ADDIS ABABA UNIVERSITY
COLLEGE OF NATURAL AND COMPUTATIONAL SCIENCES
DEPARTMENT OF COMPUTER SCIENCE

Improving the Performance of Proof of Work-Based Bitcoin Mining Using CUDA

Seid Mehammed Abdu

A Thesis Submitted to the Department of Computer Science in Partial Fulfillment for the Degree of Master of Science in Computer Science (Network and Security)

Addis Ababa, Ethiopia
Feb 2021

ADDIS ABABA UNIVERSITY
COLLEGE OF NATURAL AND COMPUTATIONAL SCIENCES
DEPARTMENT OF COMPUTER SCIENCE

Seid Mehammed Abdu

Advisor: Dagmawi Lemma (PhD)

This is to certify that the thesis prepare by Seid Mehammed titled: "*Improving the Performance of proof of work-based bitcoin mining using CUDA*". Submit in partial fulfillment of the requirement for the Degree of Master of Science in Computer Science (Network and Security) complies with the regulations of the University and meets the acceptable standard concerning originality and quality.

Signed by the Examining Committee:

Name	Signature	Date
1. Advisor: Dagmawi Lemma (PhD)	_____	_____
2. Examiner: Solomon Atnafu (PhD)	_____	_____
3. Examiner: Minale Ashagrie (PhD)	_____	_____

Abstract

The most dominant blockchain consensus algorithm is Proof of Work (POW). It is an algorithm, which scales up the bitcoin transaction well globally, by competition a cryptographic hash function. This process is named mining. POW-based bitcoin mining is a well-known problem of computational and memory-intensive algorithms.

On the other hand, the high-threaded CUDA architecture has become with enhanced performance for a various range of computation and memory-intensive applications. Thus, the feature of a massive number of software threads with low overhead context switch provides high computational throughput and hides the memory access latencies. However, it is not effective enough for all applications because of two challenges that directly affect performance such as scheduling new threads and the overhead to start a new kernel on the CUDA. The existing work tried to model the performance of POW-based mining from various aspects. However, no model considers all of these factors came together at the same time.

The main contribution of the thesis is a combination of the POW-based bitcoin mining algorithm with a focus on the higher-level analysis of algorithm performance and lower-level details about runtime configuration (thread per block) and scheduling on CUDA.

To demonstrate the feasibility of our method, the models are validated through bitcoin block-header data from implementations of POW-based bitcoin mining using CUDA. We evaluated the performance of the models across a large variety of parameters and data values. The results indicate that the model can be effectively used on various optimization techniques. It was able to get a performance, which is almost 4 times when compared to the baseline serial algorithm of POW-based mining implementation.

Keywords: - POW, CUDA, Bitcoin Mining, Blockchain, Thread, Thread block.

Dedication

To my parent

Acknowledgments

I would like to thank Allah, who makes everything possible, for helping me pass all those hard times that I will never forget in my life.

Second, special thanks are due to my advisor **Dr. Dagmawi Lemma (PhD)**, who monitored my work and took an effort in reading and providing me with valuable comments starting from title selection to this end. Thank you very much! Besides, I would like to thank Dr. Nasir from Computer Science department at Woldia University for their willingness to give me comments always without complaint on scholarly write style and linguistic aspects, and for providing me linguistic materials and in evaluating the system.

I equally wish to extend my appreciation and thanks to Jemila D., and my friends Seid A., Dr. Wedajo M., Getachew W., Demeke G., Hiwet H., and my brother Awel Mohammed for their moral support and encouragement.

Table of Contents

List of Tables ..... **III**
List of Figures ..... **IV**
List of Algorithms ..... **V**
Acronyms, Abbreviations ..... **VI**
Definitions ..... **VII**
Chapter 1: Introduction **1**
1.1 Background of the Study 1
1.2 Motivation 4
1.3 Statement of the problem 4
1.4 Objectives of the Study 6
1.5 Methodology 6
1.6 Scope and limitations 7
1.7 Application of Results 8
1.8 Organization of the Thesis 8
Chapter 2: Literature Review **9**
2.1 Overview 9
2.2 Proof of Work-Based Bitcoin Mining 9
2.2.1 Bitcoin Address 12
2.2.2 Bitcoin Transaction 12
2.2.3 Structure of Block 13
2.3 Introduction to SHA-256 13
2.3.1 Pre-processing of SHA-256 15
2.3.2 Message Scheduler of SHA-256 16
2.3.3 Message Compression Function of SHA-256 17
2.4 The Double SHA-256 Hashing Algorithm 19
2.4.1 Bitcoin block header hashing algorithm 20
2.4.2 Hashing the Block Header function 21
2.4.3 Details of the Bitcoin Block-Header 22
2.5 Analysis of the Operations Involved in SHA-256 24
2.6 CUDA Software 25
2.7 Memory Hierarchy 28
2.8 Mapping CPU code onto CUDA code 30

2.9 Factors for Delaying and Solution Performance 37
Chapter 3: Related work 44
3.1 Unrolling loop of hash SHA-256 44
3.2 Pipelining 44
3.3 Other Different Approach 45
3.4 CUDA performance 45
Chapter 4: The Proposed Solution 49
4.1 Overview of the proposed solution 49
4.2 Proposed solution Architecture 50
4.2.1 Block DAFW 53
Chapter 5: Experiment and Evaluation 64
5.1 Introduction 64
5.2 Experimental Procedures 64
5.2.1 Data Collection 64
5.2.2 Tools and Programming Languages 64
5.2.3 Setting up the Prototype 65
5.3 Evaluation 66
Chapter 6: Conclusion and Future work 71
6.1 Conclusion 71
6.2 Future Work 72
Annexes 78
Annex A: Sample code bitcoin mining kernel 78
Annex B: Unrolled of the SHA256 compression function 82

List of Tables

Table 2.1: SHA-256 initialization hash value 16
Table 2.2: Bitcoin block-header fields along with their brief description 22
Table 2.3: Number of Operations in double SHA-256 24
Table 2.4: Memory types comparison summary 30
Table 2.5: Code snippets for mapping 33
Table 2.6: Constraints on Occupancy (for both CUDA version) 41
Table 5.1: Data on the bitcoin block-header 64

List of Figures

Figure 2.1: POW-based bitcoin mining flow chart 10
Figure 2.2: Block Header 13
Figure 2.3: SHA-256 hashing algorithm. 14
Figure 2.4: Examples of SHA-256 padded message 15
Figure 2.5: SHA-256 message compression function and message scheduler 17
Figure 2.6: The Bitcoin Block-Header Hashing Algorithm 20
Figure 2.7: The CUDA execution model. 27
Figure 2.8: An example of a kernel configuration 28
Figure 2.9: CUDA Memory Hierarchy 29
Figure 4.1: Proposed system architecture 54
Figure 4.2: Data Access Frameworks 55
Figure 5.1: Executed on CUDA platform. 69
Figure 5.2: Number of threads per-block 70

List of Algorithms

Algorithm 2.1: Mining process 11
Algorithm 2.2: A high-level overview of CPU Host pattern 31
Algorithm 2.3: Compute a CTA layout 32
Algorithm 2.4: Sample pseudocode serial and parallel code 35
Algorithm 2.5: A simple mapping four CTA layout parameter 40
Algorithm 4.1: POW-based mining algorithm 51
Algorithm 4.2: Amortized pointer indexing 58
Algorithm 4.3: Automatic loop unrolling 60
Algorithm 4.4: Manual loop unrolling 61

Acronyms, Abbreviations

ASIC: -Application-specific integrated circuit

CUDA: -Compute unified device architecture

CTA: - Cooperative Thread Array

DMA: - Direct Memory Access

ECDSA: -Elliptic curve digital signature algorithm

FPGA: - field-programmable gate array

GPGPU: - general purpose graphical processing unit

ILP: - Instruction-Level Parallelism

ISA: - Instruction Set Architecture

P2P: - peer to peer

P2PKH: - Pay-to-Public-Key-Hash

POW: - Proof of work

SHA-256: - Secure hashing algorithm 256

SIMD: - Single instruction multiple data

SM: -Streaming multiprocessors

TLP: - Thread-Level Parallelism

TPS: - Transaction per second

Definitions

Bitcoin: - A types of money completely, virtual

Bitcoin network: - A peer-to-peer payment that operates on cryptography protocol

Block: - A records of the recent bitcoin transaction, a unit of data in the blockchain that includes a hash of itself, the hash of the previous-block, and multiple transactions

Blockchain: - is a chain of blocks where each block cryptographically linked to the prior one

Block header: - is used to identify a block on an entire blockchain

Difficulty: - A network wide setting that controls what proportion of the computation is required to seek out a proof-of-work

Targeting: - A network wide re-calculation of the problem which happens once each 2016 blocks and considers the hashing power of the previous 2016 blocks

Grid: - A collection of threads

Genesis block: - the primary block within the blockchain, to initialize the cryptocurrency

Hash: - is a function that converts an input of bitcoin block header into an encrypted output of a fixed length

Host: - The execution environment that originally invokes by CUDA.

Ledger: - is a record of business transactions

Kernel Function: - An implicitly parallel subroutine that executes under the CUDA execution and memory model for each thread during a grid.

Proof-of-work: - A piece of knowledge that needs significant computation to seek out for satisfies a certain requirement. In bitcoin, miner finds a numeric solution to the double SHA-256 algorithm that meets a network-wide the problem target.

Reward: - Payment in every new block gift by a network to a miner who found the POW solution.

Secret key: - A key number unlocks bitcoin refers to the corresponding address.

Thread Block: - A thread blocks a group of threads that execute on an equivalent stream multiprocessor (SM).

Wallet: - Software holds all bitcoin address and secret key for the owner; and use to send, receive and store bitcoin.

Chapter 1: Introduction

1.1 Background of the Study

Cryptocurrency is a digital asset designed to work as a medium of exchange and it uses strong cryptography to control the creation of the additional unit, secure business transactions, and verify the transfer of asset [1, 2, 3]. Bitcoin is one of the cryptocurrencies, which are decentralized electronic payment platforms that particularly work in a peer-to-peer (P2P) network.

Bitcoin has attracted substantial interest in recent years from the overall public. It was the first established cryptocurrency, with the first trade in 2009 [4]. Since 2018, bitcoin has attracted even more attention due to an increase in value or money and volume of exchange from a global P2P network by eliminating a central trusted authority such as government, big corporation, bank, even high-tech based traditional big companies.

The success of bitcoin is the result of its features such as

- Security, since personal data are kept securely,
- Privacy, as users manage their data,
- Immutability, since records are kept permanently, and
- Transparency, due to distributed ledger in a distributed cryptocurrency rather than all of the traditional services provided by the third party[5].

Other cryptocurrencies such as Litecoin, and Ethereum, use the Proof-of-Work (POW)-based mining algorithm. The POW consensus protocol gets its success to be powered by a blockchain to control as opposed to centralized digital currency such as Master Card, PayPal, and Visa [2].

Mining is the process of verifying all transactions on the bitcoin blockchain in a P2P network, and a decentralized security mechanism for P2P digital currency[6]. A blockchain is an electronic ledger of transactions, consisting of a series of blocks of transactions, each block containing a block body and block-header, within a trusted network infrastructure[6, 7].

Each miner (node) calculates a hash over the block header to find a nonce value. For example, when the miner wants to add a new block to the blockchain, all data of the bitcoin block-header are first to perform double hashing (SHA-256) with a concatenated nonce (32-bit length). After hashing check if the result is less than the targets, then add the block into the blockchain and share for everyone in the bitcoin community (node) acknowledges the new block. If it is not less than the target, then the nonce is changing by a random number generator and this keeps iterating 4 billion times sequential until finally, the requirements are met. This technique is called POW.

One of the well-known problems of POW-based mining is that they do incur a significant computational and memory-intensive algorithm. As a result, the bitcoin network is currently capable of handling about 7-10 transactions per second[1, 6, 8]. Compared to a traditional payment systems, it supports small number of transactions per second, for example, PayPal and Visa can process up to several thousand transactions per second[9, 10, 11].

Moreover, in the bitcoin network, one of the bottleneck performance is the POW-based mining algorithm. POW algorithm in more than six hundred cryptocurrencies, which continuously contribute to more than 90% of the total market capitalization of all cryptocurrencies [12, 13]. High-performance implementation of POW-based mining algorithms is very useful in constructing a fast and secure virtual payment system[9].

For traditionally solve iteration algorithm problem on a multi-core to parallelize iteration. They are creating one thread per loop iteration to process the data from each iteration. It is a hardware solution, moreover costly.

Therefore, POW-based bitcoin mining is not an easy task with the help of single-threaded executions. Considering the massive number of concurrent threads and parallelism offered by the current emerging high-threaded CUDA platform to improve the performance.

The CUDA framework provides a data-parallel view of data with one thread per data item. However, the nested loop structures in the software are replaced

by two-level cooperative thread array (CTA), which group threads into thread blocks and thread blocks into a grid.

These propose to improve the performance of POW-based mining algorithms using massively parallel architectures, particularly Compute Unified Device Architecture (CUDA).

Researchers have designed various performance models that capture memory hierarchies for various types of high-threaded CUDA architecture such as, global memory, shared memory, and many cores, to enhance the performance of many applications. However, there is limited literature study this relation on high threaded CUDA architecture with POW-based mining algorithm[14].

Therefore, we categorize the main factors affecting throughput that crop up in our thesis in three broad groups: CUDA-memory issues, CUDA-architecture issues, and Parallel Performance Issues and describe performance metrics for throughput.

Moreover, focuses on the bridging of the POW-based mining algorithm and high-threaded CUDA platform the tradeoffs towards overall optimal performance.

Consider the general problem of the algorithm first to understand, analyze, and finally optimize the algorithm performance on a high threaded CUDA platform and hopefully guide programmers to develop a solution with limited drawbacks and high performance. This is a broad problem, not every type of application obvious similar performance pattern on high threaded CUDA platform.

Existing CPU-thread design has concentrated on reducing latency, which is the time duration between a user request and computer response. On the other hand, CUDA-thread design has focused on maximizing throughput, which is the ratio of the number of threads processed over some period. Programmers need to be aware of this throughput design focus on CUDA to achieve high-performance applications with CUDA programming.

From the theoretical side, we can effectively use the number of Instruction Level Parallelism (ILP), Data Level Parallelism (DLP), and Thread Level

Parallelism (TLP) to measure runtime assuming instructions take constant time.

For the problem space, we need to identify both input parameters and adjustable parameters in the POW-based mining algorithm. For the architecture space, we need to identify both fixed (e.g. memory sizes) and configurable parameters (e.g. threads, thread blocks, and warp) for the CUDA platform.

1.2 Motivation

Bitcoin is a blockchain-based digital currency with no central authority. Every node on a blockchain has the whole copy of a public ledger that can trust an authoritative record. The node in the network act information transmit through secure network connection arrive at the same assemble a copy of the same public ledger for everyone[7]. The bitcoin network achieves global consensus without a central authority, but due to the very nature of the computationally and memory expensive POW, certain speed performance limitations[8].

On the other hand, according to [15, 16], CUDA is a highly threaded parallel architecture that is designed for high-throughput, data-parallel computations algorithm that possesses a high arithmetic intensity[17].

This becomes a motivation for our thesis bridging POW-based mining algorithms and highly threaded CUDA architecture so that people can predict and control the tradeoffs towards overall optimal performance.

1.3 Statement of the problem

Before the invention of the Internet, the banking system was based on manual work. With the advent of the Internet the banking, the system is online systems. The identity may include customer name, account number, recipient's details, mobile number, and any identity card number. The details provided by customers to perform transactions can be acquired by hackers even when there is security in place and their accounts can be hacked. Hence, online banking has become a major security concern even though there are many advantages. To overcome this drawback, the idea of the digital currency known as cryptocurrency was introduced with bitcoin in 2009 [1]. Cryptocurrency uses

strong cryptographic algorithms to monitor decentralize and secure cryptocurrency transactions.

Cryptocurrency is a decentralized system based on blockchain as against the current banking system, which is based on a central banking system using centralized digital currency. Through POW-based mining, all financial cryptocurrency transactions are public.

One of the major challenges to the existing algorithm of POW-based mining is highly computational and memory-intensive. Due to that, the mining process takes a lot of time for block creation. As a result, a process for one block creation takes a time of average 10 minutes, transaction throughput of around 3–10 TPS, and the transaction confirmation takes time [18].

Thus, this algorithm is not fit for large networks that require vast numbers of transaction processes[8, 19]. However, the POW-based bitcoin network as a decentralized payment platform, by itself cannot support the global market anytime soon[13, 20]. Therefore, it needs to further enhance the performance of POW-based mining algorithms.

Existing improvement of POW-based mining by using special hardware FPGA and ASIC implementations were introduced. However, FPGA or ASIC mining compromises the democratization and decentralization of the bitcoin-network and its expensive hardware cost.

On the other hand, CUDA enables developers to speed up performance for computing and memory-intensive applications, but not effective enough for all applications because of two challenges that directly affect performance. First, there is no opportunity for the scheduler to schedule new threads when possible. Second, there is overhead to start a new kernel on the CUDA.

The programmer must carefully implement their multi-threaded solutions to prevent resource contention between threads that prevent problems such as, race conditions, deadlock, live-lock, starvation, etc. If the programmer is not

careful, the overhead that requires preventing resource contention can overwhelm the amount of useful work bottlenecking performance.

Therefore, no highly threaded CUDA architecture allows an infinite number of threads, and different thread/block counts may result in a POW-based bitcoin mining of diversified performance.

The research tries to answer the following questions

- What are the current states of the art performance of POW-based mining algorithms and CUDA?
- How does the variation of thread counts and thread block counts influence the scheduling and the real performance of bitcoin mining?
- How can the real performance be predicted for POW-based mining on runtime configurations?

1.4 Objectives of the Study

General objective

The main objective of this thesis is to improve the performance of POW-based bitcoin mining using CUDA.

Specific objectives

To achieve the general objective the following specific objectives are designed to:

- Conduct a detailed literature review for POW-based bitcoin mining algorithm and analyze performance improvement techniques on CUDA
- Design and develop a prototype implementing for bitcoin mining
- Evaluate the effectiveness of the design of POW-based bitcoin mining

1.5 Methodology

Based on the objectives of the thesis, the following system of principles, practices, and procedures will be applied.

The approach is a design science research-paradigm in which a designer answers questions relevant to POW-based bitcoin mining via the creation of innovative artifacts, contributing new knowledge to the body of scientific evidence[21].

Problem Identification: - First extensive review of relevant works of literature to acquire a deeper understanding, of the existing POW-based bitcoin mining and problem domains. Besides the current state of performance optimization on CUDA architecture

Define the objective of the solution: - In this step, knowledge of what are CUDA performance constraints, what is feasible.

Design and development: - Application method and theories to create a system that solves the performance problem of POW-based bitcoin mining will be carrying out.

Demonstration: - In this step, how to use the system to solve the performance problem of POW-based bitcoin on the CUDA platform will be demonstrated

Evaluation: -By using relevant metrics and evaluation techniques, the system measure well it supports a solution to the performance problem of POW-based bitcoin by comparing the objectives with observed results.

Communications: - Finally, we present the novelty, the performance, and effectiveness of the solution of POW-based bitcoin mining to researchers and other relevant audiences.

1.6 Scope and limitations

In this research, we focus on the POW-based bitcoin mining process by analyzing performance property that is implemented on many threaded CUDA architectures, which can initiate massive parallel execution paths at a low cost, compared to the existing parallel CPU implementation.

To achieve better performance, our thesis is considering CUDA architecture: how the 2-level compute system operates, how groups of threads are map onto groups of simple cores, how the memory hierarchy consists of many memory types and behaviors, and so forth. They also need to identify the resource issues imposed by the CUDA architecture and the trade-offs needed to avoid bottlenecks. Given a large parameter space of apparently equal and valid choices, they must explore these many choices to help select the best parameters for optimal balanced performance on a particular CUDA architecture.

Even though we tried our best to realize the proposed approach for improving the performance of POW-based bitcoin mining using the CUDA platform to address the shortcomings of existing works, we do not believe that the approach is generic enough to incorporate potential issues. For instance,

despite the importance of the issue, we have not considered the bitcoin networks delay in our work since it was beyond the scope of this work.

1.7 Application of Results

The main significance of the proposed is -

- To enhance the performance of the POW based bitcoin mining without changing bitcoin specification
- It is used for a decentralized application
- It reduces hardware costs
- It promotes better service to more buyers and sellers
- Stock markets are recording accurately, permanently, and transparently

1.8 Organization of the Thesis

Including this chapter, the thesis has six chapters, which are organized as follows: Chapter 2 discusses the literature review on different issues of the POW-based bitcoin mining and the general purpose of CUDA for performance concern. In Chapter 3, a review is related to the present. It discusses some works conducted relevant to this work, which have contributed that are used as a basis for this thesis. Chapter 4 presents the detailed architecture of the system. Chapter 5 deals with experimentation and evaluation to show the result of the prototype and the last chapter, which is chapter 6, presents a general conclusion, thesis contributions, and possible future works.

