SIMD EXTENSIONS FOR ETHEREUM VIRTUAL MACHINE

by

Aykut Bozkurt

B.S., Computer Engineering, Boğaziçi University, 2018

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in M.S. in Computer Engineering
Boğaziçi University
2022

ACKNOWLEDGEMENTS

I would like to thank my supervisor Can Özturan for his guidance and partnership throughout my thesis. I always felt fortunate during the thesis thanks to all the professors of Computer Engineering in Bogazici University.

I dedicate this thesis to my family for the comfort, and support which they have offered me all the time in my life.

ABSTRACT

SIMD EXTENSIONS FOR ETHEREUM VIRTUAL MACHINE

Ethereum and its smart contracts have been growing their popularity. Therefore, there is a need for higher transaction throughput in every other day. Ethereum Virtual Machine is a Turing-complete computer which executes Ethereum bytecode encoded instructions of smart contracts. Every instruction uses 256-bit wide stack items as input and output operands. They pop required inputs from the stack and push the result into it after an execution. A gas consumption cost is assigned to them relative to the complexity of the instruction as it prevents halting problem. Consumed gas multiplied by gas price is charged as transaction fee by the transaction sender, so that Denial of Service (DoS) attacks can be avoided. Current supported instruction set has some weaknesses. Firstly, transactions containing large size of vector operations require excessive amount of gas cost. Secondly, transaction throughput is limited because of no parallelism in execution. Therefore, we extend the instruction set by Single Instruction Multiple Data (SIMD) operations to benefit from data level parallelism. We show how EVM can benefit from the SIMD instructions by lowering gas consumption and increasing transaction throughput.

ÖZET

ETHEREUM SANAL MAKINESI İÇİN SIMD KOMUTLARI

Ethereum ve akıllı sözleşmelerin popülerliği artmaktadır. Bu nedenle, her geçen gün daha fazla işlem hacmine ihtiyaç duyulmaktadır. Ethereum Sanal Makinesi, akıllı sözleşmelerin Ethereum-baytkoduyla kodlanmış talimatlarını yürüten bir Turing tam bilgisayarıdır. Her talimat, giriş ve çıkış işlenenleri olarak 256 bit genişliğinde yığın öğeleri kullanır. Yığından gerekli girdileri çıkarırlar ve bir yürütmeden sonra sonucu yığına geri koyarlar. Durdurma sorununu engellediği için talimatın karmaşıklığına göre onlara bir gaz tüketim maliyeti atanır. Tüketilen gazın, gaz fiyatıyla çarpımı, işlemi gönderen tarafından işlem ücreti olarak harcanır ve bu şekilde Hizmet Reddi (DoS) saldırıları önlenebilir. Mevcut desteklenen komut kümesinin bazı zayıf yönleri vardır. Bunlardan birincisi, büyük boyutlu vektör işlemleri içeren işlemlerde aşırı miktarda gaz maliyeti gerektirmesidir. İkincisi, yürütmede paralellik olmaması nedeniyle saniye başına yapılabilen işlem sayısı sınırlıdır. Bu nedenle, veri seviyesi paralelliğinden yararlanmak için Tek Komutlu Çoklu Veri (SIMD) işlemleriyle komut kümesini genişletiyoruz. Gaz tüketimini azaltarak ve işlem hacmini artırarak EVM'nin SIMD komutlarından nasıl yararlanabileceğini gösteriyoruz.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS

μ	EVM World State
σ	EVM Machine State

LIST OF ACRONYMS/ABBREVIATIONS

CFG	Context Free Grammar
DeFi	Decentralized Finance
DoS	Denial of Service
ECDSA	Ellyptic Curve Digital Signature Algorithm
EIP	Ethereum Improvement Proposal
EVM	Ethereum Virtual Machine
LC	Lane count
LW	Lane width
NFT	Non Fungible Token
RSA	Rivest–Shamir–Adleman
SIMD	Single Instruction Multiple Data
SHA	Secure Hash Algorithm

1. INTRODUCTION

Blockchain is a technology to store data in blocks that are linked to each other in chains. It is used for the nodes in the distributed system to store transaction data. The difference from other databases is the immutability of the data stored in the chain. Its combination with cryptography has paved the way for the cryptocurrencies and smart contracts, which are currently very popular and continue to be widely adopted. Besides the immutability and security, it offers anonymity to its users via public private key transaction signatures. Cryptocurrency chains are often called as distributed ledgers due to their properties.

Bitcoin is the first ever distributed ledger that is used widely by people around the world. It allows only coin transfers to be stored in its ledger. That kind of distributed ledgers are called Blockchain 1.0. On the other hand, Ethereum not only stores coin transfer transactions, but also records immutable code and its data storage that can be changed via transactions. This addition, which boosted Blockchain 1.0 to Blockchain 2.0, means a lot for humanity, because many real life business logic can make sense in cryptocurrency world. For example, a decentralized voting system like [1] would be possible in a smart contract. New terms like NFT (non fungible token) and DeFi (decentralized finance) appeared as smart contracts [2]. In addition to external personal accounts' addresses, Ethereum lets the smart contracts have addresses that enables them to take part in transactions. What makes smart contract execution possible is the Ethereum Virtual Machine (EVM) which is a Turing complete computer. Its input is portable Ethereum bytecode. Every operation the EVM can process, which is documented in [3], has its own bytecode representation. EVM uses the concept of the gas to solve the halting problem and DoS attacks [4]. Each instruction consumes an EVM specific amount of gas which is chosen by considering the CPU cycles they take. Sender puts an estimated unitless gas quantity for its transaction to successfully complete [5]. Transaction senders also puts a gas price in terms of ether which is driven by the demand and supply among users for the gas price [6].

Senders have to pay the cost of total consumed gas for the transaction. Therefore, it would incentivize people to move into Ethereum's decentralized world if gas costs can be reduced. The network also needs to execute EVM instructions faster to scale efficiently considering the increasing popularity of Ethereum. Thus, efficient solutions are required for EVM's scalability problems.

1.1. Related Work

An EIP on [7], which is still in draft, was created by Greg Colvin. It suggests SIMD instruction set extension into EVM and speaks of the design and the rationale behind it. EVM stack consists of 256 bit items and that is appropriate to take advantage of SIMD to increase EVM performance and also to reduce gas consumption.

As shown by [8], the performance of vector operations can be boosted because SIMD instructions offer a means of improving the ratio of processor performance to power usage due to reduced and effective data movement. They obtain up to 13.88x performance improvement on ARM architectures and 5.54x performance improvement on Intel architectures.

Some other studies show how SIMD improved performance of algorithms. [9] shows that 7x speedup boost on SHA-512. [10] shows that 4x speedup boost on elliptic curve scalar multiplication. [11] shows that 3-4x speedup boost on BLAKE2b. [12] shows that 3x speedup boost on OpenSSL. [13] shows that 2-3x speedup boost on elliptic curve modular multiplication. [14] shows that 1.7-1.9x speedup boost on SHA-256. [15] shows that 1.3x speedup boost on RSA-encryption.

1.2. Contributions of the Thesis

In this thesis, we consider the scalability problems of EVM. Current version of EVM has two drawbacks in terms of scalability:

(i) Low throughput because of no parallelism in the execution,
(ii) High gas costs incurred by big loop executions.

To solve the scalability problems, we extended current EVM with the help of SIMD instructions.

In Chapter 2, we review the background material about Ethereum and EVM internals. Chapter 3 presents our solution in details. It shows and discusses internal design decisions about the bytecode representation and the gas cost model for SIMD instructions. Chapter 4 presents a helper bytecode generator for our solution. Chapter 5 shows some examples of bytecode execution with comparison between both versions of EVM. In chapter 6, we present our experiments and benchmark comparison between current version of EVM and SIMD extended EVM. Finally in chapter 7, we conclude the thesis with results and future possible extensions.

2. PRELIMINARIES

2.1. Ethereum

Ethereum is a digital ledger that securely stores all the transactions that can be either a coin transfer or a contract call. Transactions are propagated to some special nodes called "miner node" through the Ethereum network. Miners pick transactions and put them into what is called a "block" which is linked to previous blocks in the chain as depicted by Figure 2.1.

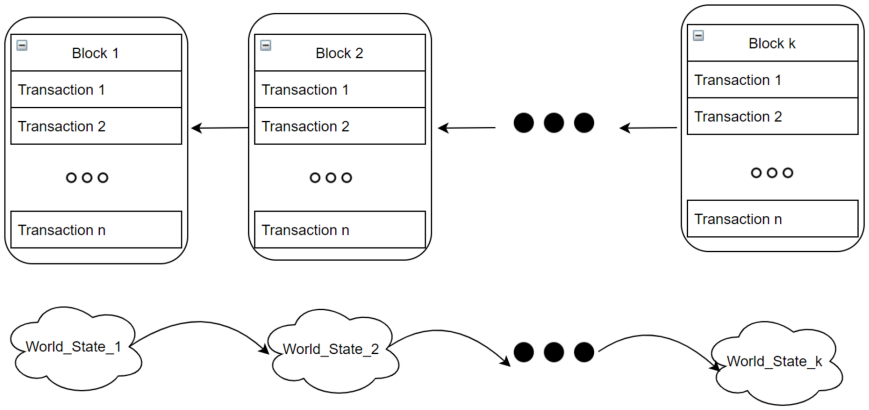

Figure 2.1. Blockchain.

2.2. Blocks

Transactions contains gas price to incentivize the miners. Each transaction consumes a certain amount of gas. Gas fee is calculated by

$$\textbf{GasFee} = \textbf{GasPrice} \cdot \textbf{ConsumedGas}. \tag{2.1}$$

Miners often prioritizes the transactions with high gas fees and include them first in the block.

After picking transactions for the block, the block finalisation step is taken place by applying each transaction into current world state. Finally, they try to solve a mathematical puzzle to prove their effort to create the block. After solving the puzzle, they dissipate the block with the solved puzzle parameters to all other nodes. Puzzle solving makes the irreversibility of the past transactions extremely difficult [16].

2.3. Consensus and Common State

Because there are many miners that are trying to solve the puzzle, many blocks will race with each other in the network. The network should decide on the same state after a period. That is where the consensus protocol comes in. Despite of the many racing blocks, the consensus protocol will choose the longest valid chain which contains the longest proof of work chain as shown in Figure 2.2. In this way, the network agrees on a common state [17].

