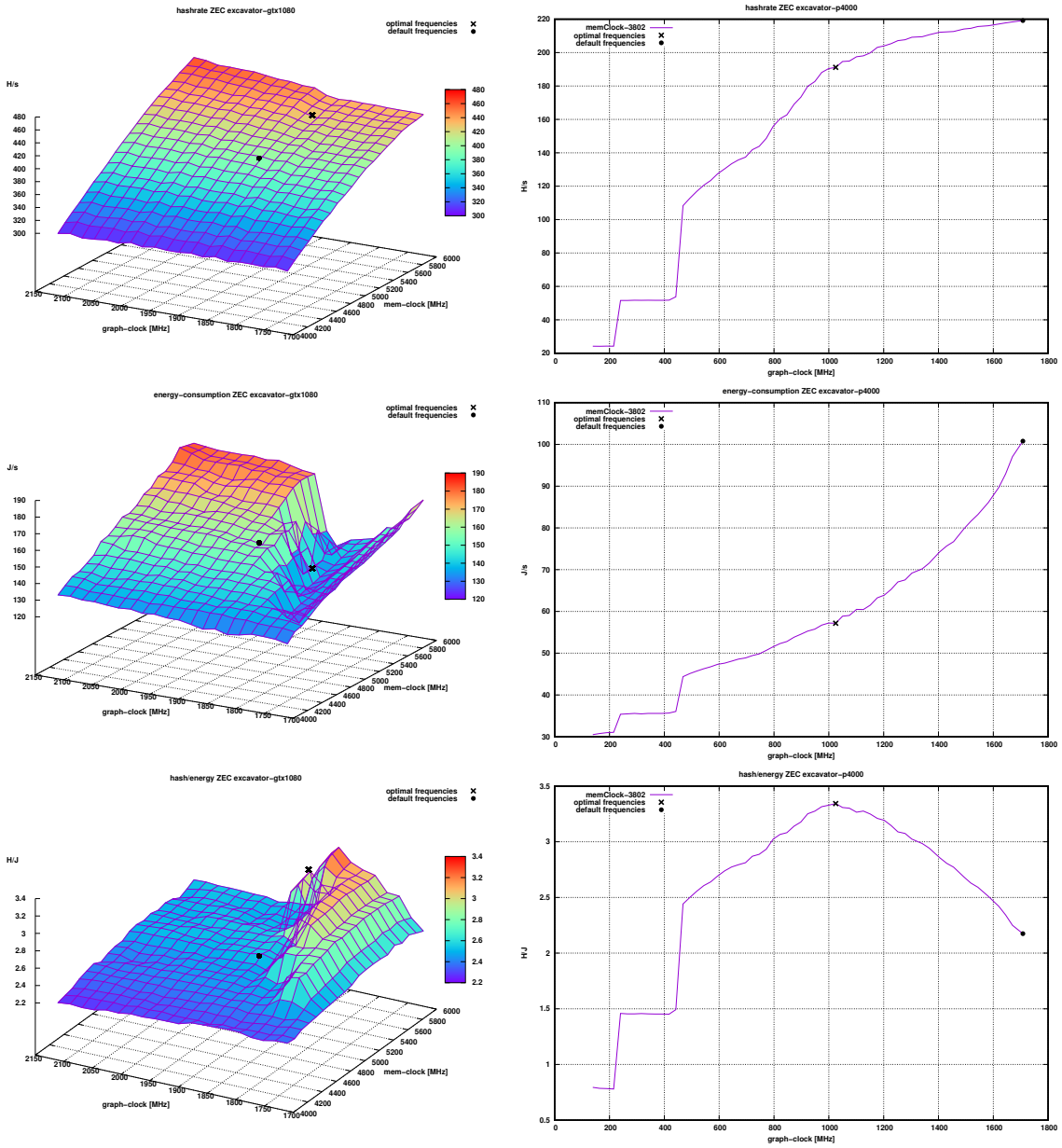

Abbildung 5.8: ZEC: Hashrate (oben), Energieverbrauch (mitte) und Hashes pro Joule (unten) bei allen einstellbaren Frequenzen auf der Geforce GTX 1080 (linke Spalte) und der Quadro P4000 (rechte Spalte)

5.3 Frequenzoptimierung

In diesem Abschnitt werden die verschiedenen Optimierungsalgorithmen (siehe Abschnitt 4.3), welche während der *Frequenzoptimierungsphase* (siehe Abschnitt 4.4) verwendet werden, untersucht. Dazu wird sowohl eine 2D-Optimierung auf der Titan X, als auch eine 1D-Optimierung auf der Quadro P4000 betrachtet. Als Währung wird ZEC genutzt.

Jeder der Algorithmen (Hill Climbing, Simulated Annealing und Nelder-Mead) wird dreimal mit verschiedenen Startfrequenzen ausgeführt. Die verschiedenen Startpunkte liegen bei maximalen, minimalen und mittleren Frequenzen. Außerdem werden alle Algorithmen noch einmal mit der Nebenbedingung einer minimal einzuhaltender Hashrate ausgeführt. Die maximale Anzahl an Iterationen wird für alle Ausführungen auf sechs gesetzt.

In folgenden Abschnitten wird Optimierungsverlauf und Performance der einzelnen Algorithmen vorgestellt. Die Performance der Optimierung wird mit folgenden Kriterien gemessen:

- Abweichung zwischen gefundenem und echtem Optimum.
- Benötigte Anzahl an Funktionsauswertungen zum Finden des besten Wertes.

5.3.1 Performance Hill Climbing

Die Ergebnisse der Optimierung mit Hill Climbing sind in Tabelle 5 zu sehen. Die erste Tabellenzeile zeigt das gesuchte Optimum aus Tabelle 4. In der zweiten Tabellenzeile wird die gefundene Energieeffizienz (Hashes pro Joule) mit den dazugehörigen Frequenzen in Klammern für die verschiedenen Startpunkte aufgelistet. Die dritte Tabellenzeile zeigt die benötigte Anzahl an Funktionsauswertungen zum Finden der besten Energieeffizienz, sowie die Gesamtzahl der Funktionsauswertungen bis zur Terminierung in Klammern. Das Ganze ist spaltenweise für die Titan X (2D-Optimierung) und die Quadro P4000 (1D-Optimierung) zu sehen.

Abbildung 5.9 zeigt die dazugehörigen Optimierungsverläufe. Es ist jeweils der Verlauf des Algorithmus zusammen mit der zu optimierenden Funktion bei den verschiedenen Startpunkten für beide GPUs dargestellt.

Das Hill Climbing findet von allen Startpunkten einen guten Wert nahe am Optimum auf beiden GPUs. Durchschnittlich wird ein Wert von 3.98 H/J auf der Titan X gefunden. Das Optimum aus Tabelle 4 liegt bei 4.13 H/J. Auf der Quadro P4000 liegt der durchschnittlich gefundene Wert (3.31 H/J) fast beim Optimum aus Tabelle 4 (3.34 H/J). So kleine Unterschiede sind aber im Bereich der Messungenauigkeiten. Generell wird bei einer 1D-Optimierung wie auf der Quadro P4000 leichter das Optimum gefunden. Die zum Finden des besten Wertes durchschnittlich benötigte Anzahl an Funktionsauswertungen, beträgt auf der Titan X 15 und auf der Quadro P4000 fünf.

Hill Climbing ZEC		TITAN X	Quadro P4000
Optimum		4.13566 (4155,1151)	3.34324 (3802,1025)
Result	Start-Min	3.97659 (4006,1126)	3.31102 (3802,961)
	Start-Mid	4.02499 (4248,1177)	3.31719 (3802,1088)
	Start-Max	3.94662 (4024,1265)	3.31502 (3802,999)
	Average	3.98273 (4093,1189)	3.31438 (3802,1016)
Evaluations	Start-Min	8 (19)	6 (9)
	Start-Mid	18 (23)	3 (6)
	Start-Max	18 (19)	6 (8)
	Average	14.66 (20.33)	5 (7.66)

Tabelle 5: Hill Climbing: Ergebnis und Anzahl an Funktionsauswertungen der Optimierung von ZEC mit Start bei minimalen, mittleren und maximalen Frequenzen

5.3.2 Performance Simulated Annealing

Die Performance von Simulated Annealing ist ähnlich zu der von Hill Climbing auf beiden GPUs. Die Stärke von Simulated Annealing lokalen Optima zu entkommen wird hier nicht benötigt, da die zu optimierende Funktion keine besitzt. Tabelle 6 zeigt Ergebnis und benötigte Anzahl an Funktionsauswertungen der Optimierung. Der Tabellenaufbau ist wie in Abschnitt 5.3.1 beschrieben. Die dazugehörigen Optimierungsverläufe bei den verschiedenen Startpunkten sind in Abbildung 5.10 zu sehen.

Auf der Titan X wird durchschnittlich ein Wert von 4 H/J gefunden. Das entspricht einer Abweichung von 0.13 H/J vom Optimum (4.13 H/J) aus Tabelle 4. Auf der Quadro P4000 wird wie beim Hill Climbing auch, abgesehen von Messungenauigkeiten, der Maximalwert gefunden. Die durchschnittlich benötigte Anzahl an Funktionsauswertungen ist identisch zum Hill Climbing mit 15 auf der Titan X und fünf auf der Quadro P4000. Das liegt zum Teil daran, dass Simulated Annealing für die Exploration neuer Lösungskandidaten dasselbe Muster (siehe Abbildung 4.2) wie Hill Climbing verwendet.