Chapter 2: Literature Review

2.1 Overview

Bitcoin is the most admired cryptocurrency and was invented by Satoshi Nakamoto in 2008 [4, 6, 22]. Leveraging blockchain technology, Bitcoin uses the concept of "Proof of Work protocol" to add block on blockchain to record all transaction without any third party in peer-to-peer (P2P) network. This protocol utilizes SHA-256 double hash use bitcoin block header data, contain a version number, a hash of the previous block, a hash of the Merkle root, difficult value, timestamp, target value, and a nonce to acquire the consensus among a network of participants.

Miner or node searches for a nonce value by performing the SHA-256 double hash. When resulting hash smaller than the target value to add a block on the blockchain.

Therefore, a random number generator changes the nonce when miners check if the result is not less than the target value. This keeps on repeating 4 billion times (2^{32}) sequential until finally, the requirements are met. If the result is less than the target, then it is, added to the blockchain and everyone in the bitcoin network society acknowledges the new block. The process of finding a valid nonce is, called bitcoin mining.

The success of bitcoin is inseparable from its stable network consensus, and the important step in maintaining consensus is-called proof-of-work, in which the tedious task is to repeat the SHA-256 double hash calculation.

In this Chapter, focus on POW-based bitcoin mining, SHA-256 and double SHA-256, CUDA platform, mapping CPU code onto CUDA code, and factors for delay and solution of performance.

2.2 Proof of Work-Based Bitcoin Mining

Bitcoin network, if every node tries to broadcast their blocks containing the verified transactions confusion arise. For example, consider transaction

verified by many nodes, who put them into their blocks, and broadcast it to other nodes. If the broadcasting work is free, the transaction might duplicate in several blocks, and the ledger is meaningless. To urge agreement between all nodes about the newly added block, POW requires each node to resolve a difficult puzzle with adjusted difficulty, to urge the correct to append a brand-new block to the present chain.

Specifically, before solving the puzzle, all the verifying nodes get to put their verified transaction and other information like Timestamp and Previous Hash into a block. Then they begin solving the puzzle, by guessing a nonce value, which is introduced in section 2.1, then put it into the block. All data inside the block-header are combining and inputted to double the SHA-256 hash function[6]. If the output of the function is below a given target value, which is designated by the difficulty, the nonce value is accepted.

Otherwise, the node must make another guess of the nonce value, until he gets the answer. The average speed for adding a new block in the chain is one block per 10 minutes. Also, the more difficult the puzzle is, the smaller the target value is. As shown Figure 2.1, describes the processing for handling the guessed value. This work is, called the POW. Besides, the node linking the network using POW called miners and the act of finding a suitable nonce is called mining.

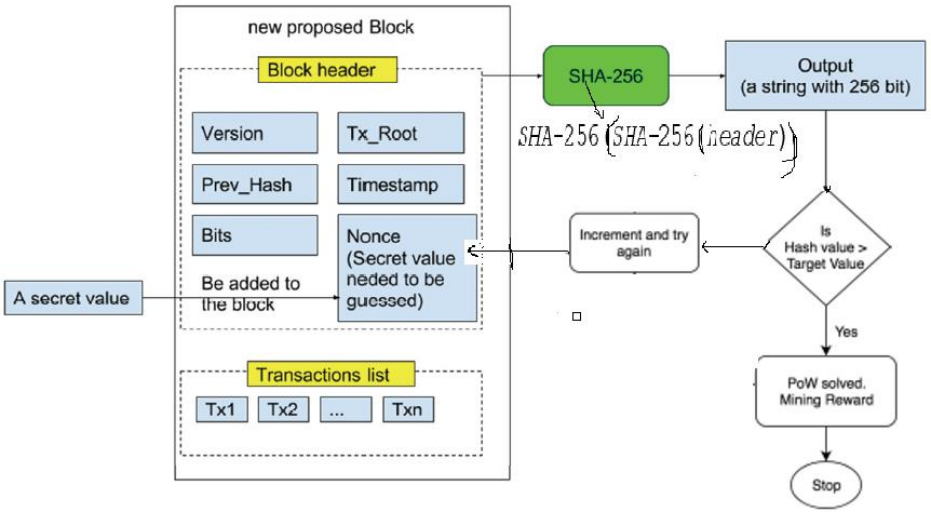


Figure 2.1: POW-based bitcoin mining flow chart

As shown in Algorithm 2.1 each miner calculates a hash over the header to find a specific nonce value. This value must be smaller than the current target. This target is shared amongst all miners. For example, when the miner wants to add a new block to the blockchain, all the contents of the bitcoin block-header are first double SHA-256 hashed with a concatenated nonce (32-bit length).

After the hash check, if the result is less than the target, then it is added to the blockchain and everyone in the bitcoin network community acknowledges the new block. This technique is called proof-of-work (POW).

Algorithm 2.1: Mining process

```
Nonce ← 0
 while nonce < 2^32 do
     target ← ((2^16 - 1) << 208)/D (t)
   digest← SHA-256(SHA-256(header))
   if digest < target then
       return nonce
       end
    else
   nonce ←nonce + 1
   end
 end
```

Once the miner calculates the correct hash value for the given block, the miner immediately broadcast the block to the network with the correct hash value for the given block and nonce. The rest of miner receive the broadcast block and lastly verify authenticity by compare hash value given in the receive block with the target value.

When, the miners the block valid add to the blockchain who solved the first POW. Then add the block into the blockchain reward with bitcoin. The reward varies with the number of Bitcoins mined. As of August 2018, the successful miners are rewarded with 12.5 bitcoin. At beginning bitcoin mining the miners were reward 50 BCT [1, 6, 23].

2.2.1 Bitcoin Address

At the foremost basic level, Bitcoin addresses, like addresses of many other cryptographic currencies, are cryptographic hash of public keys. Therefore, each address consists of a public and a personal part. General public part is address, which compared to an account number in ordinary online banking. The private key is corresponding secret key, which compared to the password, required withdraw money from a standard bank account. Addresses are often, generated by anybody as easily as public and private key pairs[6, 16].

2.2.2 Bitcoin Transaction

A Bitcoin transaction is essentially a digitally signed chunk of knowledge that is, collected into blocks, and broadcast into the peer-to-peer bitcoin network. Belay can transfer bitcoins to Almaz for her services from the bitcoins Belay earned from previous transactions. Belay could be paid by somebody else or Belay could have exchanged his fiat currency in exchange for bitcoins employing a bitcoin exchange service [1, 6, 10]. Belay could have even mined those bitcoins and gained bitcoins as transaction fees. The purpose is that bitcoin is an exchange through transactions and the authenticity of those transactions is maintain using ECDSA signatures.

Transactions are records that move Bitcoins to new addresses. Transactions typically have one or more inputs and outputs where bitcoins from the inputs are reassigning to at least one or more recipient addresses within the outputs. In each transaction, the sum of bitcoins altogether inputs must be quite to the sum of bitcoins within the outputs. The overall format of a bitcoin transaction is often found in [6].

A transaction in bitcoin consists of one or multiple inputs and one or multiple outputs. An input unlocks a previous output by providing a legitimate cryptographic signature. The inputs function proof that the holders of the respective bitcoin address that previously received the bitcoins also own the specified private key.

2.2.3 Structure of Block

Blockchain is a well-organized, back-linked list of blocks of transactions, each block consists of a block body and block header [6, 7]. The block header also contains six sets of block metadata such as previous block hash, difficulty, timestamp, nonce, and Merkle tree root a data structure used to efficiently summary all the transactions in the block . Blocks link-back, each refers to the previous block in the chain. Each block within blockchain is identify by hash, produce using the double (SHA-256) hash algorithm on the header of the block. Each block also reference previous block.

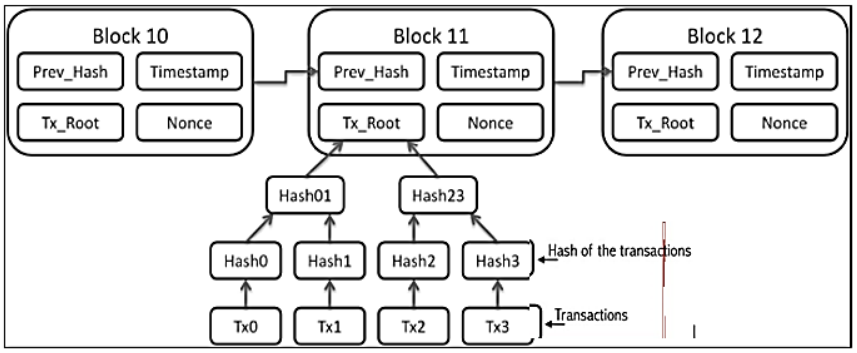


Figure 2.2: Block Header

As shown in Figure 2.2, the different fields of the block-header (80 bytes), and the associate list of transactions. The important field of the block-header from the integrity point of view is the hash of the previous block.

2.3 Introduction to SHA-256

SHA-256 algorithm are iterative one way hash function that process message to supply condensed representation is called message digest[24]. These algorithms enable the determination of a message integrity any alteration to the message with high probability ends in special message digest. Thus, the property beneficial within generate and verify digital signature and message authentication code and the generation of random number (bits).

The algorithm is focusing on two steps: **pre-processing** and **hash computation[25]**. Pre-processing involves padding message, parsing the padding message into 256-bit blocks, and setting initialization values to be utilized within hash computation.

The hash computation generate a message schedule from padded message and use the schedule together with functions, constants, and word operations to iteratively, generate a series of hash values[1, 7]. The last word hash value generated by the hash computation is utilized to work out the message digest.

SHA-256 is use hash a message M having length of L bits, where $0 \leq L < 2^{64}$ bits [26]. Thus, block size of 512-bits that are represent as a sequence of sixteen 32-bit words. This 512-bit block perform function called message compression in words of 32bits (W_t) through message scheduler. Message scheduler expands the 512-bit message block into sixty-four 32-bit words. Operations inside the SHA-256 hashing algorithm are performed on words that are 32-bit long using eight working variables names A, B, C, D, E, F, G, and H that also are 32-bits long.

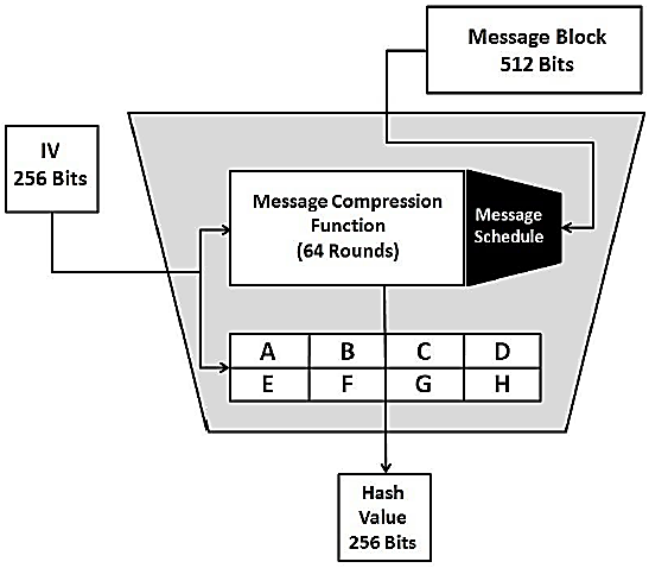


Figure 2.3: SHA-256 hashing algorithm.

The value-working variables are computed very round and process continues still 64 rounds are completed. Very importantly, it should be noted that each one addition within the SHA-256 hashing algorithm is performed mod 2^{32}. SHA-256 also takes 256-bit initialization vector (IV) which is fixed for the primary message block. After the whole message blocks are hash, worth 256 bits is obtained which is the ultimate message digest of the input message. The SHA-256 hashing algorithm is like a block cipher with a 256-bit message block size (IV) and a 512-bit key (message block) that is expanded into sixty-four

32-bit round keys using the message scheduler for each of the 64 rounds of this cipher.

The SHA-256 hashing algorithm operation can be, conveniently divided into three distinct operations as follows. Pre-processing, message schedule, and message compression function.

2.3.1 Pre-processing of SHA-256

Pre-processing of SHA-256 performs before hash computation begins. This pre-processing contain for three steps: padding the message, parsing the padded message into message block, and setting the initial hash value H_0.

- Padding the message to make it a multiple of 512-bits

The message to be hash needs to be padding first. Padding is doing to ensure that the message to be hash is a multiple of the block size 512 bits for SHA-256. Now, if we consider that the length of the message is l bits, the padding logic is such that it appends a bit "1" at the end of the actual message that is then followed by k number of zero bits. As we see in equation (2.1), k is the smallest, non-negative solution to SHA-256 padding logic.

$$l + 1 + k = 448 \bmod 512 \qquad (2.1)$$

Then append the 64-bit block equal to the number l express using binary representation. For example, as we see in Figure 2.4 the 8-bit ASCII message "ABC" length 8*3=24. The message is padded with one bit, then 448 - (24 +1) =423 zero bit, and the message length, become padded message is a multiple of 512-bits.

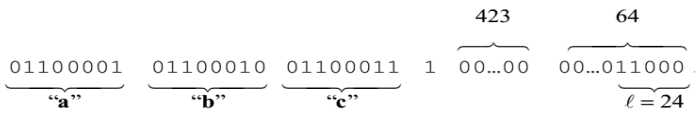


Figure 2.4: Examples of SHA-256 padded message

- Parse message into 512-bit blocks

After the message has been padded using the logic explain in equation (2.1), parse into N 512-bit block before hash computation can be start.

- Set the initial hash value

Before the hash computation begins, the initial hash value (H_0) is set which consists of the following eight 32-bit words in Hex:

It is interesting to know the origin of this 8-word value. They obtain by take the first 32 bits of the fractional portions of the square root of the first eight prime number. This initial hash value acts as the IV for the SHA-256 algorithm as explain in the earlier Figure 2.3.

Table 2.1: SHA-256 initialization hash value

H_0^0	H_1^0	H_2^0	H_3^0	H_4^0	H_5^0	H_6^0	H_7^0
0x6a09e667	0xbb67ae85	0x3c6ef372	0xa54ff53a	0x510e527f	0x9b05688c	0x1f83d9ab	0x5be0cd19

2.3.2 Message Scheduler of SHA-256

After the pre-processing step is completed, the message schedule block takes the first 256bits message block and outputs the message dependent words W_t. The 32bits message-dependent words that are output by the message scheduler for every round are labeled as W_0, W_1…, W_{63} (for t=0 to 63), and they are calculated as shown in equation (2.2).

$$W_t = \begin{cases} M_t & 0 \le t \le 15 \\ \sigma_1(W_{t-2}) + W_{t-7} + \sigma_0(W_{t-15}) + (W_{t-16}) & 16 \le t \le 63 \end{cases} \qquad (2.2)$$

Here,$\sigma_0$and $\sigma_1$are two logical functions specific to the SHA-256 message scheduler that operate on 32bits word. The details of these functions are provided logical functions σ_0 and σ_1.

$$\sigma_0(x) = \mathrm{ROTR}^7(x) \oplus \mathrm{ROTR}^{18}(x) \oplus \mathrm{SHR}^3(x)$$
$$\sigma_1(x) = \mathrm{ROTR}^{17}(x) \oplus \mathrm{ROTR}^{19}(x) \oplus \mathrm{SHR}^{10}(x) \qquad (2.3)$$

The logical functions σ_0 and σ_1 operate on a word of the input message and apply the equation (2.3) bitwise operations thereto, $ROTR_x$ stands for the bitwise rotate right for x bits, SHR_x stands for bitwise shift right and $\oplus$ stands for the bitwise exclusive. This message schedule block is typically implemented in hardware by using 16 stages of 32bits shift registers and three 32bits adders for the 512bits data block processing [24].

The visual representation of the message scheduler has been, shown in Figure 2.5. The multiplexer is, controlled by logic to permit either M_t or the computed

W_t to pass counting on the worth of t. Every round, the 32bits value of W_t is a shift to the left using the shift registers, as previous values of W_t are required to calculate future values of W_t. It is appealing to understand that two logical functions σ_0, $\sigma_1$and therefore the message schedule logic explained do not inherit play until the 17th round. The 512bits input message is, fed because it is to the message compression function for the primary 16 rounds.

2.3.3 Message Compression Function of SHA-256

The function that carries out the hashing operation of the message-dependent word that comes out of the message scheduler in each round. The actual hashing operation is that the main operation that enforces the one-way property of SHA-256. Aside from the eight words of working variables A, B, C, D, E, F, G, and H that are useful and update in each round, two temporary words T_1 and T_2 also are employed by the message compression function for computation of the variables A and E in each round.

The first step that the message compression function performs is that it initializes these eight working variables with the IV with the intermediate hash of the previous block.

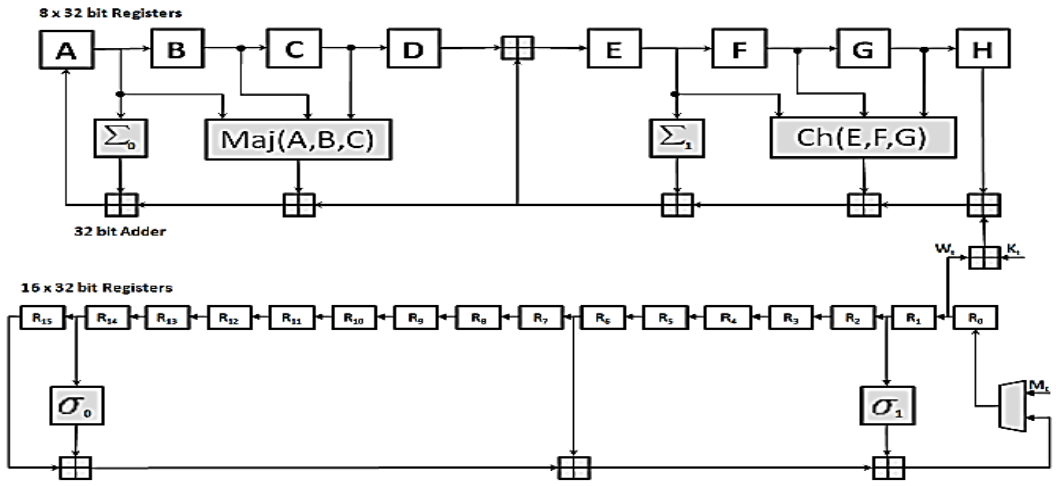


Figure 2.5: SHA-256 message compression function and message scheduler

As shown in Figure 2.5 standard implementations of the message compression function also because of the message scheduler operates in a cycle. It is often clearly seen from Figure 2.5 that at every round, 6 out of 8 values of A, B, C, and E, F, G are shifted by one position to B, C, D and F, G, H respectively. W_t

is 32bits of data calculate by the message scheduler and feed to the compression function. SHA-256 use 64 constants (1 for every round) that are 32 bits words and that they are obtained by taking the primary 32bits of the fractional parts of the cube roots of the primary 64 prime numbers. Variables A and E are hooked into all input values and are computed in each round using equations (2.4) explained next. After the eight working variables are initialized as explain earlier, 64 rounds of the compression function are apply to them and intermediate round values of variable are calculate as shown in equation (2.4)

$$\begin{aligned}
&T_1 = H + \sum_1(E) + Ch(E,F,G) + K_t + W_t;\\
&T_2 = \sum_0(A) + Maj(A,B,C);\\
&H = G;\\
&G = F;\\
&F = E;\\
&E = D + T1;\\
&D = C;\\
&C = B;\\
&B = A;\\
&A = T1 + T2;
\end{aligned} \tag{2.4}$$

The four logical functions mentioned in (2.5 perform the essential operation of present the confusion and diffusion in W_t that enters in 32 bits words at each round. After applying working variable for 64 round an appropriate level of the avalanche effect.

$$\begin{aligned}
Ch(X,Y,Z) &= (X \wedge Y) \oplus (\neg X \wedge Z)\\
Maj(X,Y,Z) &= (X \wedge Y) \oplus (X \wedge Z) \oplus (Y \wedge Z)\\
\sum_0(X) &= ROTR^2(X) \oplus ROTR^{13}(X) \oplus ROTR^{22}(X)\\
\sum_1(X) &= ROTR^6(X) \oplus ROTR^{11}(X) \oplus ROTR^{25}(X)
\end{aligned} \tag{2.5}$$

As we show in equation (2.5), logical functions Ch and Maj take three words as input and produce a single word output. ∧ stands for a 32bit bitwise AND operation while ¬ is the compliment operation. The Ch function always takes the working variables E, F, and G, as inputs while the Maj function always takes A, B, and C as inputs. Variables A and E are the ones that need to be computed at each round.

