Evaluating Memory-Hard Proof-of-Work Algorithms on Three Processors

Zonghao Feng
Hong Kong University of Science and Technology
Kowloon, Hong Kong
zfengah@cse.ust.hk

Qiong Luo
Hong Kong University of Science and Technology
Kowloon, Hong Kong
luo@cse.ust.hk

ABSTRACT

Most public blockchain systems, exemplified by cryptocurrencies such as Ethereum and Monero, use memory-hard proof-of-work (PoW) algorithms in consensus protocols to maintain fair participation without a trusted third party. The memory hardness, or the amount of memory access, of these PoW algorithms is to prevent the dominance of custom-made hardware of massive computation units, in particular, application-specific integrated circuit (ASIC) and field-programmable gate array (FPGA) machines, in the system. However, it is unclear how effective these algorithms are on general-purpose processors.

In this paper, we study the performance of representative memory-hard PoW algorithms on the CPU, the Graphics Processing Unit (GPU), and the Intel Knights Landing (KNL) processors. We first optimize each algorithm for individual processors, and then measure their performance with number of threads and memory size varied. Our experimental results show that (1) the GPU dominates the CPU and the KNL processors on each algorithm, (2) all algorithms scale well with number of threads on the CPU and KNL, and (3) the size of accessed memory area affects each algorithm differently. Based on these results, we recommend CryptoNight with scratchpads of different sizes as the most egalitarian PoW algorithm.

PVLDB Reference Format:
Zonghao Feng and Qiong Luo. Evaluating Memory-Hard Proof-of-Work Algorithms on Three Processors. *PVLDB*, 13(6): 898-911, 2020.
DOI: https://doi.org/10.14778/3380750.3380759

1. INTRODUCTION

Blockchain technologies have been under rapid development, and a major application is cryptocurrencies, such as Bitcoin [34], Ethereum [44], and Monero [42]. Promising applications have also been demonstrated in healthcare [21], supply chain [36], financial services [46], and other areas. Recently, database researchers have started analyzing private blockchains [20], scaling up blockchains [19], and developing blockchain-based storage engines [43], databases [23, 35], cloud data management [33], and data provenance [39]. In this paper, we study the performance of several representative memory-hard proof-of-work (PoW) algorithms, which are key components in leading cryptocurrency systems.

Cryptocurrencies are public blockchain systems, where distributed participants do not fully trust each other but must reach consensus for their data blocks (transactions) to be appended to the blockchain. PoW algorithms [22] are used to reach such consensus. The core idea is that a service requestor must prove to a service provider that the requestor has spent a certain amount of computational effort, and that the provider can verify this effort in a short time.

Because the success of appending data blocks to the chain depends on the speed of executing PoW algorithms, cryptocurrency participants, or called miners, utilize powerful computers for this purpose. These computers include not only general-purpose CPUs and GPUs, but also application-specific integrated circuit (ASIC) and field-programmable gate array (FPGA) machines. Such imbalance in computing power between participants challenges the democratization of the system, and may create opportunities for security threats.

To prevent the dominance of powerful computers, in particular, ASICs and FPGAs, recent cryptocurrencies adopt memory-hard PoW algorithms, which require a large amount of memory access. The goal is to have a similar PoW performance on various processors. These PoW algorithms effectively discourage the use of ASICs and FPGAs because their memory performance is limited and will be costly to boost. However, it is unclear how effective these memory-hard PoW algorithms achieve the democratization purpose on general-purpose computers, as CPUs and GPUs differ considerably on processor architectures as well as memory performance. Therefore, we study the performance of memory-hard PoW algorithms on three types of general-purpose processors - the CPU, the GPU, and the Intel Xeon Phi Knights Landing (KNL) processors.

We select three representative memory-hard PoW algorithms for our study. The first algorithm under study is CryptoNight [18], which is one of the most popular memory-hard PoW algorithms, designed by the CryptoNote foundation. Its performance is bound by memory latency because the algorithm performs a sequence of random memory accesses mixed with reads and writes in a 2MB scratchpad in the memory. The second algorithm we study is Ethash

Proceedings of the VLDB Endowment, Vol. 13, No. 6
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3380750.3380759

[24], which is another notable memory-hard PoW algorithm, used by Ethereum [44]. It works by fetching data from a randomly generated dataset, specifically a directed acyclic graph (DAG) in memory, which is typically several gigabytes in size. The third algorithm under study is Cuckoo Cycle [41]. It finds cycles in a randomly generated bipartite graph by visiting all edges and marking vertices whose degrees are less than or equal to one. The bipartite graph takes several gigabytes of memory, the accesses to the graph are sequential reads, and those to the mark arrays are random reads and writes.

In this paper, we study memory-hard PoW algorithms on general-purpose processors, because these techniques are not only essential for public blockchains, but also relevant to in-memory data-intensive operations. However, few studies have been performed to compare the performance of a memory-hard PoW algorithm between general-purpose processors or analyze the performance characteristics of these algorithms. To the best of our knowledge, this study is the first to fill the gap.

Specifically, we first take the open-source versions of the memory-hard PoW algorithms, and implement and optimize them on the CPUs, the GPUs, and the KNL. We then evaluate the overall performance of these optimized implementations in their default parameter settings. Next, we vary the number of threads and the size of accessed memory area to examine the scalability and memory performance characteristics. Finally, we analyze the results on processor profiling, memory throughput, and power consumption.

Our contributions can be summarized as follows:

- We evaluate the overall performance of each optimized algorithm on seven processors (two CPUs, one KNL, and four GPUs).
- We identify the performance factors of each algorithm and examine their effect.
- We collect and analyze various hardware profiling results, such as cache hit rate, memory throughput, and power consumption.
- We provide suggestions for selecting PoW algorithms in blockchains and designing egalitarian PoW algorithms based on our experiment results.

The remainder of this paper is organized as follows. We first introduce the background on blockchain, PoW algorithms, and computer processors in Section 2. Then we describe details of the memory-hard PoW algorithms under study in Section 3. The implementations and optimizations of each algorithm on different processors are presented in Section 4, and the experimental results along with our analysis and findings are shown in Section 5. Finally, we discuss related work in Section 6 and conclude in Section 7.

2. BACKGROUND

2.1 Blockchain

We first introduce the concepts and terminologies in blockchain systems using the Bitcoin as an example. The Bitcoin whitepaper [34] was published in 2008 by Satoshi Nakamoto, whose identity remains unknown. It proposed a fully decentralized implementation of a *distributed ledger*, i.e., an asset database that is shared among multiple sites. In this database, transactions are recorded in a chain of *blocks* connected by hash pointers, as shown in Figure 1. This chain of blocks is called the *blockchain*, and the first block of the blockchain is called the *genesis block*. Except the genesis block, each block contains the hash value of the previous block. Thus, the content of each block cannot be modified, given that the hash function is secure. Nodes (participants) can freely join or leave the Bitcoin network, and every node can verify the validity of past transactions. This way, every node can be the *validator*. To save storage space and reduce verification time, the *Merkle tree* (a binary hash tree) is used to store the transactions.

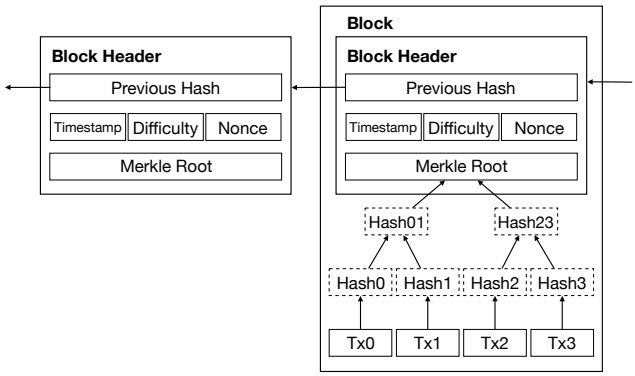

Figure 1: The Bitcoin blockchain

Blockchain systems can be categorized into two types - public or private. *Public blockchains* or *permissionless blockchains* keep the decentralization feature, i.e., nodes are untrusted and can freely join or leave the network. In contrast, in *private blockchains* or *permissioned blockchains*, the identities of participants are known, the data might be private, and trusted third parties may act as transaction validators.

Due to propagation delays or malicious attacks, nodes may have different views of the blockchain, called *forks*. The forks may result in *double spending* problems, i.e., an asset is spent in two distinct transactions. To resolve this issue, consensus protocols must be developed. For example, Bitcoin's consensus protocol is that only the longest chain among the forks is accepted as the finalized transactions, because the longest chain is likely to be the work of honest nodes. The assumption is that honest nodes in the blockchain system possess more than half of the computation power, so that these honest nodes can always produce the longest chain. Finally, forks are not always undesirable. Some consensus protocols accept forks and turn the blockchain into a tree or graph, in order to improve the performance [32].