5.3.3 Performance Nelder Mead

Die Optimierung mit dem Nelder-Mead Verfahren schneidet insgesamt etwas schlechter ab als mit Hill Climbing und Simulated Annealing. Die gefundenen Werte, sowie die benötigte Anzahl an Funktionsauswertungen sind in Tabelle 7 zu sehen. Der Aufbau der Tabelle ist wie in Abschnitt 5.3.1 beschrieben. Abbildung 5.11 zeigt die dazugehörigen Optimierungsverläufe bei den verschiedenen Startfrequenzen.

Die durchschnittlich gefundene Energieeffizienz beträgt auf der Titan X 3.88 H/J, was einer Abweichung von 0.25 H/J vom Optimum aus Tabelle 4 entspricht. Besonders bei Start mit maximalen Frequenzen wird der VRAM-Frequenzbereich nicht ausreichend exploriert und deshalb nur ein Wert von 3.72 H/J gefunden. Auf der Quadro P4000 wird, wie mit Hill Climbing und Simulated Annealing auch, der Maximalwert abgesehen

Simulated Annealing ZEC		TITAN X	Quadro P4000
Optimum		4.13566 (4155,1151)	3.34324 (3802,1025)
Result	Start-Min	4.02485 (4127,1126)	3.32891 (3802,987)
	Start-Mid	4.05375 (4036,1189)	3.31112 (3802,1063)
	Start-Max	3.9611 (4091,1126)	3.32744 (3802,1037)
	Average	4.013 (4085,1147)	3.32249 (3802,1029)
Evaluations	Start-Min	9 (16)	10 (14)
	Start-Mid	12 (19)	4 (13)
	Start-Max	24 (26)	2 (15)
	Average	15 (20.33)	5.33 (14)

Tabelle 6: Simulated Annealing: Ergebnis und Anzahl an Funktionsauswertungen der Optimierung von ZEC mit Start bei minimalen, mittleren und maximalen Frequenzen

von Messungenauigkeiten gefunden. Zum Finden des besten Wertes benötigt das Nelder-Mead Verfahren sowohl auf der Titan X, als auch auf der Quadro P4000 durchschnittlich zehn Funktionsauswertungen. Damit ist das Nelder-Mead Verfahren auf der Titan X zwar schneller (10 vs. 15 Funktionsauswertungen), dafür auf der Quadro P4000 langsamer (10 vs. 5 Funktionsauswertungen) als Hill Climbing und Simulated Annealing.

Generell ist das Nelder-Mead Verfahren bei der Exploration neuer Lösungskandidaten relativ unabhängig von der Dimension. Zwar hat ein Simplex je nach Dimension weniger bzw. mehr Punkte, aber in jeder Iteration muss, unabhängig von der Dimension, der schlechteste Punkt des Simplex ersetzt oder der Simplex komprimiert werden (siehe Abschnitt 4.3.4). Nur bei der Berechnung des initialen Simplex und bei einer Komprimierung wird pro Dimension höher eine Funktionsauswertung mehr benötigt.

Nelder Mead ZEC		TITAN X	Quadro P4000
Optimum		4.13566 (4155,1151)	3.34324 (3802,1025)
Result	Start-Min	4.01787 (4050,1177)	3.33445 (3802,1025)
	Start-Mid	3.90598 (4732,1151)	3.33988 (3802,1037)
	Start-Max	3.72097 (5549,1164)	3.34415 (3802,1037)
	Average	3.8816 (4777,1164)	3.3395 (3802,1029)
Evaluations	Start-Min	4 (12)	5 (12)
	Start-Mid	16 (18)	15 (16)
	Start-Max	11 (15)	11 (14)
	Average	10.33 (15)	10.33 (10.66)

Tabelle 7: Nelder Mead: Ergebnis und Anzahl an Funktionsauswertungen der Optimierung von ZEC mit Start bei minimalen, mittleren und maximalen Frequenzen

5.3.4 Optimierung unter Nebenbedingungen

Für die Evaluation der Optimierung unter Nebenbedingungen wurden die drei verschiedenen Algorithmen mit einer minimal einzuhaltenden Hashrate von 80% bzw. 95% der maximalen Hashrate auf der Titan X bzw. Quadro P4000 ausgeführt. Die Startfrequenzen müssen bei der Optimierung mit minimaler Hashrate immer beim Maximum liegen, denn auf Basis des Messwertes bei maximalen Frequenzen wird der Absolutwert der einzuhaltenden Hashrate ausgerechnet.

In Tabelle 8 sind die Ergebnisse, sowie die benötigte Anzahl an Funktionsauswertungen der Optimierung mit den verschiedenen Algorithmen zu sehen. Das Tabellenformat ist wie in Abschnitt 5.3.1 beschrieben. Statt der unterschiedlichen Startpunkte sind hier allerdings die verschiedenen Algorithmen dargestellt. Abbildung 5.12 zeigt die zur Tabelle gehörenden Optimierungsverläufe. Das lila Raster (Titan X) bzw. die grüne Linie (Quadro P4000) markiert den Bereich der Funktion auf dem die Nebenbedingung der einzuhaltenden Hashrate erfüllt ist. Der beste gefundene Wert für die Energieeffizienz der auf diesem Raster bzw. dieser Linie liegt ist das Ergebnis. Das Optimum unterliegt Messschwankungen und beträgt etwa 3.6 H/J auf der Titan X und 3.05 H/J auf der Quadro P4000.

Die Resultate der verschiedenen Algorithmen sind relativ ähnlich. Dennoch liefert das Nelder-Mead Verfahren etwas schlechtere Werte. Die Anzahl der Funktionsauswertungen zum Finden des besten Wertes verhält sich wie bei der Optimierung ohne Nebenbedingung auch. Das Nelder-Mead Verfahren braucht bei der 2D-Optimierung auf der Titan X weniger bzw. bei der 1D-Optimierung auf der Quadro P4000 mehr Funktionsauswertungen als Hill Climbing und Simulated Annealing.

Currency: ZEC		**TITAN X**	**Quadro P4000**
Optimum		ca. 3.6 (4755,1379)	ca. 3.05 (3802,1303)
Result	Hill Climbing	3.44178 (5421,1430)	3.05715 (3802,1341)
	Simulated Annealing	3.48682(4749,1430)	3.0806 (3802,1316)
	Nelder Mead	3.3656 (4836,1468)	3.00388 (3802,1328)
Evaluations	Hill Climbing	12 (22)	3 (6)
	Simulated Annealing	18 (21)	2 (9)
	Nelder Mead	6 (16)	8 (12)

Tabelle 8: Ergebnis und Anzahl an Funktionsauswertungen der Optimierung von ZEC mit den verschiedenen Algorithmen unter der Nebenbedingung einer minimal einzuhaltenden Hashrate

5.3.5 Verwendung eines Power-Limits

Die *NVML*-Bibliothek erlaubt es für eine GPU ein Power-Limit mithilfe der Funktion *nvmlDeviceSetPowerManagementLimit()* zu setzen. Der NVIDIA-Treiber sorgt dann

durch Beschränkung der Core-Frequenz dafür, dass der Energieverbrauch der entsprechenden GPU dieses Limit nicht übersteigt. Setzt man jetzt die VRAM-Frequenz auf das Maximum und startet das Mining, erzielt man meistens die maximale Hashrate unter der Bedingung eines maximalen Energieverbrauchs. Die maximale Hashrate wird nicht ganz erreicht, falls die Währung Compute-Bound ist, da man in diesem Fall durch Herabsetzen der VRAM-Frequenz ohne Einbußen der Hashrate Energie sparen würde. Allerdings hat die VRAM-Frequenz im Vergleich zur Core-Frequenz nur einen geringen Einfluss auf den Energieverbrauch (siehe Abschnitt 5.2), weshalb man auch in diesem Fall nahe am Optimum ist. Dieses Verfahren hat zudem den Vorteil, dass man keinen Optimierungsalgorithmus benötigt.

Tabelle 9 zeigt die erzielte Hashrate unter der Verwendung des minimalen Power-Limits von 125W auf Titan X bzw. von 60W auf der Quadro P4000 für die Währungen ETH, XMR und ZEC. Neben der erzielten Hashrate ist auch die vom Treiber eingestellte Core-Clock und der tatsächliche Energieverbrauch im Format Core-Clock|Energiebedarf|Hashrate zu sehen. Auf der Titan X werden zudem verschiedene VRAM-Frequenzen untersucht. Mit Ausnahme von ZEC wird die höchste Hashrate bei der maximalen VRAM-Frequenz erzielt. Und auch bei ZEC ist, obwohl relativ Compute-Bound, die Abweichung nicht groß.