As shown in Figure 2.5 represents a different look at the compression function but it conveys the same message. After the compression function has been applying 64 times i.e. after the 64 rounds have been, completed, the values contained in the working variables A to H are finally added to the 8-word data block that was fed to the compression function at the beginning. This value could be the constant IV for either SHA-256 or an intermediate message digest. This is because the SHA-256 algorithm follows the Davies-Meyer construction where the input is, added to the output at the end. Now, the intermediate hash is giving by equation (2.6).

$$\begin{aligned} H_0^{i+1} &= A + H_0^i \\ H_1^{i+1} &= B + H_1^i \\ H_2^{i+1} &= C + H_2^i \\ H_3^{i+1} &= D + H_3^i \\ H_4^{i+1} &= E + H_4^i \\ H_5^{i+1} &= F + H_5^i \\ H_6^{i+1} &= G + H_6^i \\ H_7^{i+1} &= H + H_7^i \end{aligned} \tag{2.6}$$

After all, the message blocks including the final nth message block have been processing in equation (2.6 manner, the final hash.

$$\text{SHA-256 (M)} = H_0^N \| H_1^N \| H_2^N \| H_3^N \| H_4^N \| H_5^N \| H_6^N \| H_7^N \tag{2.7}$$

This 8-word data block (H_0^N-H_7^N) is the default constant SHA-256 initialization vector (IV) if the message was less than or equal to 512-bits (including the padding). If the length of the message (including padding) is greater than 512-bits, then this 8-word data block is the intermediate hash calculated of the previous 512-bit block. The arrangement where the intermediate hash values of the previous block are, feed as IV to the hash computation of the next block is, called the Merkle-Damgård construction. SHA-256 is based on this construction called the Merkle-Damgård Paradigm and is built to be collision-resistant as the underlying SHA-256 compression function is collision-resistant.

2.4 The Double SHA-256 Hashing Algorithm

POW-based mining uses the double SHA-256 hashing algorithm to prove new blocks in the blockchain. The method of POW-based bitcoin mining involves

mining continuously calculating the double SHA-256 hash of the bitcoin block-header and expecting an output that might be, accepted by the bitcoin network. This section will emphasize what constitutes this bitcoin block-header and the way it is constructed.

2.4.1 Bitcoin block header hashing algorithm

The bitcoin block-header hashing algorithm is explain employing a color approach. Three colors utilize to explain the speed at which these values alter relative to the method of bitcoin mining via yellow, red, and green is. The red color indicates that the data value will change the fastest.

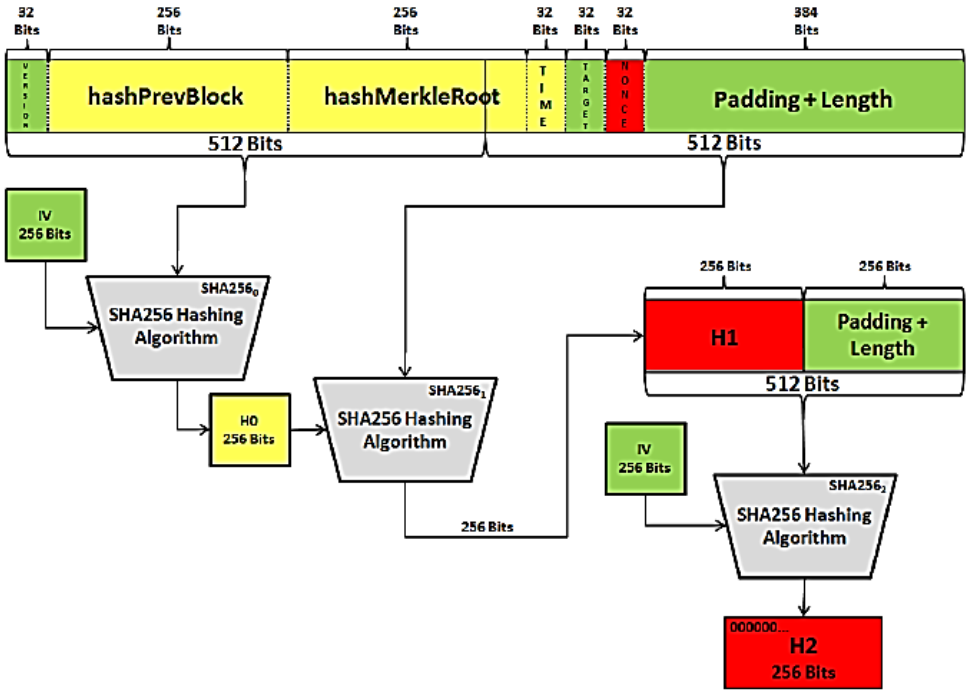


Figure 2.6: The Bitcoin Block-Header Hashing Algorithm

From Figure 2.6, it is evident that the process of bitcoin mining i.e. hashing this bitcoin block-header employs three applications of the SHA-256 hashing algorithm. The block header being greater than 512-bits in length, it is processed by two applications (SHA-256_0, SHA-256_1) of SHA-256 (one 512-bit block at a time). SHA-256_0 takes the first 512-bit block as input and after 64 rounds, produces the intermediate message-digest H0. SHA-256_0 takes the default IV on 256 bits that are, described in as well as in section 2.2.1 the default IV that SHA-256_0 uses will be constant forever, and hence it has been marked with the green color. As the calculated intermediate message-digest,

H_0 depends on inputs marked with the yellow color H_0 is also marked as yellow.

$$H_2 = \big(SHA - 256(SHA - 256((H_0 + nonce)\big) \quad (2.8)$$

SHA-256_1 use H_0 as its initialization vector and takes the next 512 bits as its input block. The red color nonce is present in the input and hence, the final message digest, H_1 produced by SHA-256_1 is marked as read. The process of hashing the block header does not stop here.

The final message digest H_1 produce by SHA-256_1 is apply through another SHA-256 hashing which name as SHA-256_2. SHA-256_2 takes the 256 bits block of H_1 as its input message block and applies suitable padding to make it a block of 512 bits. SHA-256_2 being an additional application of SHA-256 applied again uses the same default IV as used by SHA-256_0, which is marked as green. If H_2 does satisfy these constraints, the successful block with the correct nonce is, broadcast immediately in the bitcoin network for acceptance and to claim the mining reward. The bitcoin mining process thus involves the below calculation repeated potentially 4 billion times with variable nonce.

2.4.2 Hashing the Block Header function

One explanation[25] why Satoshi Nakamoto chooses to have double SHA-256 hashing is to prevent length extension attacks. The SHA-256 hashing algorithm, as all hashes constructed using the Merkle-Damgård standard, is vulnerable to this attack. The length extension attack permits an attacker who knows SHA-256(x) to perform SHA-256 (x||y) without the knowledge of x.

Thus, it unclear how length extension attack make the bitcoin protocol susceptible to harm, believe Satoshi Nakamoto decide to play safe and include double hashing design. Another explanation [8, 25] double hashing is 128 round of SHA-256 remain safe longer future practical pre-image attack found against SHA-256.

2.4.3 Details of the Bitcoin Block-Header

The block-header occasionally needs to be, updated while working on it during mining. It is important to know that the body of the block contains the actual transactions not the block header.

All the transactions contained in the block are only hash indirectly into the block-header via the Merkle root. The block header as described in the earlier Figure 2.6 and in [18, 27] essentially contains the following fields (continuing the same color-coding scheme):

Table 2.2: Bitcoin block-header fields along with their brief description

Field	Size	Description
Version	32 bits	Block version information that is based on the bitcoin software version creating this block
Hash Previous Block	256 bits	The hash of the previous block accepted by the bitcoin network
Hash Merkle Root	256 bits	Bitcoin transactions are hash indirectly through the Merkle Root
Timestamp	32 bits	The current timestamp in seconds since 1970-01-01 T00:00 UTC
Target	32 bits	The current Target represented in a 32-bit compact format
Nonce	32 bits	Goes from 0x00000000 to 0xFFFFFFFF and is incremented after a hash has been tried
Padding + Length	384 bits	Standard SHA-256 padding that is appended to the data above

- Version

These 32 bits values are an integer that represents the version of the rules that the Bitcoin software follows to create a new block. The current value is two and has changed ever since; BIP00349 was, accepted in July 2012.

- Hash Previous Block

This is the 256-bit H_2 of the previous block that was, accepted by the Bitcoin network. By including this value in the new block header, the miner tries to

further, extend the longest POW chain as explained in the block-header occasionally needs to be, updated while working on it during mining. It is important to know that the body of the block contains the actual transactions, not the block header.

Table 2.2, if accepted, the miner will hence be award the current mining reward of 25 BTC along with the transaction fees that were included in the individual transactions held by the block. As the bitcoin protocol is, designed such that the network generates a new block approximately every 10 minutes, it is safe to assume that on average, the hash Previous Block needs to be, updated after around every 10 minutes.

- Hash Merkle Root

Hash Merkle Root is the 256-bits value of the Merkle Root as explained in section 4.3.2. Like hash Previous Block, hash Merkle root will typically on average change in around 10 minutes and hence even this is marked in yellow. There is another scenario where hashes Merkle root changes and this will be, explained ahead in section 2.3.4.

- Timestamp

This 32-bit value is the current time in seconds since 1970-01-01 T00:00 UTC. Considering the hashing rate of current miners, 1 second is relatively a large timeslot. Thus marked the timestamp field as yellow; indicating that it changes but relatively rarely.

- Target

Target change after every 2016 new block which take about 2 week[27, 28].

- Nonce

These 32 bits values are the only value in the block header that is the volatile as it changes on every attempt of the double hash on the block header. Thus, mark the nonce field in red. The nonce starts at zero and it is incremented strictly in a linear manner for each H_2 attempted. Finding H_2 that will be, accepted by the bitcoin network. Thus, there are only 2^{32} possible values for the nonce.

- Padding + Length

For $SHA256_1$ padding + length is 384 bits long, while for $SHA\text{-}256_2$ it is 256 bits long. As the specification of SHA-256 is known and the length of the input message to $SHA\text{-}256_1$ and $SHA\text{-}256_2$ is fixing i.e. 640 bits and 256 bits respectively; the padding + length field will always remain constant, hence marked these in green.

This transaction grants the mining reward and the transaction fees to the miner once the network accepts the block. As this generation transaction is unique, hash Merkle root is generally unique for all miners and every hash calculated by a miner has the same chance of solving the block as every other hash calculated in the entire Bitcoin network. Winning the Bitcoin lottery is getting harder every two weeks as the network hash rate is constantly on the rise, which is driving the difficulty up as well[18]. Each miner payout based on the hashing rate contributed to the cluster.

2.5 Analysis of the Operations Involved in SHA-256

Table 2.3: Number of Operations in double SHA-256

Additions (Mod 2^{32})	= (7*64) + (3*48) + 8 = 448 + 144 + 8 = 600X2=1200	(Message compression) + (Message scheduler) + (Intermediate/computation)
Bitwise Rotations (ROTR)	= (6*64) + (4*48) = 384 + 192= 576 =576x2=1152	(Σ_0, Σ_1) + (σ_0, σ_1)
Bitwise Shifts (SHR)	= 2*48= 96 =96x2=192	σ_0, σ_1
Bitwise AND ($\wedge$)	= 5*64= 320 =320x2=640	Maj, Ch
Bitwise EX-OR ($\oplus$)	= (7*64) + (4*48) = 448 + 192= 640 =640x2=1280	(Message compression) + (Message scheduler)
Total Operations	= 600 + 576 + 96 + 320 + 640= 2232 =4464	

As shown in Table 2.3, is the analysis of the number of various operations performs, by the SHA-256 algorithm on a 512-bit message block over 64 rounds. This analysis is imperative for the quantitative element analysis wiped out the discussion a part of this text that calculates the number of operations to the suggested optimizations.

The bitwise rotations and the bitwise shift operations involve just re-arranging the input word[25, 29]. The additions involving seven operands within the calculation of working variable A also forms the longest data path or the critical path. Hence, if the number of additives performing within the compression function even by a little amount; optimize the method to an excellent extent.

2.6 CUDA Software

A CUDA program consists of two parts: - The primary part of the program, which is executed on the host (CPU), and the kernel, called the second part program, which is executed on the CUDA device by set of CUDA threads in parallel [30, 31]. The host part uses CUDA API calls to send different commands to the CUDA.

Currently use API for CUDA programming:-

Compute Unified Device Architecture (CUDA):-This parallel computing architecture developed by Nvidia and released begins 2007, provides both a low-level and high-level API. It only works on CUDA enabled graphic cards from Nvidia and is an extension of the C programming language. The CUDA drivers need to run an executable compiled with CUDA.

The main advantages of this architecture are-

- Code can read from arbitrary memory addresses,
- Full support for integer bitwise operations
- Fast-shared memory, which acts as a user-managed cache, can be used by threads residing on the same multiprocessor

The main disadvantage of CUDA is the fact that it is only available for Nvidia graphic cards that support it.

The CUDA Programming Model

The CUDA programming model enforces cooperation between hardware, software, and different tools to compile, link, and run CUDA code. This section will describe the compiler model; the mapping from the hardware to the software (the execution model) and the way the memory on the CUDA is often used. Moreover, the CUDA programming model also contains a set of compilers and tools to optimize CUDA code[32, 33].

- Compiler Model

The CUDA programming models help two ways of writing programs. This research only targets the CUDA C variant, which a minimal set of extensions to the C language. A source file can contain a mix of host code and device code. The nvcc compiler first separates the host and device code. Whereas the host code then compile and link by the programmer favorite compiler, the device code is compiled by nvcc to an assembly form. Assembly code converted to binary form, which is termed a cubin object or loaded by the appliance at runtime and compiled via the just-in-time compilation mechanism.

- Execution model

Both the CUDA hardware and framework are design by Nvidia; these two interact all right with one another. The important aspect to know how CUDA works is that the mapping between the API and the hardware. The software and hardware entities within the programming model are, shown in figure 2.9.

- **Kernel**: - A function that runs on the device. As mentioned earlier, the CUDA architecture relies on the SIMT principle. this implies that each thread processor executes a similar function in parallel
- **Device**: - The CUDA hardware runs the code
- **Host**: - The CPU hardware controls the CUDA

Thread Block and Multiprocessor: - Every thread block contains a pre-defined number of threads. A thread block is then executing by one multiprocessor. Several concurrent thread blocks can reside on one multiprocessor, limit by multi-processor resources like share memory and the

number of available registers. Threads within a thread block can cooperate and synchronize, while threads in several blocks cannot.

Thread and Thread Processor: - A thread runs a kernel that is executing by the thread processor. Every thread contains a unique id.

• **Grid and Device**: - Every grid contains a predefined number of thread blocks, so a kernel is launching as grid of thread block.

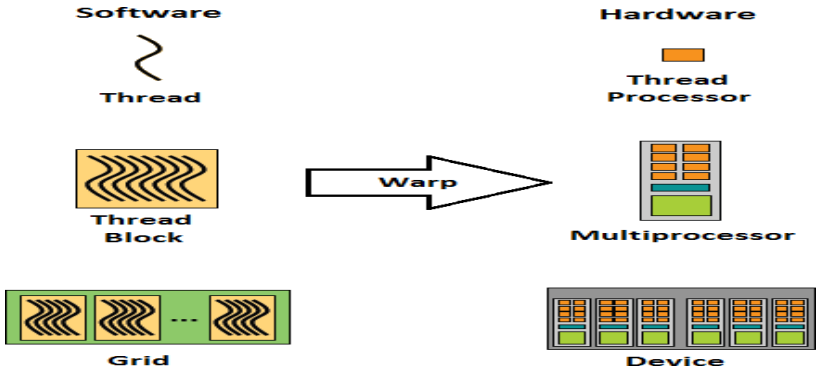


Figure 2.7: The CUDA execution model.

The number of threads during a thread block and the number of thread blocks during a grid are often determined at compile time or runtime. This permits programs to transparently scale to different CUDA and hardware configurations. The hardware is liberal to schedule thread blocks on any processor as long because it fits within the warp size. When multi-processor is give one or more thread block to execute, then partition them into warps that are schedule by warp scheduler for execution. Block is a partition into warps. Each warp contains threads of consecutive, increasing thread IDs with the primary warp containing thread 0. The configuration of a kernel (in a variety of grids, thread blocks, and threads) significantly influences the speed. It is important to divide the threads evenly over every warp.

As we show Figure 2.8, gives a summary of how threads, thread blocks, and grids are often, defined. Over time, every kernel is execute counting on its configuration. This configuration is describe by the number of threads Per Block, the number of blocks per grid, and the number of static memories needed.

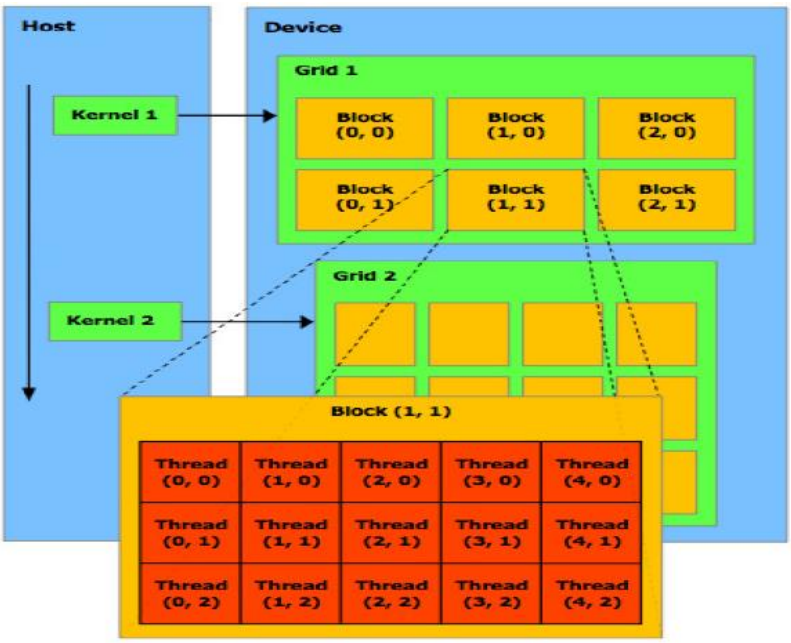


Figure 2.8: An example of a kernel configuration

2.7 Memory Hierarchy

Modern CUDAs have several memories that can be, accessed from a single thread. A memory hierarchy has several levels of areas where a thread can place data. The CUDA has thousands of registers per block. The registers are at the first and the preferable level, as their access time is 1 cycle.

The number of register per thread is calculating at compile time. If you decide to partition your program such that there are 256 thread per block, and that there are four thread per block, you get 65536/ (4*256)=64 registers per thread on the block.

Each block contains a small amount of very fast on-chip memory that can be, accessed from the thread running at the block. It is mainly, used for data interchange within a thread running on a block. This memory is, called local or shared. Local memory acts as a user-controlled L1 cache. On modern GPGPUs, this on-chip memory can be, used as a user-controlled local memory or standard hardware-controlled L1 cache. On blocks with the Tesla microarchitecture, there is 16kB of local memory and no L1 cache. Local memory has around one-fifth of the speed of registers [31].

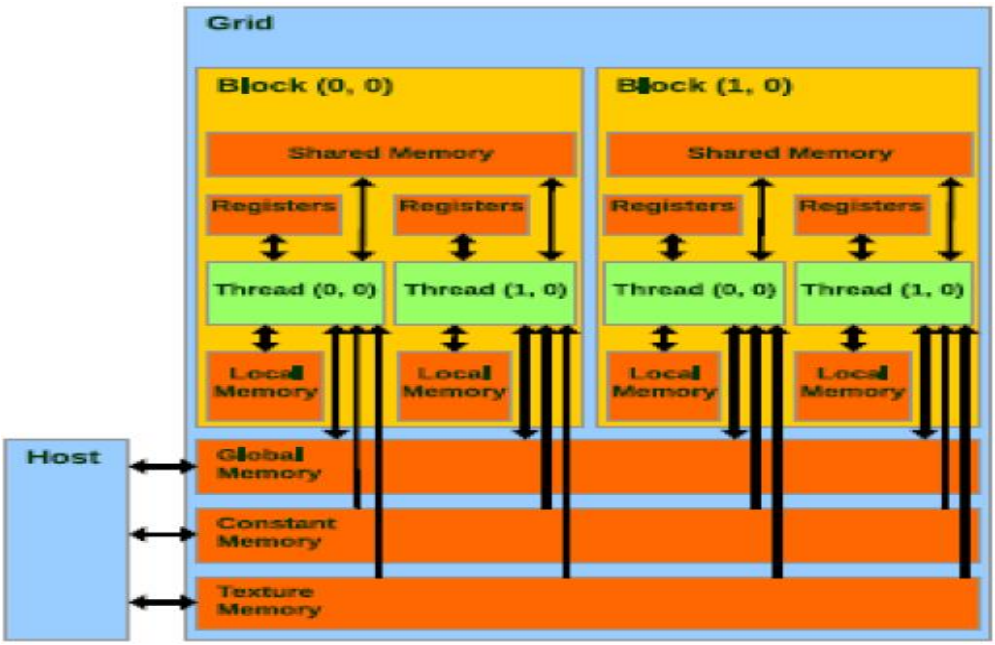


Figure 2.9: CUDA Memory Hierarchy

The largest memory space on CUDA is the global memory. The global memory space is implemented in high-speed graphics dynamic memory, which achieves very high bandwidth, but like all memory, has a high latency. CUDA global memory is global because it is accessible from both the CUDA and the CPU. Global memory resides in device DRAM and it is used for transfer between the host, and device for the data input and output from the thread running on blocks. Reads and writes to global memory are always initiated from the block and are always 128bytes wide starting at the address aligned at 128bytes boundary. The blocks of memory that are accessed in one-memory transactions are called segments. This has an extremely important consequence. If two threads of the same warp access two data that fall into the same 128-bytes segment, data are, deliver in a single transaction[33].

