Co-Mining: A Processing-in-Memory Assisted Framework for Memory-Intensive PoW Acceleration

Abstract

Recently, HBM (High Bandwidth Memory) and PIM (Processing in Memory) integrated technology such as Samsung function-in-memory DRAM opens a new door for memory-intensive PoW acceleration by jointly exploiting GPU, PIM and HBM. In this paper, we for the first time propose a GPU/PIM Co-Mining framework to accelerate memory intensive PoW by fully exploiting HBM-PIM's bandwidth and coordinately scheduling mining tasks in both GPU and PIM. Specifically, we first design a linear programming model to intelligently guide the GPU/PIM task scheduling. An extended finite-state-machine model is designed for the GPU memory controller to switch PIM working mode (compute/memory mode) accordingly. Finally, considering the speed difference between intra-/inter-channel memory accesses, a hybrid memory access method is proposed to minimize inter-channel data movements. We evaluate Co-Mining based on Samsung's HBM2-based function-in-memory architecture. The experimental results show that it can achieve up to 38.5% hashrate improvement compared with the method by directly integrating PIM into PoW acceleration with GPU.

1 Introduction

With the great potential to establish decentralized trust between untrusted nodes, blockchain technologies have gained great attention from both academic and industry [25]. To protect the distributed ledger from being tampered by malicious nodes, Proof-of-Work (PoW) algorithms are adopted to achieve consensus among all nodes. It requires contributors to solve a hard puzzle (mining) to generate new blocks, such as in Bitcoin [21] and Ethereum [34]. Based on different design principles, the PoW algorithms can be categorized into computation-intensive and memory-intensive ones.

For computation-intensive PoW algorithms, many ASIC-based and FPGA-based accelerators have been designed to solve the puzzle with hardware parallelism, such as Bitmain ASIC miner [1]. However, different from computation-intensive PoW algorithms, the internal designs of memory-intensive ones enforce randomized memory accesses to make the off-chip memory bandwidth become the PoW bottleneck, making the ASIC and FPGA designs, with the focus on computation acceleration, not effective anymore.

To accelerate memory-intensive PoW algorithms, some recent works [14, 35] start to exploit hiding memory access latencies or increasing memory bandwidth. Particularly, emerging processing-in-memory (PIM) architectures show promising potentials with their computation capability of in-memory NDP (Near-Data-Processing) engines for less

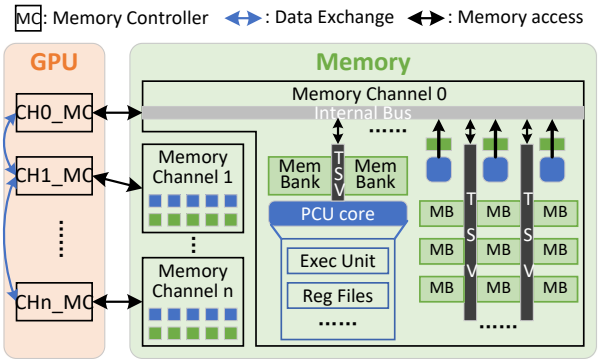


Figure 1. A PIM-integrated GPU architecture.

data movements [10]. However, with the increased memory requirement (e.g. more than 4GB in Ethereum), simply offloading the PoW computation to PIM [10] cannot effectively speed up the PoW anymore because of their unrealistic assumption that all datasets can be fit into one memory channel. When the datasets are too large to be entirely put into one memory channel, the PIM cores need to frequently access inter-channel data, which, as illustrated in Figure 1, must be handled by threads in GPU or CPU, incurring significant overheads with even more data movements than without using PIM. On the other hand, the widely-used GPU-only PoW solution, cannot fully exploit memory bandwidth due to memory channel conflict caused by huge amount of randomized memory accesses issued by the memory-intensive PoW algorithms.

Recent HBM (High Bandwidth Memory) and PIM integrated technology such as Samsung function-in-memory DRAM [18] opens a new door for memory-intensive PoW acceleration by jointly exploiting GPU, PIM and HBM. In the GPU-PIM hybrid architecture, GPU cores can achieve higher hashrate than PIM cores, but they cannot fully exploit the memory bandwidth. Thus, PIM cores can utilize the remaining memory bandwidth to accelerate the mining process. However, since the PIM cores can only handle very simple calculations (no control instructions like branches or loops), GPU cores must be involved to handle the PIM task scheduling [23], which in turn consumes some GPU computation resources and memory bandwidth.

In this paper, we aim to develop a hybrid GPU-PIM Co-Mining framework, in which we separate the GPU threads into hash calculation threads and PIM control threads. The hash calculation thread is used to perform the normal mining process and the PIM control thread is used to schedule the PIM cores. With this Co-Mining framework, we can jointly schedule hash calculation threads and PIM control threads

to accelerate memory-intensive PoW algorithms. To achieve this, however, two challenges must be addressed.

The first challenge is how to coordinately schedule hash calculation threads and PIM control threads to fully exploit memory bandwidth during the mining process. We need to determine when and how to schedule those two types of threads. To address this, we formulate the problem as a linear programming problem, which takes the memory channel bandwidth utilization and the current hashrate throughput into consideration. Since the number of PIM cores is limited, issuing too many PIM control threads may consume too much memory bandwidth and block hash calculation threads, thus degrading the overall performance. By striking a balance between hash calculation threads and PIM control threads, we can explore the memory bandwidth while achieving higher hashrate throughput.

The second challenge is that PIM cores allow to switch between computation mode and memory mode (e.g., HBM-PIM from Samsung [18]), which is controlled via setup instructions by GPU. Improper switches between these two modes would decrease the PIM working efficiency. However, the GPU is unaware of PIM task status because PIM cores cannot actively notify GPU whether their tasks have been accomplished. To tackle this challenge, we extend the finite-state-machine model inside the GPU memory controller to support mode switch of PIM cores, thus enabling to perform fine-grained control for PIM cores. Specifically, we determine whether to switch the mode of one PIM core (and its related memory banks) by evaluating a state transfer probability based on history memory accesses. We divide all the PIM core memory accesses into intra-channel and inter-channel accesses, and propose different control flows for them. By eliminating unnecessary mode switches and minimizing costly inter-channel data movements, the efficiency of PIM cores is maximized.

We have implemented a trace-driven hashrate simulator from the ground to evaluate the efficiency of our Co-Mining framework. For the HBM-PIM platform, we choose a recently fabricated HBM2-based PIM architecture from Samsung [18], which can achieve 307GB/s bandwidth and 1.2TFLOPS FP16 throughput in one cube (6GB, 16 Channels) with 128 programmable computing units (PCUs) integrated. The PoW algorithm used in Ethereum (i.e., Ethash), which is memory intensive, is adopted for illustration and evaluation. By properly scheduling in-memory computation tasks to PIM cores, our Co-Mining framework can leverage both the GPU and PIM cores to increase the overall performance. Our main contributions can be summarized as follows:

- We formalize the GPU/PIM task scheduling problem into a linear programming model and strike a balance between hash calculation threads and PIM control threads, to maximize the memory bandwidth utilization as well as the hashrate throughput.
- We extend the finite-state-machine model in the GPU memory controller to perform PIM mode switches for fine-grained control of PIM cores and introduce different control flows for different memory accesses.
- We adopt a recent HBM-PIM architecture in our Co-Mining framework and evaluate our techniques with Ethash based on a trace-driven hashrate simulator. The simulator has been open-sourced for public access [2]. The results show that we achieve up to 38.5% hashrate improvement compared with the method by directly integrating PIM into PoW acceleration with GPU.