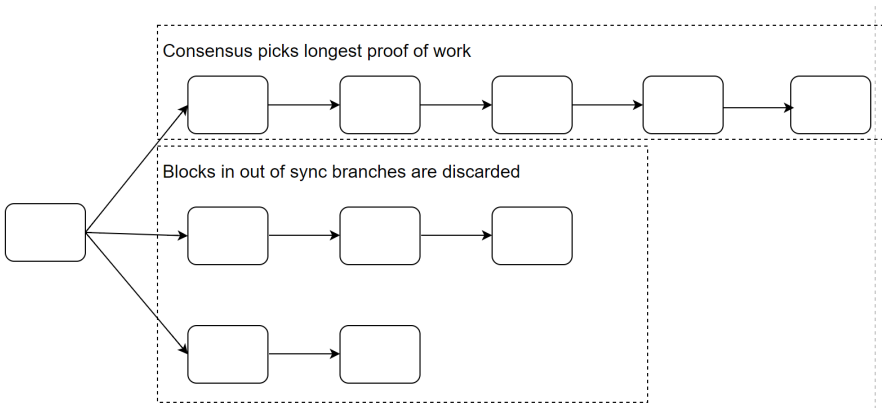

Figure 2.2. Consensus.

2.4. Block Verification

Blocks that are spread to the network are verified by the other nodes before being counted in the chain. Verification steps can be summarized by [18]:

(i) Existence of the previous block is checked.

(ii) The block's timestamp should not be longer than 15 min after the previous block's timestamp,

(iii) Validate the block number, difficulty, transaction root, uncle root and gas limit are valid,

(iv) Validate the proof of work on the block,

(v) Apply transactions in the block one by one and update world state accordingly. Abort transaction in case of an error,

(vi) Add block reward to the miner's account.

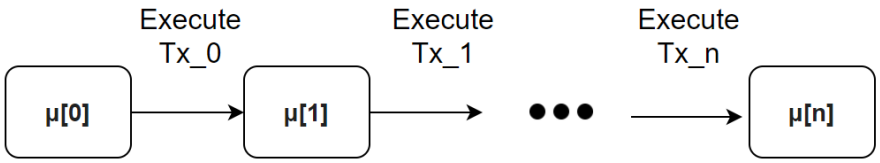

Figure 2.3. Finalisation.

2.5. Accounts

An account can be either an externally owned account(EOA) or a contract account as shown in Figure 2.4. Account state consists of nonce and balance for both types of accounts. Nonce helps to prevent double spending of the same transaction. Nonce is incremented after every successful transaction and if a transaction has lower nonce than current nonce, it is rejected. Storage hash and code hash only exists for contract accounts. They respectively maps to data storage and code for the contract.

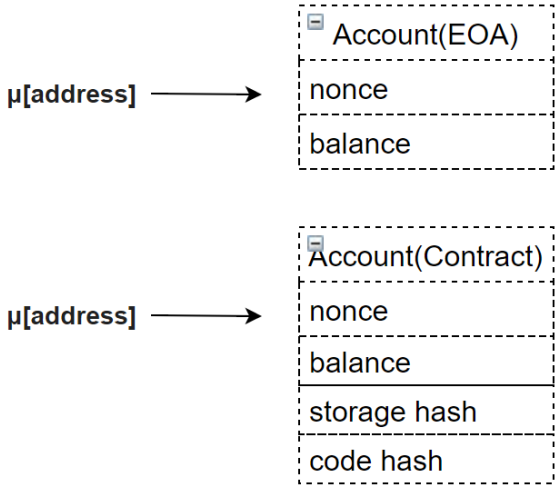

Figure 2.4. Externally owned and contract accounts' properties.

2.6. Transactions

There are 2 types of transactions which are, namely, contract creation and message call. When an EOA sends a contract creation transaction, a contract address is created by using transaction sender address and nonce. Contract code is put into transaction's init parameter by transaction sender. Message transactions can be between any types of accounts. In case of a contract call, input parameters is passed to the transaction's data parameter. Transaction contains "to" parameter to specify recipient address which is 0 in case of contract creation. "Value" parameter specify the amount of the transfer if any. It also contains "gas price" and "gas limit" parameters [19]. Gas price assigns a value per gas unit. Gas limit specifies the maximum amount of gas unit that transaction sender would accept. If gas limit is exceeded by EVM, transaction is aborted [20]. All the parameters are specified by a transaction structure as shown by Figure 2.5.

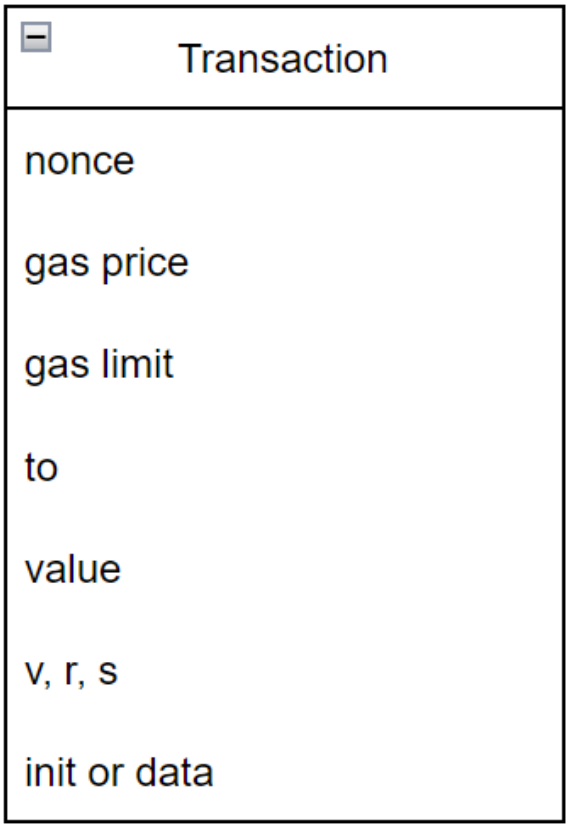

Figure 2.5. Transaction properties.

2.7. Ethereum Virtual Machine

EVM is a Turing compete machine which updates its world state upon every transaction execution. World state stores all account related information belonging to all account address.

It is an run time environment for smart contracts. It mainly consists of immutable virtual ROM that contains EVM code, machine state and world state. Machine state contains virtual registers which store "available gas" and "program counter" as shown by Figure 2.6. Machine state also contains a stack with 1024 items and memory. On the other hand, it has a non-volatile account storage area, which is called world state, stores mapping between account addresses and account information.

Each stack item has a width of 256 bits. All operations are performed on the stack. That means all inputs and outputs of an EVM instruction is pushed and popped onto the stack. After every transaction EVM updates its world state.

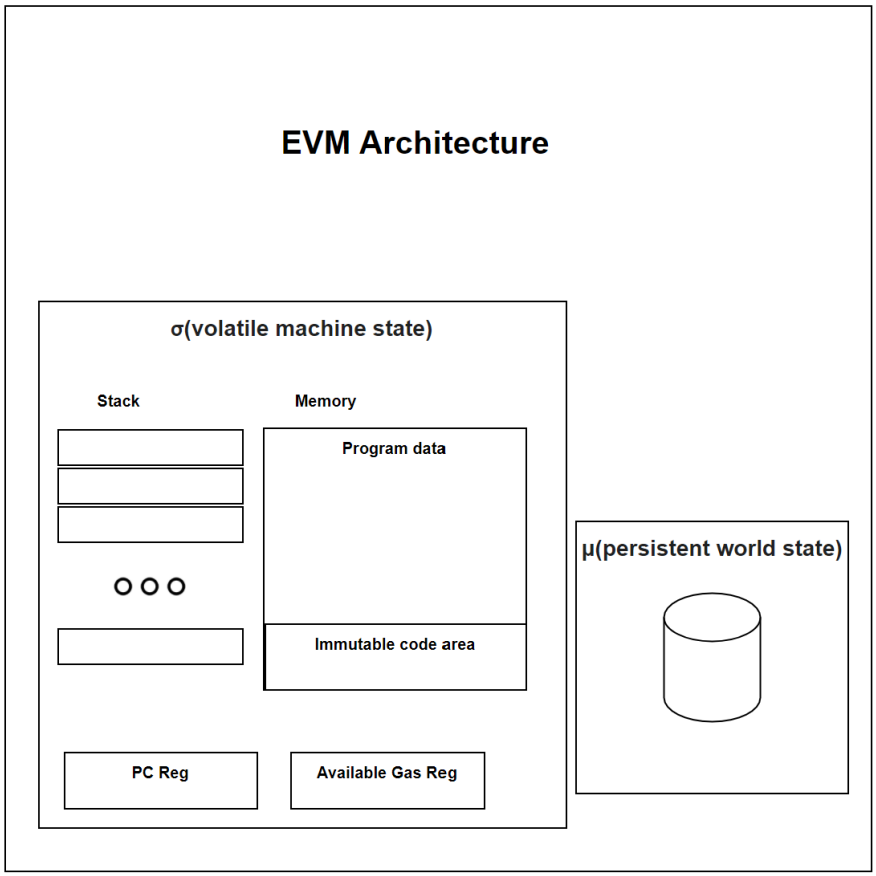

Figure 2.6. EVM Internal State.

2.8. SIMD

According to Flynn's Taxonomy, SIMD means that the same instruction is executed on multiple parts of a large register as shown in Figure 2.7. Every scalar operation needs to operate in CPU registers. After the operation finalizes, its result is written back to a memory location from register. Even though registers can accommodate multiple data and execute the same operation for them, they have to execute the same operation for multiple data in sequence if there is no support for SIMD.

Intel first introduced MMX instruction set which support SIMD operations. Registers are named MM0-7 and had 64 bits width. They could operate up to 8 integers of 8 bits. Then, they extended MMX with SSE instruction set. MMX was only capable of operating on integers. 128 bit width SSE registers (XMM0-7) can also operate on single precision floating point numbers. Then SSE2 further extended SSE to be able to operate on double floating point numbers. After those extensions, AVX and AVX2 registers appeared with its 256 bit width registers (YMM0-7).