2.2 Proof-of-Work Algorithms

Proof-of-Work (PoW) algorithms are run by service requestors to solve some computationally hard puzzles for their requests to the service provider. The service provider will first verify if the requestor has finished the work, before the requestor is allowed to have the service. In the context of blockchain systems, the service requestors are the users who want to append a new block to the blockchain, and the service provider is all the users in the blockchain network.

Based on PoW algorithms, consensus protocols in blockchain systems can be formally defined as follows. Given a

Table 1: Comparison of PoW algorithms

Name	Hash Function	Memory-Hardness	Application	Hardware
Hashcash [11]	SHA-256	None	Bitcoin [34]	ASIC
Scrypt [37]	Scrypt	Latency	Litecoin [30]	ASIC
Equihash [15]	BLAKE2b	Bandwidth	Zcash [12]	ASIC
CryptoNight [18]	CryptoNight	Latency	Monero [42]	CPU, KNL, GPU
Ethash [24]	SHA-3 (Keccak)	Bandwidth	Ethereum [44]	GPU
Cuckoo Cycle [41]	-	Latency and Bandwidth	Grin [28]	GPU

cryptographic hash function H, the block content b, and the difficulty of the work d, the block can be appended to the chain if the PoW algorithm can find a nonce n such that $H(n||H(b)) < d$ ($||$ represents concatenation). For example, in Bitcoin, the hash function H is SHA256d (SHA256 done twice), the block b contains the transactions produced during a period of time, and the difficulty d represents the threshold for the final hash value and will increase over time.

Since the hash function used in the PoW algorithm of Bitcoin is simple and fast to run on ASICs, ASIC miners dominate the computing power of the Bitcoin network. This domination is detrimental to the decentralization of Bitcoin because individuals without powerful hardware cannot succeed in executing transactions. To address this problem, complex PoW algorithms are designed and adopted by blockchains. Table 1 shows the PoW algorithms in use by mainstream blockchain applications. They usually combine several hash functions and require a certain amount of memory access.

In our study, we focus on the most popular and representative memory-hard PoW algorithms, namely CryptoNight, Ethash, and Cuckoo Cycle. We categorize these algorithms by the types of memory-hardness, i.e., latency bound and bandwidth bound. Specifically, CryptoNight is memory latency bound, Ethash is bounded by memory bandwidth, and Cuckoo Cycle combines both types of memory-hardness. The two PoW algorithms we omitted from study are Scrypt and Equihash, because they are less memory hard than CryptoNight and Ethash correspondingly.

2.3 Blockchain, PoW, and Database

Blockchain is an increasingly important topic in database research. On the one hand, blockchains have the potential of enhancing various database applications. Recent papers have explored blockchain applications in relational databases [35], data provenance [39], and storage engines [43]. On the other hand, database technologies can be applied to improve blockchain performance. For example, the sharding strategy widely used in distributed databases has been applied to blockchains [19, 23] to increase the throughput and lower the latency.

PoW algorithms play an indispensable role in permissionless blockchains, because they are the core component that enables consensus between untrusted parties. Furthermore, memory-hard PoW algorithms are closely related to in-memory query processing. For example, the Cuckoo Cycle PoW algorithm is inspired by the Cuckoo hash algorithm, so they share common optimizations with hash probing in query processing [38].

In practice, memory-hard PoW algorithms are accelerated by hardware. Similarly, modern parallel processors are widely used to improve the performance of query processing with memory intensive operations [40, 45]. Finally, the design of consensus protocols in distributed databases is a classic problem [33], and PoW-based consensus protocols in blockchains are a special case of the problem.

In summary, memory-hard PoW algorithms and their performance on modern processors are of significance to the database area. Therefore, we are motivated to conduct this experimental study. Based on our study, we make recommendations for selecting and adjusting PoW algorithms in permissionless blockchains.

2.4 Processors

Modern CPUs consist of up to a few tens of cores, and use multiple levels of caches to improve memory access speed. Specifically, the L1 and L2 caches are privately owned by each core, and the L3 cache is shared among all the cores. Depending on the memory type, high-end server CPUs can achieve a bandwidth of more than 100 GB/s. Even though the CPU is the most versatile general-purpose processor, its thread parallelism is limited compared with the GPU.

GPUs are many-core processors that contain thousands of streaming processors (cores), organized into tens of streaming multiprocessors. GPUs use the single instruction, multiple threads (SIMT) execution model, i.e., each instruction is executed concurrently in multiple threads. Threads on the GPU are organized in thread *blocks*, and all threads in a thread block must be assigned to the same multiprocessor. A single routine on the GPU, called a *kernel*, can be run in different *thread configurations*, in terms of number of thread blocks and number of threads per block. Since the thread scheduling unit, called *warp*, consists of 32 consecutive threads, usually the number of threads per block is set to a multiple of 32. The number of thread blocks is set to a multiple of the number of multiprocessors. The memory bandwidth of GPUs is high, which can be close to 900 GB/s.

The CPU and the GPU differ significantly on processor architecture and memory hierarchy. The second generation Intel Xeon Phi processor, code named Knights Landing (KNL), is a many-core processor that represents a middle-of-the-road approach. The number of cores on KNL is 64, and its instruction set and programming APIs are compatible to the CPUs. It is a fully independent self-bootable processor, which does not rely on the CPU and eliminates the data transfer cost. KNL has two vector processing units (VPU) per core, which support the AVX-512 SIMD instruction set. KNL contains 16 GB on-board MCDRAM, which can provide a memory bandwidth of 450 GB/s, about five times higher than DDR RAM. Although existing CPU-based code can run on KNL without any modifications due to the x86 compatibility, architecture-aware optimizations can further improve the performance. Through tuning programs on KNL, some optimizations can also boost the performance on the CPU, which is called the "dual-tuning" effect [29].

Compared with modern multi-core CPUs, many-core processors have a large number of low-performance cores and are especially suitable for data-parallel tasks. Therefore, to achieve the best performance on many-core processors, we need to reduce serial execution and increase parallelism in the program.

In addition to general-purpose processors, specialized processors, e.g., FPGAs and ASICs, are widely used by non memory-hard PoW algorithms for cryptocurrency mining, because customized hardware provides faster instruction execution and reduces power consumption. For example, the mining of Bitcoin is dominated by ASIC miners, which are thousands of times faster than GPUs. However, for memory-hard PoW algorithms, specialized processors have little advantage over CPUs and GPUs due to the complexity of algorithms and high cost of memory. For example, the Antminer E3 [16] for Ethereum has a hash rate of 190 MH/s, which is only 2 times faster than the V100 GPU at the cost of consuming 3 times more energy. Therefore, we target our study on general-purpose processors, because they are most suitable for memory-hard PoW algorithms.

3. MEMORY-HARD POW ALGORITHMS

3.1 CryptoNight

The CryptoNight algorithm was released in 2013, as part of the CryptoNote blockchain system. It contains a memory-hard loop, which performs a sequence of random reads and writes in a small memory area, called a scratchpad. It was designed for running on the CPUs efficiently, because the scratchpad fit into the CPU L2 cache. In contrast, such sequences of random memory accesses were inefficient on the GPU and ASIC machines.

The CryptoNight algorithm consists of three steps. The first step is scratchpad initialization. The size of the scratchpad memory is 2 MB. First, we use the Keccak (SHA3) [13] function to generate a result of 200 bytes. We use the first 32 bytes, byte $0 \cdots 31$, as an AES-256 key. Then we split the next 128 bytes, bytes $64 \cdots 191$, into eight blocks of 16 bytes each, and encrypt each of these eight blocks for 10 AES rounds using the AES-256 key. The encryption result is stored as the first 128 bytes of the scratchpad memory. Then we iteratively perform AES encryption on the current 128 bytes of the scratchpad and append the result as the next 128 bytes in the scratchpad. This process continues until the entire scratchpad is filled.

The second step is the memory-hard loop, as listed in Algorithm 1 and illustrated in Figure 2. Both A and B are 16-byte integers, and are initialized to the XOR result of bytes $0 \cdots 31$ and $32 \cdots 63$ generated using the Keccak function. These integers are then served as addresses in the scratchpad by looking up the 21 least significant bits. We read the value S[A] in the scratchpad using A as the address, and perform AES encryption of S[A] with A to get result C. Then we perform XOR on B and C, and write the XOR result back to S[A]. Next we use C as the address, read S[C] into D, multiply C and D, add the multiplication result to A, and store A to S[C]. Finally, we get the XOR result of A and D as the new A, and C as the new B, and feed the new A and B to the next iteration. Since the output of the AES encryption is random, we can not predict the address, thus the loop can not be parallelized. The loop contains 524,288 iterations, so a sequence of 2 million random reads and writes in total will be performed on the scratchpad.