2 Background and Motivation

2.1 Proof-of-Work in Blockchain

A blockchain system can be seen as a distributed public ledger that stores transactions with a linked list of blocks. Blocks are generated and shared over the entire blockchain network to defend against system failure, data manipulation, and cyber attacks [27, 37]. These systems generally adopt a mining process (e.g., PoW, which requires solving a computation and energy-hungry puzzle) [22] to generate new blocks. A newly generated block will be broadcast to the whole network to achieve consensus.

The PoW process is usually implemented by performing several hash functions with the previous block information and a nonce as inputs. The target is to find an output satisfying some constraints (e.g., a fixed number of '0's in front of the result) after a series of hash operations. Since the hash functions are not reversible, the only way to generate such a satisfied output is to keep trying different input nonce. Once a proper nonce is found, it will be utilized to seal the newest block and the miners will get some rewards.

Currently, PoW algorithms can be categorized into computation intensive and memory-intensive ones. The computation intensive PoW algorithms, represented by Bitcoin, focus on complicating the hash functions. Thus, attempting a nonce requires more computation, which makes the mining process more difficult. However, once a blockchain is implemented and released to public, its PoW algorithm cannot be changed (otherwise it will cause a fork), which makes the ASIC and FPGA based mining accelerators popular. For example, there are lots of ASIC solutions accelerating SHA-256 algorithm that used in Bitcoin [1].

To prevent the blockchain from ASIC-caused centralization (more and more mining power is controlled by several big ASIC miners), memory-intensive PoW algorithms are developed such as ***Ethash*** used in Ethereum [34] and ***Scrypt*** used in Litecoin [24]. Different from computation-hungry PoW algorithms, those memory-intensive ones integrate plenty of random memory accesses into their hash calculations. Those algorithms generally contain many iterations with data-dependency and several memory pages will be fetched to finish calculation in each iteration. Although the

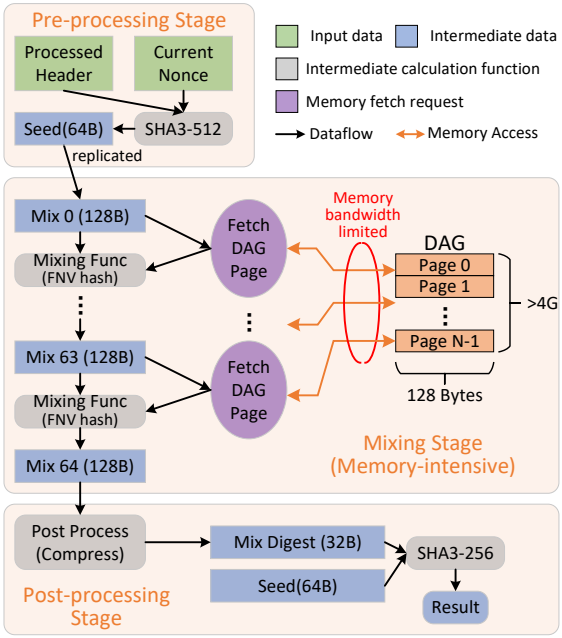


Figure 2. Ethash algorithm details.

ASIC/FPGA solutions can increase computation density for computation-intensive PoW algorithms, it is hard to provide enough memory bandwidth for memory-intensive ones.

2.2 Ethash in Ethereum

The most representative memory-intensive PoW algorithm is ***Ethash*** used in Ethereum. In this paper, we will use ***Ethash*** to illustrate the working process of memory-intensive PoW algorithms. ***Ethash*** keeps trying to find a nonce that generates a satisfied hash output which is less than a predefined precision. The in-memory calculation dataset is called DAG and is divided into multiple 128-byte chunks. Each nonce try involves many random memory accesses, making it hard to exploit data parallelism during the execution. Furthermore, the size of the in-memory dataset is growing over time, which has reached about 4.5GB now [3].

The ***Ethash*** procedure concurrently issues many searching threads with each trying a unique nonce. Once an acceptable nonce is found, the search stops. Figure 2 shows its basic working flow. To test each nonce, it firstly generates a seed using the SHA-512 function during the pre-processing stage and then begins memory fetching in each step of calculation in the mixing stage. The purple ellipse shows the memory fetch request, which involves a 128-byte page at once. Specifically, *FNV* hash function [12] is used for mixing the intermediate result (128 Bytes) with the newly fetched memory data (128 Bytes). Each memory fetching address is determined by the previous mixing result. Because of data dependencies, parallelism can hardly be exploited among those steps.

To finish the whole computation, the mixing function needs to be invoked 64 times (i.e., 64 steps) with non-sequential and wide-spread memory addressing. Due to the dispersion properties, the *FNV* hash produces pseudo-randomized and non-predictable addresses that have no locality, which invalidates caching and prefetching for the mixing stage [11]. In addition, the loop-carried dependency in the mixing stage disables loop-level parallelism. In summary, the mixing stage consumes most of the execution time and memory bandwidth, making the algorithm memory-intensive [35]. Finally, the post-processing stage compresses the mixed data and generates the final result through the SHA-256 function.

2.3 HBM-PIM architecture

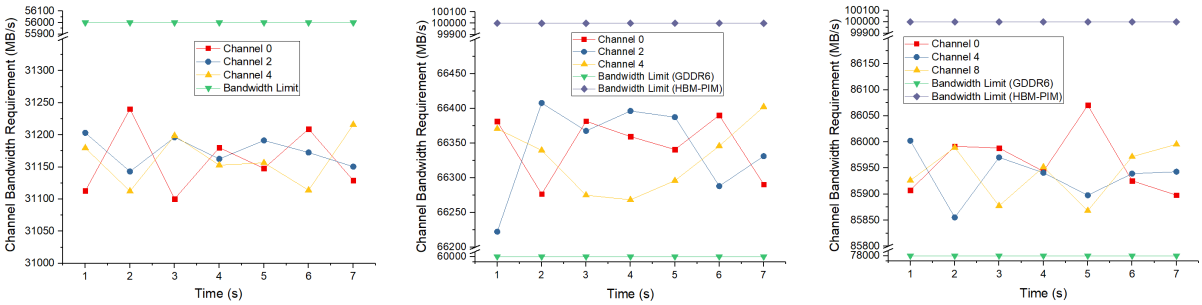
To accelerate memory-intensive applications, PIM is a promising technique since it eliminates data movements between memory and computation units (e.g., GPU). Recently, Samsung fabricates an HBM2-based PIM architecture called HBM-PIM, which reaches 307GB/s bandwidth and 1.2TFLOPS FP16 throughput in one cube (6GB, 16 CHs) with 128 programmable computing units (PCUs) integrated [18]. Figure 1 shows the basic architecture of the HBM-PIM. Specifically, the 6GB memory cube is divided into 16 pseudo channels and each channel has 8 PCU cores (128 PCUs in total). Note that there are many memory controllers in GPU and each one corresponds to one memory channel. Inside each channel, there are many memory banks, and each PCU core is connected with two of them. The PCU core can access other memory banks inside this channel through the internal bus with a high speed. However, for inter-channel memory access, since different memory channels have different timings [30], GPU cores must be involved to issue scheduling instructions.