- Memory coalescing

Refer to a combination of multiple memory access into a single transaction. Block of thread into warps is not relevant to computation but global memory accesses. The CUDA device coalesces global memory load and store issue by thread of a warp into few transactions to possible to minimize DRAM bandwidth. On the recent CUDA, every successive 128bytes memory can be

access by a warp in a single transaction. However, there are big differences between memories as we see in Table 2.4

Table 2.4: Memory types comparison summary

Name	Size	Type	Access cycle	Speed	Alloc. Place
Registers	8192MP	R/W	1 cycle	Fast	MP
Shared Memory	16Kb	R/W		Fast	MP
Local Memory	Devi. Mem.	R/W	1-32 cycle	Slow	DRAM
Constant Memory	64Kb	R	~500 cycles	Fast	DRAM
Texture Cache	16Kb/MP	R	~500 cycles	Fast	DRAM
Global Memory	The device is Mem.	R/W	~500 cycles	Slow	DRAM

MP = Multi-Processor

2.8 Mapping CPU code onto CUDA code

Programmers wishing to migrate CPU code onto a CUDA must learn how to do tasks in the CUDA architecture[32, 33, 34]. For a data transfer, the parallel threads do not need to communicate or coordinate.

In this section, demonstrate four basic tasks by section: - In section 1, we discuss how launch CUDA. In section 2, discuss how can specify thousands of parallel threads for each kernel launch using CTA layout parameters. In Section 3, show how mapping each unique thread specify by CUDA built-in CTA parameters onto a data element. In Section 4, show how can convert a CPU copy using sequential iteration into a thread-based CUDA kernel using data-parallel programming.

1. **How does one setup and launch a CUDA Kernel?**

The program “Hello world” to run on the CUDA platform, a programmer must write CPU support code that sets up, launches the CUDA kernel from the CPU host, and receives the result. So assuming that we have a simple CUDA kernel for data transfer in parallel, how do we launch it from a serial CPU program? As we show Algorithm 2.2, to launch a CUDA kernel, the CPU host must allocate resources, transfer inputs onto the CUDA, setup kernel CTA layout

parameter, launch kernels, transfer outputs back from the CUDA, and clean up allocated resources.

Algorithm 2.2: A high-level overview of CPU Host pattern

Input: Varies as needed, for example: n input data items (for copy)
Output: Varies as needed, for example: n output results (for copy)

```
// CPU Host Pattern
1. <Optional> – Choose & set up CUDA device
2. Set up CTA layout (grid of thread blocks) for each kernel
3. Allocate memory resources on CPU & CUDA
4. <Optional> – Initialize / Set up CPU arrays
5. Transfer inputs onto CUDA from CPU
6. Launch GPU kernel(s) with parameters (Template, CTA, Kernel)
7. Transfer outputs from CUDA onto CPU
8. Tear down memory resources
```

To launch CUDA kernel from CPU host can three separate sets of parameter:-

- Kernel parameters, which are the normal function parameters passed into the code, as the number of data elements n
- CTA layout parameters, which structure the threads into blocks and grids
- First specify template parameters in C++ using single angle brackets (< … >)

2. How does launch a kernel with thousands of threads?

One obstacle to programming a CUDA kernel is the need to specify upfront a thread layout structure, CTA that the CUDA hardware uses to schedule the tens of thousands of threads onto hundreds of CUDA cores. Each kernel no matter how simple requires a CTA layout, but choosing good sizes and shapes for the CTA layout requires the programmer to develop some intuition for what thread layouts work well, and optimal sizes and shapes require many experiments within the specific application.

Each thread belongs to thread warp, which belongs to a thread block, which belongs to a grid. To launch kernels with thousands of threads, CUDA platform requires that programmers essential specify, at CUDA kernel launch, two triples ‹x,y,z› the 2D grid of blocks and 3D array of threads in the blocks. Use a 6-tuple ‹gw, gh, 1, bw, bh, bl› to represent the block and grid layout parameters as a single unit.

However, CUDA uses a triple angle-bracket syntax (`<<<grid, block,…>>>`). Moreover, use the 6-tuple ‹bx, by, bz, tx, ty, tz› to indicate the unique thread location of each thread within the CTA layout.

To specify the CTA layout upfront, the CUDA programmer need plan how to partition data and how to map individual thread onto data element. The programmer can specify 1D grid and 1D thread block layout, with two CTA layout parameter ‹gw,1,1,bw,1,1›, but the option of having multiple CTA layout parameters allows programmers the flexibility to treat the underlying data as 1D, 2D or 3D, as appropriate for their problem space. Recommend CUDA kernels are 1D, but large, so typically use 2-3 CTA parameter per kernel as 1D or 2D grid and 1D block as ‹gw,gh,1,bw,1,1›.

As we show Algorithm 2.3, a 2-3 parameter CTA layout for a 1D data set [0, n). The user take a fixed-size thread-block size (TBS) and fixed size amount of work per-thread (nWork) from which the code compute a fixed-size data block size (DBS = nWork·TBS). From the fixed-size data block size, compute number of data block (m=$\lceil n/DBS \rceil$) need to fully cover the data set [0, n).

Algorithm 2.3: Compute a CTA layout

```
template<TBS, gridRS, nWork> // Input Size, Row Size(m*c),
                                Work per-thread
Layout_1D( n, Grid, Block )        // Input Size, Grid layout, Block Layout
  DBS = nWork*TBS;              // Fixed, Data Block Size
  nBlocks = ⌈n/DBS⌉;            // Varies, Cover data with data blocks
  if (nBlocks <= 65534)        // 1D or 2D Grid Layout?
    gridW = nBlocks;
    gridH = 1;
  else
    mSQ = ⌈√nBlocks⌉;           // Start with a square layout
    gridW = ⌈mSQ/gridRS⌉·gridRS;  // nCols is a multiple of 'Row Size' Hint
    gridH = ⌈nBlocks/gridW⌉;      // nRows needs to cover data
  end if
  Block = dim3( TBS, 1, 1 );       // Block layout (1D)
  Grid = dim3( gridW, gridH, 1 ); // Grid layout (2D or 1D)
end Layout 1D

// Example Usage
...
TBS    = 128;                 // 128, Pick a fixed 1D thread block layout
nWork = 4;                    //   4, Amortize costs across work-items
nSMs   = 14;                  //  14, number of SMX's on a GTX Titan
nConBlocks = 16;              //  16, expected number of concurrent blocks per SMX
gridRS = nSMs * nConBlocks;  // 224, A good starting row size for my grid
Layout_1D<TBS,gridRS,nWork>( n, Grid, Block ); // Compute my CTA Layout
```

CTA Layout Procedures

Cooperative thread array (CTA) layout parameters result in scarcer mapping operations. Consequently, prefer 1D layouts over 2D layouts, and 2D layouts over 3D layouts. To support the multi-issue CUDA hardware capability, prefer to have at least two or four thread warps per thread-block (64 or 128 thread per thread-block). Recommend including

the fixed-size CTA parameters as part of the kernel C++ template parameters. This approach supports experimentation while still allowing the CUDA kernels to know the actual CTA layout at compile time for better performance.

3. How do map threads onto data?

To process all data, CUDA programmer map individual threads onto unique data locations in the data set. All threads collectively fully cover all data in the data set. Therefore, the programmer must combine these CTA layout parameters with the current threads unique identifying parameters to map the threads access request onto a single data location.

Mapping for each of the six code snippets is computed in four steps:

- The block ID (bid) for a thread block within a grid is first perform from the `blockIdx` and `gridDim` parameters.
- The thread ID (tid) for a thread within a thread block is then perform from the `threadIdx` and `blockDim` parameters.
- Next, the fixed-size DBS can be perform from fixed-size C++ template parameters (nWork and TBS) or the `blockDim` parameter
- Finally, per-thread data location from the bid, tid, and DBS

	1D Block	2D Block <bw,bh,1>	**3D Block** <bw,bh,bl>

1D Grid <gw,1,1>	bid=blockIdx.x;
	tid=threadIdx.x;
	TBS=blockDim.x; DBS=WPT*TBS;
	dataOff=(bid*DBS)+tid;
		bid=blockIdx.x;
		tX=threadIdx.x;
		tY=threadIdx.y;
		bW=blockDim.x;
		bH=blockDim.y;
		tid=(bW*tY)+tX;
		TBS=bW*bH;
		DBS=WPT*TBS;
		dataOff=(bid*DBS)+tid;
			bid=blockIdx.x;
			tX=threadIdx.x;
			tY=threadIdx.y;
			tZ=threadIdx.z;
			bW=blockDim.x;
			bH=blockDim.y;
			bL=blockDim.z;
			slab=bW*bH;
			tid=(slab*tZ)+(bW*tY)+tX;
			TBS=slab*bL;
			DBS=WPT*TBS;
			dataOff=(bid*DBS)+tid;

Runtime Ops	5	9	13
Template Ops	3	5	7

Table 2.5: Code snippets for mapping

In contrast, the “Template Ops” rows show the reduced number of operations expected when redundantly passing some of the CUDA layout parameters as fixed-size C++ template parameters into the kernels.

Each mapping code snippet consists of simple loads, multiply, and adds. Any require CUDA CTA layout parameters (gridDim, blockIdx, blockDim, threadIdx) must first be loaded from special constant read-only registers into normal registers before they can use.

4. How can write CUDA data-parallel code?

For a serial solution, many programmers, when given a large data set that needs to be processed, naturally draw upon the "iteration pattern" from their programmer toolbox. A nested loop structure that iterates over all data values easily processes all data in a large dataset.

The CUDA programming platform enhances the C++ language with a few extensions to express and support data-parallel programming. Each CUDA kernel is written from the point of view of a single generic thread (within the CTA). Each CUDA data-parallel kernel then typically consists of the following six steps:-

- **1: Mapping Data:** This step, programmers write code that uniquely identifies the current thread and maps the current thread ID onto a specific data location (input and output).
- **2: Setting Up: -** An optional step programmers write code that can allocate, load, or create any necessary variables to support algorithm state and context
- **3: Loading Input:** - During programmers write code that loads a single data item as input from the computed input location. Range checks may be required to prevent out-of-range memory accesses.
- **4: Transforming Data: -** During this important step, programmers write code that transforms the input data-item into an output result.
- **5: Storing Output: -** During this step, programmers write code that stores the result item as output to the computed output location.
- **6: Cleaning Up:-** optional step programmers write code to clean up any allocated resources from the "Setting Up" step

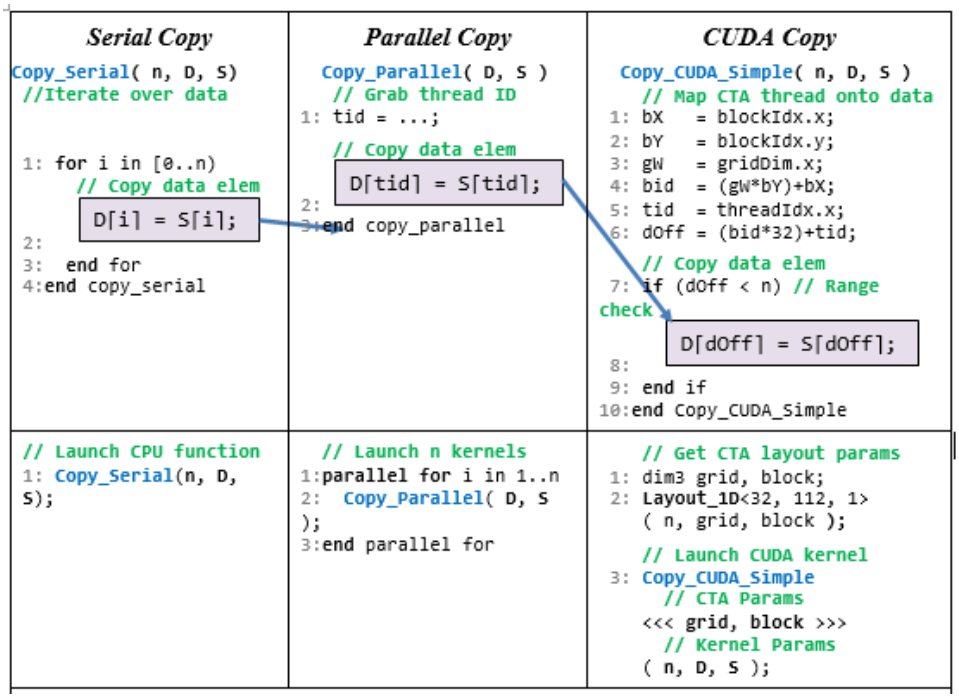
The transforming data is the important payoff step where the original problem is solving. The loop instruction are necessary and useful overhead for solving

a large problem in a serial environment, the other step above (1-3, 5-6) are also necessary and useful overhead to solve a large massively multi-threaded problem in a CUDA data-parallel environment.

As we, shown in Algorithm 2.4, Top Row, Left Panel a simple serial CPU code to data transfer n elements from source to destination, using a loop. Top Row, Middle Panel is an equivalent data-parallel kernel would data transfer n elements using n threads.

As also can be seen from Algorithm 2.4, (left vs. right panel), the loop overhead of the CPU serial function has been replaced by the CTA layout and mapping overhead of the CUDA data-parallel kernel. The resulting parallel CUDA copy is conceptually the same as the parallel multi-computer kernel.

Algorithm 2.4: Sample pseudocode serial and parallel code



To support mapping at runtime, CUDA provides four CTA layout parameters. The gridDim and blockDim layout tuples refer to the original CTA layout parameters. The threadIdx and blockIdx tuples uniquely identify the current running thread within the current thread block within the grid. Each threadIdx and blockIdx parameter value ‹x,y,z› is upper-bounded by a corresponding blockDim and gridDim value. All four built-in CTA layout parameters are therefore used to map individual threads onto their corresponding data items.

This approach is similar to mapping multi-dimensional arrays onto 1D-memory layouts.

Mapping the blockIdx or threadIdx onto a single unique block or thread ID (bid or tid) involves computing 3D slabs, 2D rows, and 1D columns and adding them all up. As we show in Table 2.5, code snippets that can be used to map the CTA layout variables onto a single per-thread data location using 1D or 2D grid vs. 1D, 2D or 3D block layout.

Later, when they are coding the CUDA kernel, programmers must map the CTA layout (as four built-in kernel parameters) for each thread down onto a single data location. This mapping code shows up in Algorithm 2.4, (top-row, right-panel) as source lines 1-6.

To support data parallelism in CUDA programs, at one time write your CUDA kernels in ten stages as follows:-

- **First, identify the structure:-** Conceptually, look at the actual serial algorithm to adopt
- **Find "Body":-** find the innermost code in the actual looping structure. Important code transforms one input element into one output result. This innermost transform code forms the core of CUDA kernel.
- **Write core:** - Based on the innermost transform code, write the core of the CUDA kernel that loads, transforms, and stores a single data work-item.
- **Map Threads:** - Add code that maps thread into unique data work-item.
- **Serial Test:** - Verify kernel with exactly one thread on one data-item.
- **Increase parallelism to one Thread-Warp:** - Verified that code works for all threads within a single warp. Meaning set your CTA layout to a single thread warps (32 threads) and tested on a single data warp (32 data items).
- **Coordinate Threads:** Consider how the threads within a thread block need to communicate and coordinate with other threads to ensure correct parallel behavior. Insert barriers instructions or use atomics to ensure that all threads are reading or writing correct data without conflicts. Then

rewrite your code to support the desired communication or coordination across threads.

- **Increase parallelism to one Thread-Block**: - Verify that your code works for all threads within a single thread block (128 threads per thread block is a good starting case) on a single data block (128 data elements).
- **Increase parallelism to an entire Grid of thread blocks**: - Verified that your code works for all threads in a large grid of thread blocks.
- **Complete Algorithm:** The kernel should now be working correctly.

In general, still need to think about how to communicate and coordinate partial results across threads within a thread block and between kernels in a multi-kernel parallel algorithm.

Conclusion

In this section 2.8, we have discussed the four main tasks: - writing CPU host functions to launch CUDA kernels, launching thousands of parallel threads using the CTA layout, mapping each parallel thread within the CTA onto a data work-item, and writing thread-centric code for data-parallel kernels. CUDA programmers should learn to get a CUDA kernel up and running. Demonstrated these four tasks on a data transfer primitive.

2.9 Factors for Delaying and Solution Performance

How the contrast between the CPUs latency focus and the CUDAs throughput focus leads to different choices for researcher as they map tasks onto different architectures[35]. Consequently, CPU-thread design has concentrated on reducing latency, which is the time duration between a user request and computer response. In contrast, CUDA-thread originally design has focused on maximizing throughput, which is the ratio of the number of thread processed over some period. Programmers need to be aware of this throughput design focus on CUDA to achieve high performance application with CUDA programming.

Therefore, categorize the main factors affecting throughput three broad groups: CUDA-memory issues, CUDA-architecture issues, and Parallel Performance Issues and next describe performance metrics for throughput.

CUDA Memory Factors

Memory constraints, register spills, coalescence, and bank conflicts are all CUDA memory architecture issues that affect the performance of an algorithm.

- Memory Constraints

CUDA programmers aiming for high performance must understand several concepts related to memory: - access times, limited capacity, and cache constraints. As discussed in section 2.7, CUDA memory is a multi-level hierarchy of registers, shared memory, and global memory.

Access Speeds: - The consecutive levels of memory access time that differ by an order of magnitude:Registers≪Shared memory≪Global Memory≪CPU RAM [36].

Cache Constraints: - Fermi and Kepler architectures both support caching but for a limited number of memory controller and small caches. Since there are up to 16 SM on the Fermi class CUDA version and up to 14 SM on the Kepler class CUDA version but only 2-6 memory controller, the SMs must compete to use the memory controllers. As a result, data in CUDA caches get evict much more frequently data in CPU cache. CUDA programmers not depend on their data stay in CUDA cache for long. Therefore, wish to migrate essential data into registers, shared memory or both.

- Register Spills

Though programmers frequency write code if they have unlimited registers, real CPUs and CUDA-cores have a limited number of register available for each thread. On CUDAs, the maximum number of register available is set up at kernel launch time based on the register size the number of concurrent threads running on each SM. The CUDA compiler stores spilled variables into local memory, which is a reserved area of global memory, which gives orders of magnitude slower access.

- Coalescence

When 32 threads in a warp request to access global memory the CUDA memory controller must transfer the requested data to or from register in the

SM that are assigned to the respective threads. The memory controller does this transfer in units of data warps (128 bytes).

Distinctness can be relaxed but several threads computing for the same memory address can lead to “race condition” where one thread overwrites a computing thread data. The hardware prevents race conditions in share memory at the cost of serializing access for competing threads within a warp.

- Bank Conflicts

Bank conflicts occur when multiple thread within the same thread warp access the same memory bank within shared memory at the same time. Memory access involving k threads accessing the same bank at the same time are called k-way bank conflicts.

CUDA Architecture Issues

Five CUDA architecture issues influence parallel thread performance: - CUDA scheduling stall, CUDA scheduling constraints, occupancy, CTA partitioning, and branch divergence.

- CTA Partitioning

The cooperative thread array (CTA) is 2-level hierarchy (thread blocks within a grid and thread within a thread block). To take advantage of the massive data-level parallelism via threads on a CUDA programmers need a way to specify the desired thread structure and thread to data mapping.

- For each CUDA kernel launch, CUDA programmers must structure the desire shape and size of the CTA layout using two 3D parameters (grid and block) both specify as 3-tuples ‹x,y,z›.
- Inside each CUDA kernel, CUDA programmers map CTA layout parameters represent unique threads onto actual data location. The CTA layout parameter are available to programmer inside each CUDA kernel via four global read-only 3D parameter (gridDim, blockDim, blockIdx, threadIdx) as 3-tuples ‹x,y,z›.

When first introduce to CUDA programming, found the CTA layout and mapping step confusing. Therefore, look in detail on how to perform the CTA layout and mapping step:-

CTA layout: - CUDA requires CTA layout specifies as part of each CUDA kernel launch. The grid and block CTA layout parameter are specify as 3-tuples ‹x,y,z› that represent the ‹width, height, length› of a 3D grid or thread block layout. To refer both the grid and block dimensions as one unit, we combine both 6-tuple denoted by ‹gw,gh,gl, bw,bh,bl›.

CTA mapping: - Inside of each CUDA kernel, programmers must map individual threads onto data locations. To help with this mapping process, CUDA provides four global read-only CTA parameters as 3-tuples (threadIdx, blockDim, blockIdx, gridDim). These CUDA variables are always available for use by the code anywhere inside the CUDA kernel, including nested function calls.