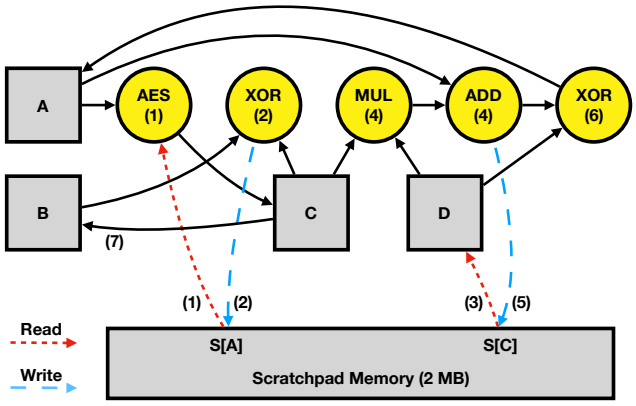

Figure 2: The memory-hard loop of the CryptoNight algorithm. The numbers in parentheses correspond to the line numbers in Algorithm 1.

Algorithm 1: The Memory-hard Loop of the CryptoNight algorithm

Input : Integer A, B, Scratchpad memory S
Output: Modified scratchpad memory S

for $i \leftarrow 1$ **to** 524288 **do**
1 $C \leftarrow AES(S[A], A)$;
2 $S[A] \leftarrow XOR(B, C)$;
3 $D \leftarrow S[C]$;
4 $A \leftarrow ADD(A, MUL(C, D))$;
5 $S[C] \leftarrow A$;
6 $A \leftarrow XOR(A, D)$;
7 $B \leftarrow C$;
8 **end**

The final step is result calculation. Similar to the first step, bytes $32 \cdots 63$ of the initial Keccak result are used as an AES-256 key, and bytes $64 \cdots 191$ are encrypted with the contents in the scratchpad memory, 128 bytes a time. Then the Keccak permutation is performed once on the encryption result. Depending on the two least significant bits of the first byte of the Keccak result, one of the four hash algorithms will be chosen: BLAKE-256, Groestl-256, JH-256, or Skein-256. The selected hash algorithm is applied on the Keccak result to produce the final output of CryptoNight.

With CryptoNight as the consensus protocol of the blockchain, the users who request to append a new block will be required to perform the CryptoNight algorithm repeatedly until they find such a nonce n that holds $H(n||b) \times d < 2^{256}$, where H is the CryptoNight hash function, b is the content of the new block, and d is difficulty. When a new block is broadcast to the blockchain network, the validator will perform the CryptoNight hash function on the new block to verify if the hash value of the new block is less than the given threshold.

3.2 Ethash

Ethash [24], another memory-hard PoW algorithm, is used by Ethereum, which is the second largest cryptocurrency in the world. The memory-hard loop randomly reads slices of

a memory area, called the DAG, which is significantly larger than the scratchpad in CryptoNight. The DAG area is initially 1GB and increases over time. Each slice of the DAG is 128 consecutive bytes.

Ethash updates its parameters every 30000 blocks, which is called an *epoch*. In each epoch, the Ethash algorithm works in four steps. Firstly, the *seed* of the epoch is generated by calculating the SHA3-256 hash value of the previous seed. The seed of the first epoch is set to 32 bytes of zeros. In practice, the seed can be determined by the current block number.

In the second step, Ethash allocates a *light cache*, whose size is determined by the current block number and increases over time. At block number 0, the size of the light cache is 16 MB. The light cache is generated by first filling up the memory sequentially with SHA3-512 hash values, and then applying the RandMemoHash algorithm [31] twice.

The third step generates the *DAG*. The initial size of the DAG is 1 GB. This step is intended to prevent efficient execution on ASICs, since large memory is very costly. The DAG is only updated at the start of each epoch. Each 64-byte item in the DAG depends on 256 items in the light cache and is calculated by the Fowler–Noll–Vo (FNV) hash function [25].

Algorithm 2: The Main Loop of Ethash

Input : *header*, *nonce*, *dag*, *dag_size*
Output: *mix*

```
1  MIX_BYTES ← 128;
2  HASH_BYTES ← 64;
3  mixhashes ← MIX_BYTES/HASH_BYTES;
4  n ← dag_size/HASH_BYTES;
5  w ← MIX_BYTES/4;
6  s ← sha3_512(header + nonce);
7  mix ← [];
8  for j ← 0 to MIX_BYTES/HASH_BYTES do
9  |  mix[j] ← s;
10 end
11 for i ← 0 to 64 do
12 |  p ← fnv(i ∧ s[0], mix[i%w])%(n/mixhashes) ∗
   |    mixhashes;
13 |  newdata ← [];
14 |  for j ← 0 to MIX_BYTES/HASH_BYTES do
15 |  |  newdata[j] ← dag[p + j];
16 |  end
17 |  mix ← fnv(mix, newdata);
18 end
```

The final step is known as the mining process, as shown in Algorithm 2. It is a memory-hard loop with 64 iterations. Each iteration fetches 128 bytes of data from the DAG and mixes these bytes with the results from the previous iteration. Since this loop requires randomly reading data from a large area of memory, which cannot be cached, it is bound by memory latency. However, because the accesses to the DAG in the loop are all reads, a large number of reads can be parallelized to utilize memory bandwidth effectively.

To verify if the Ethash PoW output is valid, a validator will calculate the light cache of the corresponding epoch and regenerate the involved pieces of data in the DAG, without constructing the entire DAG. This design reduces the memory capacity requirement for the validator and also improves the efficiency of validation.

3.3 Cuckoo Cycle

The Cuckoo Cycle [41] algorithm solves a PoW puzzle that finds cycles or other structures in large random graphs. It has the simplest specification among all the PoW puzzles, but its security is not comprised, and has been used in many cryptocurrencies, e.g., Grin [28] and Aeternity [27].

Initially, a bipartite graph is generated using the hash value of the block as the seed. The size of the graph is defined by edge bits N, i.e., the number of edges is 2^N and the number of vertices on each side of the bipartite graph is 2^N. The task is to find a list of edges that form a cycle of a given length in the graph. The difficulty of this puzzle can be adjusted by limiting the hash value of the edge list to be less than a threshold. When the graph is big (e.g., $N = 29$ in Grin), the memory capacity requirement is high.

To solve the Cuckoo Cycle puzzle, one can maintain a directed forest and perform union-find on the forest to locate cycles. However, this simple solver is inefficient in practice. The edge trimming algorithm [8], as shown in Algorithm 3, can greatly reduce the running time and memory usage. The idea is to preprocess the graph by deleting all the vertices of degrees less than or equal to one, which are not in any cycle. Since the bipartite graph is sparse, nearly $2/e \approx 73\%$ of the vertices can be excluded after the first round of trimming. After several rounds of trimming, the trimmed graph is passed to the simple solver to get the result. In practice, edge trimming is the most time-consuming step of the Cuckoo Cycle PoW.

Algorithm 3: The Edge Trimming Step of Cuckoo Cycle

Input : edge bits N, trim rounds r, *edge*

```
1  n ← 2^N;
2  live_left ← [true, n] ;      ▷ New array of size n
3  live_right ← [true, n];
4  for i ← 0 to r do
5  |  count ← [0, n];
6  |  for j ← 0 to n do
7  |  |  if live_right[edge[j].right] then
8  |  |  |  count[edge[j].left] + +;
9  |  |  end
10 |  end
11 |  for j ← 0 to n do
12 |  |  if count[j] < 2 then
13 |  |  |  live_left[j] = false;
14 |  |  end
15 |  end
16 |  Repeat for the right side
17 end
```

4. IMPLEMENTATION & OPTIMIZATION

4.1 Implementation

We first introduce the implementation details of each algorithm. Since the memory sizes and access patterns of these algorithms vary, we select the most suitable parallelization strategies for individual algorithms to improve the performance.

For the CryptoNight algorithm, we let each thread execute the entire CryptoNight function with its private scratchpad and nonce value, because each scratchpad is small, and the accesses to a single scratchpad cannot be parallelized (a subsequent access depends on the address obtained from a previous access). With private nonce values and scratchpads, multiple instances of the PoW puzzle are running independent from each other, and no exchange of information occurs between threads.