SIMD vector operations takes fewer CPU cycles than sequential calls to a scalar instructions. EVM uses its 256 bit wide stack for instruction operands. Therefore, SIMD instructions can be perfect fit for EVM in order make it to consume less CPU cycles and, hence, less gas. That is why data parallelism via SIMD registers is important for EVM to improve performance and reduce the cost of data parallelizable algorithms, e.g. by elementwise vector operations.

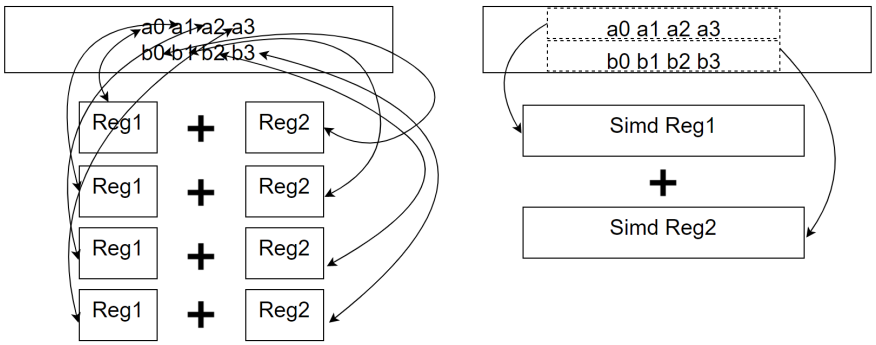

Figure 2.7. a) Scalar vs b) SIMD operation.

3. ETHEREUM VIRTUAL MACHINE SIMD EXTENSION

We chose Aleth, which is the C++ implementation of EVM, to perform the SIMD extension to EVM operations. After detailed benchmarking [21] between compiler intrinsics, and libsimdpp (a user friendly and performant SIMD library) versus scalar operations, we chose libsimdpp because of both competitive performance to the compiler intrinsics and easier integration than the compiler intrinsics. We integrated libsimdpp into Aleth in order to extend EVM operation set. libsimdpp uses the most advanced SIMD instruction set available on host CPU to get maximum performance. In case of no SIMD support for any operation, then it fallbacks to scalar operation mode.

3.1. SIMD Instruction Representation

We base our approach on [7] in order to implement our solution. As shown by Table 3.1, each SIMD instruction is encoded into 2 bytes in bytecode. First byte represents SIMD opcode. First bit of the second byte is utilized to differentiate integer and floating arithmetic. While bits 3-4 of second byte contains information about operand width, bits 6-8 tells us operand count for the operation.

Table 3.1. SIMD Opcode Specification [7].

N bits	**Field**
8	Opcode
1	scalar type 0:=unsigned integer, 1:=floating
1	reserved
2	Lane width (00:=1byte, 01:=2bytes, 10:=4bytes, 11:=8bytes)
1	reserved
3	Lane count (000:=2, 001:4, 010:=8, 011:=16, 100:=32)

3.2. SIMD Instructions

Supported SIMD instructions is shown on Table 3.2. Args represents the count of stack items popped whereas Returns shows the count of the stack items pushed for the related operation.

Table 3.2. EVM SIMD Operations [7].

Opcode	Operation	Args	Returns	Gas
0xC1	XADD	2	1	3
0xC2	XMUL	2	1	5
0xC3	XSUB	2	1	3
0xC4	XDIV	2	1	5
0xC6	XMOD	2	1	3
0xD0	XLT	2	1	3
0xD1	XGT	2	1	3
0xD4	XEQ	2	1	3
0xD5	XISZERO	1	1	3
0xD6	XAND	2	1	3
0xD7	XOOR	2	1	3
0xD8	XXOR	2	1	3
0xD9	XNOT	1	1	3
0xDB	XSHL	2	1	3
0xDC	XSHR	2	1	3
0xE0	XPUSH	0	1	3

3.2.1. Operation Descriptions

3.2.1.1. XADD. Pops 2 vectors from stack, adds them, and pushes the vector result into the stack back.

3.2.1.2. XSUB. Pops 2 vectors from stack, subtracts them, and pushes the vector result into the stack back.

3.2.1.3. XMUL. Pops 2 vectors from stack, multiplies them, and pushes the vector result into the stack back.

3.2.1.4. XDIV. Pops 2 vectors from stack, divides them, and pushes the vector result into the stack back.

3.2.1.5. XMOD. Pops 2 vectors from stack, modulos them, and pushes the vector result into the stack back.

3.2.1.6. XLT. Pops 2 vectors from stack, compares them with less than relation, and pushes the boolean vector result into the stack back.

3.2.1.7. XGT. Pops 2 vectors from stack, compares them with greater than relation, and pushes the boolean vector result into the stack back.

3.2.1.8. XEQ. Pops 2 vectors from stack, compares them with equality relation, and pushes the boolean vector result into the stack back.

3.2.1.9. XISZERO. Pops 1 vector from stack, compares it with zero, and pushes the boolean vector result into the stack back.

3.2.1.10. XAND. Pops 2 vectors from stack, bitwise-ands them, and pushes the vector result into the stack back.

3.2.1.11. XOOR. Pops 2 vectors from stack, bitwise-ors them, and pushes the vector result into the stack back.

3.2.1.12. XXOR. Pops 2 vectors from stack, xors them, and pushes the vector result into the stack back.

3.2.1.13. XNOT. Pops 1 vector from stack, bitwise nots it, and pushes the vector result into the stack back.

3.2.1.14. XSHL. Pops 1 vector and 1 integer from stack, shifts left the vector by the integer, and pushes the vector result into the stack back.

3.2.1.15. XSHR. Pops 1 vector and 1 integer from stack, shifts right the vector by the integer, and pushes the vector result into the stack back.

3.2.1.16. XPUSH. Pushes a vector into the stack.

3.3. Gas Consumption

The reason gas consumption for XMUL and XDIV is chosen 5 while it is chosen 3 for all other operations is because by comparison, on most Intel and ARM SIMD units, instructions take approximately cycle counts as shown in Table 3.3, independent of register width.

Table 3.3. EVM SIMD Operation Cycles [7].

Operation	**Intel Cycle**	**ARM Cycle**
XADD	0.5	1
XMUL	2	1
XSUB	0.5	1
XDIV	10	12

3.4. EVM Interpreter Loop

EVM interprets the bytecode in its loop as described by the equations given below:

$$\mathbf{SP} = \mathbf{SPP} \tag{3.1}$$

$$\mathbf{SPP} = \mathbf{SPP} + \mathbf{Returns} - \mathbf{Args} \tag{3.2}$$

and the steps given as follows:

(i) Fetch instruction from code,
(ii) Adjust input and output stack pointers SP and SPP using Equation (3.1),
(iii) Go and execute fetched instruction and update gas,
(iv) Increment instruction counter and repeat all the steps again.

3.4.1. Scalar XOR

For XOR operation, stack is updated by

$$\mathbf{SPP}[\mathbf{0}] = \mathbf{SP}[\mathbf{0}]\mathbf{xor}\mathbf{SP}[\mathbf{1}]. \tag{3.3}$$

3.4.2. SIMD XOR

For SIMD XOR operation, stack is updated by the following algorithm:

(i) Parse first byte and obtain SIMD related information which contains data type, lane width and lane count,

(ii) Jump and execute XOR SIMD operation as shown by Figure 3.1 [22].

```
template<class SimdVec, class UnderlyingType>
void simdXor(){
    auto vec_1bytesA = reinterpret_cast<UnderlyingType*>(&m_SP[0]);
    auto vec_1bytesB = reinterpret_cast<UnderlyingType*>(&m_SP[1]);

    SimdVec xmmA = simdpp::load_u(vec_1bytesA);
    SimdVec xmmB = simdpp::load_u(vec_1bytesB);
    SimdVec xmmC = simdpp::bit_xor(xmmA, xmmB);
    simdpp::store_u(reinterpret_cast<UnderlyingType*>(m_SPP), xmmC);
}
```

Figure 3.1. EVM SIMD XOR Execution.

4. ETHEREUM VIRTUAL MACHINE SIMD BYTECODE GENERATOR

We need a SIMD opcode parser because we extend EVM instruction set and there is no compiler support for the extension yet. We used EVM SIMD parser [23], which is developed by us in Haskell, that converts human readable SIMD instruction format into EVM bytecode. This helps us generate massive lines of SIMD bytecodes to benchmark our implementation.

4.1. Grammar

Attribute grammar rules for the parser can be summarized as below:

$\langle Start \rangle$ ::= $\langle Op \rangle$

| $\langle SIMDOp \rangle$ $\langle SIMDByte \rangle$

| $\langle Push \rangle$ $\langle SIMDByte \rangle$ $\langle Vec \rangle$

if ($\langle Vec \rangle$.elems ! = $\langle SIMDByte \rangle$.elems) {

error;

}

| $\langle Xpush \rangle$ $\langle SIMDByte \rangle$ $\langle Vec \rangle$

if ($\langle Vec \rangle$.elems ! = $\langle SIMDByte \rangle$.elems) {

error;

}

$\langle SIMDByte \rangle$ ::= $\langle OpType \rangle$ $\langle LW \rangle$ $\langle LC \rangle$

if ($\langle LW \rangle$.width $*$ $\langle LC \rangle$.elems > 32) {

error;

} else {

$\langle SIMDByte \rangle$.width = $\langle LW \rangle$.width;

$\langle SIMDByte \rangle$.elems = $\langle LC \rangle$.elems;

}

⟨*OpType*⟩ ::= ⟨*I*⟩ | ⟨*F*⟩

⟨*LW*⟩ ::= ⟨*LW1*⟩
{ LW.width = LW1.width; }
| ⟨*LW2*⟩
{ LW.width = LW2.width; }
| ⟨*LW4*⟩
{ LW.width = LW4.width; }
| ⟨*LW8*⟩
{ LW.width = LW8.width; }

⟨*LC*⟩ ::= ⟨*LC2*⟩
{ LC.elems = LC2.elems; }
| ⟨*LC4*⟩
{ LC.elems = LC4.elems; }
| ⟨*LC8*⟩
{ LC.elems = LC8.elems; }
| ⟨*LC16*⟩
{ LC.elems = LC16.elems; }
| ⟨*LC32*⟩
{ LC.elems = LC32.elems; }