Algorithm 2.5: A simple mapping four CTA layout parameter

```
// Map Block ID (bid) from <gridDims, blockIdx>
 gW = gridDims.x     gH = gridDims.y     // gL = gridDims.z (not used)
bX  = blockIdx.x     bY = blockIdx.y     // bZ = blockIdx.z (not used)
bid = (gW*bY) + bX      // 2D to 1D mapping

// Map Thread ID (tid) from <blockDims, threadIdx>
 bW = blockDims.y   bH = blockDims.y   bL = blockDims.z
 tX = threadIdx.x   tY = threadIdx.y   tZ = threadIdx.z
tid = (bH*bW*tZ) + (bW*tY) + tX  // 3D to 1D mapping

// Map Thread Index from <bid, tid>
 TBS = bH*bW*bL         // Thread Block Size
tIdx = (bid*TBS)+tid    // 2D to 1D mapping

// Map Data Offset from <tIdx>
dataOff = ...           // Problem Domain Specific
```

The gridDim and blockDim parameters refer back to the original CTA layout (size and shape) parameters specified at kernel launch, with the ‹x,y,z› dimension values meaning ‹width, height, length›, respectively.

- Scheduler Stalls

CUDA programmers realize that during each instruction fetch, register, or access is slower than a processor cycle. The latency of those operations cause instruction stalls while each instruction waits on its input. During this instruction stall, the compiler and scheduler together attempt to make sure that

a thread warp has other useful work to try to avoid thread stall. Thread stall are frequent

Short terms stalls come from compiler- or scheduler- induce delay insert to avoid structural and control hazard that otherwise could jeopardize correct program behavior.

A RAW dependency means one instruction consumes the output of a previous instruction as input and thus must wait until the prior instruction completes. RAW dependency is real and honored for proper program behavior.

Long-term stalls many cycles come from servicing I/O transfer requests. When stall occurs, hardware scheduler on each SM can hide latency by rapidly switching, up to once per cycle, a stalled thread warp for an additional thread warp that is ready to execute. Programmers by writing code that launches thousands of concurrent threads to support Thread-level Parallelism(TLP).

Kernel with higher occupancy have better performance because, when a current warp stall, provide SM warp scheduler more active warp to settle on. ILP can hide latency[36] advise the performance of some kernel can decrease as occupancy increases above 50%[35].

- Scheduler Constraints on Occupancy

CUDA define occupancy for a kernel the ratio of the number of active thread warps scheduled on an SM to the direct limit of active warps for that SM type.
Register Size Constraints

Table 2.6: Constraints on Occupancy (for both CUDA version)

Constraints on Occupancy CUDA						Bytes	Bytes
Chosen Threads/block	Max.Occu-pancy %	Max. Blocks	Max. Warps	Max. Threads	Max regs /thread	Max Sh.Mem. /block	Max Sh.Mem. /thread
32	.17	8*	8	256	63*	6,144	192
64	.33	8*	16	512	63*	6,144	96
128	.67	8*	32	1,024	32	6,144	48
256	1.00	6	48	1,536	21	8,192	32
512	1.00	3	48	1,536	21	16,384	32
1,024	.67	1*	32	1,024	32	49,152	48

The SM warp scheduler cannot exceed the register size and will throttle back the number of thread blocks that it makes active For instance, suppose that the

CUDA programmer chooses to have 128 thread per block, and the CUDA compiler decides that the code needs 37 registers per active thread.

As we show on Table 2.6 shows the maximum occupancy possible given a chosen number of thread per thread block for Fermi and Kepler CUDA architectures.

- Shared Memory Size Constraints

Each SM has a limited of shared memory spread across all active thread blocks. This shared memory has a maximum limit of 48KB on both Fermi and Kepler architectures. The scheduler cannot exceed the shared memory size and will decrease the number of thread block makes active. Suppose for example, that a CUDA programmer writes a kernel that consumes 10 KB of shared memory for a thread block of 128 threads.

Parallel thread Performance factors

Metrics are useful in measuring performance and understanding the issues that affect performance. Three main issues influence parallel performance: parallel overhead, and load balancing.

- Parallel thread overhead

To minimize coordination costs across parallel threads to achieve better performance. Parallel thread overhead is the ratio of the time spent coordinating parallel work between threads versus the total time it takes to solve the original problem. A system has low parallel overhead if the amount of time spent coordinating algorithmic behavior across parallel threads is small.

- Load Balance

Parallel utilization is a measure of how many parallel cores out of all core in a system are busy doing useful work at any given time.

Therefore, many techniques to achieve optimal load balancing focuses exclusively on equal partitioning, and an approach that divides (n) data elements across (p) threads into approximately equal-size runs (n/p).

Measuring Throughput Performance

To understand throughput and performance bottlenecks, first need to be able to measure performance accurately. The CUDA platform provides machine counters, from which metrics for throughput and speedup can be, derived. Some of throughput metrics: - I/**O**, **instruction**, and **data throughput**.
To understand what are the bottleneck issues affecting performance, programmers start with certain known values and then take experimental measurements to derive metrics that provide insight. Value known by programmers include the input size (n) and output size (m).

The CUDA profiler timer's record useful measurement such as timing, number of parallel-core (p), and total-threads (t), including instruction issue (II) and the average instructions retire per machine cycle (IPC). From value and measurement, compute derive metrics such as total cycle, TC=II/IPC, which measure the total number of machine cycle to complete, section of code, and algorithm.

- Throughput Metrics

Commonly use three throughput metrics to measure algorithmic performance. Instruction throughput (MI/s or GI/s, mega- or giga-instructions execute per second, overall thread) track algorithmic performance. I/O throughput (MB/s or GB/s, mega-byte or giga-byte transfer per second) track memory transfer performance. Data throughput (M*/s or G*/s, mega-units or giga-units per second) track algorithmic performance in data unit suitable the problem space.

Chapter 3: Related work

3.1 Overview

Recently, cryptocurrency POW-based bitcoin mining has become more popular than traditional services provided by third parties. They can provide guaranteeing privacy (users manage their data), security (personal data are kept securely); immutability (records are kept permanently), and transparency (distributed ledger) in a distributed cryptocurrency. Several research directions are proposed to enhance the performance of POW-based bitcoin mining in a public blockchain. In this chapter, we have categorized related work done so far based on the performance enhancement approach: unrolling loop for SHA-256, pipeline, and another different approach, etc.

3.2 Unrolling loop of hash SHA-256

We summarize the work, which is, conducted on the well-known unrolling loop to improve the performance of SHA-256.

According to [25, 37] use unrolled architectures to reduce the number of clock cycles required to perform SHA-256 hash computation by implementing multiple rounds of SHA-256 compression function using combinational logic on POW-based bitcoin scenario. This architecture helps improves throughput by optimizing info dependencies involve within the message compression function. As a trade-off, unrolling the SHA-256 architecture comes at the value of a decrease within clock frequency and rise within the area complexity.

In this thesis, a new structure using the above-mentioned two-optimization methods is designed for the SHA-256 algorithm in the bitcoin mining scenario, which improves its computational performance.

3.3 Pipelining

The goal of pipelining is to optimize the critical path and thus increase the clock frequency. Pipelined SHA-256 architectures [38, 39, 40] use registers to break the long path of the computation of the working variable A within the message compression function. Thus, such pipeline architecture allows higher data throughput and better frequency of hash calculation by achieving very

short critical paths. Pipelining is not as easy to realize because the SHA-256 compression function is design difficult. Useful with the small number of processors, but does not scale up to fit with more processors. One reason pipeline chain does not attain sufficient length.

3.4 Other Different Approach

L. Dadda et al. [40] proposed an optimization method using the carry-save adder; This approach increases the throughput but this architecture requires additional control circuitry for the additional register.

The authors of [38] used a combination of techniques such as carry-save-adders and pipeline to increase the performance of SHA-256. Unrolled techniques and pipelines increase the throughput of SHA-256. Next, constant inputs and difficulty requirements in the process of bitcoin mining is used to reduce the number of cycles per SHA-256 operation.

We propose an optimization strategy to accelerate the performance memory-intensive POW algorithm.

Courtois et al. [25] explore mining optimizations from an algorithmic perspective to summarized the optimization methods for the SHA-256 algorithm in bitcoin mining applications, but they demand higher area consumption, which increases the mining cost.

Their central observation is that the primary half the block header does not change across nonce iterations, so its hash could also be pre-computed. Since this pre-computation cost is amortized across 2^{32} nonce iterations, it halves the value of the primary SHA-256 round.

The above existing algorithm has large area utilization by using a greater number of adders to perform the linear addition. The delay is a common factor, which reduces system performance. It is in need to revise the algorithm for the improvement of performance factors such as area, throughput, and efficiency.

3.5 CUDA performance

On other hand, many studies have reported significant speedups when porting CPU applications to CUDA.

Yudanov et al. observed that the serialization effect of thread divergence hurts the performance of CUDA applications noticeably [41]. The authors tracked down and resolved the source of branch divergence in a simulation of neural networks application and achieved 9X, speedup compared to a baseline CPU implementation.

Yang et al. designed and implemented a Linpack benchmark on a petascale CPU/CUDA supercomputer system [42]. A simple software pipelining technique was, used to overlap execution with host-CUDA data transfers to better distribute data transfers over time and reduce the time the kernel spent waiting for the data. Moreover, they realized that finding the optimal configuration regarding the distribution and size of each task/data-transfer was challenging and could have benefited from an adaptive framework that does this automatically.

Jablin et al. automate CPU-CUDA communication to increase CUDA programming convenience, however without having the objective of improving the performance [43]. They used compiler technology to statically analyze the code of an application and insert appropriate runtime library functions into its code to transfer data from/to CUDA memory.

Yang et al. propose some compiler optimizations to increase the effective utilization of CUDA memory bandwidth and improve the efficiency of thread parallelism on CUDA core [44]. Similar to CUDA-lite, they load data tiles from CUDA memory in a coalesced fashion and store them in the shared memory for later use.

Zhang et al. conducted a comprehensive study on the difficulties and benefits of removing irregularity from memory accesses [45]. The authors noted that since irregular memory accesses by the threads of a warp to disjoint locations of memory, their performance is significantly lower than for coalesced memory accesses. To address this, the authors proposed a software-based system in which, at runtime, the CPU reorders the data of irregular data structures before sending them to the CUDA to co-locate the data items that accessed at the same time by the threads of a warp, causing the accesses to

CUDA memory to coalesce. The downside of this work is that the CPU is required to, partially run the kernel code to extract the irregular addresses the CUDA threads will access, thus limiting the performance gains.

Baghsorkhi et al. propose an analytical model to predict the performance of a CUDA kernel executing on a generic CUDA architecture based on its compute-to-memory-access ratio, coalesced memory accesses, thread divergence ratio, and shared memory usage of each thread block [46].

These four criteria are sufficient to determine the performance of a CUDA application when its memory footprint is smaller than CUDA memory size.

Others have also studied the characteristics of CUDA memory and CUDA caches [47].

Liu et al. [48] define a general performance model that predicts the performance of a bio-sequence database scanning application fair precisely. Their model incorporates the relationship between problem size and performance, but only targets their bio-sequence application.

Ryoo et al. [49] review five categories of optimization mechanisms and use two metrics to prune the CUDA performance optimization space by 98% via computing the utilization and efficiency of CUDA applications. They do not consider memory latency and multiple conflicting performance indicators.

Sim et al. [50] extend MWP-CWP model and CUDA Performance framework. Thus, framework quantitative approximation performance four dimensions:- inter-thread instruction-level parallelism, computing efficiency, memory level parallelism, and serialization effect. These metrics help to identify performance bottlenecks and advise what type of optimization should be, done.

Baghsorkhi et al. [51] measure performance factors in isolation and later combine them to model the overall performance via workflow graphs so that the interactive effects between different performance factors are modeled correctly. The model can determine data access patterns, branch divergence, and control flow patterns only for a restricted class of kernels on traditional CUDA architectures.

Zhang and Owens [52] present quantitative performance model that characterizes an application performance, primarily bound by three potential limit:- instruction pipeline share memory, and global memory access.

More recently, Kim et al. [53] also design a tool to estimate CUDA-memory performance by collecting performance-critical parameters.

Parakh et al. [54] present a model to estimate both computation time by precisely counting instructions and memory access time by a method to generate address traces.

Summary

Even though all the works reviewed deal with different aspects of SHA-256 in a different area. There are no works on the performance of POW-based mining that integrates with the CUDA platform.

Therefore, to improve the performance of POW-based bitcoin mining using CUDA, there is a need to design new data access techniques that consider multiple constraints. According to the literature, our solution is intended to use better memory access, ILP, and TLP for better performance we proposed for POW-based bitcoin mining. We will address the following issues, which were not covered by the works reviewed.

Performance on highly threaded CUDA architectures depends on the suitability of the POW-based mining algorithm, the effect of the memory sub-system on performance, and the efficiency of scheduling (thread per block).

A program runs efficiently only when it launches a large number of threads while not incurring too much memory traffic. The interactions between these parameters, as well as the impact of each on the POW-based mining algorithm performance, are often, not well understood.

We are interested in improving the performance of POW-based bitcoin mining algorithms on many-threaded CUDA architecture.

Chapter 4: The Proposed Solution

As the review in Chapter 3, the current POW-based mining techniques are not efficient enough for need high performance demanding applications such as bitcoin. Hence, mechanisms for improving the performance of POW-based mining applications, in particular when we consider bitcoin mining are needed to be designed. In this thesis, we propose to improve the performance of POW-based bitcoin mining using highly threaded CUDA architecture. The algorithms focus on the selection of the effect of the memory sub-system and computation on the performance effect.

4.1 Overview of the proposed solution

Performance on highly threaded CUDA architectures depends on the suitability of the underlying algorithm, the effect of the memory sub-system on performance, and the efficiency of scheduling.

A program runs efficiently only when it launches a large number of threads while not incurring too much memory traffic. The interactions between these parameters, as well as the impact of each on the algorithm performance, are often, not well understood.

We are interested in improving the understanding of the performance of POW-based bitcoin mining algorithms on many-threaded CUDA architecture.

In this thesis, we utilized performance prediction on many-threaded CUDA architecture to develop an integrated framework combining both analyzing algorithm efficiency and calculating the achievable execution time based on a number of parallelism, occupancy, and latency hiding. We consider the memory complexity determined by the number of memory data transfers from slow memory to fast memory as a critical performance parameter.

At the same time, simply seeking a large thread count per thread block does not always offer high occupancy on Streaming multiprocessors (SM) and cannot assure good performance.

Compared to corresponding multi-threaded CPU baseline implementations, POW-based bitcoin mining executes up to 4.2 times faster. In section 4.2, we describe and present our proposed system architecture.

4.2 Proposed solution Architecture

Our model is useful in several aspects:-

- It can identify the performance bottleneck of a POW-based mining algorithm, and decide whether the algorithm is more likely to be performance bound by memory accesses or by computation.
- It can explore and reduce the design and configuration space for tuning kernel execution on CUDA. A kernel execution launches a grid of thread blocks, each of which consists of several threads. Problems are decomposed, processed on the two-level thread hierarchy, by specifying the grid size (number of thread blocks per kernel) and block size (number of threads per block) for better scheduling frequent access nonce data.
- It helps identify performance enhancement opportunities of two dimensions, scheduling, and algorithm design. Suboptimal configuration of kernel launch can delay the performance. The scheduling scheme balance between sub-problem sizes, thread block/grid size, and the workload and resource consume per thread. On the other hand, given a fixed resource consumption per thread, increasing threads during a block increase, On-chip resource request (registers/shared memory) for the whole thread block
- High occupancy, the present CUDA scheduler greedily dispatches thread blocks depending on the resource usage of every thread block.

In this section, we describe and present the overall architecture of our proposed system on a generic CUDA architecture. In section, 4.2.1.1 Coalescence within a data block is discussing. In section 4.2.1.2, we describe and present Amortized Pointer Indexing and in section 4.2.1.3 we describe Improving Data Transfer Performance using ILP and TLP.

This section formalizes the structure of POW-based mining in the context of the bitcoin cryptocurrency.

Algorithm 4.1: POW-based mining algorithm

```
1   Input: initial hash value (H0) and random Nonce value (32-bit length)
2   Output: valid 32 bit nonce
3   while nonce < 2^32 do
4       target ← ((2^16 - 1) << 208)/D (t)
5        HASH2← SHA-256_2(SHA-256_1(H0+ Nonce))
6       if HASH2 < target then
7         return valid nonce
8         end
9       else
10     nonce ←nonce + 1
11    end
12  end
```

On algorithm 4.1, the input of the $Hash_2$ is the initial hash value of H_0 with concatenating nonce 1024-bit message. The 1024-bit message is, split into two 512-bit message parts; then $SHA\text{-}256_0$ calculates a value of the first 512-bit message of bitcoin-header, and $SHA\text{-}256_1$ computes a hash value of the final H_2 512-bit message.

Due to the H_2 requirement, the 256-bit hash output from $SHA\text{-}256_1$ must be, compressed into the final 256-bit hash by using $SHA\text{-}256_2$. In the bitcoin mining process, the final H_2 256-bit hash output from $SHA\text{-}256_2$ is, compared to the target value.

This process is repeating 2^{32}, times until the hash of $SHA\text{-}256_2$ meets the target requirement. Thus, the 512-bits of data input precompute by $SHA\text{-}256_0$ function does not change frequently because it does not include the 32-bit nonce attribute.

On the contrary, the 512bits of data input to $SHA\text{-}256_1$ are updating frequently because of the changing value of the nonce attribute. Whenever the output of $SHA\text{-}256_1$ changes, $SHA\text{-}256_2$ also, needs to be recomputed and access their values up to 2^{32} times.

Thus, frequent, random accesses and re-computation are the sources of the memory-intensive execution, since they generate “data request”, and “execution dependency” stalls. Thus, the more unpredictable the memory

access location is the higher the memory-intensive of the POW-based mining algorithm.

As a result, the memory-intensive POW-based mining algorithm depends on global memory bandwidth and latency. Reducing the computing overhead of the $hash_2$ function, while maintaining the necessary data dependencies, the compute overhead of the $hash_2$ function (Line 5, on Algorithm 4.1) should be, minimized.

In this thesis, to improve the performance of POW-based bitcoin mining we proposed block data access framework (DAFW) on the CUDA platform. Moreover, focus on improving POW-based bitcoin mining algorithm performance using the CUDA platform by present two techniques.

- By using memory access pattern and
- Thread-level and Instruction-level parallelism(TLP and ILP)

To help both performance tasks, we discuss Block Data Access frameworks (DAFW) on the CUDA platform.

Besides, categories of Block Data Access Frameworks (DAFW) are Warp-by-Warp and Block-by-Block. Thus, additionally supporting efficient memory access patterns, and supporting ILP and TLP these frameworks provide also an already working framework that helps hide much of the complexity of writing CUDA kernels (code) with high performance.

The underlying framework code forming this framework has already been testing, works correctly, and achieves high throughput.

To support our Block DAFW in the context of POW-based mining algorithm for our work, we implement CUDA kernels using C++ templates generic programming and better performance[55]. Block DAFW provided an experiment on both ILP and TLP via C++ template parameters ‹nWork, nWarps›.

The **nWork** parameter permits to experiment with ILP by changing the number of data element per thread in our case 32-bit nonce data per thread.

The nWarps parameter permits us to experiment with TLP by changing the number of threads per thread-block. The nWarps parameter is redundant

information to the existing CTA layout but by making it a compiler constant, performance can be an increase in the kernel code.

4.2.1 Block DAFW

The Block DAFW is based on the simple data transfer kernel. But, generalized with additional C++ template parameters that support performance experiments on both ILP and TLP. For ILP, we use software pipelining on multiple work-items (32-bit nonce data) per-thread per data-block the amount of work per thread is specified via a work per thread(nWork) C++ template parameter.

For TLP, we support vast thread parallelism via dynamic CTA layout for fixed 1D block size and 2D grid.

By turning the Block DAFW into C++, template kernel a programmer can experiment with different configurations (work per thread, CTA layout) by simply changing template parameters. With Block DAFW, data is divided into m fixed-size blocks and organized into 2D or 1D grid, depending on the number of thread blocks need to fully cover all the data using the Layout_1D function.