Similar to CryptoNight, we parallelize Ethash by having each thread execute the entire Ethash main loop. The difference is that the DAG of Ethash is shared by all the threads, because the DAG size is at several gigabytes and is read-only by all threads. As a result, each thread is running its own instance of Ethash without communicating to each other, and no conflict exists between threads even if they may access the same proportion of the DAG.

Similar to Ethash, the edge trimming step of the Cuckoo Cycle algorithm has read-only access to a graph of several gigabytes in size. Different from the other two algorithms, each round of the edge trimming step visits all edges of the bipartite graph, and updates the counts of the neighbors of each vertex. Therefore, we parallize each round by partitioning the edge set and having each thread visit a subset of edges and update the neighbor counts. The read accesses to the edges have no conflict; however, different threads may update the neighbor count and livelihood mark of the same vertex, which will result in a write-write conflict. Therefore, we use atomic operations on count and livelihood mark updates to ensure the correctness of results.

4.2 Optimizations on CPU

4.2.1 SIMD Vectorization

The cryptographic hash functions are important primitives in PoW algorithms. The computation in these functions is usually suitable for vectorization. Therefore, we use SIMD instructions to vectorize the hash functions in PoW algorithms. Specifically, the SHA-256 hash function in Hashcash, the Keccak hash function in CryptoNight and Ethash, and the Siphash [10] hash function in Cuckoo Cycle are vectorized. We use the AVX2 instruction set on the CPU, which has a vector width of 256 bits.

4.2.2 AES Instruction Set

The Advanced Encryption Standard (AES) encryption is widely used and is an important component of the CryptoNight algorithm. Currently, most CPUs have built-in instructions for AES encryption, called AES-NI (Advanced Encryption Standard New Instructions). We utilized these instructions to improve the efficiency of CryptoNight on the CPU and KNL.

4.2.3 Huge Pages

The default page size on the CPU is 4 KB. Memory-hard PoW algorithms may incur a large number of page faults due to random memory access, so pages will be frequently swapped. The working set of CryptoNight is 2 MB, which requires at least 512 pages of 4 KB page size. By enabling the huge pages option on the CPU and increasing the page size to 2 MB, we can fit the entire scratchpad memory of CryptoNight into a single page. As such, the random reads and writes happen in a single page, and page faults are effectively eliminated.

4.2.4 Loop Unrolling

Most PoW algorithms require multiple rounds of computation, and the number of iterations is known. Therefore, we unroll the loops in the CPU implementations through the `#pragma unroll` directive. Through loop unrolling, the branch instructions are minimized, and the resource used per thread can be reduced. We also fine tune the unroll factor to achieve best performance.

4.3 Optimizations on KNL

Most optimizations on the CPU are applicable to the KNL processor as well. Moreover, the KNL processor supports additional features including the AVX-512 instruction set and high-bandwidth MCDRAM.

Three memory modes are available on the KNL. They differ by the strategy of utilizing the MCDRAM on the KNL. In the cache mode, the MCDRAM is completely assigned as a cache between the KNL processor and the CPU's DDR main memory. In the flat mode, the MCDRAM is used as program-allocatable memory to increase the total size of available memory for the KNL. In the hybrid mode, the MCDRAM is partitioned into a cache and a program-allocatable memory area with a given parameter on the portion [29].

We use the flat mode in our experiments, because the programmer can explicitly manage the MCDRAM in the flat mode, so that frequently accessed data can be manually allocated in the MCDRAM. In contrast, in the cache mode the MCDRAM cannot be explicitly managed in the programs.

As a KNL program can program both the CPU's DDR RAM and its own MCDRAM in the flat mode, we use the `numactl` utility to specify the memory type, MCDRAM and DDR RAM, for memory allocation. The MCDRAM has higher bandwidth but larger latency than DDR RAM. Therefore, using MCDRAM in bandwidth-bound PoW algorithms can improve the performance.

4.4 Optimizations on GPU

The implementations of PoW algorithms on the GPU share the main parallelization strategy with the CPU and the KNL. Some optimizations, e.g., loop unrolling, can also be applied to the GPU. However, other optimizations must be adjusted to suit the GPU architecture.

Firstly, the number of threads in the CPU and KNL programs is limited roughly to the number of cores, and the SIMD vectorization is also limited to the vector width. In contrast, the GPU program can be executed by a massive number of light-weight threads organized in thread blocks, so the execution model is SIMT (single instruction, multiple threads). Therefore, we experiment with a wide parameter range on number of thread blocks and number of threads per block for each algorithm on the GPU.

Secondly, the GPU cache sizes are very limited, but its global memory bandwidth is high. Furthermore, the concurrent memory accesses of a warp of threads to consecutive addresses can be coalesced. Specifically, the memory coalescing optimization is suitable for many cryptographic hash functions, which are essential for PoW algorithms. For CryptoNight, we achieve memory coalescing for AES encryption using the CUDA build-in vector data type. The

Table 2: Hardware configurations

Processor	CPU1	**CPU2**	**KNL**	GPU1	GPU2	GPU3	**GPU4**
Model	i7-3770	Xeon Gold 5115	Xeon Phi 7210	GTX 670	Tesla K80	GTX 1080 Ti	Tesla V100
# Cores	4	20	64	1344	4992	3584	5120
Base Frequency (MHz)	3400	2400	1300	915	562	1481	1246
Max Frequency (MHz)	3900	3200	1500	980	824	1582	1380
Memory Type	DDR3	DDR4	MCDRAM	GDDR5	GDDR5	GDDR5X	HBM2
Memory Size (GB)	32	256	16	4	24	11	16
Bandwidth (GB/s)	25.6	115.2	450	192	480	484	897
L1 Cache (KB)	32	32	32	16	16	48	128
L2 Cache (MB)	8	20	32	0.5	1.5	2.75	6
TDP (W)	77	85	215	170	300	250	250

Table 3: Performance of PoW algorithms. H/s denotes hashes per second, MH/s denotes million hashes per second, and G/s denotes graphs per second.

	CPU1	**CPU2**	**KNL**	GPU1	GPU2	GPU3	**GPU4**
Hashcash (MH/s)	25.21	190.26	191.18	159.31	577.74	1832.95	2724.92
CryptoNight (H/s)	128.513	552.95	1005.2	91.21	478.57	780.14	1541.10
Ethash (MH/s)	0.75	2.31	3.09	16.30	27.97	32.63	94.21
Cuckoo Cycle (G/s)	0.24	0.67	0.59	-	0.85	3.92	5.39

SHA3 hash functions in Ethash are optimized in a similar manner. The Cuckoo Cycle algorithm, however, contains no hash function, but its sequential access to the edge set is coalesced.

5. EXPERIMENTS

After implementing and optimizing the PoW algorithms, we compare and analyze their performance. Firstly, we compare the overall performance of each algorithm on different processors and identify the key performance factors. Secondly, we investigate the impact of these performance factors. Finally, we examine the hardware profiling results on latency, bandwidth, and power consumption.

5.1 Experimental Setup

The list of processors is shown in Table 2. We use both low-end and high-end CPUs and GPUs. In the experiments that perform comparison between different platforms, unless specified, we select the processors with the best performance, i.e., CPU2, KNL, and GPU4. All the experiments are conducted on the CentOS system.

Our implementations are based on the state-of-the-art open-source projects [1, 2, 3, 4, 5] with our optimizations and modifications. Our programs are compiled using GCC 5.5 for CPU and KNL, and NVCC 10.1 for GPU. To collect statistics of hardware events, we use the GNU `perf` tool to monitor the performance on CPU and KNL, and `nvprof` to record GPU events and metrics.

The workloads of the algorithms under study are set as follows, following typical setups of the original algorithms. For Hashcash and CryptoNight, the input data is randomly generated. For Ethash, we start from block number 10000, which has a DAG size of 1 GB. For Cuckoo Cycle, we set edge bits to 29, which corresponds to the number of 2^{29} vertices on each side of the bipartite graph, and 2^{29} edges between the two sides.

The PoW algorithm performance is usually measured by the hash rate, i.e., the number of hash function calls divided by the elapsed time. In the case of Cuckoo Cycle, the hash rate is replaced by the number of graphs processed divided by the elapsed time. However, the absolute hash rates of different algorithms are not comparable, since they are determined by the algorithm design and parameter settings. Rather, we mainly compare the hash rate of each algorithm on different processors to examine its effectiveness in achieving the original goal of memory-hard PoW algorithms. In addition, we include Hashcash, Bitcoin's PoW algorithm as a baseline, since it has no memory-hardness design.