⟨*Op*⟩ ::= ⟨*Add*⟩ | ⟨*Mul*⟩ | ⟨*Sub*⟩ | ⟨*Div*⟩ | ⟨*Mod*⟩ | ⟨*Lt*⟩ | ⟨*Gt*⟩ | ⟨*Eq*⟩
| ⟨*Iszero*⟩ | ⟨*And*⟩ | ⟨*Or*⟩ | ⟨*Xor*⟩ | ⟨*Not*⟩ | ⟨*Shl*⟩ | ⟨*Shr*⟩ |
⟨*Pop*⟩

⟨*SIMDOp*⟩ ::= ⟨*Xadd*⟩ | ⟨*Xmul*⟩ | ⟨*Xsub*⟩ | ⟨*Xdiv*⟩ | ⟨*Xmod*⟩ | ⟨*Xlt*⟩ | ⟨*Xgt*⟩
| ⟨*Xeq*⟩ | ⟨*Xiszero*⟩ | ⟨*Xand*⟩ | ⟨*Xoor*⟩ | ⟨*Xxor*⟩ | ⟨*Xnot*⟩ |
⟨*Xshl*⟩ | ⟨*Xshr*⟩

⟨*I*⟩ ::= ‘00’

⟨*F*⟩ ::= ‘10’

⟨*LW1*⟩ ::= ‘000’
{ ⟨*LW1*⟩.width = 1 }

⟨*LW2*⟩ ::= ‘000’
{ ⟨*LW2*⟩.width = 2 }

⟨*LW4*⟩ ::= ‘100’
{ ⟨*LW4*⟩.width = 4 }

⟨*LW8*⟩ ::= ‘110’
{ ⟨*LW8*⟩.width = 8 }

⟨*LC2*⟩ ::= ‘000’
{ ⟨*LC2*⟩.elems = 2 }

⟨*LC4*⟩ ::= ‘001’
{ ⟨*LC4*⟩.elems = 4 }

⟨*LC8*⟩ ::= ‘010’
{ ⟨*LC8*⟩.elems = 8 }

⟨*LC16*⟩ ::= ‘011’
{ ⟨*LC16*⟩.elems = 16 }

⟨*LC32*⟩ ::= ‘100’
{ ⟨*LC32*⟩.elems = 32 }

⟨*Xadd*⟩ ::= ‘c1’

⟨*Xmul*⟩ ::= ‘c2’

⟨*Xsub*⟩ ::= ‘c3’

⟨*Xdiv*⟩ ::= ‘c4’

⟨*Xmod*⟩ ::= ‘c6’

⟨*Xlt*⟩ ::= ‘d1’

⟨*Xgt*⟩ ::= ‘d2’

⟨*Xeq*⟩ ::= ‘d4’

⟨*Xiszero*⟩ ::= ‘d5’

⟨*Xand*⟩ ::= ‘d6’

⟨*Xoor*⟩ ::= ‘d7’

⟨*Xxor*⟩ ::= ‘d8’

⟨*Xnot*⟩ ::= ‘d9’

⟨*Xshl*⟩ ::= ‘db’

⟨*Xshr*⟩ ::= ‘dc’

⟨*Add*⟩ ::= '01'

⟨*Mul*⟩ ::= '02'

⟨*Sub*⟩ ::= '03'

⟨*Div*⟩ ::= '04'

⟨*Mod*⟩ ::= '06'

⟨*Lt*⟩ ::= '10'

⟨*Gt*⟩ ::= '11'

⟨*Eq*⟩ ::= '14'

⟨*Iszero*⟩ ::= '15'

⟨*And*⟩ ::= '16'

⟨*Or*⟩ ::= '17'

⟨*Xor*⟩ ::= '18'

⟨*Not*⟩ ::= '19'

⟨*Shl*⟩ ::= '21'

⟨*Shr*⟩ ::= '22'

⟨*Pop*⟩ ::= '50'

⟨*Vec*⟩ ::= '[' ⟨*VecItem*⟩ ']'
{ ⟨*Vec*⟩.elems = ⟨*VecItem*⟩.elems }

⟨*VecItem*⟩ ::= ⟨*Number*⟩ ',' ⟨*VecItem*⟩
{
⟨*VecItem_l*⟩.elems =
⟨*Number*⟩.elems + ⟨*VecItem_r*⟩.elems
}
| ⟨*Number*⟩
{ ⟨*VecItem*⟩.elems = ⟨*Number*⟩.elems }

⟨*Number*⟩ ::= ⟨*Nonzero*⟩ ⟨*Number*⟩ | ⟨*Nonzero*⟩ | ⟨*Zero*⟩
{ ⟨*Number_l*⟩.elems = 1 }

⟨*Nonzero*⟩ ::= '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

⟨*Zero*⟩ ::= '0'

We take advantage of attribute grammar because we give permission only to specific pairs of LC and LW values. We add CFG with that semantic rule. Each EVM stack item spans 256bits (32 bytes) width in memory. Therefore, we only allow specific counts of elements for a specific element width. For example, when an element width 8 bytes is used in SIMD operation, maximum element count would be 4 in order not to overflow EVM stack size.

4.2. Example Bytecode Generation

We generate bytecode by parsing input files containing SIMD operations or scalar operations and generate corresponding EVM bytecodes.

Parser takes consecutively 4 command line arguments:

- rawRepeat = Bytecode generated for scalar input file is repeated rawRepeat times and appended to stdout.
- rawPath = Path of input file for scalar operations.
- SIMDRepeat = Bytecode generated for SIMD input file is repeated SIMDRepeat times and appended to stdout.
- SIMDPath = Path of input file for SIMD operations.

An input file consists of some SIMD operations to add 2 SIMD vectors of element width=8 and element count=4.
SIMD Input File:

```
Xpush (SIMDByte I LW8 LC4) [1,2,3,4]
Xpush (SIMDByte I LW8 LC4) [1,2,3,4]
Xadd (SIMDByte I LW8 LC4)
Pop.
```

Another input file consists of a scalar operations to add 2 scalars of width=8.
Scalar Input File:

```
Push LW8 15
Push LW8 15
Add
Pop.
```

After running ethparser with the proper parameters and an input file, output would be Ethereum bytecode representation for the input file.

4.3. Input Generation for Benchmark

Note that, element count=4 for 1 SIMD addition computationally equal to 4 scalar additions. The examples above computes the same amount of additions which corresponds to 4 scalar additions.

We generate computationally same amount of computation for both SIMD and scalar operations in our benchmarks. Then, we compared scalar version of the operation to SIMD version of it. For example, total of 3,200,000 1 byte scalar addition is equal to 100,000 SIMD addition operation when LC=32.

We generated every combination of bytecodes of possible lane width and lane count for all SIMD operations for both unsigned int and floating types. Anyone who want to generate SIMD bytecode can check how it works on [10].

5. EXAMPLE SIMD EXECUTIONS ON ETHEREUM VIRTUAL MACHINE

5.1. Addition of 2 Vectors of 8 Items in Scalar

In first iteration as shown by Figure 5.1, we push first items of the vectors, then pop and add them.

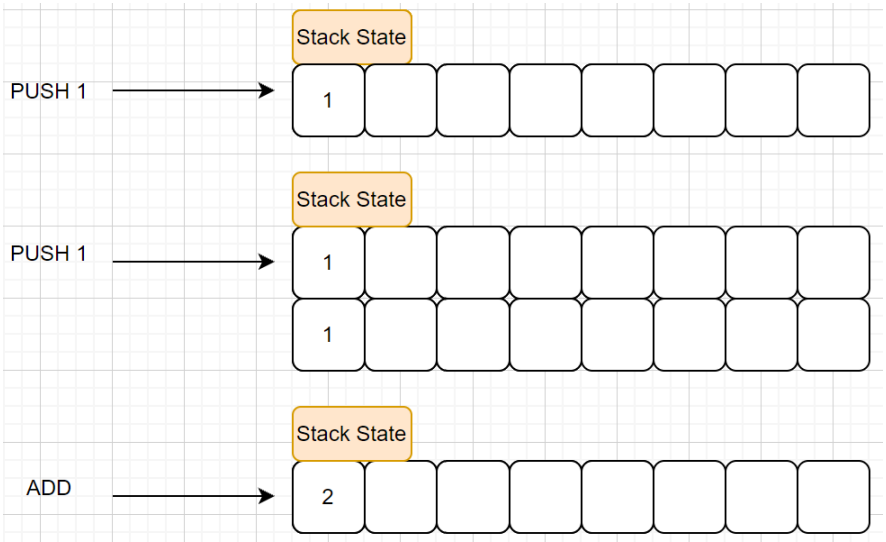

Figure 5.1. Scalar Vector Addition Step 1.

In second iteration as shown in Figure 5.2, we push second items of the vectors, then pop and add them.

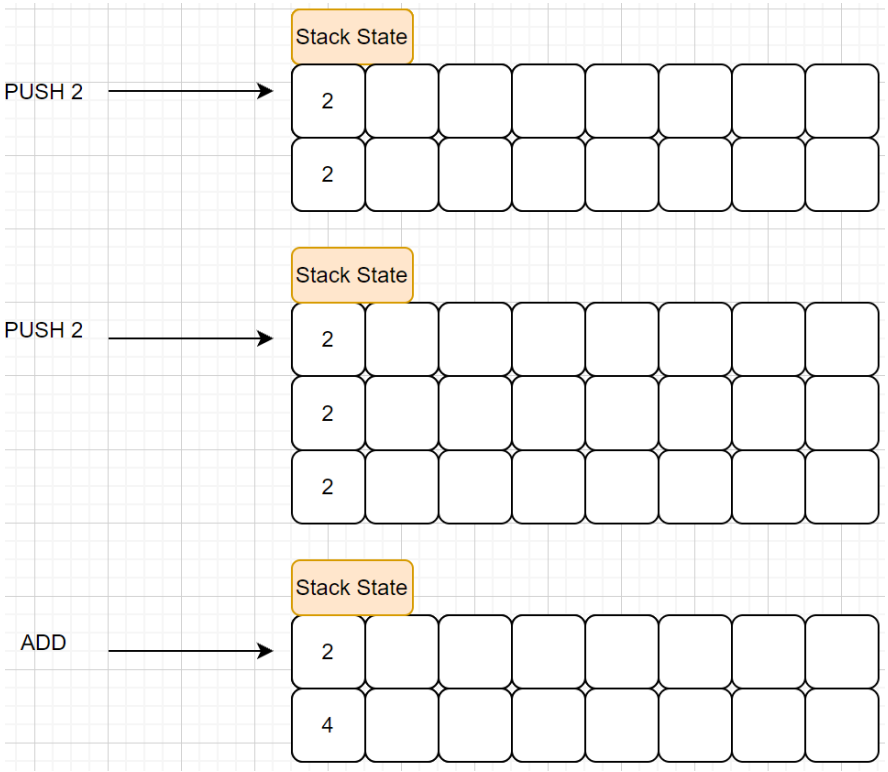

Figure 5.2. Scalar Vector Addition Step 2.