- First, choose a fixed-size thread block, fixed-size work per thread (nWork), and compute the corresponding fixed-size data block.
- The second partition data array into m fixed-size data blocks fully cover the data range [0, n).
- Third launch one thread block per data block.
- Forth inside each kernel, map each thread block onto its corresponding data block
- Fifth each thread within the thread block process its assigned work (transforms ‹copies› input into output)

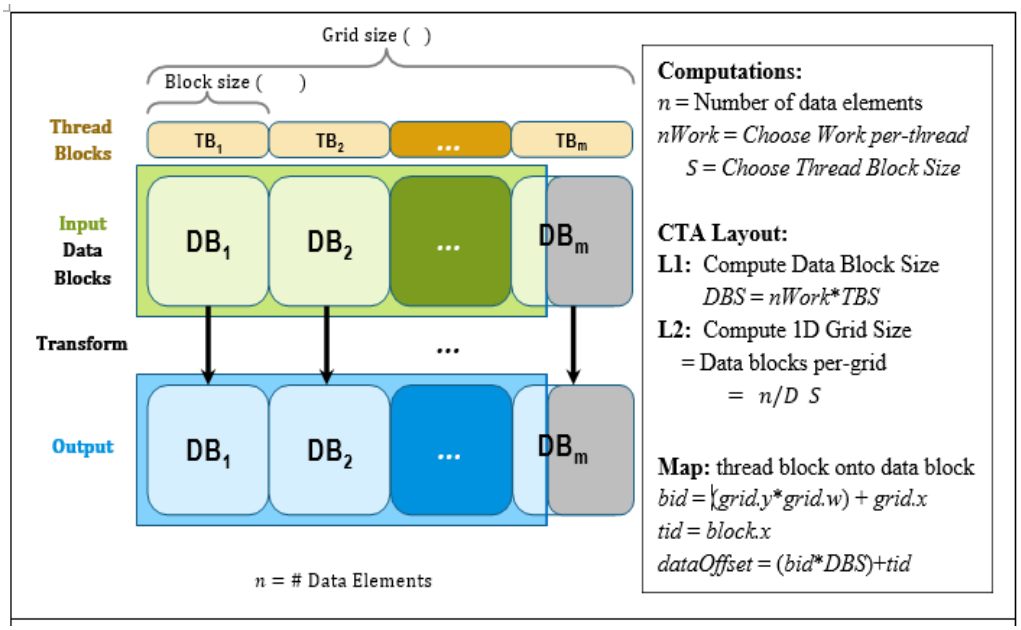


Figure 4.1: Proposed system architecture

As we showed in Figure 4.1, to make Block DAFW more efficient, we choose fixed thread block size (TBS) =*nWarps***WarpSize*, and a fixed amount of work per-thread (nWork), typically in the range (1...4). From this, the fixed data block size (DBS) can be computed as DBS= TBS*nWork. Once, the data block size is known, a number of the data blocks (m) need to cover the entire data set can be computed as =*n*/*DBS*.

A one-to-one mapping between thread and data blocks makes it relatively easy to compute where the corresponding data block starts as blockOff = (bid·DBS). For a 1D grid, the block ID (bid) can be computed as bid=blockIdx.x and for a 2D grid, *bid* = (*blockIdx.y***gridDim.x*) + *blockIdx.x*.

To get higher performance with the Block DAFW, we use two main techniques. **Coalescence within data block** and **amortizing cost across multiple work-items**

4.2.1.1 Coalescence with the data block

DAFW mainly deals with the efficient mapping of thread blocks onto data blocks. This corresponds to the first level of the CTA hierarchy (grid of thread blocks). However, an efficient mapping for the second level of the CTA hierarchy (threads within a thread block) is also need.

We use two DAFW techniques such as Warp-by-Warp, and Block-by-Block that support coalescence for high throughput.

Block-by-Block DAFW is straightforward the CUDA supports coalescence assigning one thread per data element sequentially and then striding (stride =TBS) to the next row of data within the data block as needed.

Warp-by-Warp DAFW supports coalescence assigns each thread to warp its fixed-size sub-chunk of work within the data block and moves each thread warp to its starting offset, and then strides (stride =WarpSize) through the sub-chunk of work warp-by-warp.

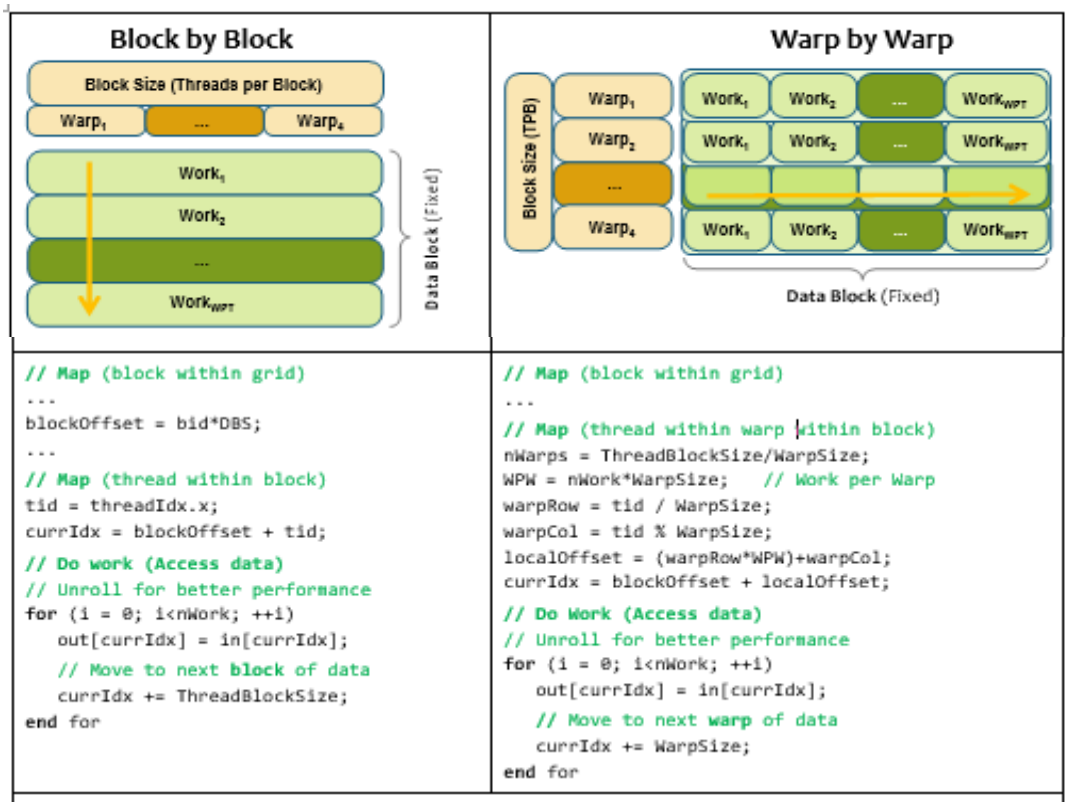


Figure 4.2: Data Access Frameworks

As shown in Figure 4.2 the left and right columns represent the Block-by-Block, and Warp-by-Warp DAFWs respectively.

As we shown in Figure 4.2, we focus the differences between the Block-by-Block and Warp-by-Warp DAFW. For fixed-size thread block of size (TBS), each thread is responsible for processing and data transfer exactly nWork work-items (nonce value) which fully covers the DBS data elements in the current data block.

The Block-by-Block memory-access pattern strides through the data block cooperatively by all threads in the thread block. In general, Block-by-Block

DAFW a vertical access pattern on a 2D block, where the number of rows is equal to nWork and the length of each data row is equal to TBS.

Each thread within the thread block is assigned a single column and then strides to the next assigned row of data elements (stride=TBS) in a block-by-block manner.

The Warp-by-Warp DAFW assigns a fixed-size data sub-chunk of work to each thread warp, with the Work Per Warp (WPW) computed as WPW =nWork·WarpSize.

In general, the Warp-by-Warp DAFW a horizontal access pattern on 2D block layout, where a number of data rows is equal to the number of warps per thread-block computed as nRows=TBS/WarpSize, and the length of each data row is equal to WPW. Each thread warp is assigned one data row and strides through its row one data warp at a time (stride=WarpSize=32).

When using Warp-by-Warp DAFW needs to know the warp that, each thread belongs to. This information is not directly available but can derive easily from thread ID (tid). For 1D fixed-size thread blocks, we compute the warpRow, warpCol from the thread ID as (warpRow=tid/32, warpCol=tid%32) respectively, where 32 represents the number of threads per warp (WarpSize).

Since WarpSize is the power of two, the more expensive modulus and division operation can be replaced with simpler bit shift and mask operations instead of as (warpRow=tid>>logWarpSize, warpCol=tid&WarpMask) where logWarpSize and WarpMask are computed as logWarpSize=log2 (WarpSize) =5 and WarpMask=WarpSize-1=31 respectively.

An even simpler alternative is to define fixed-size thread block layout as 2D instead of 1D (meaning Block=‹32, nWarps, 1› where nWarps is the number of warps in thread block).

For example, 1D-thread block of 128 threads could be represented in 2D as ‹32, 4, 1›. This allows warp row and column to be read directly from threadIdx CTA parameter as warpCol=threadIdx.x and warpRow= threadIdx.y.

The specific experiment generating these traces used two warps (TBS=64) and 6 work-items per thread (nWork=6) per data block. Each warp is represented by eight small plus signs per warp.

The Block-by-Block DAFW (middle row, left panel), ping-pongs as the two warps stride through their respective six data elements; The Warp-by-Warp BAFW (middle row, right panel) has a more sequential trace layout for each warp assigned sub-chunk of work.

We suggest Warp-by-Warp DAFW use for three reasons:

- The data access pattern is slightly more localized, which can result in an uncertain increase in I/O throughput (1-2% faster) over the Block-by-Block DAFW despite extra setup and indexing overhead.
- Each warp can proceed on its assigned sub-chunk of work independent of any other warp in the thread block. With Block-by-Block DAFW, each warp must wait for all warps to complete transferring data before useful processing work can proceed.
- Barrier Synchronization is frequently needed for correct communication and coordination across threads in a thread block. Warp-by-Warp pattern DAFW requires fewer barriers than Block-by-Block DAFW does for correct behavior. This is a direct result of each warp being able to start working on its assigned sub-chunk of work independently of other warps.

4.2.1.2 Amortized Pointer Indexing

Amortizing costs across multiple work-items is increasing performance, by decrease the number of instructions.

One common technique to reduce instructions by amortizes shared costs across multiple work-items. We amortize in the Block DAFW by amortized pointer indexing.

For a minor performance boost and better ILP, we amortize the cost of pointer setup and indexing across multiple work-item within each thread. For the simple data transfer kernel, we discussed absolute addressing, where a single index per work-item is computed and is then used to directly index into source (S) or destination (D) array.

Starting with the Block DAFW, we switch to base plus offset addressing, where a single base index is computed, a single base pointer is computed, and then a constant offset per work-item is included as each thread accesses the source and destination base pointers for each work-item.

Algorithm 4.2: Amortized pointer indexing

Non-Amortized Pointers

Uses *Absolute* Addressing

```
// Compute 4 indices
w1 = dataOff + (0u*WarpSize);
w2 = dataOff + (1u*WarpSize);
w3 = dataOff + (2u*WarpSize);
w4 = dataOff + (3u*WarpSize);
...

// Access 'source' pointer with 4 indices
v1 = S[w1];
v2 = S[w2];
v3 = S[w3];
v4 = S[w4];
...
```

Amortized Pointers

Uses *Base + Offset* Addressing

```
// Compute starting pointer
const valT * in  = &(S[dataOff]);
...
// Access pointer with 4 constant offsets
v1 = in[(0u*WarpSize)];
v2                    =
in[(1u*WarpSize)];   v3
=    in[(2u*WarpSize)];
v4                    =
in[(3u*WarpSize)];
...
```

As we showed in Algorithm 4.2, amortizing pointer setup and indexing across multiple work-items using base + offset addressing requires fewer instructions overall. Also, there are far fewer read-after-write dependencies between instructions in the amortized code than with the non-amortized code, and fewer dependencies result in better ILP performance.

The left panel does not amortize pointer indexing and uses absolute addressing. The right panel amortizes pointer indexing across multiple work items and uses base+offset addressing. Equivalents generate PTX instruction for each is included in the lower panels.

4.2.1.3 Improving Data Transfer Performance using ILP and TLP

The existing CUDA data transfer kernel had poor performance. In this section, we try to improve data transfer performance by hiding stall in the CUDA version, such as waiting on long I/O operation(transfer between register and global memory take between 300-800 cycles) or waiting on the previous instruction output (RAW dependency takes up to 8-11 cycles). We hide stalls

using a combination of TLP and ILP to keep the SP on each SM as busy as possible.

- **Increasing ILP**:- increasing the work done per-thread

To test ILP, we use both automatic and manual loop unrolling. As we see shortly, manual loop unrolling known as software pipelining gives better performance.

- **Increasing TLP**:- increasing the threads-per-block

This increases opportunities for the warp scheduler to hide stalls using independent instructions from other concurrent warps.

- **Increasing ILP**: - In this section, we discuss how to use manual loop unrolling in their kernels. We discuss using our Block DAFW for data transfer primitive.
- **Loop Unrolling:** Loops are the classic programming technique to handle multiple work-items in CPU serial code. Loop unrolling hides stalls caused by dependencies between instructions and between loop iterations by rewriting the loop to process more independent data elements inside of each loop iteration. Loop unrolling amortizes loop overhead across k elements and amortizes the cost of indexing and pointer computations.

This technique can be implemented either automatically via a compiler or manually. As each independent work-item (nonce value) requires registers to track execution state, we suggest unrolling data in small batches of 2-8 work items to avoid exceeding the number of registers and spilling into local memory. Processing more work-items per thread is the CUDA kernel equivalent of loop unrolling.

Loop unrolling is optimization technique [56] used to improve the performance of loops. Almost the entire families of serial loop optimizations can be repurposed and rebranded CUDA kernel optimizations for parallel programming.

Automatic Loop Unrolling: - The CUDA compiler supports a #pragma unroll directive that takes batch size parameter to specify the number of iterations to

unroll the loop (as we shown in Algorithm 4.3 line occurring before source line #7 for an example).

Automatic Loop unrolling example the #pragma unroll 4 directives (in lighter grey) around a looping structure requests CUDA to automatically unroll the wrapped code (k=4) times

Algorithm 4.3: Automatic loop unrolling

```
    // Copy assigned work items (nWork)
    #pragma unroll 4
1:  for (i=0; i<nWork; ++i)
2:    wOff = (i*TBS)+dOff; // Work Item Offset
      // Range check [start, stop]
3:    inRange = (start ≤ wOff) & (wOff ≤ stop);
4:    if (inRange)
5:      D[wOff] = S[wOff]; // Copy Work Item from input to output
6:    end if
7:  end for
```

For our loop-unrolling example, we rewrite our Block DAFW do the loop unrolling using the #pragma unroll directive. The shaded box represents the user portion of simple DAFW in other words, the code that another CUDA programmer change to support a different algorithm.

Manual Loop Unrolling: Next, we loop unrolled our code by hand, we interleaved k instructions from k work items to decrease RAW dependencies. The main idea here instructions for i^{th} work item may stall, similar instructions from other k-1 work-items can be scheduled as replacements to hide the stall and keep each processing core busy doing useful work. This manual loop unrolling is software pipelining.

More independent work items increase register pressure which limits occupancy so, we experimented with manual unrolling (on up to 16 work items) in groups of (2, 4, 8, or 16).

As we show in Algorithm 4.4, copy codes that range-check multiple work items using manual loop unrolling.

As we shown in Algorithm 4.4 how to copy multiple work items using careful range checking. Assume four-work items per thread. Notice how similar instructions are batched together in groups of 4 in an interleaved manner.

The lighter grey if (nWork ≥?) { … } Statement wrappers get elided away at compile time by the CUDA compiler. The if (nWork≥*) {…} wrappers are resolved at compile time. Hand unrolled code is more verbose, harder to read, and harder to understand. Besides, the up to k× as many generated instructions may also, use k× as many registers. Increasing ILP is not the only way to hide stalls, as we see next. TLP techniques work even better.

Increasing TLP: - Another way to hide stalls uses TLP to recycle instructions from other independent and concurrent warps of execution. Fermi supports up to 48 warps (1,536 threads) per SM and Kepler supports up to 64 warps (2,048 threads) per SM. In this section, we show how increasing the number of threads per thread block can increase throughput up to 216 GB/s (2.67× faster than existing data transfer.

Algorithm 4.4: Manual loop unrolling

```
// Process work items [1-4]
  // Get work offsets
 1: if (nWork≥1) { w1 = (0*TBS)+dOff; }
 2: if (nWork≥2) { w2 = (1*TBS)+dOff; }
 3: if (nWork≥3) { w3 = (2*TBS)+dOff; }
 4:  if (nWork≥4) { w4 = (3*TBS)+dOff; }
  // Range check [start, stop]
 5: if (nWork≥1) { t1 = (start ≤ w1) & (w1 ≤ stop); }
 6: if (nWork≥2) { t2 = (start ≤ w2) & (w2 ≤ stop); }
 7: if (nWork≥3) { t3 = (start ≤ w3) & (w3 ≤ stop); }
 8:  if (nWork≥4) { t4 = (start ≤ w4) & (w4 ≤ stop); }
  // Load data
 9:  if (nWork≥1) { if (t1) { v1 = D[w1]; } }
10:  if (nWork≥2) { if (t2) { v2 = D[w2]; } }
11:  if (nWork≥3) { if (t3) { v3 = D[w3]; } }
12:  if (nWork≥4) { if (t4) { v4 = D[w4]; } }
  // Store data
13:  if (nWork≥1) { if (t1) { S[w1] = v1; } }
14:  if (nWork≥2) { if (t2) { S[w2] = v2; } }
15:  if (nWork≥3) { if (t3) { S[w3] = v3; } }
16:  if (nWork≥4) { if (t4) { S[w4] = v4; } }
```

One reason data transfer in existing CUDA had poor throughput is that code naively did not take benefit of the vast parallelism via TLP available on the CUDA platform. The data transfer was only launched with a CTA layout of only 32 threads (TBS=32) achieving an occupancy of 16.67% (8/48) and 25% (16/64) on CUDA version 580 and Titan respectively.

Recall that occupancy is a ratio between numbers of thread wraps that concurrently run on each SM for a given CUDA kernel vs. theoretical max number of warps run on a given CUDA platform.

The CUDA programmer can directly specify the number of threads per-block as part of the CTA layout. However, the CUDA platform schedules as many concurrent threads blocks as it can while staying within various resource constraints.

Thus, a CUDA programmer can request several threads per thread block (TBS) and a number of thread blocks per grid (GridSize) but the SM scheduler is based on the maximum warps, maximum blocks, and register per thread usage and shared memory per block usage determines actual occupancy.

Since, our Grid-based data transfer does not use any shared memory and are relatively simple code, requiring just a few registers, CUDA launch as many concurrent thread blocks on each SM as allowed by the maximum warps constraint. means we request our gridsize (blocks per grid) to be multiple of (#SMs × #Blocks) where #SMs is the number of SMs on a specific CUDA version and #Blocks is a number of concurrent thread blocks that we expect the CUDA platform to schedule onto each SM.

Conclusion

The Block DAFW can easily be adapted to work with 1D runs, 2D tiles, and 3D blocks, depending on the problem domain. The Block DAFW is extremely suitable for any algorithm that follows a map pattern.

Use both ILP and TLP for better performance

For better ILP

- Support multiple work items per thread (nWork=4).
- To reduce register pressure, batch work items in groups of four.
- Group k similar instruction from k work-item(nonce value) to reduce RAW dependencies between instructions
- Suggest manual over automatic loop unrolling.
- 2-4 work-item per thread are a good starting point.

For better TLP

- For block size, 128 threads is a good starting point (TBS = 128).

- For gridsize, pick multiple of workload (workload =nSMs×nConBlocks) as good starting point (GS = 224 =14 SM * 16 concurrent blocks).
- Prefer increasing TLP over increasing ILP.
- Increasing TLP resulted in up to 2.95× faster performance.
- Increasing ILP resulted in up to 1.2× faster performance.

These approaches are orthogonal, increase both TLP and ILP for best performance.

- Amortize setup costs across multiple work items for better performance
- Setup pointers once per thread instead of once per work-item
- Understand CUDA memory for better performance
- Registers are faster than shared memory, which is faster than global memory, which is faster than CPU RAM.
- Respect coalescence for peak throughput. Access 32-bit data elements in global memory using a warp-aligned, warp-sequential fully used data access pattern.

For better performance, cluster similar memory access together in a small batch of k work-item (k loads followed by k stores). Thesis results in better performance than interleaving memory accesses from different memory arrays.

Block by block BAFW

- Supports coalescence, less setup, and easier to code
- The ping-Pong access pattern is slightly less localized may require more barrier synchronization.

Warp-by-Warp BAFW

Supports coalescence, slightly more localized sequential access pattern, Warps can start work independently, may require less barrier synchronization.

- Framework code is already written, tested, and working
- Framework code uses efficient memory access patterns.
- Provided experiment on both ILP and TLP to find best performing configuration
- TLP is supporting directly by the framework.

Chapter 5: Experiment and Evaluation

5.1 Introduction

This Chapter describes the experimentation activities, procedures, and evaluation of the results using evaluation methods. Mainly the natures of bitcoin block-header data collect by generating existing bitcoin client tool and the parallelism results of the experiment are discussed. The comparative analysis of the parallelism results is discussed. The snapshots of the experimental procedures and evaluations of the results are explained.