Each PCU-connected memory bank can only operate in compute mode or memory mode. In the memory mode, all the memory banks act as the normal HBM2 memory to serve GPU memory requests. When one PCU core is activated, all the PCU-related memory banks are transferred to the compute mode, in which those banks can only be accessed by the connected PCU core while blocking other memory requests (from the GPU side or the other PCU cores). The GPU can make a typical memory bank enter its compute mode by issuing an ACT command with the most significant bit of the bank address set to 1, and the exit by issuing an ACT (with highest bit set to 0) followed by a PRE command [20]. When all PCU-connected memory banks exit compute mode, the corresponding PCU core will stop working and reserve its current status in its register files.

PCU cores can execute some basic bit and arithmetic operations. Some fundamental logic, such as ALU/FPU, register files, and load/store unit are implemented in the PCU core. However, it cannot handle any control logic (i.e., branches or loops). Thus, all the PCU control logics should be managed via GPU threads.

2.4 Motivation

Although there are several works that adopt PIM architectures to accelerate memory-intensive PoW algorithms [10],



(a) RTX2060 (6 Channels, 192bit/6GB) (b) RTX3060 (6 Channels, 192bit/12GB) (c) RTX3090 (12 Channels, 384bit/24GB)

Figure 3. The memory bandwidth requirement of Ethash.

Ethash cannot be easily accelerated due to its large memory dataset. On the above-mentioned 6GB HBM-PIM chip with 16 pseudo channels, each channel only has 384MB memory. When we fit the current Ethash dataset ($\approx 4.5GB$) into it, due to the dispersion of memory accesses, more than 90% of data accesses ($\frac{4500MB-384MB}{4500MB} \approx 91.5\%$) will involve another channel, which needs help from GPU cores. However, GPU cores need extra memory bandwidth to issue control commands to PCU cores, which makes it even less efficient than GPU-only implementation. Thus, in this work, we intend to make PIM cores work with GPU cores with a Co-Mining framework.

Our main purpose is to make PCU cores to explore the remaining memory bandwidth that can not be utilized by GPU cores. We have done some preliminary experiments and found that both the mid-end (e.g., RTX2060/3060) and the high-end (e.g., RTX 3090) GPUs can not fully consume their memory bandwidth when running memory-intensive PoW algorithms. Figure 3 indicates the actual memory bandwidth requirement issued by the real Ethash workload. Due to the limited number of CUDA cores, GPU can only generate a certain number of memory requests in a period. When executing on RTX2060 (1920 CUDA cores), Ethash occupies at most 31.5GB/s memory bandwidth in one channel (as Figure 3(a) shows), which is far less than the provided memory bandwidth (336GB/s in 6 channels, 56GB/s per channel). Furthermore, even if the FNV hash function generates widespread and randomized memory accesses to all the channels, the load on each memory channel still reveals slight imbalance ($\frac{31250MB-31100MB}{31100MB} \approx 0.5\%$) because the requests are not ideally evenly distributed.

The conditions in RTX3060 (Figure 3(b)) and RTX3090 (Figure 3(c)) are different from that in RTX2060. Benefiting from more CUDA cores (3584 in RTX3060, 10496 in RTX3090), they have more intensive memory requests. With the original GDDR6 memory, the memory bandwidth can be overwhelmed. But if we adopt Samsung's HBM-PIM architecture, to build the same capacity (12GB in RTX3060, 24GB in RTX3090) with its original configuration, we can provide more memory bandwidth in total (307GB/s per 6GB-chip). In that case, we still have available memory bandwidth to issue PIM control commands.

3 Co-Mining Architecture

In this section, we will introduce the overall architecture of the Co-Mining framework. As shown in Figure 4, the right-bottom part shows a typical memory channel from the HBM-PIM architecture. Our techniques mainly concentrate on the software layer and left-bottom GPU side, including the GPU/PIM thread scheduler for the streaming processors (SPs) and the extended finite-state-machine model in the GPU memory controller. Since the real-time requirement of GPU/PIM thread scheduling is not high, to reduce the performance impact of the scheduling task decision, we offload the linear programming solving to the CPU side and communicate with GPU through the PCIe bus.

Specifically, in our Co-Mining architecture, when we start the Ethereum mining process (*main thread* on the CPU side), a lot of *Ethash_search* threads (i.e. *hash calculation threads*) will be created in GPU with each trying one nonce until find the satisfied one. At that time, all PCU cores are halted and the memory banks are working in memory mode. After a while, based on the collected bandwidth utilization of each memory channel and the current hashrate throughput, we will solve a linear programming problem on the CPU side (linear programming thread) to determine whether we should add more PCU control threads (See details in Section 4.1) that executing on GPU. Because the decision is based on the status collected periodically, this is not to be done in real-time. Thus, we can offload the problem-solving process to the CPU side (the gray part in Figure 4) to save the computation resources on GPU.

The decision results will guide the creation of PCU control threads. Then, the PCU control thread will activate one PCU for mining assistance, which requires the corresponding memory banks to switch to compute mode. To support PIM-related memory commands, we extend the original finite-state-machine model by adding several extra states to perform PIM operations as well as to maintain PCU cores execution information. The extended finite-state-machine model in the GPU memory controller (the middle-bottom of Figure 4) will control mode switch (i.e., state transfer) using a probability-based algorithm according to the history memory accesses (See details in Section 4.2). After the PCU core

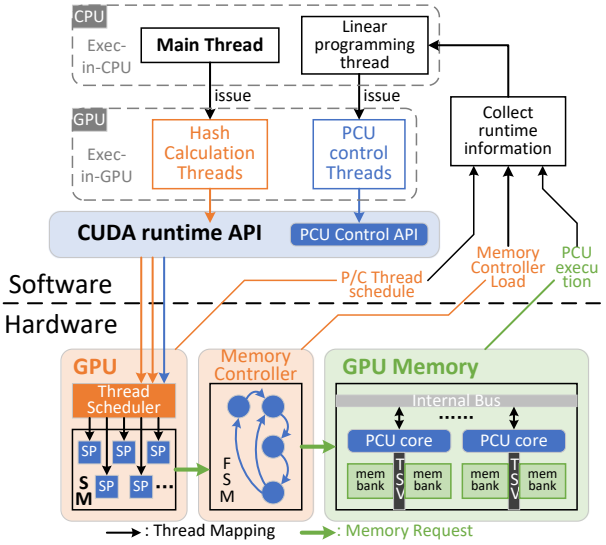


Figure 4. The overall architecture of Co-Mining.

finishes one step of the mixing stage and generates the next required memory address, we propose two different control flows for intra- and inter-channel memory fetching based on the address range (See details in Section 4.3). When one PCU control thread finishes the total 64 steps of the mixing stage, one nonce try is completed and the thread is terminated.

4 Co-Mining Techniques

4.1 GPU/PIM task scheduling

Since PCU cores must be activated by memory commands from GPU cores, in our design, we indicate several PCU control threads in GPU cores to read intermediate results and dispatch tasks to PCU cores. Hence, we separate the GPU threads into two types: the hash calculation thread (C thread) and the PCU control thread (P thread). The workflow of the PCU control thread is shown in Algorithm 1.