	VRAM-Clock	**ETH**	**XMR**	**ZEC**
Titan X (125W)	4005	1683|125|24230325	1847|115|591.2	1506|125|450.6
	5005	1455|125|31828634	1809|125|767.4	1303|125|454.8
	6005	1265|125|34486963	1721|125|934.2	1151|125|425.7
P4000 (60W)	3802	620|60|17187522	1708|60|586.1	1177|60|202.9

Tabelle 9: Eingestellte Core-Clock, tatsächlicher Energieverbrauch und erzielte Hashrate auf der Titan X bei einem Power-Limit von 125W und auf der Quadro P4000 bei einem Power-Limit von 60W für verschiedene Währungen

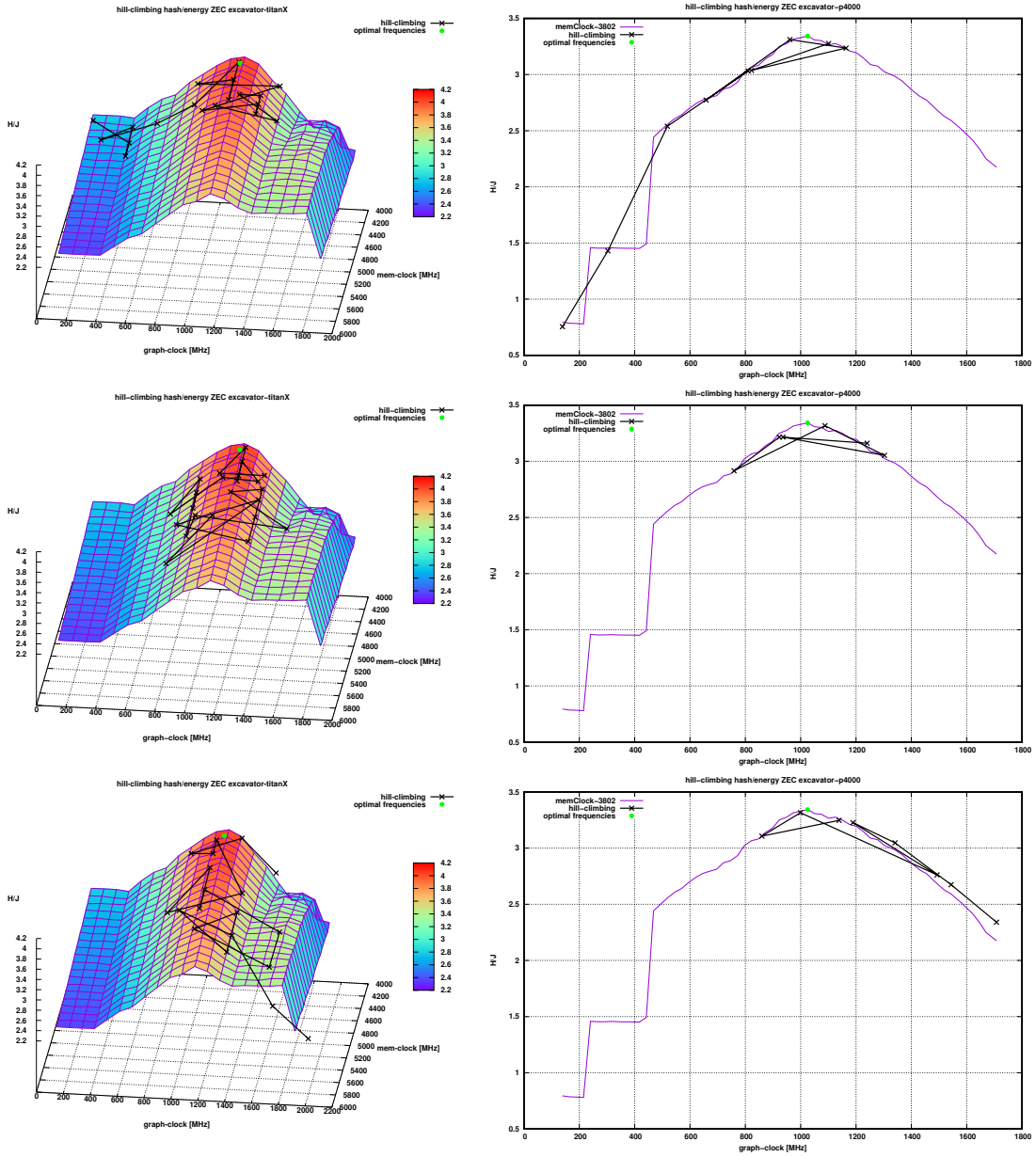

Abbildung 5.9: Hill Climbing: Verlauf der Frequenzoptimierung von ZEC mit Startpunkt bei minimalen (oben), mittleren (mitte) und maximalen (unten) Frequenzen auf der Titan X (linke Spalte) und der Quadro P4000 (rechte Spalte)

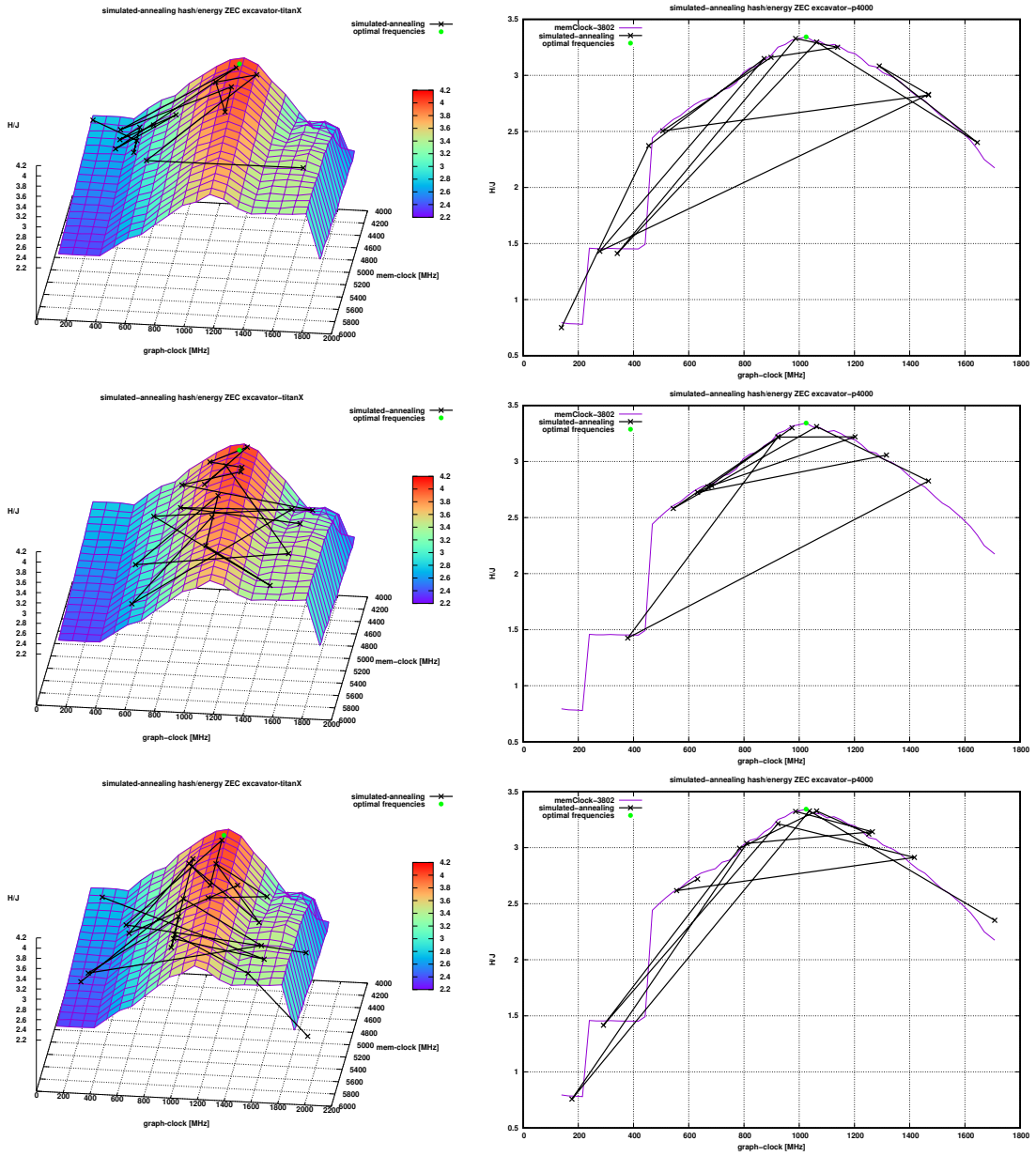

Abbildung 5.10: Simulated Annealing: Verlauf der Frequenzoptimierung von ZEC mit Startpunkt bei minimalen (oben), mittleren (mitte) und maximalen (unten) Frequenzen auf der Titan X (linke Spalte) und der Quadro P4000 (rechte Spalte)