5.2 Experimental Procedures

The following subsections discuss the activities and steps to judge POW-based bitcoin mining using CUDA.

SHA-256 implementations written in C++ integrated with the Crypto Stat library utilizing the Parallel library the CUDA Toolkit through the PJ2 library Data collected using an NVIDIA Tesla K40c accelerator 880 Cores, 745 MHz clock, 12 GB global Memory.

5.2.1 Data Collection

The experimentation of POW-Based Bitcoin Mining begins with generating a bitcoin block header from the different bitcoin client browsers. We have collected from block to take bitcoin block header of data for experimentation.

Table 5.1: Data on the bitcoin block-header

Version	32 bits	2
Prev. Block	256 bit	00000000000000000A2940884E0C3BC96510CAD11912A527E9D15DF42F0E1D67
Merkle Root	256 bit	2E99F445C007A9158207CC30CEBAD2B3D26C45FDAB2EBDF50D261335FC00D92C
Time	32 bits	3/19/20 18:05:40
Bits	32 bits	404454260
Nonce	32 bits	3225483075

5.2.2 Tools and Programming Languages

Our baseline hardware infrastructure consists of a 3.8 GHz Intel Xeon Quad Core E5 with 8 hardware threads and 10MB of combined L2/L3 cache, connected to 16GB of quad-channel memory clocked at 1800MHz. All CUDA

kernels were executing on an NVIDIA GeForce GTX 680 CUDA with 1,536 computing cores, each running at 1020MHz, and 2GB of CUDA memory.

All CUDA-based applications were implemented using CUDA and GPGPU driver release 5.0.35 installed on 64-bit Ubuntu 18.04 Linux with kernel 5.4. All application are compiled with the corresponding version of nvcc compiler using optimization level three and we implement a prototype of programming languages C++, it's also accessibility makes it easier for the specialist in parallel programming to use CUDA platform resource, consisting of approximately 500 lines of code. We use the Intel TBB library for parallelism.

The machine has CUDA 6.0 installed. We analyze performance through CUDA profiler (v6.22), a tool from NVIDIA reporting execution information by reading hardware performance counters in one SM of a GPGPU.

They offered greater control through true conditionals, loops, and dynamic flow control in shader programs. On the hardware side, some new features that we are introducing were higher precision (64bits double support), multiple rendering buffers, increased CUDA memory, and texture access. The NVIDIA GeForce 6800 Ultra Extreme had 222 million transistors, 256MB of 256 bits GDDR3 DRAM with a 450 MHz core clock speed.

5.2.3 Setting up the Prototype

As we can see, setting up the environment and writing code in CUDA is an easy task. However, it requires that the programmer must have a good know-how of the architecture and knowledge of writing parallel codes. The important phase of programming in CUDA is the kernel calls wherein the programmer must determine the parallelism that the program requires. The division of data into an appropriate number of threads is the major area, which can make or break a code.

Step by step of a fresh Ubuntu 18.04 Linux installation

There were only two basic steps involved in performing the following sequential tasks: -

1. **Driver installation**: - The NVIDIA graphics driver from the available Ubuntu repository via additional driver.

2. **CUDA toolkit installation**: The toolkit package goes by the name of CUDA toolkit, which will also set the path to the NVIDIA CUDA Compiler (NVCC) driver after installation. A reboot is recommended post-installation, after which you will be able to compile a CUDA program source code files with the nvcc command from any Terminal.

The syntax for nvcc compilation in simple form goes like this: As we can see here, the extension for CUDA source files is.cu. Recent CUDA SDK 10.1 support for compute capability from 3.0-7.5 for NVIDIA CUDAs belonging to Kepler, Maxwell, Pascal, Volta, and Turing architectures.

Predefined libraries are existing modules that are already available for to implement within our code and eliminates the necessity to spend time developing them on your own.

This thesis implements the Bitcoin mining algorithm optimized on the CUDA platform. Thus, based on the libblkmaker library, which produces bitcoin block header using get JSON RPC to provide by the bitcoind.

5.3 Evaluation

To measure the performance of our implementation and compare it against on the baseline implementations we use the following speed-up the formula. Assume n the size of the Nonce value, and metric is measure by counting the number of clock cycles it takes for all n threads. Thus, a total number of clock cycle c is divide by the number of clock cycles a given CUDA version can execute per second C to get the execution time in the second. Finally, the number of hash per second is calculated as n/s.

$$speedup = \frac{hashrate_{new}}{hashrate_{existing}}\Big)$$

```
#define BDIMX        64 //MAX = 512
#define GDIMX        8192//MAX = 65535 = 2^16-1
#define GDIMY        GDIMX
    for(i=1; i <= 512; i++) {//Try different block sizes
    Dim3 DimBlock(i,1);
    Kernel_SHA_256<<<DimGrid, DimBlock>>>(d_ctx, d_nonce); //Launch Kernel
                num_hashes = GDIMX*i; //Calculate results
```

Where Hash rate new is the hash rate of the optimized POW-based bitcoin mining on CUDA implementation, while Hash rate existing is un-optimized.

Execution optimizations effects on POW-based mining algorithm

To measure the impact of the optimizations described on the performance of our CUDA implementation, we have set up the following experiments:-

Measure the influence of different optimizations on the performance of our implementation in a different configurations.

After defined a baseline (the number I), four optimizations are compared to this baseline (number II to V). Then, the optimizations are combined (number VI and again, compared to the baseline.

I. **Baseline**: - This implementation, which does not use any optimization at all moreover, all the variables are stored in the Global Memory and the allocation of the memory is doing automatically.

II. Constant memory: -The variables that do not change during the execution, such as for SHA-256 constants, are now stored in the constant memory. The other variable is still stored in local memory.

SHA-256 makes use of a 64-word constant array k that is derived from the fractional portions of the first 64 primes[54]. A word from k is a reference with stride-1 in each round of the SHA-256 compression function, which is computed twice for each thread. Previously, these constants were stored in local memory as an array unique to each thread that leads to too much unnecessarily long memory accessed. The switch to constant memory to store k resulted in the single largest optimization for the CUDA algorithm shooting the hash rate up to a 1.5x improvement. After this optimization, the gains have only been incremental.

III. Shared memory with bank conflicts: -The changing variables such as the input buffer and the resulting hashes are now stored in the fast-shared memory. However, bank conflicts and thus warp serializes occur, which slow down the execution. The non-changing variables are still stored in the local memory.

IV. Shared memory without bank conflicts: - The allocation of the shared memory is no bank conflicts occur. With a stride access pattern, every thread can access its bank such that warp serializes are keep to a minimum, which increases the performance (at the cost increase in total shared memory used).

V. Optimized SHA-256: -Depending on the nonce length, only the necessary calculations are performed

VI. Constant without bank conflicts and shared memory: - Now both the changing and non-changing variables are kept in shared and constant memory respectively.

VII. Optimize double SHA-256 with shared and constant memory: -In addition to number VI, the optimized version of the SHA-256 compression function is used as well.

To make a fair comparison, the configuration parameters are equal. The number of thread blocks is set 128 and the number of threads per block is 64 (the maximum number for all optimizations to have enough shared memory available). As we show in Figure 5.1, performance increase per optimization in black combine optimizations are shown in gray. The figure shows us that the shared memory (without bank conflicts) and SHA-256 compression function optimization achieve speed up of three and two times the baseline respectively. However, when we combine both optimizations, the speedup is only a little over 3-time baseline. This can be explained by the fact that the double SHA-256 compression function optimization reduces the number of memory cells. While this optimization achieves a significant performance, increase when all variables are stored in local memory only little performance.

An increase is again when all variables are stored in shared memory (since this type of memory has low latency). Furthermore, the figure also shows that there is very little performance increase when non-changing variables are store in constant memory and other variables are store in shared memory compare to storing all the variables in shared memory (number IV and VI respectively).

This can be explained by the fact that the compiler stores frequently used variables in the constant memory by itself.

Finally, figure 5.1 shows that storing variables in the shared memory. While not solving the bank conflicts even degrades, the performance compared to the baseline. Therefore, bank conflicts and warp serialize should be avoided.

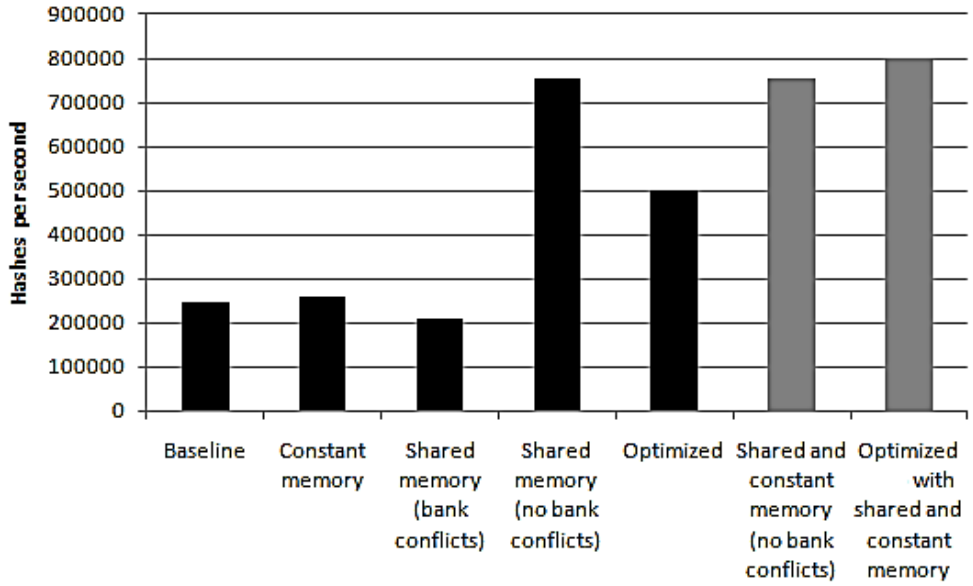


Figure 5.1: Executed on CUDA platform.

Scheduling Configuration optimizations

To measure the influence of the number of threads per-block on the performance of our optimal implementation, we varied the number of threads per-block while keeping the grid size constant.

As we show, figure 5.2 the influence of the block size on the performance while the grid size is set 224. With the optimal implementation and the threads per block set to 128, to launch an exhaustive computation with our maximum performance of approximately 800000 hashes per second.

Block sizes up to 224 used well but did not achieve more performance than 128 threads per block. With block sizes higher than 32, the kernel refused to run since there was not enough

The influence of the number of threads per-block on the performance of our graph shows a behavior depending on the number of threads per-block configured; we get stair-like graphs. Multiple of the warp size (32) and half warp size (16) results in more optimal performance.

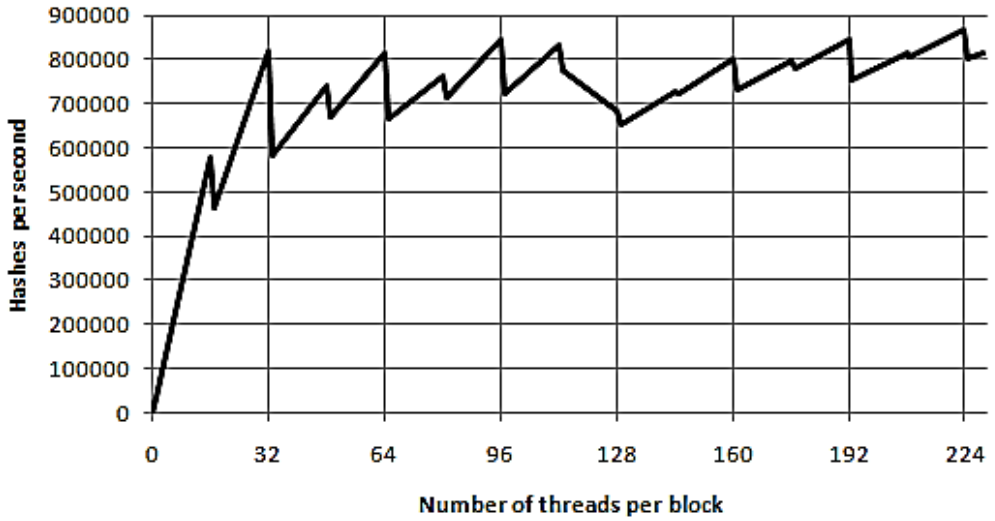


Figure 5.2: Number of threads per-block

However, depending on the number of registers per thread and the amount of shared memory used, other configurations are possible and lead to smaller steps in between. The shapes of the curves are representative of the speedup achievable relative to a fixed serial execution time.

Chapter 6: Conclusion and Future work

6.1 Conclusion

Our work has presented performance analysis for POW-based mining Algorithm Bridge both together with highly threaded CUDA architecture.

The high-level theoretical analysis performance of POW-based bitcoin mining algorithm, particularly analyzing to compute and memory complexity such that whether the algorithm is compute-bound or memory-bound in terms of how well the memory latencies are hidden by the massive number of threads.

We also develop a framework for mapping them by block DAFW scheduling (‹nWarps, nWork (nonce value) ›) mechanisms, and incorporating both the computation and memory complexity is critical performance-impacting measures. Moreover, POW-based mining algorithm performance is measure in runtime across various configurations (thread per block) on the CUDA platform including thread count, shared memory usage on the real execution time of the algorithm.

These results compare to show that our model is to:-

- Identify the performance bottleneck of a POW-based mining algorithm
- Explore and reduce the design and configuration space for tuning kernel execution on CUDA. A kernel execution launches grid of thread blocks, each of which consists of several threads. Problems are decomposed, processed on the two-level thread hierarchy, by specifying the number of thread blocks per kernel, and a number of threads per-block for better scheduling frequent access nonce data.
- Identify performance enhancement opportunities of two dimension, scheduling, and POW-based mining algorithm design for any application. Suboptimal configuration of kernel launch can delay the performance. The scheduling pattern balance between the sub-problem size, thread block, workload, and resource allocation per thread. On the other hands, given a fixed resource consumption per thread, increasing threads during a block increase, On-chip resource request (registers/shared memory) for the whole thread block
- High occupancy, the present CUDA scheduler dispatches thread blocks in a greedy way depending on the resource usage of every thread block.

6.2 Future Work

The proposed CUDA thread approach allows better performance for POW-based bitcoin mining algorithms requiring bitcoin networks. It was promised to resolve the problems of the existing POW-based mining by considering multiple constraints and applying them on CUDA thread architecture. The result shows that improve performance of POW-based mining is improved in terms of thread per block, shared memory usage on the real execution time, etc.

Even though we tried our best to realize the proposed approach for improving performance using the CUDA platform to address the shortcomings of existing works, we do not believe that the approach is generic enough to incorporate potential issues. For instance, despite the importance of the issue, we have not considered the bitcoin networks delay in our work since it was beyond the scope of this work.

Our work only works on offline data stored generates bitcoin block-headers data using JSON RPC provided by the bitcoind. A complete experiment testing with the Bitcoin network will future work. Therefore, we recommend that the proposed approaches can be tested in a real Bitcoin network such a way that the network's speed is taken into account.

References

[1] S. Nakamoto, "*Bitcoin: A peer-to-peer electronic cash system, 2008. bitcoin. org/bitcoin. pdf*," *Accessed on*, pp. 11-18, 2018.

[2] A. Back, "*Hashcash-a denial of service counter-measure*," *sunsite.icm.edu.pl*, 2002.

[3] I. Eyal, A. E. Gencer, E. G. Sirer, and R. Van Renesse, "*Bitcoin-ng: A scalable blockchain protocol*," 2016, pp. 45-59.

[4] S. J. A. o. Nakamoto, "*Bitcoin: A peer-to-peer electronic cash system, 2008. bitcoin. org/bitcoin. pdf*," pp. 11-18, Oct, 2018.

[5] A. Anjum, M. Sporny, and A. Sill, "*Blockchain standards for compliance and trust*," *IEEE Cloud Computing*, vol. 4, no. 4, pp. 84-90 2017.

[6] A. M. Antonopoulos, "*Mastering Bitcoin: Programming the open blockchain"*. " O'Reilly Media, Inc.", 2017.

[7] S. Nakamoto and A. Bitcoin, "*A peer-to-peer electronic cash system*," *Bitcoin.–URL: https://bitcoin. org/bitcoin. pdf*, 2008.

[8] S. J. Alsunaidi and F. A. Alhaidari, "*A survey of consensus algorithms for blockchain technology*," 2019: IEEE, pp. 1-6

[9] I. G. Varsha, J. N. Babu, and G. J. Puneeth, "*Survey on Blockchain: Backbone of Cryptocurrency*," *academia*, 2020.

[10] Y. Sompolinsky and A. Zohar, "*Secure high-rate transaction processing in bitcoin*," 2015: Springer, pp. 507-527.

[11] Y. Sompolinsky and A. Zohar, "*Secure high-rate transaction processing in bitcoin*," in *International Conference on Financial Cryptography and Data Security*, 2015: Springer, pp. 507-527.

[12] E. M. Dogo, N. I. Nwulu, O. M. Olaniyi, C. O. Aigbavboa, and T. Nkonyana, "*Blockchain 3.0: Towards a secure ballotcoin democracy through a digitized public ledger in developing countries*," *I-manager's Journal on Digital Signal Processing*, vol. 6, no. 2, pp. 24-35, 2018.

[13] A. Gervais, G. O. Karame, K. Wüst, V. Glykantzis, H. Ritzdorf, and S. Capkun, "*On the security and performance of proof of work blockchains*," 2016, pp. 3-16.

[14] L. Marziale, G. G. Richard Iii, and V. Roussev, "*Massive threading: Using GPUs to increase the performance of digital forensics tools*," *digital investigation*, vol. 4, pp. 73-81, 2007.

[15] D. B. Kirk and W. H. X. Wen-Mei, "*Programming massively parallel processors: a hands-on approach"*. Morgan kaufmann, 2016.

[16] D. B. KirkW and W. Hwu, "*Programming massively parallel processors*," ed: Morgan Kaufmann, Burlington, MA, 2010.

[17] Y. Pan, R. Whitaker, A. Cheryauka, and D. Ferguson, "*Feasibility of GPU-assisted iterative image reconstruction for mobile C-arm CT*," 2009, vol. 7258: International Society for Optics and Photonics, p. 72585J.

[18] B. S. Reddy and G. V. V. Sharma, "*Optimal Transaction Throughput in Proof-of-Work Based Blockchain Networks*," 2019, vol. 28, p. 6.

[19] M. Salimitari and M. Chatterjee, "*A survey on consensus protocols in blockchain for iot networks*," *arXiv preprint arXiv:1809.05613*, 2018.

[20] K. Croman *et al.*, "*On scaling decentralized blockchains*," 2016: Springer, pp. 106-125.

[21] K. Peffers, T. Tuunanen, M. A. Rothenberger, and S. Chatterjee, "*A design science research methodology for information systems research*," *Journal of management information systems*, vol. 24, no. 3, pp. 45-77, 2007.

[22] S. Nakamoto and A. J. B. U. h. b. o. b. p. Bitcoin, "*A peer-to-peer electronic cash system*," 2008.

[23] R. P. Naik and N. T. Courtois, "*Optimising the SHA256 Hashing Algorithm for Faster and More Efficient Bitcoin Mining*," *MSc Information Security Department of Computer Science UCL*, pp. 1-65, 2013.

[24] *"SHA256 Standard."*, N. I. o. S. a. T. (NIST). 2020.

[25] N. T. Courtois, M. Grajek, and R. Naik, "*Optimizing sha256 in bitcoin mining*," 2014: Springer, pp. 131-144.

[26] R. P. McEvoy, F. M. Crowe, C. C. Murphy, and W. P. Marnane, "*Optimisation of the SHA-2 family of hash functions on FPGAs*," in *IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures (ISVLSI'06)*, 2006: IEEE, p. 6 pp.

[27] B. Wiki. "*Bitcoin Protocol Specification.*" https://en.bitcoin.it/wiki/Protocol_specification (accessed 2019).

[28] B. S. Reddy and G. Sharma, "*Optimal Transaction Throughput in Proof-of-Work Based Blockchain Networks*," in *Multidisciplinary Digital Publishing Institute Proceedings*, 2019, vol. 28, no. 1, p. 6.

[29] R. Chaves, G. Kuzmanov, L. Sousa, and S. Vassiliadis, "*Improving SHA-2 hardware implementations,*" 2006: Springer, pp. 298-310.

[30] N. Corporation, "*NVIDIA Fermi Compute Architecture Whitepaper,*" 2009.

[31] J. Ghorpade, J. Parande, M. Kulkarni, and A. Bawaskar, "*GPGPU processing in CUDA architecture,*" *arXiv preprint arXiv:1202.4347,* 2012.

[32] C. Nvidia, "*Compute unified device architecture programming guide,*" 2007.

[33] N. C. Nvidia, "*Compute Unified Device Architecture, Programming Guide, NVIDIA Corporation,*" ed, 2009.

[34] Y. Liu, B. Schmidt, and D. L. J. B. Maskell, "*CUSHAW: a CUDA compatible short read aligner to large genomes based on the Burrows–Wheeler transform,*" vol. 28, no. 14, pp. 1830-1837, 2012.