The first line of Algorithm 1 corresponds to the preprocessing stage of Ethash. Then we calculate the memory address and activate related PCU core (line 2-3). The calculations in PCU will be invoked 64 times which corresponds to the 64 mixing steps of Ethash (line 4-15). Note that we will reschedule those 64 calculations based on the memory address range and the memory access type (See details in Section 4.3). Finally, we will retrieve the result from the last scheduled PCU core (line 16-18).

To determine whether the overall performance can benefit from adding more P threads, we model it into a linear programming problem with the memory channel utilization as constraints.

According to the working process of Ethash [4], it uses 8 threads to compute 128-byte data in parallel (16-byte per thread). Based on our preliminary investigation, for each C thread, about 240 cycles will be consumed before generating each 16-byte memory access [13]. So we define the ***memory access density*** of each C thread as ***16 Bytes/240 Cycles***. For the P thread, since it only has a branch to check if the PCU involves intra-channel or inter-channel memory requests (See section 4.3 for details), it only executes about 12 cycles and then generates 256-byte memory access (read 128B intermediate result and write 128B to another PCU core). Thus, its ***memory access density*** is ***256 Bytes/12 Cycles***.

Algorithm 1 The workflow of the PCU control thread.

Input: Previous block header: $BlkHeader$
New nonce to try: $Nonce$
Output: Calculated hash result

1: Calculate $Seed \leftarrow SHA3\_512(BlkHeader + Nonce)$
2: Retrieve first memory address $Addr_i$ from $Seed$
3: Activate PCU core at address $Addr_i$
4: **while** repeat times < 64 **do**
5: Wait until PCU calculation finished
6: Read the next memory request address $Addr_i$ from PCU core at address $Addr_{i-1}$
7: **if** $Addr_i$ and $Addr_{i-1}$ are in the same channel **then**
8: Perform intra-channel data movement through global bitlines
9: **else**
10: Read intermediate data from PCU: $RD(Addr_{i-1})$
11: Terminate PCU core at address $Addr_{i-1}$
12: Write intermediate data to PCU: $WR(Addr_i)$
13: Activate PCU core at address $Addr_i$
14: **end if**
15: **end while**
16: Wait until PCU calculation finished
17: Read final result from PCU: $READ(Addr_i)$
18: Terminate PCU core at address $Addr_i$

Table 1. Parameters in our linear programming model.

N	Number of memory channels
P_s	Number of times that GPU schedules P thread
C_s	Number of times that GPU schedules C thread
T	The average time that PCU core finishes one step
B_p	The size of blocked memory request from P thread
B_c	The size of blocked memory request from C thread
x	$\in \{0, 1\}$, if we need to add one more P thread
y	$\in \mathbb{N}$, if we need to reduce some C threads

Taking Table 1 with the above analysis about the thread memory access density, we can list the constraints as follows:

$$P_s + x \leq \text{the number of PCU cores} \quad (1)$$

$$12\times(P_s+x)+240\times(C_s-y) \leq \text{GPU cores}\times\text{Core freq.}\times T \quad (2)$$

$$256\times(P_s+x)+16\times(C_s-y) \leq \text{Total memory bandwidth}\times T \quad (3)$$

In the above constraints, we fix the time slot T to be the average time that one PCU core finishes one step (i.e., one

mixing function), so the number of PCU control threads is equal to the number of times they are scheduled.

For the C thread, when its 128-byte memory access has been processed, one step is finished; for the P thread, a processed 256-byte memory access can be regarded as one step. Due to the PCU execution, the size of the blocked memory request has a linear relationship with the number of P threads. Thus, the throughput (how many steps are finished) can be estimated as follows:

$$Throughput = \frac{16 \times (C_s - y) - \frac{B_c}{P_s} \times (P_s + x)}{128} + \frac{256 \times (P_s + x) - \frac{B_p}{P_s} \times (P_s + x)}{256} \quad (4)$$

We will find the value of x on the CPU side to maximize the throughput (Equation 4) with the consideration of the constraints (1), (2) and (3). If x is determined as 1, we will find the corresponding y that satisfies the constraints. Then, the decision of x and y will guide us for the creation of P threads as well as the deconstruction of C threads.

By solving this linear programming problem in each time slot T, we can strike the balance between the hash calculation thread and the PCU control thread, and find the optimal configuration towards higher throughput.

4.2 Fine-grained PIM core controller

The memory controllers inside GPU are used to translate assembly LOAD/STORE code to memory operations (e.g., ACT, PRE, RD/WR), and those memory states are maintained in a finite-state-machine (FSM) inside memory controllers. We exhibit several important states shown in Figure 5 (the left part, with blue background). With PCU cores integrated, some extra states are needed to issue PCU-related commands. However, currently, the state machine can only handle the entry and exit of PIM mode (compute mode) without any PCU core execution status information, as the middle part with orange background of Figure 5 shows.

As PCU cores cannot actively communicate with GPU, the GPU core needs to explicit poll (e.g., check if the memory address that stores the PCU task output has been written by PCU cores) to get all the PCU task status. To avoid unnecessary polling request from GPU, the PCU core execution status must be maintained on the GPU side. Hence, we extend the finite-state-machine model in the GPU memory controller to handle PCU core execution status to save unnecessary GPU requests. Specifically, the right part of Figure 5 with green background shows our fine-grained PIM core control flow, including all five states: *uninterruptable PIM compute state, interruptable PIM compute state, PIM compute finish state, PCU state save state and PCU terminate state* in our extended FSM model. Combined with the original DRAM-only FSM (blue part), we can achieve more fine-grained control of PCU cores compared with the traditional PIM-integrated FSM.

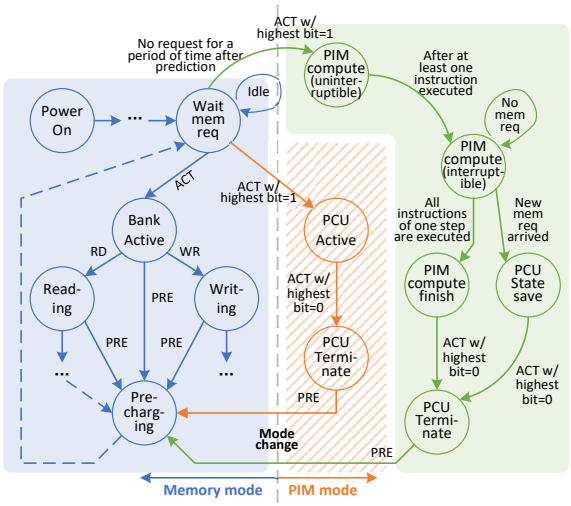


Figure 5. The extended finite-state-machine of the fine-grained PIM core controller.

The most important state transfers of PCU cores are related to mode changes. The first is the switch from the memory mode to the compute mode. The switch will only occur when the memory channel is idle. As the PCU core frequency is far less than the GPU core (300MHz vs. 1680MHz) and it can only perform 16-bit operations, executing one 32-bit instruction on PCU core may take the time equivalent to more than 10 GPU clock cycles. Hence, when PCU is executing a instruction, if a new normal memory request arrives, we can either block this request until the PCU core finishes this instruction or abort the PCU execution to serve this request immediately. The latter would incur two unnecessary mode switches: exiting compute mode immediately after entering it without doing anything.