Similar operations are executed until all items are added. In the last iteration as shown in Figure 5.3, we push last items of the vectors, and then pop and add them.

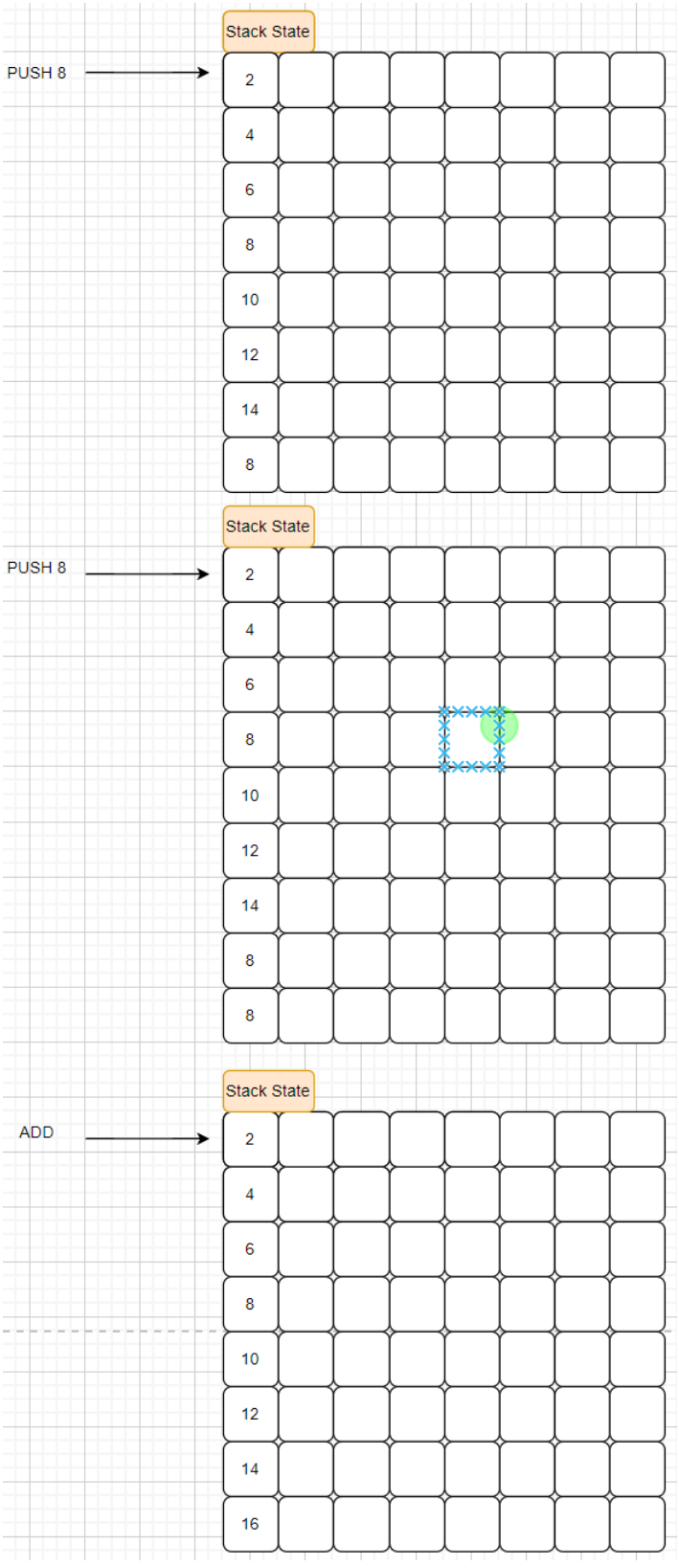

Figure 5.3. Scalar Vector Addition Step 8.

5.2. Addition of 2 Vectors of 8 Items in SIMD

As shown in Figure 5.4, we push both the vectors in one step and pop and add them using SIMD instruction in one step again.

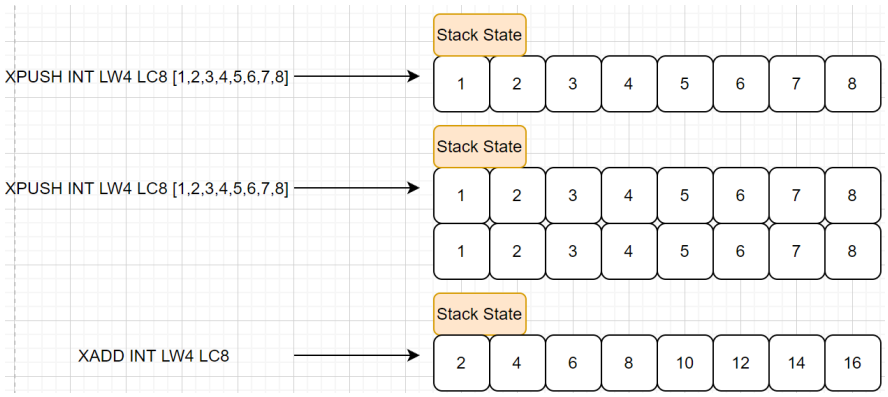

Figure 5.4. SIMD Vector Addition.

6. EXPERIMENTS AND RESULTS

6.1. Benchmark Setup

We benchmarked optimized release binary of SIMD-extended aleth on a system as shown in Table 6.1. Note that total number of operations is set to 3,200,000 scalar operation. Bytecodes for both SIMD and scalar operations, which are computationally equal to 3,200,000 scalar operations, are generated by [23]. We generated bytecodes, and evaluated benchmark results for each SIMD instruction in the next sections.

Table 6.1. Benchmark Environment

Parameter	**Specification**
Operating System	Linux/Ubuntu 20.04 LTS
Memory	16 GB
CPU	i7-10510U CPU @ 1.80 GHz (8 CPUs)
Supported SIMD Instruction Set	AVX2
Total Scalar Operation Count	3,200,000

6.2. Operation Support

Even though EVM does not natively support floating point arithmetic for scalar operations, EVM SIMD instructions support both integer and floating addition ranging from lane width (LW) of 1 byte to LW of 8 bytes. It is added because of the future possible floating point support. Currently, there is no way for nodes to agree on the same floating point values due to precision errors yet by [24]. For example, lane count (LC) of 32 and LW of 1 byte integers can be operated for the same instruction considering EVM stack item size which is 256 bits. The best possible SIMD instruction set is chosen by the libsimdpp library. libsimdpp lets us check SIMD capabilities of the host during compile time.

When there is no SIMD support for an operation in the chosen instruction set, it fallbacks to scalar operations and hence there is no gain or loss in speedup and gas consumption at the end. All supported operation results are listed in the next sections.

6.3. Addition Benchmark

As shown in Table 6.3, speedups up to 7.3x for 1 byte, 5.6x for 2 byte, 4.6x for 4 byte, 3.9x for 8 byte can be obtained for integer vector addition if all possible register space is used. Speedups up to 5.7x for single precision floating, and 3.9x for double precision floating point can be obtained.

6.4. Subtraction Benchmark

As shown in Table 6.4, speedups up to 7.3x for 1 byte, 5.6x for 2 byte, 5.7x for 4 byte, 3.9x for 8 byte can be obtained for integer vector subtraction if all possible register space is used. Speedups up to 9.2x for single precision floating, and 3.9x for double precision floating point can be obtained.

6.5. Multiplication Benchmark

As shown in Table 6.2, speedups up to 7.9x for 1 byte, 6.6x for 2 byte, 5.6x for 4 byte, 4.4x for 8 byte can be obtained for integer vector multiplication if all possible register space is used. Speedups up to 6.3x for single precision floating, and 4.5x for double precision floating point can be obtained.

6.6. Division Benchmark

As shown in Table 6.5, speedups up to 14.5x for 1 byte, 8.3x for 2 byte, 6.8x for 4 byte, 4.8x for 8 byte can be obtained for integer vector division if all possible register space is used. Speedups up to 8.0x for single precision floating, and 6.11x for double precision floating point can be obtained.

6.7. Modulo Benchmark

As shown in Table 6.6, speedups up to 3.6x for 1 byte, 2.7x for 2 byte, 3.3x for 4 byte, 2.2x for 8 byte can be obtained for integer vector modulo if all possible register space is used. There is no modulo operation for floating point.

6.8. Push Benchmark

Table 6.7 shows that speedups up to 3.1x for 1 byte, 3.7x for 2 byte, 3.2x for 4 byte, 2.0x for 8 byte can be obtained for integer vector push if all possible register space is used. There is no push operation for floating point.

6.9. Equality Benchmark

Table 6.8 shows that speedup up to 6.9x for 1 byte, 5.4x for 2 byte, 5.6x for 4 byte, 3.9x for 8 byte can be obtained for integer vector equality if all possible register space is used. Speedups up to 5.6x for single precision floating, and 3.9x for double precision floating point can be obtained.

6.10. Iszero Benchmark

Table 6.9 shows that speedup up to 8.8x for 1 byte, 5.6x for 2 byte, 5.5x for 4 byte, 3.8x for 8 byte can be obtained for integer vector zero equality if all possible register space is used. Speedup up to 5.5x for single precision floating, and 3.8x for double precision floating point can be obtained.

6.11. Greater Than Benchmark

Table 6.10 shows that speedup up to 6.6x for 1 byte, 5.3x for 2 byte, 5.5x for 4 byte, 3.8x for 8 byte can be obtained for integer vector greater than relation if all possible register space is used.

Speedup up to 5.5x for single precision floating, and 3.8x for double precision floating point can be obtained.

6.12. Less Than Benchmark

Table 6.11 shows that speedups up to 6.8x for 1 byte, 5.3x for 2 byte, 5.6x for 4 byte, 3.8x for 8 byte can be obtained for integer vector less than relation if all possible register space is used. Speedups up to 5.5x for single precision floating, and 3.8x for double precision floating point can be obtained.