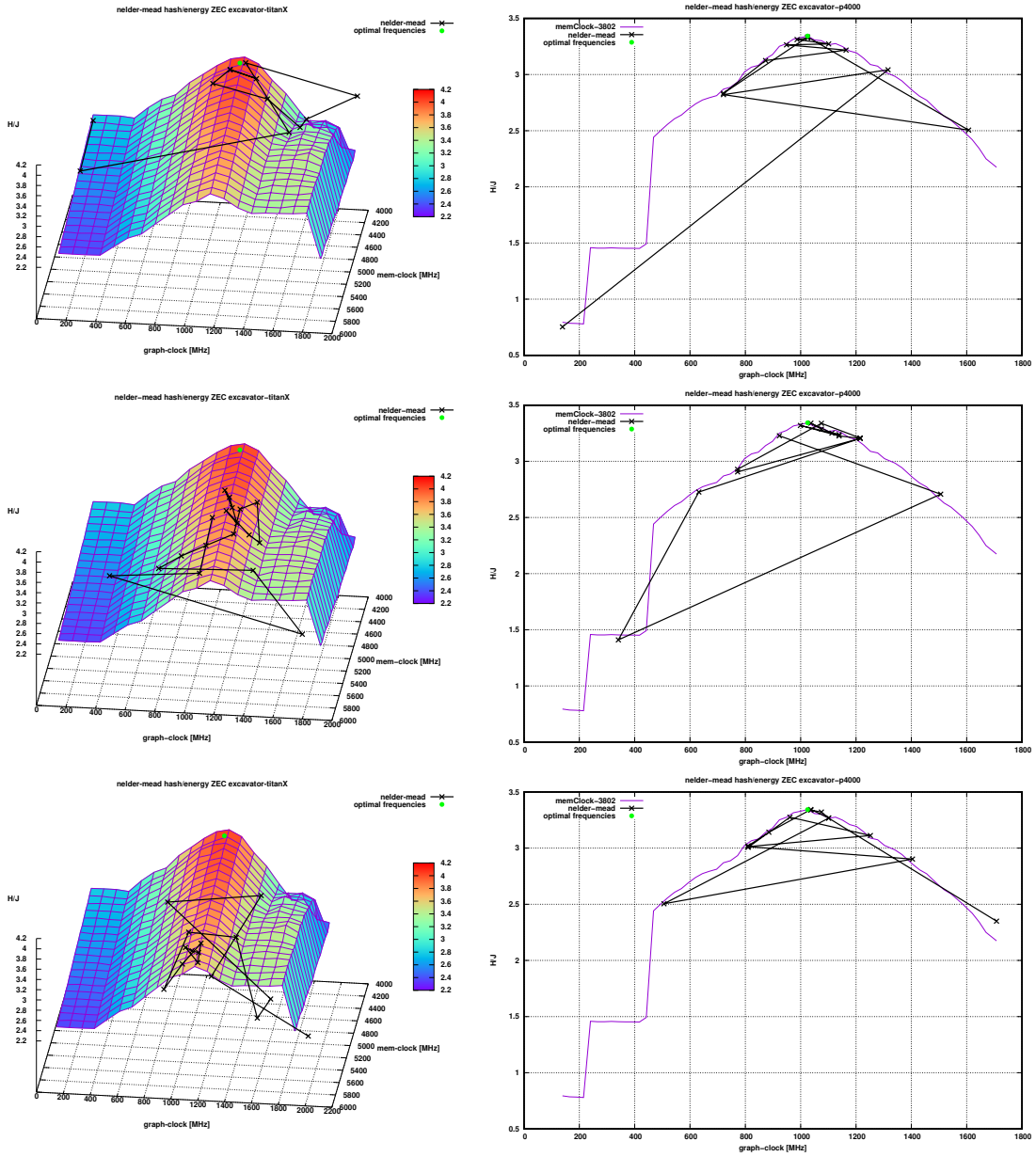

Abbildung 5.11: Nelder Mead: Verlauf der Frequenzoptimierung von ZEC mit Startpunkt bei minimalen (oben), mittleren (mitte) und maximalen (unten) Frequenzen auf der Titan X (linke Spalte) und der Quadro P4000 (rechte Spalte)

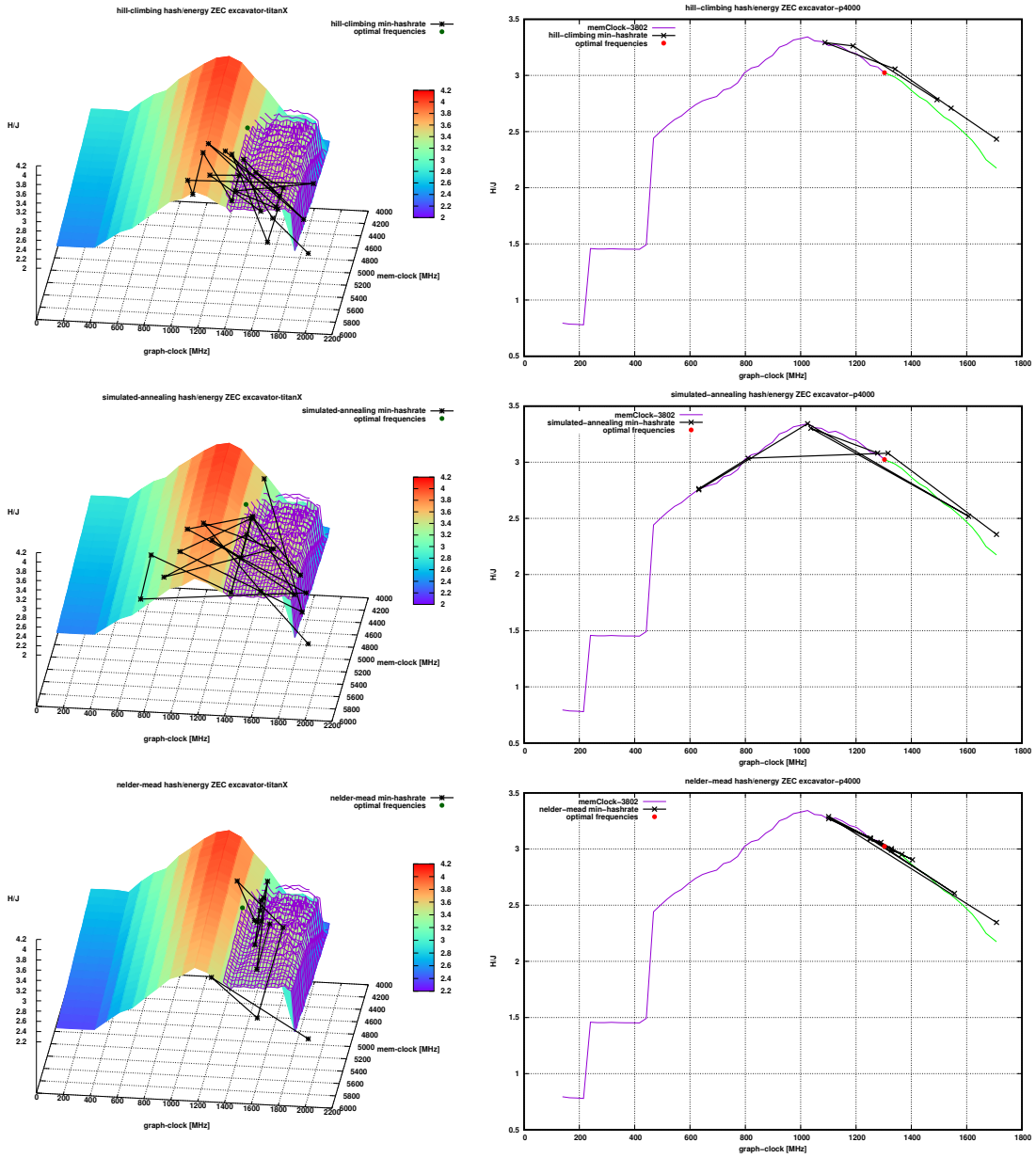

Abbildung 5.12: Verlauf der Frequenzoptimierung von ZEC mit Hill Climbing (oben), Simulated Annealing (mitte) und Nelder Mead (unten) unter der Nebenbedingung einer minimal einzuhaltenden Hashrate auf der Titan X (linke Spalte) und der Quadro P4000 (rechte Spalte)

5.4 Profitüberwachung

Für die Evaluierung der *Profitüberwachungsphase* wird das Framework auf einem Rechner mit allen vier GPUs aus Tabelle 1 im Zeitraum vom 04.10.2018 20:00h bis 07.10.2018 20:00h ausgeführt. Als Währungen wird ETH, XMR und ZEC, sowie die Altcoins LUX, RVN, BTX und VTC verwendet. Die Energiekosten werden auf 0.1 Euro/kWh festgesetzt.

Abbildung 5.13 zeigt für diesen Zeitraum den berechneten Mining-Profit bei energieoptimalen Frequenzen für jede Währung auf den einzelnen GPUs. Es wird jeweils immer nur die profitabelste Währung gemint. Abbildung 5.14 zeigt die dabei erzielten Erträge und die Energiekosten, sowie den daraus resultierenden Profit für alle GPUs einzeln und insgesamt.

In den Abbildungen ist zu erkennen, dass die Titan V die höchsten Erträge gefolgt von der Titan X erzielt. Die GTX 1080 und die Quadro P4000 liegen im Durchschnitt etwa gleichauf. Die Energiekosten sind bei Titan V und Titan X ungefähr gleich hoch, gefolgt von der GTX 1080 und der Quadro P4000, welche am sparsamsten sind. Von den Währungen wird auf der Titan V, der Titan X und der Quadro P4000 hauptsächlich ETH, und auf der GTX 1080 überwiegend LUX gemint.