To avoid such unnecessary mode switches, we choose to block the incoming memory request until the PCU core finishes the current instruction. To do this, we introduce two states in the FSM: the *uninterruptible-* and the *interruptible PIM compute* states. Since the PCU core is a simple RISC core, the instruction execution time can be accurately measured by the memory controller. Thus, after a fixed clock cycles, we will jump to interruptible PIM compute state from uninterruptible PIM compute state. However, those blocked memory requests will degrade the GPU core performance (because it delays GPU core's memory accessing). To minimize the number of blocked requests, we present a probability-based state transfer mechanism considering the history memory accesses in each memory channel as follows:

$$P_i = \frac{\text{avr}\left(H_j\right) - \left(H_i - \text{avr}\left(H_j\right)\right)}{\sum_{j=1}^{N} H_j} \quad (5)$$

Equation 5 is used to predict the incoming request probability in the next period, in which N is the number of memory channels, P_i indicates the incoming request probability of $channel_i$ and H_i counts the number of $channel_i$ served requests in the last period. Due to the dispersion properties of

the *FNV* hash function and its pseudo-random characteristic, we assume that it should generate nearly evenly distributed memory requests to all the channels. If H_i is higher than the average memory request of all the channels, which means memory channel i receives more requests than the average, we expect that it will receive less requests in the next period. Thus, the corresponding P_i will be smaller than $\frac{1}{N}$. Then, We can compare P_i with a threshold. If P_i is larger than the threshold, we expect it may receive another memory request in a short time, in which we will not enter compute mode. We only switch the selected channel into compute mode when the calculated P_i is smaller than the threshold.

However, the *FNV* hash function does not generate strictly evenly memory request across all the memory channels, as our preliminary experiment shows in Section 2.4. For this, we propose an online probability threshold adjustment method to achieve dynamic load balance. Specifically, we initially set the threshold to $P_{thres} = \frac{1}{N}$ which is the original incoming request probability (evenly across all memory channels). When we determine not to enter compute mode but wait at least one PCU instruction execution time, which means the threshold is set relatively low. Thus, we will increase the threshold by 1% (i.e. $P_{new\_thres} = 101\% * P_{old\_thres}$). In contrast, if we decide to enter compute mode with normal memory requests blocked, we would decrease the threshold by 1% (i.e. $P_{new\_thres} = 99\% * P_{old\_thres}$). With this dynamic threshold, we can achieve a fine-grained memory-to-compute mode switch by a simple yet effective control strategy.

Another challenge is that GPU cores are unaware of PCU task status because the in-memory PCU cores cannot actively transfer data to the GPU side. Thus, we add extra states in the memory controller to record if one PCU task has been completed or not. If the PCU task is interrupted by normal incoming memory requests, we will first save the PCU execution state then issue ACT and PRE command to change PCU-related banks back to memory mode. To get the PCU task execution status, we maintain the number of executed instructions for each PCU core in the memory controller. Note that we cannot directly get how many instructions are executed, we use clock cycles instead to measure the execution since RISC-based PCU cores have fixed execution cycles for each instruction. Once all the instructions are completed, it will enter the *PIM compute finish* state and the memory controller will notify the PIM control threads, making it explicitly change PCU-related banks back to memory mode.

4.3 Intra-channel and Inter-channel memory access

PCU cores cannot directly issue inter-channel memory accesses due to the individual timing of different memory channels. The inter-channel data transfer must involve GPU cores. Certainly, we can simply require GPU cores to retrieve data from another channel and write back to the channel where PCU cores are activated. However, it is not an optimal way because it may incur severe memory channel load imbalance.

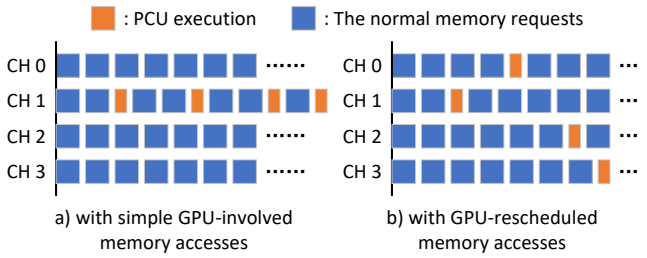


Figure 6. Two ways for inter-channel data movement.

As Figure 6(a) shows, if we make one PCU core calculate all the 64 steps of the mixing stage, the corresponding channel load will be significantly higher than the other channels due to the large number of memory requests blocked by the PCU execution.

Thus, we get GPU cores involved to re-schedule the total 64 steps and dispatch each step as an individual task to the different channels' PCU cores, as Figure 6(b) shows. Instead of retrieving the requested 128-byte memory data, we make GPU cores read the 128-byte intermediate result from the current working PCU core and dispatch the calculation of the next step to another PCU core according to the next required memory address. In this way, we can re-balance the memory channel load to achieve more efficient bandwidth utilization.

Besides, if the next required memory address locates in the same channel with the previous one, the PCU core can issue an intra-channel data transfer utilizing the global bitlines inside channel [26]. Thus, we do not need to transfer data to external GPU. According to the dispersion property of *FNV* hash functions, the probability of the next memory request address being on the same channel is only $\frac{1}{N}$, ideally. Although only a small fraction of PCU memory requests are intra-channel accesses, we still prevent them from being transferred out and in.

Specifically, we make the GPU get the calculated memory address first (line 8 in algorithm 1). If the next address involves an intra-channel request, we will not read the intermediate result out of the channel but directly inform the target PCU core to issue an intra-channel data retrieving through the global bitlines inside the channel (line 10 in algorithm 1). Otherwise, we make GPU cores read the intermediate data and re-schedule the next step to the other PCU core. Note that the internal bandwidth with global bitlines is 4X higher than the external bandwidth [19], so the intra-channel data movements will be faster. Thus, we can reduce the data access latency when two contiguous memory requests are located in the same channel.

5 Evaluation

5.1 Simulator Design

Simulator architecture. We develop a GPU-PIM simulator from the ground dedicating in the Ethash mining hashrate simulation for our experiment. Figure 7 shows the basic architecture of our simulator, mainly including three parts:

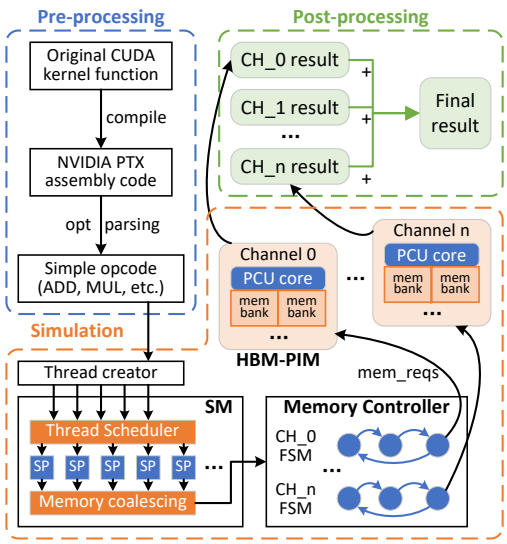


Figure 7. The trace-driven hashrate simulator architecture.

pre-processing, mining simulation and post-processing. In the pre-processing stage, we compile the CUDA kernel function and retrieve all the key operators from the assembly codes, which will be transferred to the thread creator.