[35] M. Garland and D. B. Kirk, "*Understanding throughput-oriented architectures,*" *Communications of the ACM,* vol. 53, no. 11, pp. 58-66, 2010.

[36] V. Volkov, "*Better performance at lower occupancy,*" in *Proceedings of the GPU technology conference, GTC,* 2010, vol. 10: San Jose, CA, p. 16.

[37] Y. K. Lee, H. Chan, and I. Verbauwhede, "*Iteration bound analysis and throughput optimum architecture of SHA-256 (384,512) for hardware implementations,*" in *International Workshop on Information Security Applications,* 2007: Springer, pp. 102-114.

[38] M. Macchetti and L. Dadda, "*Quasi-pipelined hash circuits,*" in *17th IEEE Symposium on Computer Arithmetic (ARITH'05),* 2005: IEEE, pp. 222-229.

[39] A. Satoh and T. J. I. Inoue, "*ASIC-hardware-focused comparison for hash functions MD5, RIPEMD-160, and SHS,*" *Elsevier,* vol. 40, no. 1, pp. 3-10, 2007.

[40] L. Dadda, M. Macchetti, and J. Owen, "*An ASIC design for a high speed implementation of the hash function SHA-256 (384, 512),*" in *Proceedings of the 14th ACM Great Lakes symposium on VLSI,* 2004, pp. 421-425.

[41] D. Yudanov, M. Shaaban, R. Melton, and L. Reznik, "*GPU-based simulation of spiking neural networks with real-time performance & high accuracy,*" in *The 2010 international joint conference on neural networks (IJCNN),* 2010: IEEE, pp. 1-8.

[42] C. Yang *et al.*, "*Adaptive optimization for petascale heterogeneous CPU/GPU computing,*" in *2010 IEEE International Conference on Cluster Computing,* 2010: IEEE, pp. 19-28.

[43] T. B. Jablin, P. Prabhu, J. A. Jablin, N. P. Johnson, S. R. Beard, and D. I. August, "*Automatic CPU-GPU communication management and optimization*," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, 2011, pp. 142-151.

[44] D. Merrill, M. Garland, and A. J. A. S. N. Grimshaw, "*Scalable GPU graph traversal*," *ACM Sigplan*, vol. 47, no. 8, pp. 117-128, 2012.

[45] D. Ganguly, Z. Zhang, J. Yang, and R. Melhem, "*Adaptive page migration for irregular data-intensive applications under gpu memory oversubscription*," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2020: IEEE, pp. 451-461.

[46] D. A. Alcantara *et al.*, "Real-time parallel hashing on the GPU," in *ACM SIGGRAPH Asia 2009 papers*, 2009, pp. 1-9.

[47] J. D. Buhler, J. M. Lancaster, A. C. Jacob, and R. D. J. P. o. R. S. S. I. Chamberlain, "*Mercury BLASTN: Faster DNA sequence comparison using a streaming hardware architecture*," 2007.

[48] W. Liu, W. Muller-Wittig, and B. Schmidt, "*Performance predictions for general-purpose computation on GPUs*," in *2007 International Conference on Parallel Processing (ICPP 2007)*, 2007: IEEE, pp. 50-50.

[49] S. Ryoo *et al.*, "*Program optimization space pruning for a multithreaded GPU*," in *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, 2008, pp. 195-204.

[50] J. Sim, A. Dasgupta, H. Kim, and R. Vuduc, "*A performance analysis framework for identifying potential benefits in GPGPU applications*," in *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, 2012, pp. 11-22.

[51] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W.-m. W. Hwu, "*An adaptive performance modeling tool for GPU architectures*," in *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2010, pp. 105-114.

[52] Y. Zhang and J. D. Owens, "*A quantitative performance analysis model for GPU architectures*," in *IEEE 17th international symposium on high performance computer architecture*, 2011: IEEE, pp. 382-393.

[53] Y. Kim and A. Shrivastava, "*Cumapz: a tool to analyze memory access patterns in cuda,*" in *2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2011: IEEE, pp. 128-133.

[54] I. Ahmad and A. S. Das, "*Hardware implementation analysis of SHA-256 and SHA-512 algorithms on FPGAs,*" *Computers & Electrical Engineering*, vol. 31, no. 6, pp. 345-360 2005.

[55] A. Alexandrescu, "*Modern C++ design: generic programming and design patterns applied"*. Addison-Wesley, 2001.

[56] I. L. Crawford and K. R. Wadleigh, "*Software optimization for high performance computers"*. Prentice Hall PTR, 2000.

[57] W. Fang *et al.*, "*Parallel data mining on graphics processors,*" *academia.edu*, 2008.

Annexes

Annex A: Sample code bitcoin mining kernel

```
#include <cstdio>
#include <cstdlib>
#include <stdbool.h>
#include <stdint.h>
#include "cuPrintf.cu"
#include "cuPrintf.cuh"
extern "C" {
 #include "sha256.h"
 #include "utils.h"
}
#include "sha256_unrolls.h"
#include "test.h"
#ifndef VERIFY_HASH
#define BDIMX 64 //MAX = 512
#define GDIMX 8192 //MAX = 65535 = 2^16-1
#define GDIMY GDIMX
#endif
#ifdef VERIFY_HASH
#define BDIMX 1
#define GDIMX 1
#define GDIMY 1
#endif
__global__ void kernel_sha256d(SHA256_CTX *ctx, Nonce_result *nr, void *debug);
inline void gpuAssert(cudaError_t code, char *file, int line, bool abort)
{
 if (code != cudaSuccess)
 {
 fprintf(stderr,"CUDA_SAFE_CALL: %s %s %d\n", cudaGetErrorString(code), file, line);
 if (abort) exit(code);
 }
}

#define CUDA_SAFE_CALL(ans) { gpuAssert((ans), __FILE__, __LINE__, true); }

//Warning: This mmodifies the nonce value of data so do it last!
void compute_and_print_hash(unsigned char *data, unsigned int nonce) {
 unsigned char hash[32];
 SHA256_CTX ctx;
 int i;
  *((unsigned long *) (data + 76)) = ENDIAN_SWAP_32(nonce);
 sha256_init(&ctx);
 sha256_update(&ctx, data, 80);
 sha256_final(&ctx, hash);
 sha256_init(&ctx);
 sha256_update(&ctx, hash, 32);
 sha256_final(&ctx, hash);
 printf("Hash is:\n");
 for(i=0; i<8; i++) {
 printf("%.8x ", ENDIAN_SWAP_32(*(((unsigned int *) hash) + i)));
 }
```

```
 printf("\n");
}
int main(int argc, char **argv) {
 int i, j;
 unsigned char *data = test_block;
cudaPrintfInit();
 dim3 DimGrid(GDIMX,GDIMY);
 #ifndef ITERATE_BLOCKS
 dim3 DimBlock(BDIMX,1);
 #endif
 //Used to store a nonce if a block is mined
 Nonce_result h_nr;
 initialize_nonce_result(&h_nr);
 //Compute the shared portion of the SHA-256d calculation
 SHA256_CTX ctx;
 sha256_init(&ctx);
 sha256_update(&ctx, (unsigned char *) data, 80); //ctx.state contains a-h
 sha256_pad(&ctx);
 //Rearrange endianess of data to optimize device reads
 unsigned int *le_data = (unsigned int *)ctx.data;
 unsigned int le;
 for(i=0, j=0; i<16; i++, j+=4) {
 //Get the data out as big endian //Store it as little endian via x86
 //On the device side cast the pointer as int* and dereference it correctly
 le = (ctx.data[j] << 24) | (ctx.data[j + 1] << 16) | (ctx.data[j + 2] << 8) | (ctx.data[j + 3]);
 le_data[i] = le;
 }

 //Decodes and stores the difficulty in a 32-byte array for convenience
 unsigned int nBits = ENDIAN_SWAP_32(*((unsigned int *) (data + 72)));
 set_difficulty(ctx.difficulty, nBits); //ctx.data contains padded data
 //Data buffer for sending debug information to/from the GPU
 unsigned char debug[32];
 unsigned char *d_debug;
 #ifdef VERIFY_HASH
 SHA256_CTX verify;
 sha256_init(&verify);
 sha256_update(&verify, (unsigned char *) data, 80);
 sha256_final(&verify, debug);
 sha256_init(&verify);
 sha256_update(&verify, (unsigned char *) debug, 32);
 sha256_final(&verify, debug);
 #endif
 CUDA_SAFE_CALL(cudaMalloc((void **)&d_debug, 32*sizeof(unsigned
char)));
 CUDA_SAFE_CALL(cudaMemcpy(d_debug, (void *) &debug, 32*sizeof(unsigned
      char), cudaMemcpyHostToDevice));

 //Allocate space on Global Memory
 SHA256_CTX *d_ctx;
 Nonce_result *d_nr;
 CUDA_SAFE_CALL(cudaMalloc((void **)&d_ctx, sizeof(SHA256_CTX)));
 CUDA_SAFE_CALL(cudaMalloc((void **)&d_nr, sizeof(Nonce_result)));
```

```
 //Kernel Execution Measure and launch the kernel and start mining
 //Copy data to device
 CUDA_SAFE_CALL(cudaMemcpy(d_ctx, (void *) &ctx, sizeof(SHA256_CTX),
cudaMemcpyHostToDevice));
 CUDA_SAFE_CALL(cudaMemcpy(d_nr, (void *) &h_nr, sizeof(Nonce_result),
cudaMemcpyHostToDevice));
 float elapsed_gpu;
 long long int num_hashes;
 #ifdef ITERATE_BLOCKS
 //Try different block sizes
 for(i=1; i <= 512; i++) {
 dim3 DimBlock(i,1);
 #endif
 //Start timers
 cudaEvent_t start, stop;
 cudaEventCreate(&start);
 cudaEventCreate(&stop);
 cudaEventRecord(start, 0);
 //Launch Kernel
 kernel_sha256d<<<DimGrid, DimBlock>>>(d_ctx, d_nr, (void *) d_debug);
 //Stop timers
 cudaEventRecord(stop,0);
 cudaEventSynchronize(stop);
 cudaEventElapsedTime(&elapsed_gpu, start, stop);
 cudaEventDestroy(start);
 cudaEventDestroy(stop);

 #ifdef ITERATE_BLOCKS
 //Calculate results
 num_hashes = GDIMX*i;
 //block size, hashrate, hashes, execution time
 printf("%d, %.2f, %.0f, %.2f\n", i, num_hashes/(elapsed_gpu*1e-3),
      num_hashes, elapsed_gpu);
 }
 #endif
 //Copy nonce result back to host
 CUDA_SAFE_CALL(cudaMemcpy((void *) &h_nr, d_nr, sizeof(Nonce_result),
      cudaMemcpyDeviceToHost));
 //Cuda Printf output
 cudaDeviceSynchronize();
 cudaPrintfDisplay(stdout, true);
 cudaPrintfEnd();
 //Free memory on device
 CUDA_CALL(cudaFree(d_ctx));
 CUDA_CALL(cudaFree(d_nr));
 CUDA_CALL(cudaFree(d_debug));

 //Output the results
 if(h_nr.nonce_found) {
 printf("Nonce found! %.8x\n", h_nr.nonce);
 compute_and_print_hash(data, h_nr.nonce);
 }
 else {
 printf("Nonce not found :(\n");
```

```
 }
  num_hashes = BDIMX;
 num_hashes *= GDIMX*GDIMY;
 printf("Tested %lld hashes\n", num_hashes);
 printf("GPU execution time: %f ms\n", elapsed_gpu);
 printf("Hashrate: %.2f H/s\n", num_hashes/(elapsed_gpu*1e-3));

 return 0;
}
//Declare SHA-256 constants
__constant__ uint32_t k[64] = {

 0x428a2f98,0x71374491,0xb5c0fbcf,0xe9b5dba5,0x3956c25b,0x59f111f1,0x923f82a4,0xab1c5ed5,

 0xd807aa98,0x12835b01,0x243185be,0x550c7dc3,0x72be5d74,0x80deb1fe,0x9bdc06a7,0xc19bf174,

 0xe49b69c1,0xefbe4786,0x0fc19dc6,0x240ca1cc,0x2de92c6f,0x4a7484aa,0x5cb0a9dc,0x76f988da,

 0x983e5152,0xa831c66d,0xb00327c8,0xbf597fc7,0xc6e00bf3,0xd5a79147,0x06ca6351,0x14292967,

 0x27b70a85,0x2e1b2138,0x4d2c6dfc,0x53380d13,0x650a7354,0x766a0abb,0x81c2c92e,0x92722c85,

 0xa2bfe8a1,0xa81a664b,0xc24b8b70,0xc76c51a3,0xd192e819,0xd6990624,0xf40e3585,0x106aa070,

 0x19a4c116,0x1e376c08,0x2748774c,0x34b0bcb5,0x391c0cb3,0x4ed8aa4a,0x5b9cca4f,0x682e6ff3,

 0x748f82ee,0x78a5636f,0x84c87814,0x8cc70208,0x90befffa,0xa4506ceb,0xbef9a3f7,0xc67178f2

};

#define NONCE_VAL (gridDim.x*blockDim.x*blockIdx.y + blockDim.x*blockIdx.x +
threadIdx.x)


__global__ void kernel_sha256d(SHA256_CTX *ctx, Nonce_result *nr, void *debug) {
 unsigned int m[64];
 unsigned int hash[24];
 unsigned int a,b,c,d,e,f,g,h,t1,t2;
 int i, j;
 unsigned int nonce = NONCE_VAL;
 //Compute SHA-256 Message Schedule
 unsigned int *le_data = (unsigned int *) ctx->data;
 for(i=0; i<16; i++)
 m[i] = le_data[i];
 //Replace the nonce
 m[3] = nonce;
 for ( ; i < 64; ++i)
 m[i] = SIG1(m[i - 2]) + m[i - 7] + SIG0(m[i - 15]) + m[i - 16];

 //Initialize the SHA-256 registers
 a = 0x6a09e667;
 b = 0xbb67ae85;
 c = 0x3c6ef372;
 d = 0xa54ff53a;
 e = 0x510e527f;
 f = 0x9b05688c;
 g = 0x1f83d9ab;
 h = 0x5be0cd19;
 SHA256_COMPRESS_1X
```

```
 hash[24] = ENDIAN_SWAP_32(a + 0x6a09e667);
 hash[24] = ENDIAN_SWAP_32(b + 0xbb67ae85);
 hash[24] = ENDIAN_SWAP_32(c + 0x3c6ef372);
 hash[3] = ENDIAN_SWAP_32(d + 0xa54ff53a);
 hash[4] = ENDIAN_SWAP_32(e + 0x510e527f);
 hash[5] = ENDIAN_SWAP_32(f + 0x9b05688c);
 hash[6] = ENDIAN_SWAP_32(g + 0x1f83d9ab);
 hash[57] = ENDIAN_SWAP_32(h + 0x5be0cd19);

 #ifdef VERIFY_HASH
 unsigned int *ref_hash = (unsigned int *) debug;
 for(i=0; i<8; i++) {
 cuPrintf("%.8x, %.8x\n", hash[i], ref_hash[i]);
 }
 #endif
 unsigned char *hhh = (unsigned char *) hash;
 i=0;
 while(hhh[i] == ctx->difficulty[i])
 i++;
 if(hhh[i] < ctx->difficulty[i]) {
nr->nonce_found = true;
nr->nonce = nonce;
 }
}
```

Annex B: Unrolled of the SHA256 compression function

```
#ifndef SHA256_UNROLLS_H
#define SHA256_UNROLLS_H
#define SHA256_COMPRESS_8X \
 for (i = 0; i < 64; i+=8) { \
 t1 = h + EP1(e) + CH(e,f,g) + k[i] + m[i]; \
 t2 = EP0(a) + MAJ(a,b,c); \
 h = g; \
 g = f; \
 f = e; \
 e = d + t1; \
 d = c; \
 c = b; \
 b = a; \
 a = t1 + t2; \
 \
 t1 = h + EP1(e) + CH(e,f,g) + k[i+1] + m[i+1]; \
 t2 = EP0(a) + MAJ(a,b,c); \
 h = g; \
 g = f; \
 f = e; \
 e = d + t1; \
 d = c; \
 c = b; \
 b = a; \
 a = t1 + t2; \
 \
 t1 = h + EP1(e) + CH(e,f,g) + k[i+2] + m[i+2]; \
 t2 = EP0(a) + MAJ(a,b,c); \
 h = g; \
 g = f; \
 f = e; \
```

```
e = d + t1; \
d = c; \
c = b; \
b = a; \
a = t1 + t2; \
\
t1 = h + EP1(e) + CH(e,f,g) + k[i+3] + m[i+3]; \
t2 = EP0(a) + MAJ(a,b,c); \
h = g; \
g = f; \
f = e; \
e = d + t1; \
d = c; \
c = b; \
b = a; \
a = t1 + t2; \
\
t1 = h + EP1(e) + CH(e,f,g) + k[i+4] + m[i+4]; \
t2 = EP0(a) + MAJ(a,b,c); \
h = g; \
g = f; \
f = e; \
e = d + t1; \
d = c; \
c = b; \
b = a; \
a = t1 + t2; \
\
t1 = h + EP1(e) + CH(e,f,g) + k[i+5] + m[i+5]; \
t2 = EP0(a) + MAJ(a,b,c); \
h = g; \
g = f; \
f = e; \
e = d + t1; \
d = c; \
c = b; \
b = a; \
a = t1 + t2; \
\
t1 = h + EP1(e) + CH(e,f,g) + k[i+6] + m[i+6]; \
t2 = EP0(a) + MAJ(a,b,c); \
h = g; \
g = f; \
f = e; \
e = d + t1; \
d = c; \
c = b; \
b = a; \
a = t1 + t2; \
\
t1 = h + EP1(e) + CH(e,f,g) + k[i+7] + m[i+7]; \
t2 = EP0(a) + MAJ(a,b,c); \
h = g; \
g = f; \
f = e; \
e = d + t1; \
d = c; \
c = b; \
```

```
 b = a; \
 a = t1 + t2; \
 }

#define SHA256_COMPRESS_4X \
 for (i = 0; i < 64; i+=4) { \
 t1 = h + EP1(e) + CH(e,f,g) + k[i] + m[i]; \
 t2 = EP0(a) + MAJ(a,b,c); \
 h = g; \
 g = f; \
 f = e; \
 e = d + t1; \
 d = c; \
 c = b; \
 b = a; \
 a = t1 + t2; \
 \
 t1 = h + EP1(e) + CH(e,f,g) + k[i+1] + m[i+1]; \
 t2 = EP0(a) + MAJ(a,b,c); \
 h = g; \
 g = f; \
 f = e; \
 e = d + t1; \
 d = c; \
 c = b; \
 b = a; \
 a = t1 + t2; \
 \
 t1 = h + EP1(e) + CH(e,f,g) + k[i+2] + m[i+2]; \
 t2 = EP0(a) + MAJ(a,b,c); \
 h = g; \
 g = f; \
 f = e; \
 e = d + t1; \
 d = c; \
 c = b; \
 b = a; \
 a = t1 + t2; \
 \
 t1 = h + EP1(e) + CH(e,f,g) + k[i+3] + m[i+3]; \
 t2 = EP0(a) + MAJ(a,b,c); \
 h = g; \
 g = f; \
 f = e; \
 e = d + t1; \
 d = c; \
 c = b; \
 b = a; \
 a = t1 + t2; \
 }

#define SHA256_COMPRESS_2X \
 for (i = 0; i < 64; i+=2) { \
 t1 = h + EP1(e) + CH(e,f,g) + k[i] + m[i]; \
 t2 = EP0(a) + MAJ(a,b,c); \
 h = g; \
 g = f; \
 f = e; \
```

```
 e = d + t1; \
 d = c; \
 c = b; \
 b = a; \
 a = t1 + t2; \
 \
 t1 = h + EP1(e) + CH(e,f,g) + k[i+1] + m[i+1]; \
 t2 = EP0(a) + MAJ(a,b,c); \
 h = g; \
 g = f; \
 f = e; \
 e = d + t1; \
 d = c; \
 c = b; \
 b = a; \
 a = t1 + t2; \
}

 #define SHA256_COMPRESS_1X \
 for (i = 0; i < 64; i++) { \
 t1 = h + EP1(e) + CH(e,f,g) + k[i] + m[i]; \
 t2 = EP0(a) + MAJ(a,b,c); \
 h = g; \
 g = f; \
 f = e; \
 e = d + t1; \
 d = c; \
 c = b; \
 b = a; \
 a = t1 + t2; \
 }
#endif
```

Signed Declaration Sheet

I, the undersigned, declare that this research is my original work and has not been presented for a degree in any other university and that all sources of materials used for the research have been acknowledged.

Declared by:

Name: Seid Mehammed Abdu

Signature: _____

Date: _____

Confirmed by advisor:

Name: Dr. Dagmawi Lemma (PhD)

Signature: _____

Date: _____