Einer der Hauptkriterien für die beim Mining erzielten Erträge ist der Börsenpreis der Währung. Anfang 2018 waren die Börsenpreise vieler Währungen auf einem Rekordhoch, was zu einem Mining-Boom führte. Zum Ende dieser Arbeit, Ende 2018, sind die Preise allerdings wieder stark gefallen. Dies ist in Abbildung 5.15 zu sehen, in welcher der Börsenpreis in Euro sowie das Handelsvolumen von ETH, XMR und ZEC von November 2017 bis Oktober 2018 dargestellt ist.

Abbildung 5.16 zeigt die Auswirkung des schwankenden Börsenpreises auf den berechneten Mining-Profit während der *Profitüberwachungsphase*. Hier ist das Framework auf einem Rechner mit einer Titan X, einer Quadro P4000 und zwei GTX 1080, einmal im Zeitraum vom 02.09.2018 bis 06.09.2018 und dann nochmal im Zeitraum vom 25.10.2018 bis 31.10.2018 ausgeführt worden. Es ist zu erkennen, dass die Mining-Profite im Zeitraum vom 25.10.2018 bis 31.10.2018 deutlich niedriger ausfallen. Zudem ist auch innerhalb des Zeitraumes ein Abwärtstrend zu verzeichnen.

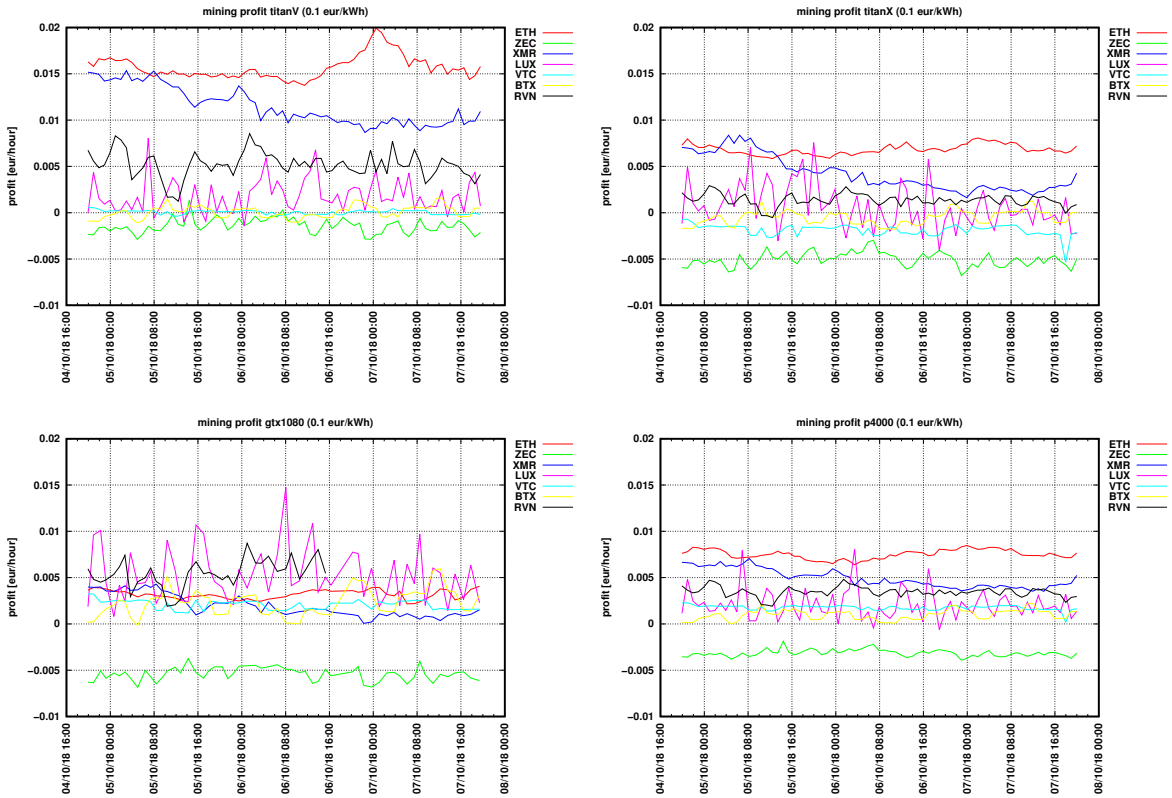

Abbildung 5.13: Berechneter Mining-Profit der verfügbaren Währungen auf den einzelnen GPUs eines Rechners während der Profitüberwachung

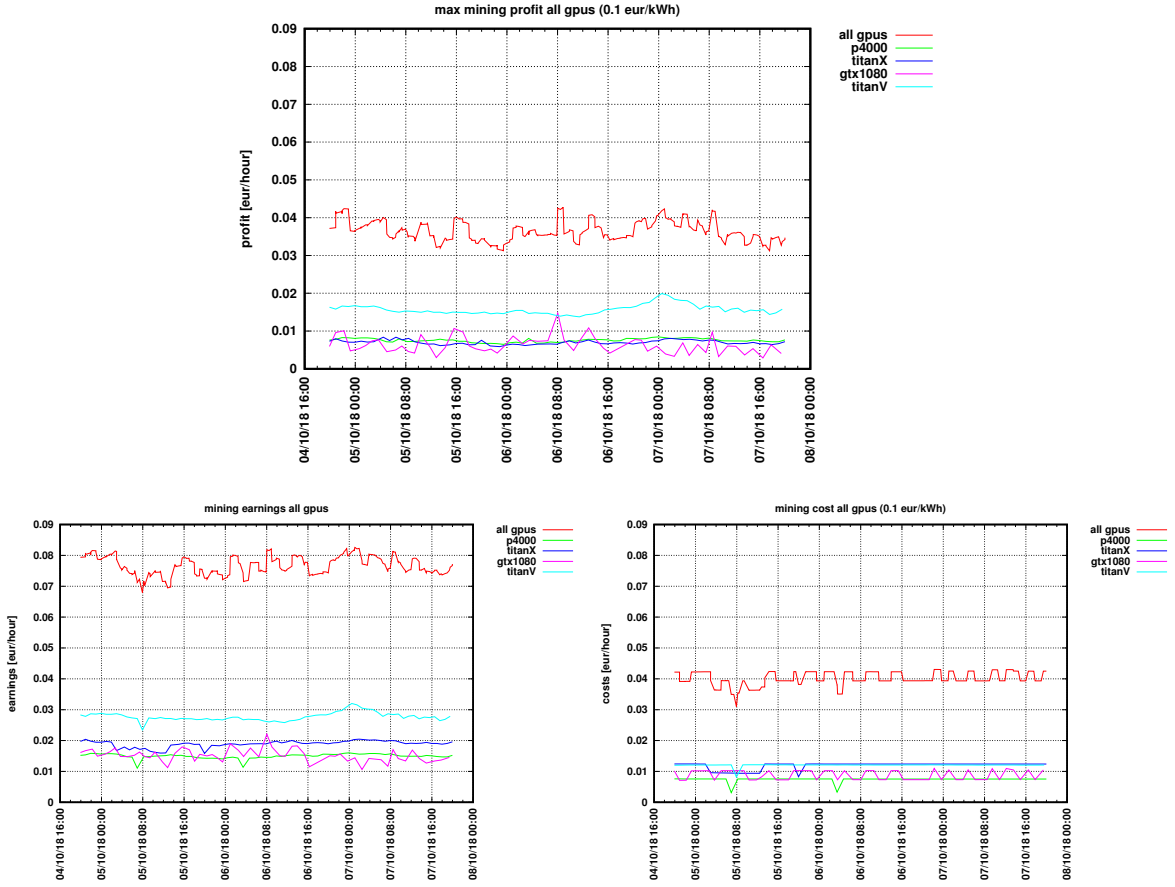

Abbildung 5.14: Berechneter Mining-Profit (oben), Miningerträge (unten links) und Energiekosten (unten rechts) der jeweils profitabelsten Währung auf allen GPUs eines Rechners während der Profitüberwachung