For the mining simulation, because all hash calculation threads are running the same code during mining process, we can simply map all the threads to the Streaming Processors (SPs) in one Streaming Matrix (SM) for synchronous execution without handling any branches. After that, all the memory requests will be coalesced and sent to the corresponding memory controller. The memory controller will hold all the memory channel status via the FSM model and output all processed requests. Due to that the mixing stage occupies most of the execution time, the hashrate throughput is linear to the number of processed memory requests. Specifically, each 8MB requests (128B/step * 64 steps) to be processed signifies one nonce calculation accomplishment. During the post-processing stage, we sum the processed request size of all the memory channels as the final hashrate. The simulator is built with C++ language and includes <1000 lines of code, which has been released for public access [2].

Simulator validation. To validate the accuracy of our simulator, we have performed a series of tests based on the Ethash mining workloads. The results are shown in Figure 8. Except some low-end graphic cards (with less than 4GB memory) that cannot perform Ethash, from the mid-end RTX2060 to the high-end RTX3090, our simulator shows similar hashrate with only 2.5% error and similar memory bandwidth utilization with only 1.6% error on average.

Co-Mining implementation. We implement our Co-Mining technologies based on our hashrate simulator. Specifically, for PCU control threads, we simulate each of them as several memory requests that only half of them contribute to the throughput (because it has an extra 128-Byte write compared to the hash calculation thread). For the extended

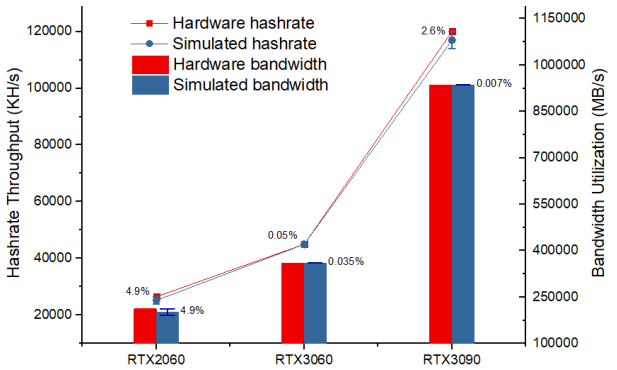


Figure 8. The trace-driven hashrate simulator validation.

FSM model, we directly implement it inside the memory controller to get the blocked request information. Those blocked requests will not contribute to the final hashrate throughput.

5.2 Experiment Setup

Table 2. The GPU configurations.

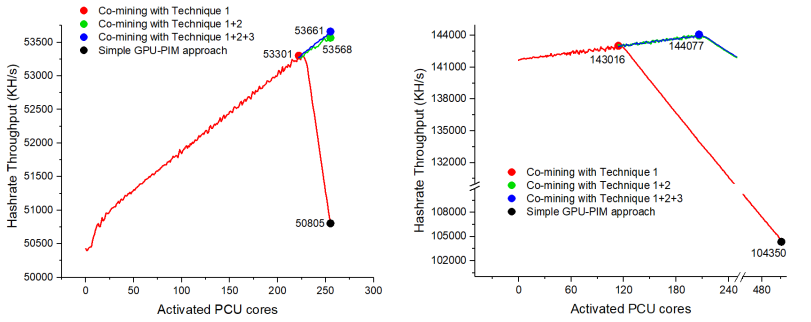
GPU	RTX2060	RTX3060	RTX3090
# of SMs	30	28	82
# of SPs per SM	64	128	128
Memory size	6GB	12GB	24GB
Core Frequency	1680MHz	1777MHz	1695MHz
with original GDDR6 (GDDR6X): [5–7]			
# of Channels	6	6	12
Bandwidth	336GB/s	360GB/s	936GB/s
MC frequency	1750MHz	1875MHz	1219MHz
with HBM-PIM (300MHz PCU, 1200MHz memory): [18]			
# of Channels	32	32	64
Bandwidth	614GB/s	614GB/s	1228GB/s
# of PCU cores	256	256	512

Table 2 shows the configuration of our simulated experimental environment based on three typical graphic cards (RTX2060/3060 for mid-end, RTX3090 for high-end) equipped with different memory systems (GDDR6/HBM-PIM).

In the evaluation, we test three different cases for each GPU: 1) the GPU-only approach simulated by our trace-driven hashrate simulator; 2) the simple GPU-PIM approach by directly integrating HBM-PIM into GPU without optimization; 3) our Co-Mining framework combining GPU and HBM-PIM with proposed techniques. Specifically, for the second configuration, we directly issue as many PIM control threads as the number of PIM cores without any optimization in either thread scheduling or the GPU memory controller. All the test cases are based on the tracefile collected from a 15-minute running of the Ethereum mining program [4].

5.3 Hashrate Throughput

Table 3 shows the hashrate throughput of the three implementations. Compared to the simple GPU-PIM approach that directly integrating HBM-PIM into GPU without any optimization, our Co-mining framework achieves up to 38.5%



(a) RTX3060 (b) RTX3090

Figure 9. The breakdown analysis of Co-mining techniques, in which Tech 1: GPU/PIM task scheduling; Tech 2: Fine-grained PIM core controller; Tech 3: Intra- and Inter-channel data access optimization.

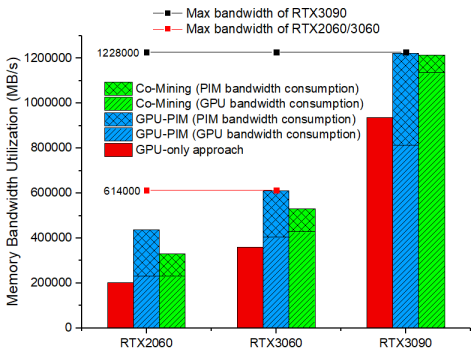


Figure 10. The memory bandwidth utilization comparison and breakdown analysis.

hashrate improvement (144MH/s vs. 104MH/s) on RTX3090. Note that the simple GPU-PIM approach has performance degradation (-10.78%) on RTX3090 while our Co-mining achieves 23.19% hashrate improvement compared to the GPU-only approach. This is mainly because we can strike a balance between GPU and PIM tasks so as to explore the available memory bandwidth more efficiently, while the simple GPU-PIM approach directly issues as many PCU threads as the number of PCU cores, which will consume too much memory bandwidth and block hash calculation threads, degrading the overall performance.

Table 3. The overall hashrate throughput (KH/s)

GPU	RTX2060	RTX3060	RTX3090
GPU-only approach	25198	44976	116952
Simple GPU-PIM approach	27908 (+10.75%)	50805 (+12.96%)	104350 (-10.78%)
Co-Mining framework	28314 (+12.37%)	53661 (+19.31%)	144077 (+23.19%)

Breakdown Analysis. We perform lots of experiments to show how we achieve the balance between hash calculation threads and PCU control threads. Figure 9 shows the breakdown of Co-mining framework by individually integrating the first technique about GPU/PIM task scheduling, the second technique about the fine-grained PIM core controller and the third technique about the intra- and inter-channel data access optimization.

Due to that a PCU control thread will occupy more memory bandwidth than a hash calculation thread, too many PCU control threads may sacrifice the hashrate throughput with the fixed bandwidth limitation, as the red line in Figure 9 shows. The GPU/PIM task scheduling in our Co-mining framework helps to find the sweet-spot number (red points in Figure 9) of the activated PCU cores, which explains why we outperform the simple GPU-PIM approach (black points in Figure 9).

The extended FSM in the fine-grained PIM core controller also contributes to the hashrate throughput, by which we can reduce unnecessary PIM mode switches while trying not to block hash calculation threads. Thus, it can reduce the memory bandwidth waste, allowing more PCU control thread without sacrificing the performance. With the extended FSM, we can achieve higher hashrate with more PCU cores activated (green lines in Figure 9).