6.13. Shift Left Benchmark

Table 6.12 shows that speedups up to 3.0x for 1 byte, 2.7x for 2 byte, 3.2x for 4 byte, 2.0x for 8 byte can be obtained for integer vector shift left if all possible register space is used. There is no shifting operation for floating point.

6.14. Shift Right Benchmark

Table 6.13 shows that speedup up to 2.9x for 1 byte, 2.8x for 2 byte, 3.1x for 4 byte, 2.0x for 8 byte can be obtained for integer vector shift right if all possible register space is used. There is no shifting operation for floating point.

6.15. Bitwise And Benchmark

Table 6.14 shows that speedups up to 7.6x for 1 byte, 5.7x for 2 byte, 5.8x for 4 byte, 4.0x for 8 byte can be obtained for integer vector bitwise and if all possible register space is used. Speedup up to 5.7x for single precision floating, and 4.0x for double precision floating point can be obtained.

6.16. Bitwise Or Benchmark

Table 6.15 shows that speedup up to 7.6x for 1 byte, 5.6x for 2 byte, 5.8x for 4 byte, 4.0x for 8 byte can be obtained for integer vector bitwise or if all possible register space is used. Speedup up to 5.7x for single precision floating, and 4.0x for double precision floating point can be obtained.

6.17. Bitwise Not Benchmark

Table 6.16 shows that speedups up to 9.6x for 1 byte, 6.0x for 2 byte, 5.8x for 4 byte, 3.9x for 8 byte can be obtained for integer vector not if all possible register space is used. Speedup up to 5.8x for single precision floating, and 3.9x for double precision floating point can be obtained.

6.18. Xor Benchmark

Table 6.17 shows that speedups up to 7.5x for 1 byte, 5.6x for 2 byte, 5.8x for 4 byte, 4.0x for 8 byte can be obtained for integer vector xor if all possible register space is used. Speedup up to 5.7x for single precision floating, and 4.0x for double precision floating point can be obtained.

Table 6.2. EVM SIMD Vector Multiplication benchmark.

Type	**LW**	**LC**	**Gas Used**	**Time Elapsed(sec)**
Int	1	1	41600000	0,5546
Int	1	2	20800000	0,3750
Int	1	4	10400000	0,2222
Int	1	8	5200000	0,1250
Int	1	16	2600000	0,0917
Int	1	32	1300000	0,0705
Int	2	1	41600000	0,7397
Int	2	2	20800000	0,2582
Int	2	4	10400000	0,1640
Int	2	8	5200000	0,1130
Int	2	16	2600000	0,1118
Int	4	1	41600000	1,1451
Int	4	2	20800000	0,3414
Int	4	4	10400000	0,2502
Int	4	8	5200000	0,2024
Int	8	1	41600000	1,6562
Int	8	2	20800000	0,4790
Int	8	4	10400000	0,3771
Floating	4	2	20800000	0,5293
Floating	4	4	10400000	0,2391
Floating	4	8	5200000	0,1808
Floating	8	2	20800000	0,4639
Floating	8	4	10400000	0,3677

Table 6.3. EVM SIMD Vector Addition benchmark.

Type	**LW**	**LC**	**Gas Used**	**Time Elapsed(sec)**
Int	1	1	35200000	0,3736
Int	1	2	17600000	0,2356
Int	1	4	8800000	0,1370
Int	1	8	4400000	0,0908
Int	1	16	2200000	0,0693
Int	1	32	1100000	0,0510
Int	2	1	35200000	0,7171
Int	2	2	17600000	0,2734
Int	2	4	8800000	0,1743
Int	2	8	4400000	0,1223
Int	2	16	2200000	0,1288
Int	4	1	35200000	1,1197
Int	4	2	17600000	0,3505
Int	4	4	8800000	0,2416
Int	4	8	4400000	0,1953
Int	8	1	35200000	1,5574
Int	8	2	17600000	0,4980
Int	8	4	8800000	0,3988
Floating	4	2	17600000	0,3469
Floating	4	4	8800000	0,2408
Floating	4	8	4400000	0,1954
Floating	8	2	17600000	0,4916
Floating	8	4	8800000	0,3993

Table 6.4. EVM SIMD Vector Subtraction benchmark.

Type	**LW**	**LC**	**Gas Used**	**Time Elapsed(sec)**
Int	1	1	35200000	0,3678
Int	1	2	17600000	0,2353
Int	1	4	8800000	0,1367
Int	1	8	4400000	0,0863
Int	1	16	2200000	0,0644
Int	1	32	1100000	1,5728
Int	2	1	35200000	0,7132
Int	2	2	17600000	0,2716
Int	2	4	8800000	0,1754
Int	2	8	4400000	0,1213
Int	2	16	2200000	0,1283
Int	4	1	35200000	1,1217
Int	4	2	17600000	0,3464
Int	4	4	8800000	0,2416
Int	4	8	4400000	0,1951
Int	8	1	35200000	1,5531
Int	8	2	17600000	0,4883
Int	8	4	8800000	0,3967
Floating	4	2	17600000	0,3447
Floating	4	4	8800000	0,2404
Floating	4	8	4400000	0,1224
Floating	8	2	17600000	0,4943
Floating	8	4	8800000	0,3992

Table 6.5. EVM SIMD Vector Division benchmark.

Type	**LW**	**LC**	**Gas Used**	**Time Elapsed(sec)**
Int	1	1	41600000	0,9696
Int	1	2	20800000	0,6414
Int	1	4	10400000	0,2878
Int	1	8	5200000	0,1741
Int	1	16	2600000	0,1343
Int	1	32	1300000	0,0669
Int	2	1	41600000	1,1823
Int	2	2	20800000	0,5333
Int	2	4	10400000	0,2416
Int	2	8	5200000	0,1655
Int	2	16	2600000	0,1417
Int	4	1	41600000	1,5846
Int	4	2	20800000	0,5700
Int	4	4	10400000	0,3700
Int	4	8	5200000	0,2338
Int	8	1	41600000	2,4218
Int	8	2	20800000	0,6766
Int	8	4	10400000	0,5069
Floating	4	2	20800000	0,5240
Floating	4	4	10400000	0,2437
Floating	4	8	5200000	0,1971
Floating	8	2	20800000	0,4853
Floating	8	4	10400000	0,3958

Table 6.6. EVM SIMD Vector Modulo benchmark.

Type	**LW**	**LC**	**Gas Used**	**Time Elapsed(sec)**
Int	1	1	35200000	0,1323
Int	1	2	17600000	0,4639
Int	1	4	8800000	0,2262
Int	1	8	4400000	0,1159
Int	1	16	2200000	0,0675
Int	1	32	1100000	0,0419
Int	2	1	35200000	0,2812
Int	2	2	17600000	0,4188
Int	2	4	8800000	0,2240
Int	2	8	4400000	0,1284
Int	2	16	2200000	0,1024
Int	4	1	35200000	0,5158
Int	4	2	17600000	0,4483
Int	4	4	8800000	0,2486
Int	4	8	4400000	0,1580
Int	8	1	35200000	0,4919
Int	8	2	17600000	0,4243
Int	8	4	8800000	0,2255
Floating	4	2	0	0,0000
Floating	4	4	0	0,0000
Floating	4	8	0	0,0000
Floating	8	2	0	0,0000
Floating	8	4	0	0,0000

Table 6.7. EVM SIMD Vector Push benchmark.

Type	**LW**	**LC**	**Gas Used**	**Time Elapsed(sec)**
Int	1	1	16000000	0,1138
Int	1	2	8000000	0,3948
Int	1	4	4000000	0,2131
Int	1	8	2000000	0,1092
Int	1	16	1000000	0,0620
Int	1	32	500000	0,0368
Int	2	1	16000000	0,2786
Int	2	2	8000000	0,4237
Int	2	4	4000000	0,2013
Int	2	8	2000000	0,1137
Int	2	16	1000000	0,0908
Int	4	1	16000000	0,4613
Int	4	2	8000000	0,4021
Int	4	4	4000000	0,2280
Int	4	8	2000000	0,1429
Int	8	1	16000000	0,4640
Int	8	2	8000000	0,4001
Int	8	4	4000000	0,2278
Floating	4	2	0	0,0000
Floating	4	4	0	0,0000
Floating	4	8	0	0,0000
Floating	8	2	0	0,0000
Floating	8	4	0	0,0000

Table 6.8. EVM SIMD Vector Equality benchmark.

Type	**LW**	**LC**	**Gas Used**	**Time Elapsed(sec)**
Int	1	1	35200000	0,3408
Int	1	2	17600000	0,2338
Int	1	4	8800000	0,1363
Int	1	8	4400000	0,0891
Int	1	16	2200000	0,0664
Int	1	32	1100000	0,0495
Int	2	1	35200000	0,6925
Int	2	2	17600000	0,2719
Int	2	4	8800000	0,1748
Int	2	8	4400000	0,1221
Int	2	16	2200000	0,1277
Int	4	1	35200000	1,0945
Int	4	2	17600000	0,3462
Int	4	4	8800000	0,2426
Int	4	8	4400000	0,1965
Int	8	1	35200000	1,5585
Int	8	2	17600000	0,4898
Int	8	4	8800000	0,4008
Floating	4	2	17600000	0,3463
Floating	4	4	8800000	0,2440
Floating	4	8	4400000	0,1972
Floating	8	2	17600000	0,4891
Floating	8	4	8800000	0,4007

Table 6.9. EVM SIMD Vector Iszero benchmark.

Type	**LW**	**LC**	**Gas Used**	**Time Elapsed(sec)**
Int	1	1	25600000	0,2375
Int	1	2	12800000	0,1615
Int	1	4	6400000	0,0917
Int	1	8	3200000	0,0541
Int	1	16	1600000	0,0350
Int	1	32	800000	0,0269
Int	2	1	25600000	0,4034
Int	2	2	12800000	0,1817
Int	2	4	6400000	0,1100
Int	2	8	3200000	0,0715
Int	2	16	1600000	0,0724
Int	4	1	25600000	0,6018
Int	4	2	12800000	0,2162
Int	4	4	6400000	0,1425
Int	4	8	3200000	0,1091
Int	8	1	25600000	0,8392
Int	8	2	12800000	0,2907
Int	8	4	6400000	0,2206
Floating	4	2	12800000	0,2180
Floating	4	4	6400000	0,1448
Floating	4	8	3200000	0,1098
Floating	8	2	12800000	0,2914
Floating	8	4	6400000	0,2210

Table 6.10. EVM SIMD Vector Greater Than benchmark.