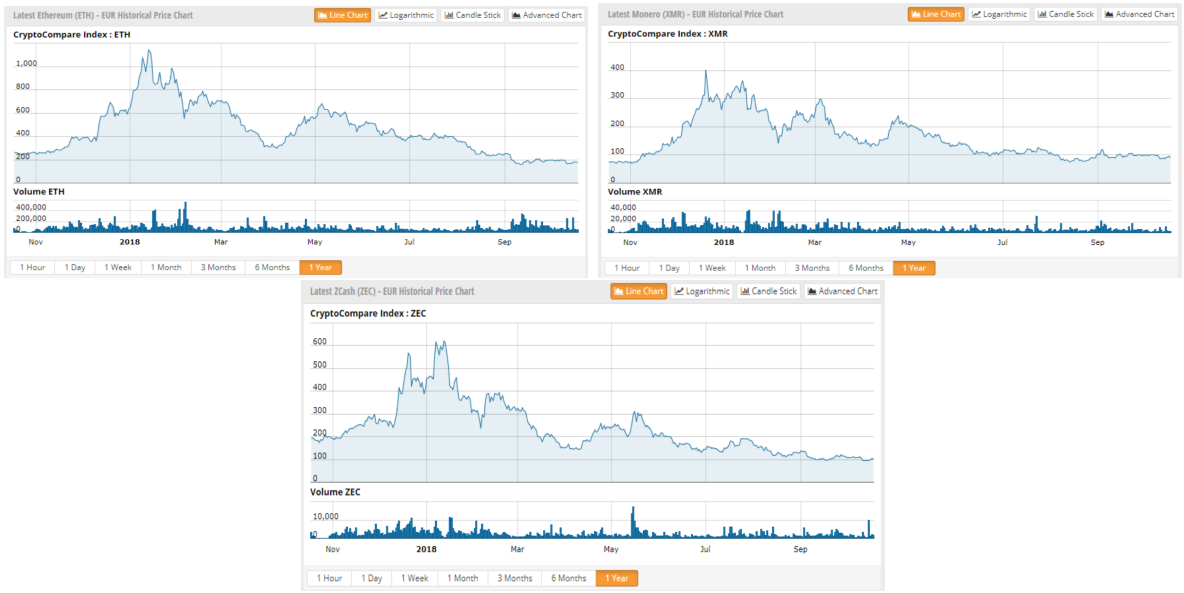

Abbildung 5.15: Börsenpreis in Euro und Handelsvolumen von ETH (oben links), XMR (oben rechts) und ZEC (unten) von November 2017 bis Oktober 2018 (Quelle: [10])

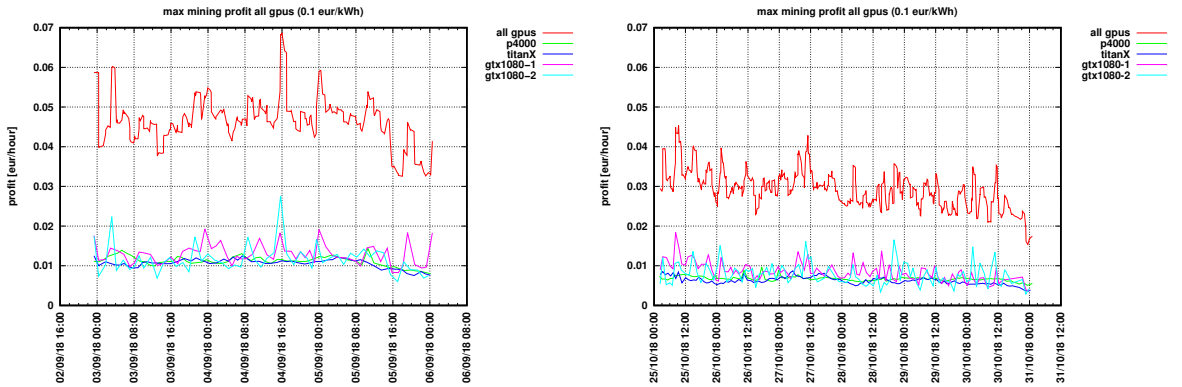

Abbildung 5.16: Berechneter Mining-Profit mit mehreren GPUs in verschiedenen Zeiträumen

6 Ausklang

Dieses abschließende Kapitel fasst die Ergebnisse der Arbeit zusammen und gibt einen Ausblick auf mögliche Erweiterungen.

6.1 Zusammenfassung

In dieser Arbeit wurde ein Autotuning-Framework zur Erhöhung der Energieeffizienz auf NVIDIA-GPUs implementiert. Dabei wurde insbesondere auf den Anwendungsfall, dem Mining von Kryptowährungen, Wert gelegt. Das Framework wurde dabei so implementiert, dass mehrere GPUs eines Rechners verwendet werden können und möglichst viele Parameter, wie die zur Verfügung stehenden Währungen, über Konfigurationsdateien angepasst werden können. Der Programmablauf ist in zwei Teile, einer Frequenzoptimierungs- und einer *Profitüberwachungsphase* aufgeteilt.

In der *Frequenzoptimierungsphase* erfolgt die Optimierung der Frequenzen parallel auf allen spezifizierten GPUs. Allerdings muss pro GPU jede verfügbare Währung einzeln optimiert werden. Das Konzept der GPU-Gruppen für gleiche GPUs schafft hier Abhilfe und ermöglicht bei der Optimierung eine Aufteilung der Währungen auf die einzelnen GPUs einer Gruppe. Um die Anpassung der Frequenzen in einem möglichst großen Bereich, sowie die Messung des Energiebedarfs auf Windows und Linux zu ermöglichen war der Einsatz von drei NVIDIA-spezifischen Bibliotheken (*NVML*, *NVAPI*, *NV-Control X*) notwendig. Für die Optimierung selbst wurden drei verschiedene Optimierungsalgorithmen (Hill Climbing, Simulated Annealing, Nelder Mead) implementiert. Diese kommen in einer auf kurzen Benchmarks basierenden Offlinephase und einer Onlinephase, während der das Mining mit Mining-Pool bereits läuft, zum Einsatz. Am Ende der *Frequenzoptimierungsphase* sind auf allen GPUs für alle verfügbaren Währungen energieoptimale Frequenzen bekannt. In der darauf folgenden *Profitüberwachungsphase* werden Energiekosten und Miningerträge bei den gefundenen optimalen Frequenzen berechnet und das Mining der profitabelsten Währung gestartet. Energieverbrauch und Miningerträge werden periodisch aktualisiert. Dabei wird der aktuelle Börsenpreis sowie die beim Mining-Pool auf Basis der eingereichten Shares erzielte Hashrate berücksichtigt. Ist die aktuell geminte Währung nicht mehr die profitabelste erfolgt ein Wechsel der Währung mit anschließender Reoptimierung der Frequenzen.

Das Framework wurde mit verschiedenen GPUs und Währungen evaluiert. Zunächst wurden die energieoptimalen Frequenzen mittels einer erschöpfenden Suche ermittelt und ein Vergleich der Energieeffizienz bei optimalen und den vom NVIDIA-Treiber eingestellten Frequenzen gemacht. Je nach verwendeter GPU und Währung wurde eine Effizienzsteigerung von bis zu 84% ermittelt. Im Anschluss wurden die verschiedenen Optimierungsalgorithmen anhand des gefundenen Optimums und der benötigten Anzahl an Funktionsauswertungen evaluiert. Abschließend wurde der während der *Profitüberwachungsphase* berechnete Mining-Profit für die verfügbaren Währungen auf einem Rechner mit vier GPUs über einen längeren Zeitraum untersucht.

6.2 Ausblick

In diesem Abschnitt werden einige Ideen für eine Erweiterung des bestehenden Frameworks vorgestellt. In der aktuellen Version des Frameworks muss für jede zu optimierende Währung der Optimierungsalgorithmus in der Miningkonfiguration eingestellt werden. Hier wäre eine automatische Auswahl des besten Algorithmus zur Laufzeit anhand von Währung und einstellbarem Frequenzbereich denkbar.

Neben der aktuellen Single-Objective-Optimierung der Energieeffizienz (Hashes pro Joule) kann noch eine Multi-Objective-Optimierung (Pareto-Optimierung) nach Hashrate und Energieverbrauch implementiert werden. Ergebnis der Pareto-Optimierung wäre dann eine Menge an Pareto-Optima (Pareto-Front). Ein Pareto-Optimum ist dabei ein Zustand, in dem es nicht möglich ist, ein Kriterium (Objective) zu verbessern, ohne zugleich ein anderes verschlechtern zu müssen.

Eine weitere Möglichkeit zum Auffinden von energieoptimalen Frequenzen ist die Erstellung eines Energiemodells. Dabei muss eine Annahme über die Form der zu optimierenden Funktion gemacht werden, hat aber den Vorteil, dass weniger Funktionsauswertungen und damit zeitintensive Messungen nötig sind.

Aktuell sind die Parameter der zu optimierenden Funktion die einstellbaren Core- und VRAM-Frequenzen. Möglich wäre auch die Launchparameter der CUDA-Kernel (Anzahl der Blöcke und Anzahl der Threads pro Block) zu verändern. Dazu muss aber der Quellcode der Miner angepasst werden. Unter Windows ist es mit der *NVAPI* zudem auch möglich die Core- und VRAM-Voltages einer GPU zu verändern. Dabei ist aber besondere Vorsicht geboten um die Hardware nicht zu beschädigen.

Besonders interessant wäre es das Framework mit anderen Anwendungen neben dem Mining von Kryptowährungen zu verwenden. Beispielsweise können statt der verschiedenen Währungen, verschiedene Implementierungsvarianten eines ODE-Solvers genutzt werden. Statt der Hashrate wird hier die Laufzeit als Metrik für die Performanz eingesetzt. Die dem Mining spezifische *Profitüberwachungsphase* entfällt dann.

7 Anhang

7.1 Installationsanleitung

7.1.1 Framework - Windows

Für das Bauen des Frameworks unter Windows sind folgende Schritte nötig:

1. Visual Studio 2015 oder neuer bzw. alternativ MinGW 5 oder neuer installieren. Bei MinGW den Pfad zu den binaries (gcc.exe, g++.exe, etc.) zur Systemumgebungsvariable PATH hinzufügen und mingw32-make.exe in make.exe umbenennen. Als IDE empfiehlt sich im Falle von MinGW CLion oder der Qt Creator, da diese eingebaute CMake-Unterstützung haben.