5.2 Performance Overview

Table 3 shows the performance of PoW algorithms on different processors. The performance number of Cuckoo Cycle on GPU1 is missing because Cuckoo requires 5.7 GB of memory to run on the test dataset, which exceeds the 4 GB memory capacity of GTX 670.

For each algorithm, we observe the performance increase from CPU1 to CPU2, and from GPU1 to GPU4. The KNL performance is mostly between that on CPU2 and GPU4 with the exception of Cuckoo Cycle where the KNL performance is slightly lower than CPU2. This increasing overall performance trend is due to the increase of computation

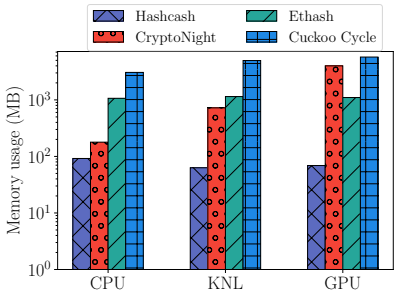

Figure 3: Comparison of memory footprint

power and/or memory bandwidth. In the end, the GPU is the best performer for each algorithm with a speedup of 1.5x-30x over KNL and 1.8x-40x over CPU2. This wide performance gap between processors suggests that current memory-hard PoW algorithms are in general ineffective in maintaining the democratization of blockchain systems. The best algorithm for the democratization purpose is CryptoNight, which yields the least performance difference across processors among the four algorithms.

Compared to CPU2, KNL has higher parallelism since it has 3 times as many as number of cores, and 3.9 times of the memory bandwidth. This performance advantage is best taken by CryptoNight, achieving a speedup of 1.8x on KNL over CPU2. This speedup is mainly from the larger number of threads on the KNL than on the CPU2, each of which executes the CryptoNight function on its small-sized private scratchpad. In comparison, KNL has no advantage over CPU2 on Hashcash, is slightly slower than CPU2 on Cuckoo Cycle, and outperforms Ethash by 30% on Ethash. These puzzling results suggest that KNL's large memory bandwidth or high core-counts do not benefit these algorithms, including the memory-hard ones.

Next we compare GPU4 with CPU2, which constitutes a 100X difference in computation power and eight times of difference in memory bandwidth. The highest speedup of GPU4 over CPU2 is achieved by Ethash, which benefits from both the high memory bandwidth and parallel computation. The second highest speedup of GPU4 over CPU2 is on Hashcash, which is computation bound and benefits from the parallel computation. In contrast, CryptoNight is memory latency bound, which sharply reduces the advantage of the GPU. Also, the GPU cache size is much smaller than the CPU's (see Table 2), which is insufficient for storing the scratchpad of CryptoNight, thus frequent global memory accesses occur. Finally, Cuckoo Cycle is bounded by global memory access bandwidth, and benefits greatly from the high memory bandwidth. The memory footprint of the four algorithms is illustrated in Figure 3.

In summary, CryptoNight is memory latency-bound and favors processors with large caches, such as CPU and KNL, but not friendly to GPU. Ethash is bandwidth-bound, which is ideal for GPU but not good for CPU or KNL. Cuckoo Cycle is both latency and bandwidth bound, and can benefit from the GPU significantly. Overall, CryptoNight achieves the best fairness among all PoW algorithms.

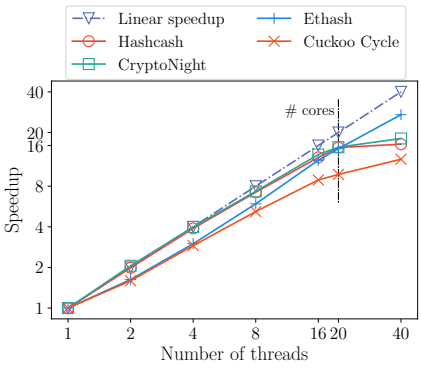

Figure 4: Scalability on CPU

5.3 Scalability

In order to understand the effect of parallelization of each algorithm on each processor, we measure the scalability of PoW algorithms with respect to the number of threads on each processor.

The speedups with the number of threads varied on the CPU are shown in Figure 4. All the PoW algorithms achieve nearly linear speedups with the number of threads up to the number of cores. When hyperthreading is enabled, only Ethash keeps speeding up, because it is bandwidth bounded and the peak bandwidth is not reached yet. Cuckoo Cycle shows the lowest speedup among all algorithms.

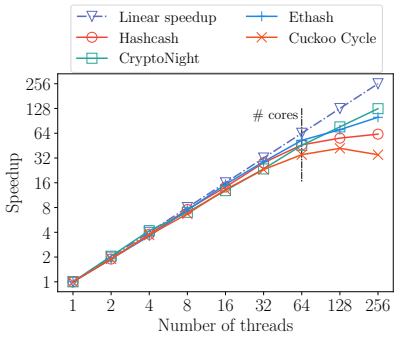

Figure 5: Scalability on KNL

The speedups with number of threads varied on the KNL are shown in Figure 5. The curves are similar to those on the CPU, which are close to linear, when the number of threads are less than the number of cores. The curves diverge considerably when the number of threads goes beyond the number of cores. Both CryptoNight and Ethash keep scaling whereas Hashcash scaling slows down and Cuckoo Cycle's performance drops when the number of threads reaches 256.

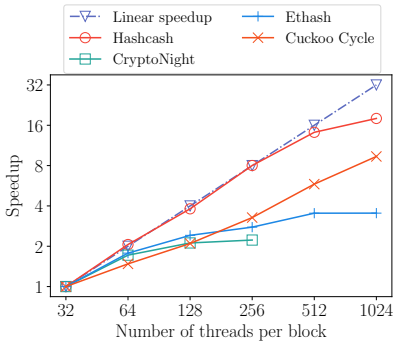

Figure 6: Scalability on GPU with the number of threads per block varied

As for the GPU, since the threads are organized in thread blocks, we evaluate the effect of both the number of threads per block and the number of thread blocks. We first fix the number of thread blocks to 80, the number of streaming multiprocessors on GPU4, and measure the speedups with the number of threads per block varied, as shown in Figure 6. The Hashcash PoW algorithm achieves a nearly linear speedup, since it is purely computation bound. The CryptoNight algorithm cannot run with 512 threads or more,

due to the limit of global memory size. Cuckoo Cycle scales better than Ethash, because its workload does not change whereas Ethash's workload increases with the number of threads, and saturates the bandwidth earlier than Cuckoo Cycle.

Next, we fix the number of threads per block to 32, the warp size, and change the number of thread blocks. The speedups are shown in Figure 7. The bottom left of the figure (i.e., 80 to 2560 thread blocks) resembles Figure 6, because the total number of threads is in the same range as in Figure 6 (80*32 to 2560*32, or 32*80 to 1024*80). When the number of thread blocks goes beyond 5120, Hashcash still maintains the best speedup. Both Ethash and Cuckoo Cycle's performance flattens out with more than 5120 thread blocks, because they both saturate the memory bandwidth.

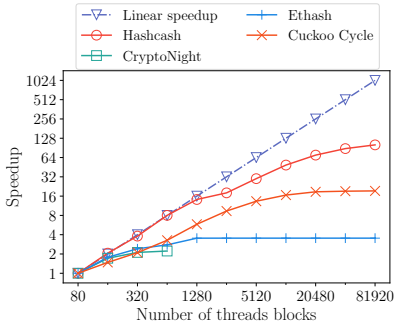

Figure 7: Scalability on GPU with the number of thread blocks varied

5.4 Effect of Memory Size

In real-life blockchain applications, the block update interval needs to be stable, which is maintained by monitoring the hash rate of the network and adjusting the difficulty of PoW. Therefore, most PoW algorithms support difficulty control by changing parameters. In addition to changing the thresholds of hash values, memory-hard PoW algorithms can be tuned by changing the size of accessed memory area. Therefore, we study the performance effect of the memory size in each algorithm.