The intra- and inter-channel data access optimization also contributes to the hashrate improvement For RTX3090, since its channels are doubled from that on RTX3060, the theoretical fraction of intra-channel accesses on RTX3090 is only a half of that on RTX3060, which limits our optimization space. Thus, the hashrate benefit would be less than that on RTX3060 (blue lines in Figure 9). With all the optimizations, our GPU/PIM task scheduling can still find the sweet-spot (blue point in Figure 9(b)), achieving the highest hashrate.

5.4 Memory Bandwidth Utilization

The reason for different hashrate improvement on different graphic cards is mainly due to the memory bandwidth consumption. We analyze the bandwidth utilization for these three approaches and exhibit the bandwidth consumed by GPU cores or PIM cores, respectively. The results are shown in Figure 10. For RTX2060, due to its limited CUDA cores, it has a large amount of remained memory bandwidth, which is enough to support the simple GPU-PIM approach for directly issuing maximum PCU control threads. In that case, our Co-mining only outperforms a little (12.37% vs. 10.75%). However, with RTX3060, the memory bandwidth waste of the simple GPU-PIM approach directly severely limits its hashrate. Thanks to the fine-grained PIM core controller and memory access optimization, Co-mining can achieve a higher hashrate with less bandwidth consumption. For RTX3090, the simple GPU-PIM approach even has negative benefit due to the overwhelmed bandwidth utilization while we can still efficiently utilize limited available bandwidth for higher hashrate throughput.

Breakdown Analysis. Figure 10 also shows the memory bandwidth consumption breakdown. For RTX2060, although Co-mining only slightly outperforms the simple GPU-PIM approach, we save half of the bandwidth consumption from

Table 4. The breakdown of hashrate throughput (KH/s)

	RTX2060		RTX3060		RTX3090	
	GPU	PIM	GPU	PIM	GPU	PIM
Simple GPU-PIM approach	25188	2720	47906	2899	98546	5804
Co-Mining	25594	2720	50735	2926	141909	2168

PIM execution, as the bar with grid shows. As table 4 shows, this bandwidth saving directly helps us gain more hashrate benefit from GPU cores on RTX3060 (50735 KH/s vs. 47906 KH/s) while the simple GPU-PIM approach only has limited hashrate improvement due to its bandwidth waste. For RTX3090, since we reduce the number of active PCU cores to save bandwidth for GPU cores, we can efficiently exploit the limited available bandwidth while not blocking hash calculation threads. Thus, we achieve significant higher throughput on GPU cores (141909 KH/s vs. 98546 KH/s) with little tradeoffs on PIM cores.

5.5 Overhead Analysis

Since we offload the linear programming to the CPU side, our system will have higher CPU usage during the PoW process. However, since the linear programming task is not real-time, we only assign one CPU core for calculation. With today's more powerful CPUs, this overhead can be ignored.

For the extended FSM, we add 3 states into its original FSM with 15+ states [30], which only occupies slightly more memory spaces but provides more fine-grained PIM control.

The HBM-PIM from Samsung consumes 5.6% more energy compared to conventional HBM2 memory chips [20]. Comparing to the major energy consumption of GPU, the slightly increased memory energy can be neglected.

6 Related Work

Proof-of-Work Acceleration. There are several works focusing on evaluating and accelerating PoW algorithms. Feng et al. evaluate memory-intensive PoW algorithms on three kinds of processors: CPU, GPU, and the Intel Knights Landing (KNL) processors and find that different processors are suitable for different PoWs [11]. Han et al. perform a detailed characterization of three most popular memory-hard PoW algorithms and then exploit data prefetching and software pipelining for acceleration [14]. Wu et al. propose a roofline-based model for the analysis of Ethash algorithm and then introduces a context switch policy to hide memory access latency and embedded NOR flash for larger bandwidth [35]. Hazari et al. propose a method for accelerating the PoW process based on parallel mining to increase transaction speed and scalability [15]. The above works indicate that PoW is the main bottleneck of blockchain system and the accelerators are in urgent necessary.

PIM-based technologies. Recently, PIM technique has shown great potential for many memory-intensive tasks. Song et al. present a novel accelerator named GraphR for graph processing based on ReRAM [29] and introduce efficient pipeline to exploit intra- and inter-layer parallelism for both training and testing in CNNs [28]. [31, 36] apply PIM to accelerate data-intensive operations in graph mining tasks and the loop code blocks of graph applications by identifying the code blocks that are best suitable for PIM execution. Deep learning applications also has been exploited with PIM accelerations. Chi et al. presents PRIME for neural network applications, in which they utilize the crossbar array of ReRAM to efficiently accelerate matrix-vector multiplications [9]. Chen et al. present ZARA, which leverages both the spatial and temporal parallelism for GAN training acceleration [8].

There are also several works for PIM-based PoW acceleration. Choe et al. utilize near-data-processing for memory-intensive algorithm Scrypt [10]. [32, 33] propose ReRAM-based accelerator for PoW algorithms in Bitcoin and IOTA, respectively. Those works explicitly show the potential of PIM architecture for memory-intensive workloads.

GPU-PIM co-optimization. There are some existing works that propose to integrate PIM-enable memories with GPU to accelerate certain computation-intensive or memory intensive applications. Pattnaik et al. introduce two new program scheduling techniques for GPU-PIM architectures, including a mechanism to decide what kernel code segments to offload to PIM cores, and a method that decides which kernels to schedule concurrently on both GPU cores and PIM cores [23]. Hsieh et al. also focus on transparently offloading computation to PIM cores with maximum potential memory bandwidth savings [16]. Kim et al. targets on a standard PIM-enabled MMU design, in which they propose a partitioned execution mechanism to decouples address translation from DRAM accesses and a cache locality-aware offload decision mechanism to prevent performance degradation for workloads with good cache locality [17].

However, existing works either only focus on data transfer/kernel offloading or do not consider some practical issues like the PIM mode switch. For memory-intensive PoWs with large dataset, simply offloading computation or optimizing data transfer only has few benefits. Thus, we propose the Co-Mining framework based on a practical HBM-PIM platform from Samsung, which dedicates on the Ethash algorithm for higher PoW efficiency.

7 Conclusion

This paper for the first time proposes a GPU/PIM co-designed framework named Co-Mining for PoW acceleration. We present a linear programming model to guide the GPU/PIM task scheduling and extend the finite-state-machine model in the GPU memory controller to gain fine-grained control of PIM mode switches. The evaluation results show that our Co-Mining can effectively utilize the available memory bandwidth for significant PoW acceleration by jointly exploiting GPU, PIM and HBM.

References

[1] 2022. BITMAIN Technologies. https://www.bitmain.com/.

[2] 2022. Co-Mining implementation. https://github.com/Anonymous007715/Hashrate-Simulator.

[3] 2022. Ethereum DAG size. https://investoon.com/tools/dag_size.

[4] 2022. nsfminer: an Ethash GPU mining implementation. https://github.com/no-fee-ethereum-mining/nsfminer.

[5] 2022. RTX2060 Specification. https://www.techpowerup.com/gpu-specs/geforce-rtx-2060.c3310.

[6] 2022. RTX3060 Specification. https://www.techpowerup.com/gpu-specs/geforce-rtx-3060.c3682.