2. Cuda Toolkit Version 9.0 oder neuer sowie aktuellen NVIDIA-Treiber installieren. Den Pfad zu den binaries (nvcc.exe, nvidia-smi.exe, etc.) zur Systemumgebungsvariable PATH hinzufügen.

3. CMake 3.2 oder neuer installieren. Den Pfad zu den binaries (cmake.exe) zur Systemumgebungsvariable PATH hinzufügen.

4. Git for Windows installieren. Bei der Wahl der Shell im Installer die Windows-Shell verwenden, nicht die voreingestellte MinTTY-Shell (siehe Abbildung 7.1). Den Pfad zu den binaries (git.exe, bash.exe und sh.exe) zur Systemumgebungsvariable PATH hinzufügen.

5. Repository im gewünschten Ordner mit `git clone https://github.com/UBT-AI2/dvfs_gpu.git` auschecken.

6. Im Verzeichnis der Top-Level-CMakeLists.txt Folgendes ausführen:
   - `mkdir build && cd build`
   - `cmake -G "MinGW Makefiles".` . für MinGW bzw. `cmake ..` für Visual Studio.
   - `make` für MinGW bzw. generierte Visual Studio Solution öffnen und bauen.

   Alternativ Top-Level-CMakeLists.txt in IDE mit CMake-Unterstützung (Visual Studio 2017, CLion, Qt Creator) öffnen und Projekt bauen.

7.1.2 Framework - Linux

Für das Bauen des Frameworks unter Linux sind folgende Schritte nötig:

1. Cuda Toolkit Version 9.0 oder neuer sowie aktuellen NVIDIA-Treiber installieren.

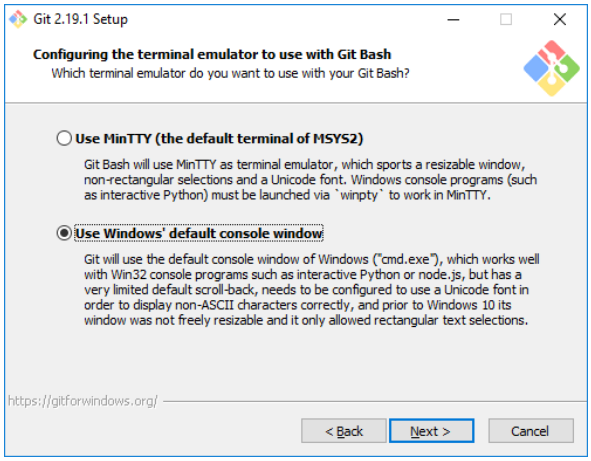

Abbildung 7.1: Wahl der Shell im Installer von Git for Windows

2. Abhängigkeiten mit Skript `intstall_linux_dependencies.sh` (Ordner utils) installieren.

3. Repository im gewünschten Ordner mit `git clone https://github.com/UBT-AI2/dvfs_gpu.git` auschecken.

4. Im Verzeichnis der Top-Level-CMakeLists.txt Folgendes ausführen:

   - `mkdir build && cd build`
   - `cmake ..`
   - `make`

   Alternativ Top-Level-CMakeLists.txt in IDE mit CMake-Unterstützung (CLion, Qt Creator) öffnen und Projekt bauen.

7.1.3 Miner

Zu den vom Framework standardmäßig verwendeten Minern sind binaries für Windows und Linux im Ordner miner vorhanden. Um die Miner mit offenem Quellcode selbst zu bauen muss das Skript `init-submodules.sh` (Ordner utils/submodule-stuff) ausgeführt werden. Das Skript initialisiert die Git-Submodule im Ordner miner und wendet die vorhandenen Patches an. Eine Anleitung zum Bauen der Miner ist in den jeweiligen Repositories der Miner vorhanden.

7.2 Benutzungsanleitung

Das Framework besteht aus mehreren Programmen, wobei das Hauptprogramm `profit_optimization` in Abschnitt 4 beschrieben wird. Die anderen Programme sind Kom-

ponenten des Hauptprogramms und erlauben das Testen und die Verwendung von bestimmten Teilfunktionalitäten (siehe Abbildung 7.2). Die anderen Programme sind:

1. `freq_optimization`: Optimierung der Frequenzen mit spezifizierten Algorithmus für eine spezifizierte Währung auf einer spezifizierten GPU. Sowohl Offline- als auch Online-Optimierung möglich. Die verfügbaren Kommandozeilenoptionen mit einer kurzen Beschreibung sind durch Ausführen von `./freq_optimization --help` zu sehen.

2. `freq_exhaustive`: Testen aller einstellbaren Frequenzen auf einer spezifizierten GPU für eine spezifizierte Währung. Sowohl Offline- als auch Online-Benchmarks möglich. Die verfügbaren Kommandozeilenoptionen mit einer kurzen Beschreibung sind durch Ausführen von `./freq_exhaustive --help` zu sehen.

3. `nvmlOC`: Setzen von Frequenzen einer spezifizierten GPU mittels der *NVML*. Ein Hilfetext zur Verwendung wird durch Ausführen von `./nvmlOC` angezeigt.

4. `nvapiOC`: Setzen von Frequenzen einer spezifizierten GPU mittels der *NVAPI* (Windows) bzw. *NV-Control X* (Linux). Ein Hilfetext zur Verwendung wird durch Ausführen von `./nvapiOC` angezeigt.

Zur Anpassung der Frequenzen benötigt das Framework root- (Linux) bzw. Administrator-Rechte (Windows). Wird das Framework unter Linux verwendet, muss in der X-Konfiguration (/etc/X11/xorg.conf) die Coolbits-Option für die GPUs gesetzt werden (siehe [36]). Wird der Linux-Rechner remote über `ssh` verwendet, muss zudem vor Programmstart das Skript `start_remote_X.sh` (Ordner utils) mit dem `source`-Befehl ausgeführt werden. Das Skript stoppt den aktuell laufenden X-Server und startet einen Neuen auf dem Display des entfernten Rechners.

7.3 Neue Währung hinzufügen

Um eine neue Währung dem Framework verfügbar zu machen muss ein entsprechender Abschnitt in der Währungskonfiguration (siehe Abschnitt 4.6.1) hinzugefügt werden. Die modifizierte Währungskonfiguration muss dann bei Programmstart mit dem entsprechenden Kommandozeilenparameter angegeben werden. Wird keine Währungskonfiguration angegeben wird eine standardmäßige erstellt und verwendet.

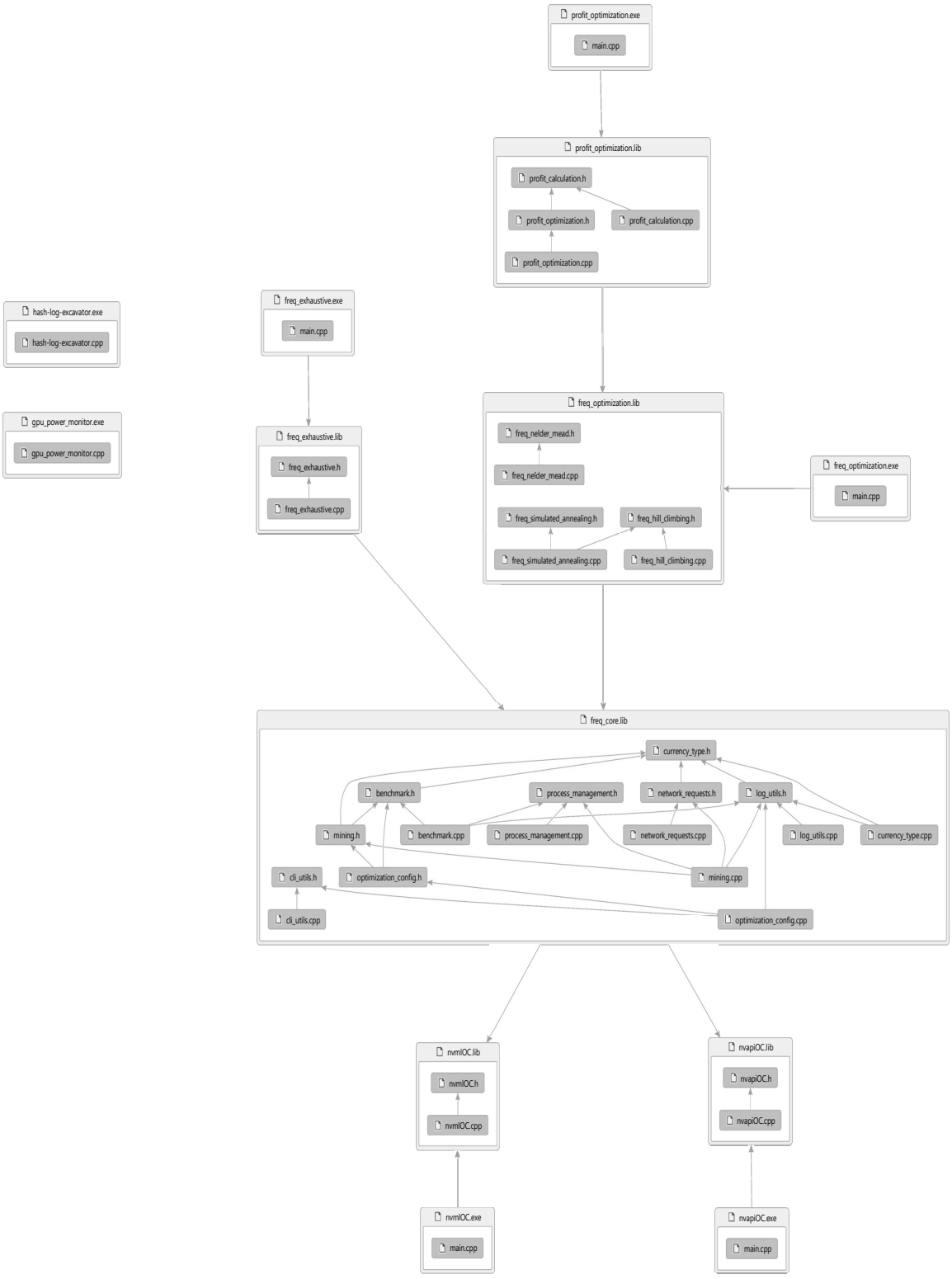

Abbildung 7.2: Struktur des Framework als Abhängigkeitsgraph

Abbildungsverzeichnis

Tabellenverzeichnis

Quellen

[1] IntelliBreeze Software AB. *Awesome Miner*. 2018. URL: http://www.awesomeminer.com/home.