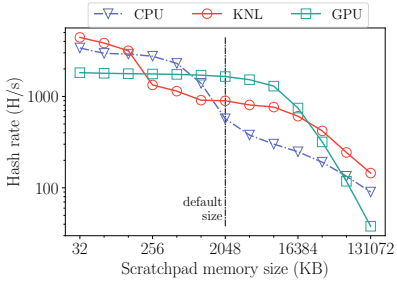

Figure 8: The hash rates of CryptoNight with scratchpad memory size varied

We first evaluate the performance of CryptoNight with the size of the scratchpad memory varied from 32 KB to 128 MB. The hash rates on different processors are shown in Figure 8. On all three processors, the hash rate decreases with the increase of scratchpad memory size. We observe that the performance of the CPU significantly drops when the scratchpad memory increases from 512 KB to 1 MB. The reason is that the L2 cache available for each thread on the CPU is 512 KB. When the scratchpad memory exceeds the capacity of L2 cache, reads and writes to the scratchpad become costly due to cache misses. Similarly, the hash rate of KNL reduces by half when the scratchpad memory increases to 256 KB, because the L2 cache per thread is 128 KB. As for the GPU, the performance is nearly constant when the scratchpad memory is less than 16 MB. This is because reads and writes to the scratchpad memory are all random global memory accesses on the GPU, regardless of the scratchpad memory size. The shared memory of the GPU cannot be utilized for storing the scratchpad, since it is dedicated to AES encryptions. When the scratchpad memory size exceeds 16 MB, the parallelism of GPU is limited by its global memory size, thus the hash rate continuously decreases and eventually becomes worse than the CPU and KNL. The crossovers of the performance curves of CryptoNight on three processors suggest that it is feasible for CryptoNight to democratize the blockchain by including multiple scratchpads of different sizes, for example, hundreds of kilobytes, tens of megabytes, and a few gigabytes. This way, the algorithm poses memory hardness to all three kinds of processors, and it will be challenging to design ASICs to perform well on all scratchpads of various sizes.

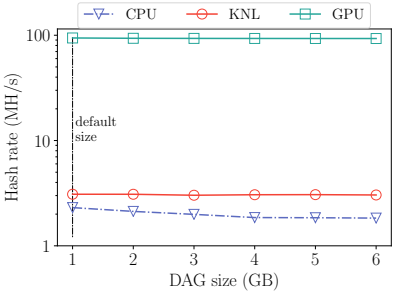

Figure 9: The hash rates of Ethash with DAG size varied

We next investigate how the DAG size of Ethash impacts the performance. The initial size of the DAG is 1 GB, and increases with the block number. The DAG size of Ethash in the real world was 3.13 GB at block number 8195807 as of July 2019. We show the hash rates of our Ethash with DAG size varied from 1GB to 6GB in Figure 9. The DAGs of size 1 GB to 6 GB correspond to block number 10000, 3839999, 7679999, 11519999, 15359999, and 19199999, respectively. With the increase of the DAG size, the hash rate on both the KNL and the GPU slightly decreased by about 1%. The hash rate on the CPU drops about 20% when the DAG size increases from 1GB to 6 GB. Overall, the DAG size of Ethash has little impact on its hash rate, and the performance advantage of the GPU is significant in Ethash.

Lastly, we analyze the impact of graph size in the Cuckoo Cycle algorithm. The Cuckoo Cycle algorithm generates a bipartite graph with N edges and $2N$ vertices, where N is set to 2^i. We show the hash rate of Cuckoo Cycle with $i = \{25, \ldots, 29\}$ in Figure 10. In the figure, the hash rate H drops logarithmically on the CPU and GPU with N. Even though the hash rates all drop on three processors with the

increase of graph size, the performance gap between the GPU and the CPU is wide, and the performance of KNL is worse than the CPU. Since Cuckoo Cycle is more efficient on small graphs, the graph size can be used for adjusting difficulty.

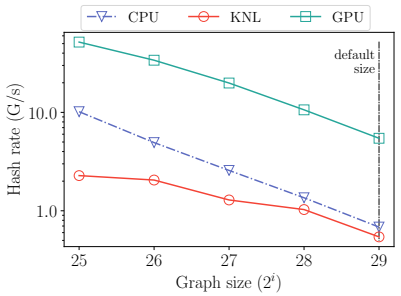

Figure 10: The hash rates of Cuckoo Cycle with graph size varied

5.5 Effect of Memory Type on KNL

The MCDRAM of KNL provides five times higher bandwidth than DDR RAM. However, the latency of MCDRAM is nearly 20% higher than DDR RAM. Therefore, utilizing MCDRAM on memory latency-sensitive applications will usually have negligible impact, or even result in worse performance.

Table 4 shows the performance of PoW algorithms on KNL using DDR RAM and MCDRAM. The performance of Hashcash is almost identical on the two types of memory, because it is computation-bound with few memory operations. Surprisingly, CryptoNight and Ethash show speedups of 1.95 times and 1.52 times respectively using MCDRAM than DDR RAM, whereas the performance of Cuckoo Cycle remains the same. These results suggest that the amounts of parallel memory accesses in CryptoNight and Ethash are large enough to hide the access latency whereas those in Cuckoo Cycle are not.

Table 4: Comparison between memory types on KNL

	DDR RAM	MCDRAM	Speedup
Hashcash (MH/s)	191.06	190.79	0.99
CryptoNight (H/s)	514.6	1004.0	1.95
Ethash (MH/s)	2.03	3.09	1.52
Cuckoo Cycle (G/s)	0.59	0.59	1.00

5.6 Latency Analysis

5.6.1 GPU Issue Stall

We analyze the reasons of GPU issue stalls to show the microperformance characteristics of these PoW algorithms. Specifically, we measured eight types of issue stall reasons, namely instruction fetch, execution dependency, memory dependency, synchronization, pipe busy, not selected, memory throttle, and others.

The statistics of issue stall reasons are shown in Table 5. Memory dependency stall is the major type of stalls in all three memory-hard PoW algorithms, which counts for more than 90 % of the stalls in CryptoNight and Ethash, and 67 % in Cuckoo Cycle. A memory dependency stall occurs in a warp when a load/store instruction cannot be issued. In Cuckoo Cycle, memory throttle stall is another significant stall reason, which takes about 22 %. Memory throttle stall is usually caused by an overwhelming number of memory instructions, which exceeds the capacity of memory data paths.

Table 5: Types of stalls on GPU. The reasons that account for more than 20% in each algorithm are highlighted.

	Hashcash	CryptoNight	Ethash	Cuckoo Cycle
Mem. dependency	0.00%	**97.65%**	**94.82%**	**67.62%**
Mem. throttle	0.00%	0.00%	0.00%	**22.86%**
Exec. dependency	11.88%	1.97%	2.35%	6.37%
Synchronization	0.00%	0.00%	0.00%	1.13%
Instruction fetch	5.23%	0.14%	0.70%	1.11%
Not selected	**40.05%**	0.00%	1.20%	0.46%
Pipe busy	**41.6%**	0.11%	0.64%	0.35%
Others	1.24%	0.13%	0.29%	0.10%

The domination of memory dependency and throttle stall indicates that these algorithms are indeed memory-hard. As a comparison, the Hashcash algorithm has no memory stall at all, but is mainly stalled by the shortage of compute resources, such as pipeline busy and not selected (the instruction is ready to issue, but the warp of threads is not selected to issue because other warps are selected).

5.6.2 Cache hit rate

We use cache hit rate to examine whether an algorithm is latency bound or not. To calculate cache hit rate precisely, we directly read the hardware event counters of processors. On CPU, we have L1 cache hit rate:

$$\frac{mem\_load\_uops.l1\_hit}{mem\_load\_uops.l1\_hit + mem\_load\_uops.l1\_miss}$$

and L2 cache hit rate:

$$\frac{mem\_load\_uops.l2\_hit}{mem\_load\_uops.l2\_hit + mem\_load\_uops.l2\_miss}$$

.

We calculate hit rates using counters of retired memory operations, i.e., operations that have been completely executed, so that failed branch prediction and prefetch operations are excluded. Similarly, we can calculate the L1 cache hit rate of KNL:

$$\frac{mem\_uops.all\_loads - mem\_uops.l1\_miss\_loads}{mem\_uops.all\_loads}$$

and L2 cache hit rate of KNL:

$$\frac{mem\_uops.l2\_hit\_loads}{mem\_uops.l2\_hit\_loads + mem\_uops.l2\_miss\_loads}$$

.

For GPU, we use the metrics provided by nvprof, including $l1\_cache\_global\_hit\_rate$, $l1\_cache\_local\_hit\_rate$, and $l2\_l1\_read\_hit\_rate$. We repeatedly execute the kernel and take the average of hit rates within a small fluctuation range.

Table 6: L1 cache hit rate

	CPU	KNL
Hashcash	99.9%	99.9%
CryptoNight	96.8%	94.2%
Ethash	99.0%	97.2%
Cuckoo Cycle	94.5%	96.8%

The hit rates of L1 cache on CPU and KNL are shown in Table 6. Most algorithms have a hit rate higher than 90%. The L1 cache hit rate of GPUs are not included, since global loads are cached in L2 cache only.