Type	**LW**	**LC**	**Gas Used**	**Time Elapsed(sec)**
Int	1	1	35200000	0,3265
Int	1	2	17600000	0,2348
Int	1	4	8800000	0,1373
Int	1	8	4400000	0,0865
Int	1	16	2200000	0,0656
Int	1	32	1100000	0,0496
Int	2	1	35200000	0,6828
Int	2	2	17600000	0,2753
Int	2	4	8800000	0,1759
Int	2	8	4400000	0,1222
Int	2	16	2200000	0,1285
Int	4	1	35200000	1,0851
Int	4	2	17600000	0,3480
Int	4	4	8800000	0,2417
Int	4	8	4400000	0,1954
Int	8	1	35200000	1,5403
Int	8	2	17600000	0,4979
Int	8	4	8800000	0,4014
Floating	4	2	17600000	0,3483
Floating	4	4	8800000	0,2428
Floating	4	8	4400000	0,1979
Floating	8	2	17600000	0,4906
Floating	8	4	8800000	0,3991

Table 6.11. EVM SIMD Vector Less Than benchmark.

Type	**LW**	**LC**	**Gas Used**	**Time Elapsed(sec)**
Int	1	1	35200000	0,3333
Int	1	2	17600000	0,2374
Int	1	4	8800000	0,1350
Int	1	8	4400000	0,0865
Int	1	16	2200000	0,0642
Int	1	32	1100000	0,0492
Int	2	1	35200000	0,6819
Int	2	2	17600000	0,2778
Int	2	4	8800000	0,1756
Int	2	8	4400000	0,1214
Int	2	16	2200000	0,1284
Int	4	1	35200000	1,0873
Int	4	2	17600000	0,3501
Int	4	4	8800000	0,2420
Int	4	8	4400000	0,1956
Int	8	1	35200000	1,5431
Int	8	2	17600000	0,4933
Int	8	4	8800000	0,4002
Floating	4	2	17600000	0,3442
Floating	4	4	8800000	0,2413
Floating	4	8	4400000	0,1966
Floating	8	2	17600000	0,4917
Floating	8	4	8800000	0,4001

Table 6.12. EVM SIMD Vector Shift Left benchmark.

Type	**LW**	**LC**	**Gas Used**	**Time Elapsed(sec)**
Int	1	1	35200000	0,1186
Int	1	2	17600000	0,4047
Int	1	4	8800000	0,2067
Int	1	8	4400000	0,1088
Int	1	16	2200000	0,0611
Int	1	32	1100000	0,0398
Int	2	1	35200000	0,2702
Int	2	2	17600000	0,4082
Int	2	4	8800000	0,2189
Int	2	8	4400000	0,1218
Int	2	16	2200000	0,0999
Int	4	1	35200000	0,4988
Int	4	2	17600000	0,4476
Int	4	4	8800000	0,2418
Int	4	8	4400000	0,1563
Int	8	1	35200000	0,4919
Int	8	2	17600000	0,4266
Int	8	4	8800000	0,2452
Floating	4	2	0	0,0000
Floating	4	4	0	0,0000
Floating	4	8	0	0,0000
Floating	8	2	0	0,0000
Floating	8	4	0	0,0000

Table 6.13. EVM SIMD Vector Shift Right benchmark.

Type	**LW**	**LC**	**Gas Used**	**Time Elapsed(sec)**
Int	1	1	35200000	0,1171
Int	1	2	17600000	0,3926
Int	1	4	8800000	0,2052
Int	1	8	4400000	0,1093
Int	1	16	2200000	0,0606
Int	1	32	1100000	0,0400
Int	2	1	35200000	0,2697
Int	2	2	17600000	0,4147
Int	2	4	8800000	0,2242
Int	2	8	4400000	0,1199
Int	2	16	2200000	0,0973
Int	4	1	35200000	0,4830
Int	4	2	17600000	0,4341
Int	4	4	8800000	0,2414
Int	4	8	4400000	0,1568
Int	8	1	35200000	0,4866
Int	8	2	17600000	0,4374
Int	8	4	8800000	0,2405
Floating	4	2	0	0,0000
Floating	4	4	0	0,0000
Floating	4	8	0	0,0000
Floating	8	2	0	0,0000
Floating	8	4	0	0,0000

Table 6.14. EVM SIMD Vector Bitwise And benchmark.

Type	**LW**	**LC**	**Gas Used**	**Time Elapsed(sec)**
Int	1	1	35200000	0,3754
Int	1	2	17600000	0,2345
Int	1	4	8800000	0,1377
Int	1	8	4400000	0,0885
Int	1	16	2200000	0,0643
Int	1	32	1100000	0,0492
Int	2	1	35200000	0,7322
Int	2	2	17600000	0,2772
Int	2	4	8800000	0,1740
Int	2	8	4400000	0,1231
Int	2	16	2200000	0,1290
Int	4	1	35200000	1,1388
Int	4	2	17600000	0,3520
Int	4	4	8800000	0,2440
Int	4	8	4400000	0,1973
Int	8	1	35200000	1,6130
Int	8	2	17600000	0,4909
Int	8	4	8800000	0,3999
Floating	4	2	17600000	0,3463
Floating	4	4	8800000	0,2426
Floating	4	8	4400000	0,1988
Floating	8	2	17600000	0,4938
Floating	8	4	8800000	0,4001

Table 6.15. EVM SIMD Vector Bitwise Or benchmark.

Type	**LW**	**LC**	**Gas Used**	**Time Elapsed(sec)**
Int	1	1	35200000	0,3741
Int	1	2	17600000	0,2345
Int	1	4	8800000	0,1355
Int	1	8	4400000	0,0872
Int	1	16	2200000	0,0639
Int	1	32	1100000	0,0495
Int	2	1	35200000	0,7302
Int	2	2	17600000	0,2770
Int	2	4	8800000	0,1759
Int	2	8	4400000	0,1219
Int	2	16	2200000	0,1298
Int	4	1	35200000	1,1335
Int	4	2	17600000	0,3468
Int	4	4	8800000	0,2428
Int	4	8	4400000	0,1967
Int	8	1	35200000	1,5904
Int	8	2	17600000	0,4893
Int	8	4	8800000	0,3978
Floating	4	2	17600000	0,3485
Floating	4	4	8800000	0,2436
Floating	4	8	4400000	0,1982
Floating	8	2	17600000	0,4899
Floating	8	4	8800000	0,3977

Table 6.16. EVM SIMD Vector Bitwise Not benchmark.

Type	**LW**	**LC**	**Gas Used**	**Time Elapsed(sec)**
Int	1	1	25600000	0,2570
Int	1	2	12800000	0,1618
Int	1	4	6400000	0,0906
Int	1	8	3200000	0,0537
Int	1	16	1600000	0,0355
Int	1	32	800000	0,0269
Int	2	1	25600000	0,4337
Int	2	2	12800000	0,1799
Int	2	4	6400000	0,1106
Int	2	8	3200000	0,0724
Int	2	16	1600000	0,0725
Int	4	1	25600000	0,6374
Int	4	2	12800000	0,2147
Int	4	4	6400000	0,1422
Int	4	8	3200000	0,1089
Int	8	1	25600000	0,8551
Int	8	2	12800000	0,3068
Int	8	4	6400000	0,2202
Floating	4	2	12800000	0,2186
Floating	4	4	6400000	0,1437
Floating	4	8	3200000	0,1100
Floating	8	2	12800000	0,2884
Floating	8	4	6400000	0,2200

Table 6.17. EVM SIMD Vector Xor benchmark.

Type	**LW**	**LC**	**Gas Used**	**Time Elapsed(sec)**
Int	1	1	35200000	0,3733
Int	1	2	17600000	0,2315
Int	1	4	8800000	0,1361
Int	1	8	4400000	0,0866
Int	1	16	2200000	0,0646
Int	1	32	1100000	0,0498
Int	2	1	35200000	0,7296
Int	2	2	17600000	0,2746
Int	2	4	8800000	0,1743
Int	2	8	4400000	0,1223
Int	2	16	2200000	0,1296
Int	4	1	35200000	1,1453
Int	4	2	17600000	0,3430
Int	4	4	8800000	0,2445
Int	4	8	4400000	0,1955
Int	8	1	35200000	1,5784
Int	8	2	17600000	0,4905
Int	8	4	8800000	0,3998
Floating	4	2	17600000	0,3471
Floating	4	4	8800000	0,2542
Floating	4	8	4400000	0,2000
Floating	8	2	17600000	0,4894
Floating	8	4	8800000	0,4001

6.19. Gas Consumption

The results indicate that for all operations when the lane count is incremented by 2 times, the gas consumption is reduced by 2 times. That is because total operations per CPU cycle is doubled when lane count is doubled. Gas consumption is lowered 32x at maximum compared to scalar operation when LW=1 and LC=32 which is maximum number of elements that can be put into 32 byte register. It is very crucial for EVM to do more reducing gas consumption.

7. CONCLUSION AND DISCUSSION

EVM suffers from low transaction throughput and high gas costs. In this thesis, we address those problems. We extend EVM instruction set with 15 different types of SIMD instruction so that we can improve performance and reduce gas cost. As a result, gas consumption is reduced by 2-32x and 2-8x speedup is obtained for different type of operations on average as shown in Chapter 6.

Portability of SIMD code is an issue because different host CPUs supports different types of SIMD support. Portable SIMD integration library (libsimdpp) made a SIMD portable EVM as much as possible. It first checks the best possible instruction set in the host CPU and fallbacks to scalar operation mode if an operation is not possible with SIMD.

One of the obstacles during the thesis is to generate SIMD bytecode. We actually need an EVM compiler that fully supports SIMD operations. All the benchmark is done thanks to our own implemented mini SIMD bytecode generator. Bytecode generation for SIMD is currently complicated because of no native compiler support.

In addition to current SIMD operations, many other useful operation that can be speedup via SIMD (e.g. reduction, shuffle), can be further added to the set in future. Secondly, we currently only support data parallelism. In future, instruction level parallelism via pipelining can be taken advantage to further improve transaction throughput [25].