[2] ETHZ ASL. *Numerical methods*. 2018. URL: https://github.com/ethz-asl/numerical_methods.

[3] Christoph Bergmann. *Kryptografie des Bitcoins für Anfänger*. 2013. URL: https://bitcoinblog.de/2013/12/22/kryptografie-des-bitcoins-fuer-anfaenger/.

[4] Blockgeeks. *What Is Hashing? Under The Hood Of Blockchain*. 2017. URL: https://blockgeeks.com/guides/what-is-hashing/.

[5] Diogo Mineiro Cameirinha. *Exploiting DVFS for GPU Energy Management*. 2015. URL: https://fenix.tecnico.ulisboa.pt/downloadFile/563345090414604/Dissertacao.pdf.

[6] Jade Cheng. *Numerical Optimization*. 2018. URL: http://www.jade-cheng.com/au/coalhmm/optimization/.

[7] Xinxin Mei; Ling Sing Yung; Kaiyong Zhao; Xiaowen Chu. *A Measurement Study of GPU DVFS on Energy Conservation*. 2013. URL: https://www.researchgate.net/publication/262365062.

[8] Bitcoin community. *Difficulty*. 2018. URL: https://en.bitcoin.it/wiki/Difficulty.

[9] Bitcoin community. *Mining*. 2018. URL: https://bitcoin.org/en/developer-guide#mining.

[10] CryptoCompare.com. *CryptoCompare API*. 2018. URL: https://www.cryptocompare.com/api/#-api-data-price-.

[11] Mirko Dölle. *Ende der Grafikkarten-Ära: 8000 ASIC-Miner für Zcash, Bitcoin Gold & Co.* 2018. URL: https://www.heise.de/newsticker/meldung/Ende-der-Grafikkarten-Aera-8000-ASIC-Miner-fuer-Zcash-Bitcoin-Gold-Co-4091821.html.

[12] Juan José Durillo; Thomas Fahringer. *From single- to multi-objective auto-tuning of programs: Advantages and implications*. 2014. URL: https://www.researchgate.net/publication/271724916.

[13] Bishwajit Dutta; Vignesh Adhinarayanan; Wu Feng. *GPU power prediction via ensemble machine learning for DVFS space exploration*. 2018. URL: https://www.researchgate.net/publication/326637320.

[14] fireice-uk. *xmr-stak*. 2018. URL: https://github.com/fireice-uk/xmr-stak.

[15] Bitcoin Forum. *Ethereum (ETH) mining profit formula*. 2017. URL: https://bitcointalk.org/index.php?topic=2262328.0.

[16] Bitcoin Forum. *PPLNS*. 2011. URL: https://bitcointalk.org/index.php?topic=39832.

[17] Sachin Joglekar. *Nelder-Mead Optimization*. 2016. URL: https://codesachin.wordpress.com/2016/01/16/nelder-mead-optimization/.

[18] Ashish Mishra; Nilay Khare. *Analysis of DVFS Techniques for Improving the GPU Energy Efficiency*. 2015. URL: https://www.researchgate.net/publication/287158852.

[19] Daniel Kober. "Erweiterung von OpenTuner zum automatischen Optimieren des Energieverbrauchs von DVFS-Prozessoren im Vergleich zu anderen Frameworks". Masterarbeit, Universität Bayreuth. 2018.

[20] Georgios Konstantopoulos. *Understanding Blockchain Fundamentals*. 2017. URL: https://medium.com/loom-network/search?q=Understanding%20Blockchain%20Fundamentals.

[21] Oak Ridge National Laboratory. *SUMMIT*. 2018. URL: https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/.

[22] Jurn-Gyu Park; Nikil Dutt; Sung-Soo Lim. *ML-Gov: a machine learning enhanced integrated CPU-GPU DVFS governor for mobile gaming*. 2017. URL: https://www.researchgate.net/publication/320850321.

[23] Nicehash. *excavator*. 2018. URL: https://github.com/nicehash/excavator.

[24] NVIDIA. *NV-CONTROL X Extension - API specificiation v 1.6*. 2018. URL: https://github.com/NVIDIA/nvidia-settings/blob/master/doc/NV-CONTROL-API.txt.

[25] NVIDIA. *NVAPI*. 2018. URL: https://developer.nvidia.com/nvapi.

[26] NVIDIA. *NVAPI Reference Documentation*. 2018. URL: https://docs.nvidia.com/gameworks/content/gameworkslibrary/coresdk/nvapi/index.html.

[27] NVIDIA. *NVIDIA Management Library (NVML)*. 2018. URL: https://developer.nvidia.com/nvidia-management-library-nvml.

[28] NVIDIA. *NVIDIA System Management Interface*. 2018. URL: https://developer.nvidia.com/nvidia-system-management-interface.

[29] NVIDIA. *NVIDIA Visual Profiler*. 2018. URL: https://developer.nvidia.com/nvidia-visual-profiler.

[30] NVIDIA. *NVML API Reference Guide*. 2018. URL: https://docs.nvidia.com/deploy/nvml-api/index.html.

[31] Stefan Oberhumer. *ethminer*. 2018. URL: https://github.com/ethereum-mining/ethminer.

[32] *OVERCLOCKING TOOLS FOR NVIDIA GPUS SUCK, I MADE MY OWN*. 2015. URL: https://1vwjbxf1wko0yhnr.wordpress.com/2015/08/10/overclocking-tools-for-nvidia-gpus-suck-i-made-my-own/.

[33] Alphonse Pace. *Chain Splits and Resolutions*. 2017. URL: https://medium.com/@alpalpalp/chain-splits-and-resolutions-d3398bddf4ab.

[34] Tanguy Pruvot. *ccminer*. 2018. URL: https://github.com/tpruvot/ccminer.

[35] Ken Shirriff. *Bitcoins the hard way: Using the raw Bitcoin protocol*. 2014. URL: http://www.righto.com/2014/02/bitcoins-hard-way-using-raw-bitcoin.html.

[36] Michael de Silva. *OC Nvidia GTX1070s in Ubuntu 16.04LTS for Ethereum mining*. 2017. URL: https://gist.github.com/bsodmike/369f8a202c5a5c97cfbd481264d549e9.

[37] TOP500.org. *TOP500 List November 2018*. 2018. URL: https://www.top500.org/lists/2018/11/.

[38] David Vorick. *The State of Cryptocurrency Mining*. 2018. URL: https://blog.sia.tech/the-state-of-cryptocurrency-mining-538004a37f9b.

[39] whattomine.com. *Coin Calculators*. 2018. URL: https://whattomine.com/calculators.

[40] Rong Ge; Ryan Vogt; Jahangir Majumder; Arif Alam; Martin Burtscher; Ziliang Zong. *Effects of Dynamic Voltage and Frequency Scaling on a K20 GPU*. 2013. URL: https://ieeexplore.ieee.org/document/6687422.

Digitale Abgabe

Die digitale Abgabe enthält die Ausarbeitung in pdf-Form mit den zur Erstellung benötigten LaTeX-Dateien im Ordner Skript sowie den aktuellen Stand des Repositories im Ordner mining.

Das Repository besteht aus folgenden Unterordnern:

- **frequency_scaling:** Enthält den Quellcode des Framework.
- **miner:** Enthält Miner für verschiedene Währungen.
- **documents:** Enthält u.a. Messdaten, Literatur und die Ausarbeitung.
- **utils:** Enthält Patches für Miner sowie Hilfsprogramme und Skripte.

Selbstständigkeitserklärung

Hiermit versichere ich, Alexander Fiebig, dass ich die vorliegende Arbeit und Implementierung selbstständig verfasst und keine anderen als die von mir angegebenen Quellen und Hilfsmittel verwendet habe. Des Weiteren versichere ich, dass diese Arbeit nicht bereits zur Erlangung eines akademischen Grades eingereicht wurde.

Bayreuth, den 07.01.2019

Alexander Fiebig