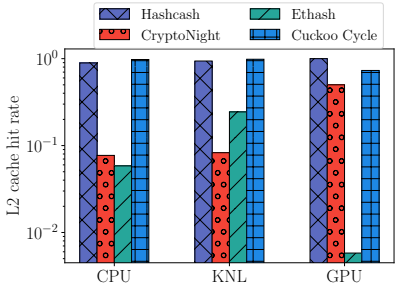

Figure 11: Comparison of L2 cache hit rate on different processors

The L2 cache hit rates of three processors are shown in Figure 11. Hashcash has the best hit rates, since it involves few memory operations. Cuckoo Cycle achieves a cache hit rate of more than 70 % on all processors, because the access on the edge set is sequential access. CryptoNight's hit rates are low on the CPU and KNL due to random accesses, but are high on the GPU due to coalesced access of the AES encryption. In comparison, Ethash's L2 cache hit rates are low on all processors, but on the GPU the latency resulted from the low cache hit rate can be hidden by overlapping data transfer.

5.7 Bandwidth Analysis

To study the memory hardness of the PoW algorithm, we also measure the peak memory throughput. On CPU, the memory throughput can be measured using the *perfmem* tool. On KNL, the metrics of *perf* are not available, so we calculate throughput using event counters. The throughput of reading and writing DDR RAM can be calculated by *unc_m_cas_count.rd* and *unc_m_cas_count.wr*, respectively. The throughput of reading and writing MCDRAM can be calculated by *unc_e_rpq_inserts* and *unc_e_wpq_inserts*, respectively. On GPU, we can get the throughput of all memory types by nvprof metrics, including the read and write throughput of L2 cache, shared memory, device memory, and system memory.

The peak memory read throughput is shown in Table 7. On all three processors, the largest throughput is achieved by Ethash, since it is a bandwidth-bound PoW algorithm. Among three processors, GPU has the highest throughput, which is 79 % of its bandwidth. KNL has demonstrated the advantage of MCDRAM over DDR RAM, but only 27% of its bandwidth is achieved.

Table 7: Peak memory read throughput (GB/s)

	CPU	KNL	GPU
Hashcash	0.47	4.67	0
CryptoNight	6.89	51.32	102.63
Ethash	7.69	119.48	708.75
Cuckoo Cycle	5.77	15.92	176.53

We also calculate the theoretical maximum hash rate of bandwidth-bound PoW as follows. Given the memory bandwidth of the processor B and the size of memory fetched per hash m, the hash rate is $H = B/m$. For the Ethash algorithm, 128 bytes of memory is fetch from the 2GB DAG per step, and each hash requires 64 steps. Therefore, $m = 128\text{bytes} \times 64 = 8\text{KB}$ memory read is performed for each hash. The memory bandwidth of V100 is $B = 897$ GB/s, so the theoretical maximum hash rate of Ethash using GTX 670 is $H = 897/8 = 112.125$ MH/s. We see that the actual hash rate is 84 percent of the theoretical value. In conclusion, Ethash is the only PoW algorithm that can make full use of the high bandwidth of GPUs.

5.8 Power Consumption

One important performance metric of PoW algorithms is power consumption. Since cryptocurrency mining is profitable, professional miners have been actively searching for hardware that can achieve high performance while maintaining a small energy footprint. To compare PoW algorithms on the energy consumption, we use `s-tui` and *nvidia-smi* to monitor the peak power consumption on CPU/KNL and GPU, respectively.

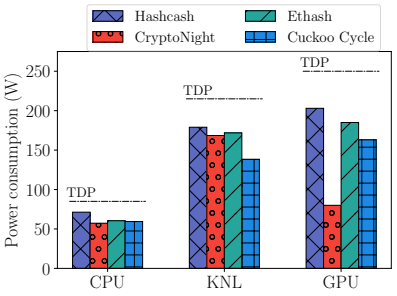

Figure 12: Comparison of power consumption

As shown in Figure 12, Hashcash is the algorithm that consumes the most power on all three processors, because the intensive computation keeps processors busy. As for memory-hard PoW algorithms, the power consumption is 80.3%, 77.3%, and 39.4% of that of Hashcash on CPU, KNL, and GPU, respectively. This is due to the stalls resulted by memory operations. The latency-bound CryptoNight algorithm uses much less power on the GPU, because frequent global memory accesses make the processor waiting for the execution of load/store instructions.

We also calculate Hashes per Joule of each processor to compare their energy efficiency. The results are shown in Table 8. GPU achieves the best energy efficiency among all three processors, while KNL spends the most energy for each hash. On the Ethash algorithm, GPU is 28.3 times

Table 8: Energy efficiency of PoW algorithms on different processors

	CPU	KNL	GPU
Hashcash (MH/J)	2.668	1.068	13.423
CryptoNight (H/J)	9.650	5.966	19.264
Ethash (MH/J)	0.038	0.018	0.509
Cuckoo Cycle (G/J)	0.011	0.004	0.033

more energy efficient than KNL, which shows the advantage of GPUs on bandwidth bound PoW algorithms. The CryptoNight algorithm has the smallest energy gap between processors, because GPUs are not suitable to latency bound tasks. However, GPU still outperforms CPU and KNL by 1.9x and 3.2x, respectively. Therefore, the manycore design of GPU makes it the most profitable mining device among the general-purpose processors.

5.9 Discussion

The selection of PoW algorithms is a critical part of blockchain system design, because it directly affects the security and performance of the blockchain. We observe diverse choices of PoW algorithms in current blockchain systems. Most blockchains are designed to be egalitarian, i.e., aim to eliminate the advantage of specialized hardware and encourage more participants to join. Other blockchains adopt PoW algorithms that are friendly to specialized hardware. Therefore, there is no standard answer to the selection of PoW algorithm, since it depends on the objective of the blockchain system. Nevertheless, in our study we focus on the representative memory-hard PoW algorithms, which are aimed for egalitarian blockchain systems.

To the best of our knowledge, this study is the first to measure and analyze the performance of representative memory-hard PoW algorithms across three types of general-purpose processors. We identify the performance factors including thread parallelism and memory size in general and study their impact. Furthermore, we obtain hardware profiling results on instruction stalls, cache hit ratios, peak memory bandwidth, and power consumption to understand the performance characteristics of the algorithms.

Based on our experimental results, we summarize the characteristics of the three memory-hard PoW algorithms. First, CryptoNight is the most egalitarian PoW algorithm. Second, Ethash is the best memory bandwidth utilizer across all three types of processors. Finally, all three PoW algorithms reduce power consumption by memory stalls.

In addition, we give our suggestion based on our experiment results: A memory-hard PoW algorithm that minimizes the difference between processors can be designed to include multiple scratchpads of different sizes.

6. RELATED WORKS

Memory-hard hash functions have been studied in cryptography. MTP and MHE [14] are two memory-hard functions designed for password hashing and encryption, respectively. These algorithms are memory-hard to prevent attacks from using ASICs to break ciphers and decrypt credential information.

Alwen and Serbinenko [7] developed theoretical tools for measuring amortized memory complexity of memory-hard functions. They define a graph pebbling game to abstract parallel computation. In their subsequent work [6], they used their model to prove the theoretical lower bound of the memory-hard PoW algorithm Scrypt.

Blockbench [20] is a framework for benchmarking the data processing capability of blockchains. It can measure the throughput, latency, and scalability of blockchain systems under various workloads. It is specifically focused on private blockchains, e.g., Hyperledger Fabric [9], where traditional Byzantine fault tolerant algorithms [17] rather than PoW algorithms are used for consensus.

Han et al. [26] conducted a study about memory-hard PoW algorithms used in cryptocurrency mining. It proposed a general structure of memory-hard PoW algorithms and benchmarked three PoW algorithms, namely Ethash, CryptoNight, and Scrypt on GPUs. The comparison between the results of [26] and our GPU results on common experiments are as follows. Firstly, we measured higher memory read throughput than their results, since the GPU we used (V100) has higher bandwidth than theirs (Titan XP). Secondly, we obtained similar results on peak memory usage, i.e., CryptoNight consumes more memory than Ethash on the GPU. Thirdly, our experimental results suggested that memory dependency is the major type of stalls on the GPU, which is consistent with their conclusion. The difference is that we distinguish between latency-bound and bandwidth-bound algorithms and perform detailed comparison between GPU and other processors.

7. CONCLUSION

The memory-hard PoW algorithms in cryptocurrencies were designed to prevent ASIC miners from having great performance advantages over CPUs and GPUs, as it is important for the fairness to allow miners on general-purpose computing hardware to participate. In this paper, we performed an experimental study of memory-hard PoW algorithms on three types of processors. We first demonstrate the impact of thread parallelism, size of accessed memory area, and memory types to the overall performance. Then, we compare PoW algorithms through quantitative analysis of low-level metrics including types of instruction stalls, cache hit rates, memory throughput and power consumption.