[7] 2022. RTX3090 Specification. https://www.techpowerup.com/gpu-specs/geforce-rtx-3090.c3622.

[8] Fan Chen, Linghao Song, Hai Helen Li, and Yiran Chen. 2019. Zara: A novel zero-free dataflow accelerator for generative adversarial networks in 3d reram. In *Proceedings of the 56th Annual Design Automation Conference 2019*. 1–6.

[9] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. 2016. Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory. *ACM SIGARCH Computer Architecture News* 44, 3 (2016), 27–39.

[10] Jiwon Choe, Tali Moreshet, R Iris Bahar, and Maurice Herlihy. 2019. Attacking memory-hard scrypt with near-data-processing. In *Proceedings of the International Symposium on Memory Systems*. 33–37.

[11] Zonghao Feng and Qiong Luo. 2020. Evaluating memory-hard proof-of-work algorithms on three processors. *Proceedings of the VLDB Endowment* 13, 6 (2020), 898–911.

[12] G Fowler. 1991. Fowler/noll/vo (fnv) hash. http://isthe.com/chongo/tech/comp/fnv. (1991).

[13] Design Guide. 2013. Cuda c programming guide. *NVIDIA, July* 29 (2013), 31.

[14] Runchao Han, Nikos Foutris, and Christos Kotselidis. 2019. Demystifying crypto-mining: Analysis and optimizations of memory-hard pow algorithms. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 22–33.

[15] Shihab Shahriar Hazari and Qusay H Mahmoud. 2019. A parallel proof of work to improve transaction speed and scalability in blockchain systems. In *2019 IEEE 9th annual computing and communication workshop and conference (CCWC)*. IEEE, 0916–0921.

[16] Kevin Hsieh, Eiman Ebrahimi, Gwangsun Kim, Niladrish Chatterjee, Mike O'Connor, Nandita Vijaykumar, Onur Mutlu, and Stephen W Keckler. 2016. Transparent offloading and mapping (TOM) enabling programmer-transparent near-data processing in GPU systems. *ACM SIGARCH Computer Architecture News* 44, 3 (2016), 204–216.

[17] Gwangsun Kim, Niladrish Chatterjee, Mike O'Connor, and Kevin Hsieh. 2017. Toward standardized near-data processing with unrestricted data placement for GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12.

[18] Young-Cheon Kwon, Suk Han Lee, Jaehoon Lee, Sang-Hyuk Kwon, Je Min Ryu, Jong-Pil Son, O Seongil, Hak-Soo Yu, Haesuk Lee, Soo Young Kim, et al. 2021. 25.4 A 20nm 6GB Function-In-Memory DRAM, Based on HBM2 with a 1.2 TFLOPS Programmable Computing Unit Using Bank-Level Parallelism, for Machine Learning Applications. In *2021 IEEE International Solid-State Circuits Conference (ISSCC)*, Vol. 64. IEEE, 350–352.

[19] Donghyuk Lee, Saugata Ghose, Gennady Pekhimenko, Samira Khan, and Onur Mutlu. 2016. Simultaneous multi-layer access: Improving 3D-stacked memory bandwidth at low cost. *ACM Transactions on Architecture and Code Optimization (TACO)* 12, 4 (2016), 1–29.

[20] Sukhan Lee, Shin-haeng Kang, Jaehoon Lee, Hyeonsu Kim, Eojin Lee, Seungwoo Seo, Hosang Yoon, Seungwon Lee, Kyounghwan Lim, Hyunsung Shin, et al. 2021. Hardware Architecture and Software Stack for PIM Based on Commercial DRAM Technology: Industrial Product. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 43–56.

[21] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. *Decentralized Business Review* (2008), 21260.

[22] Rafael Pass, Lior Seeman, and Abhi Shelat. 2017. Analysis of the blockchain protocol in asynchronous networks. In *Springer Annual International Conference on the Theory and Applications of Cryptographic Techniques*.

[23] Ashutosh Pattnaik, Xulong Tang, Adwait Jog, Onur Kayiran, Asit K Mishra, Mahmut T Kandemir, Onur Mutlu, and Chita R Das. 2016. Scheduling techniques for GPU architectures with processing-in-memory capabilities. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*. 31–44.

[24] Colin Percival. 2009. Stronger key derivation via sequential memory-hard functions.

[25] Marc Pilkington. 2016. Blockchain technology: principles and applications. In *Research handbook on digital transformations*. Edward Elgar Publishing.

[26] Vivek Seshadri, Yoongu Kim, Chris Fallin, Donghyuk Lee, Rachata Ausavarungnirun, Gennady Pekhimenko, Yixin Luo, Onur Mutlu, Phillip B Gibbons, Michael A Kozuch, et al. 2013. RowClone: Fast and energy-efficient in-DRAM bulk data copy and initialization. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. 185–197.

[27] Meng Shen, Junxian Duan, Liehuang Zhu, Jie Zhang, Xiaojiang Du, and Mohsen Guizani. 2020. Blockchain-based incentives for secure and collaborative data sharing in multiple clouds. *IEEE Journal on Selected Areas in Communications* 38, 6 (2020), 1229–1241.

[28] Linghao Song, Xuehai Qian, Hai Li, and Yiran Chen. 2017. Pipelayer: A pipelined reram-based accelerator for deep learning. In *2017 IEEE international symposium on high performance computer architecture (HPCA)*. IEEE, 541–552.

[29] Linghao Song, Youwei Zhuo, Xuehai Qian, Hai Li, and Yiran Chen. 2018. GraphR: Accelerating graph processing using ReRAM. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 531–543.

[30] JEDEC Standard. 2015. High bandwidth memory (hbm) dram. *Jesd235* (2015).

[31] Jiya Su, Linfeng He, Peng Jiang, and Rujia Wang. 2021. Exploring PIM Architecture for High-Performance Graph Pattern Mining. *IEEE Computer Architecture Letters* 20, 2 (2021), 114–117.

[32] Fang Wang, Zhaoyan Shen, Lei Han, and Zili Shao. 2019. ReRAM-based processing-in-memory architecture for blockchain platforms. In *Proceedings of the 24th Asia and South Pacific Design Automation Conference*. 615–620.

[33] Qian Wang, Tianyu Wang, Zhaoyan Shen, Zhiping Jia, Mengying Zhao, and Zili Shao. 2019. Re-tangle: A reram-based processing-in-memory architecture for transaction-based blockchain. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 1–8.

[34] Gavin Wood et al. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum yellow paper* 151, 2014 (2014), 1–32.

[35] Kun Wu, Guohao Dai, Xing Hu, Shuangchen Li, Xinfeng Xie, Yu Wang, and Yuan Xie. 2019. Memory-bound proof-of-work acceleration for blockchain applications. In *Proceedings of the 56th Annual Design Automation Conference 2019*. 1–6.

[36] Liang Yan, Mingzhe Zhang, Rujia Wang, Xiaoming Chen, Xingqi Zou, Xiaoyang Lu, Yinhe Han, and Xian-He Sun. 2021. CoPIM: a concurrency-aware PIM workload offloading architecture for graph applications. In *2021 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*. IEEE, 1–6.

[37] Guy Zyskind and Oz Nathan. 2015. Decentralizing privacy: Using blockchain to protect personal data. In *IEEE Security and Privacy Workshops*.