We share a few lessons learned through the study, which would be useful for guiding the design of memory-hard PoW algorithms. Firstly, due to the design of different processors, they are suitable for different tasks. Latency-bound tasks are friendly to CPU and KNL, because of their mature cache hierarchy. Bandwidth-bound tasks can fully exploit the potential of GPU. Secondly, since the processors are evolving rapidly, PoW algorithms should be adjustable and flexible so that their properties can be maintained as long as possible. In particular, the performance gaps between the GPU and the CPU are widening for all three memory-hard algorithms, and CryptoNight or other algorithms need to be adjusted to reduce the gap.

Our source code and performance test scripts used in this study are publicly available at `https://github.com/RapidsAtHKUST/pow-bench`.

8. REFERENCES

[1] ccminer. `https://github.com/tpruvot/ccminer`.

[2] cpuminer-multi. https://github.com/tpruvot/cpuminer-multi.
[3] cuckoo. https://github.com/tromp/cuckoo.
[4] ethash. https://github.com/chfast/ethash.
[5] ethminer. https://github.com/ethereum-mining/ethminer.
[6] J. Alwen, B. Chen, K. Pietrzak, L. Reyzin, and S. Tessaro. Scrypt is maximally memory-hard. In *Proceedings of the 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, EUROCRYPT 2017, pages 33–62. Springer, 2017.
[7] J. Alwen and V. Serbinenko. High parallel complexity graphs and memory-hard functions. In *Proceedings of the Forty-Seventh Annual ACM Symposium on Theory of Computing*, STOC '15, pages 595–603. ACM, 2015.
[8] D. G. Andersen. Exploiting Time-Memory Tradeoffs in Cuckoo Cycle. https://www.cs.cmu.edu/~dga/crypto/cuckoo/analysis.pdf, 2014.
[9] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. D. Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolic, S. W. Cocco, and J. Yellick. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 30:1–30:15. ACM, 2018.
[10] J. Aumasson and D. J. Bernstein. Siphash: A fast short-input PRF. In *Proceedings of the 13th International Conference on Cryptology in India*, INDOCRYPT 2012, pages 489–508. Springer, 2012.
[11] A. Back. Hashcash - a denial of service counter-measure. http://www.hashcash.org/papers/hashcash.pdf, 2002.
[12] E. Ben-Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza. Zerocash - Decentralized Anonymous Payments from Bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 459–474. IEEE, 2014.
[13] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche. Keccak. In *Proceedings of the 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques*, EUROCRYPT 2013, pages 313–314. Springer, 2013.
[14] A. Biryukov and D. Khovratovich. Egalitarian computing. In *25th USENIX Security Symposium*, USENIX Security 16, pages 315–326. USENIX Association, 2016.
[15] A. Biryukov and D. Khovratovich. Equihash: Asymmetric proof-of-work based on the generalized birthday problem. *Ledger*, 2(0):1–30, 2017.
[16] Bitmain. Antminer e3. https://shop.bitmain.com/product/detail?pid=00020181012201405249d5Ax96qj0686, 2018.
[17] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, pages 173–186. USENIX Association, 1999.
[18] CryptoNote. Cryptonight hash function. https://cryptonote.org/cns/cns008.txt, 2013.
[19] H. Dang, T. T. A. Dinh, D. Loghin, E.-C. Chang, Q. Lin, and B. C. Ooi. Towards scaling blockchain systems via sharding. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, pages 123–140. ACM, 2019.
[20] T. T. A. Dinh, J. Wang, G. Chen, R. Liu, B. C. Ooi, and K.-L. Tan. Blockbench: A framework for analyzing private blockchains. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 1085–1100. ACM, 2017.
[21] A. Dubovitskaya, P. Novotný, S. Thiebes, A. Sunyaev, M. Schumacher, Z. Xu, and F. Wang. Intelligent health care data management using blockchain: Current limitation and future research agenda. In *Heterogeneous Data Management, Polystores, and Analytics for Healthcare, VLDB 2019 Workshops*, pages 277–288. Springer, 2019.
[22] C. Dwork and M. Naor. Pricing via processing or combatting junk mail. In *Proceedings of the 12th Annual International Cryptology Conference*, CRYPTO '92, pages 139–147. Springer, 1992.
[23] M. El-Hindi, C. Binnig, A. Arasu, D. Kossmann, and R. Ramamurthy. Blockchaindb - A shared database on blockchains. *PVLDB*, 12(11):1597–1609, 2019.
[24] Ethereum. Ethash. https://github.com/ethereum/wiki/wiki/Ethash.
[25] G. Fowler, L. C. Noll, and K.-P. Vo. Fnv hash. http://www.isthe.com/chongo/tech/comp/fnv/index.html, 1991.
[26] R. Han, N. Foutris, and C. Kotselidis. Demystifying crypto-mining: Analysis and optimizations of memory-hard pow algorithms. In *IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS 2019, pages 22–33. IEEE, 2019.
[27] Z. Hess, Y. Malahov, and J. Pettersson. Aeternity blockchain - the trustless, decentralized and purely functional oracle machine. https://www.aeternity.com/aeternity-blockchain-whitepaper.pdf, 2017.
[28] T. E. Jedusor. Grin. https://grin-tech.org, 2016.
[29] J. Jeffers, J. Reinders, and A. Sodani. *Intel Xeon Phi Processor High Performance Programming Knights Landing Edition*. Morgan Kaufmann Publishers Inc., 2016.
[30] C. Lee. Litecoin. https://litecoin.org, 2011.
[31] S. D. Lerner. Strict memory hard hashing functions. http://www.hashcash.org/papers/memohash.pdf, 2014.
[32] C. Li, P. Li, D. Zhou, W. Xu, F. Long, and A. Yao. Scaling nakamoto consensus to thousands of transactions per second. *arXiv preprint arXiv:1805.03870*, 2018.
[33] S. Maiyya, F. Nawab, D. Agrawal, and A. El Abbadi. Unifying consensus and atomic commitment for effective cloud data management. *PVLDB*, 12(5):611–623, 2019.
[34] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf, 2008.
[35] S. Nathan, C. Govindarajan, A. Saraf, M. Sethi, and P. Jayachandran. Blockchain meets database: Design and implementation of a blockchain relational

database. *PVLDB*, 12(11):1539–1552, 2019.

[36] S. R. Niya, D. Dordevic, A. G. Nabi, T. Mann, and B. Stiller. A platform-independent, generic-purpose, and blockchain-based supply chain tracking. In *IEEE International Conference on Blockchain and Cryptocurrency*, ICBC 2019, pages 11–12. IEEE, 2019.

[37] C. Percival. Stronger key derivation via sequential memory-hard functions. `http://www.tarsnap.com/scrypt/scrypt.pdf`, 2009.

[38] K. A. Ross. Efficient hash probes on modern processors. In *Proceedings of the 23rd International Conference on Data Engineering*, ICDE 2007, pages 1297–1301. IEEE, 2007.

[39] P. Ruan, G. Chen, A. Dinh, Q. Lin, B. C. Ooi, and M. Zhang. Fine-grained, secure and efficient data provenance for blockchain. *PVLDB*, 12(9):975–988, 2019.

[40] M. Sha, Y. Li, B. He, and K. Tan. Accelerating dynamic graph analytics on gpus. *PVLDB*, 11(1):107–120, 2017.

[41] J. Tromp. Cuckoo cycle: A memory bound graph-theoretic proof-of-work. In *BITCOIN, WAHC, and Wearable, Financial Cryptography and Data Security, FC 2015 International Workshops*, pages 49–62. Springer, 2015.

[42] N. van Saberhagen. Monero. `https://www.getmonero.org`, 2014.

[43] S. Wang, T. T. A. Dinh, Q. Lin, Z. Xie, M. Zhang, Q. Cai, G. Chen, B. C. Ooi, and P. Ruan. Forkbase: An efficient storage engine for blockchain and forkable applications. *PVLDB*, 11(10):1137–1150, 2018.

[44] G. Wood. Ethereum: A Secure Decentralised Generalised Transaction Ledger. `http://gavwood.com/paper.pdf`, 2016.

[45] Y. Yuan, R. Lee, and X. Zhang. The yin and yang of processing data warehousing queries on GPU devices. *PVLDB*, 6(10):817–828, 2013.

[46] R. Zhu, C. Ding, and Y. Huang. Efficient publicly verifiable 2pc over a blockchain with applications to financially-secure computations. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, pages 633–650. ACM, 2019.