REFERENCES

1. Yao, J., L. Wei and T. Liu, "Blockchain-Based Voting System", *Computer System Networking and Telecommunications*, Vol. 3, 2020.

2. Wackerow, P., *Ethereum Developer Wiki*, 2013, `https://ethereum.org/en/developers/docs`, accessed in August 2021.

3. Wood, G., *Ethereum Yellowpaper*, 2013, `https://ethereum.github.io/yellowpaper/paper.pdf`, accessed in August 2021.

4. Chen, T., X. Li, Y. Wang and J. Chen, "An Adaptive Gas Cost Mechanism for Ethereum to Defend Against Under-Priced DoS Attacks", *International Conference on Information Security Practice and Experience*, pp. 1–3, 2017.

5. Li, C., S. Nie, Y. Cao and Y. Yu, "Trace-Based Dynamic Gas Estimation of Loops in Smart Contracts", *IEEE Open Journal of the Computer Society*, Vol. 1, pp. 295–316, 2020.

6. Donmez, A. and A. K. Karaivanov, "Transaction Fee Economics in the Ethereum Blockchain", *Economic Inquiry*, pp. 2–14, 2021.

7. Colvin, G., *EIP-616*, 2017, `https://github.com/ethereum/EIPs/blob/master/EIPS/eip-616.md`, accessed in August 2021.

8. Mitra, G., B. Johnston, A. P. Rendell and J. Zhou, "Use of SIMD Vector Operations to Accelerate Application Code Performance on Low-Powered ARM and Intel Platforms", *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*, pp. 1107–1116, 2013.

9. Bertoni, G., J. Daemen, S. Hoffert, M. Peeters, G. V. Assche and R. V. Keer, *Keccak Performance*, 2017, `https://keccak.team/sw_performance.html`, accessed

in August 2021.

10. Aoki, K., F. Hoshino, T. Kobayashi and H. Oguro, "Elliptic Curve Arithmetic Using SIMD", *International Conference on Information Security*, pp. 235–247, 2001.

11. Wessels, F., *Blake2b-SIMD*, 2014, `https://github.com/minio/blake2b-simd#readme`, accessed in August 2021.

12. Gopal, V., S. Gulley, I. Albrekht, D. Zimmerman and W. Feghali, "Improving OpenSSL Performance", *Intel Development Topics and Technologies*, pp. 9–15, 2015.

13. Pabbuleti, K. C., D. H. Mane, A. Desai, C. Albert and P. Schaumont, "SIMD Acceleration of Modular Arithmetic on Contemporary Embedded Platforms", *IEEE High Performance Extreme Computing Conference*, pp. 1–6, 2013.

14. Wessels, F., *sha256-SIMD*, 2014, `https://github.com/minio/sha256-simd#readme`, accessed in August 2021.

15. Seo, H., Z. Liu, J. Großsch and H. Kim, "Efficient Arithmetic on ARM-NEON and Its Application for High-Speed RSA Implementation", *IEEE High Performance Extreme Computing Conference*, Vol. 9, 2013.

16. van Oorschot, P. C., *Computer Security and the Internet, Tools and Jewels from Malware to Bitcoin*, Springer, 2020.

17. Bhatia, S. and S. S. Tyagi, *Blockchain for Business*, Wiley-Scrivener, 2021.

18. Buterin, V., *Ethereum Whitepaper*, 2013, `https://ethereum.org/en/whitepaper`, accessed in August 2021.

19. Jezek, K., "Ethereum Data Structures", *Association for Computer Machinery*, pp.

1–27, 2021.

20. Khan, M., H. Sarwar and M. Awais, "Gas Consumption Analysis of Ethereum Blockchain Transactions", *Concurrency and Computation Practice and Experience*, Vol. 34, 2021.

21. Bozkurt, A., *Libsimdpp Benchhmark*, 2021, `https://github.com/aykut-bozkurt/SIMDBenchmark`, accessed in August 2021.

22. Bozkurt, A., *EVM SIMD Extension*, 2021, `https://github.com/aykut-bozkurt/aleth`, accessed in August 2021.

23. Bozkurt, A., *EVM Simd Bytecode Generator*, 2021, `https://github.com/aykut-bozkurt/evmSimdOPParser`, accessed in August 2021.

24. Colvin, G., *EIP-616*, 2021, `https://eips.ethereum.org/EIPS/eip-616`, accessed in August 2021.

25. Ojha, G., *Feasibility Study of Pipelining in Ethereum Virtual Machine Architecture*, 2020, `https://www.researchgate.net/project/Advance-Computer-Architecture`, accessed in August 2021.

APPENDIX A: SIMD EVM SPEEDUP PLOTS

Speedup by SIMD can be seen for some of the SIMD instructions as shown by below figures.

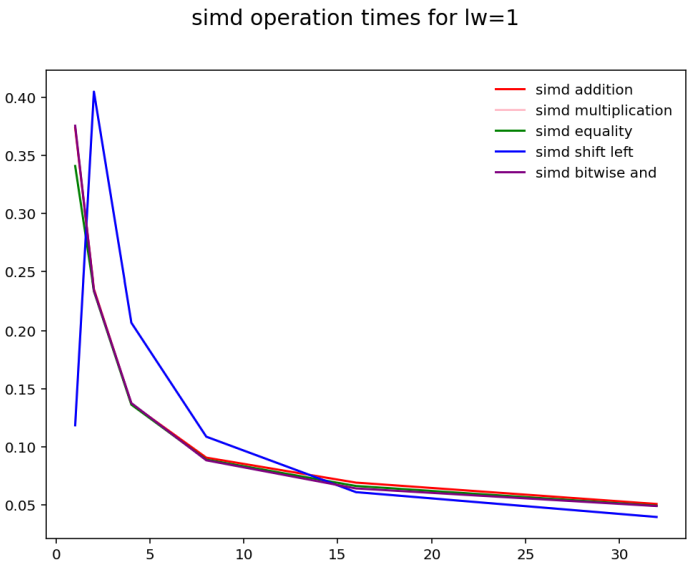

Figure A.1. LC versus time in seconds.

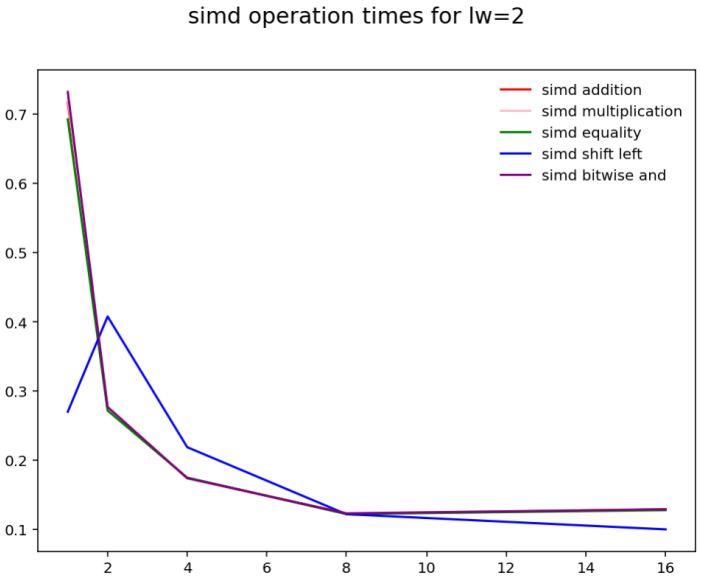

Figure A.2. LC versus time in seconds.

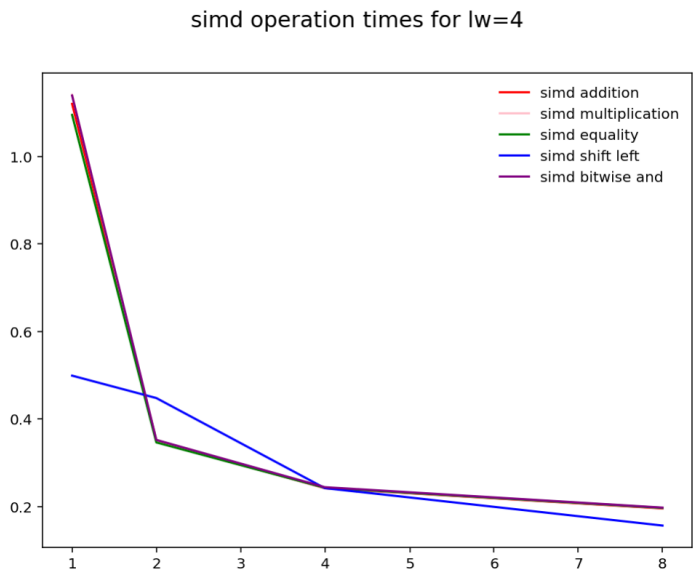

Figure A.3. LC versus time in seconds.

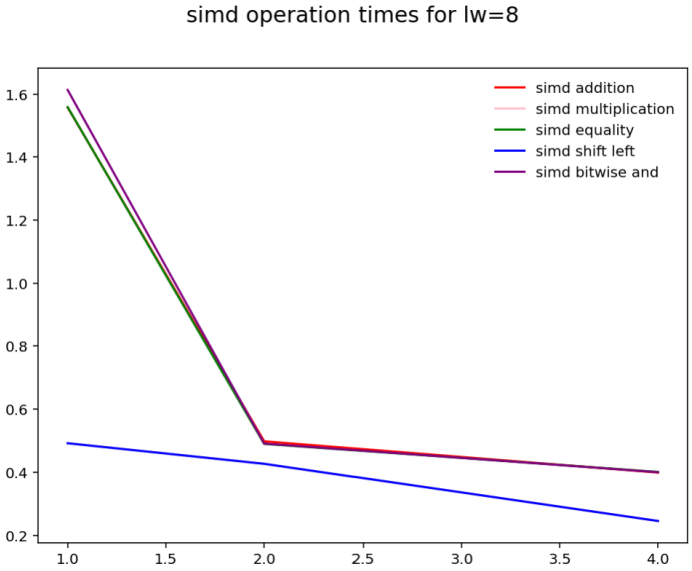

Figure A.4. LC versus time in seconds.

APPENDIX B: SIMD EVM GAS COST REDUCTION PLOTS

We explained in design section that gas costs that are assigned to operations are the same for all SIMD operations except for xmul and xdiv. For that reason the gas consumption in our benchmark branched into 2 different groups as seen by below figures.

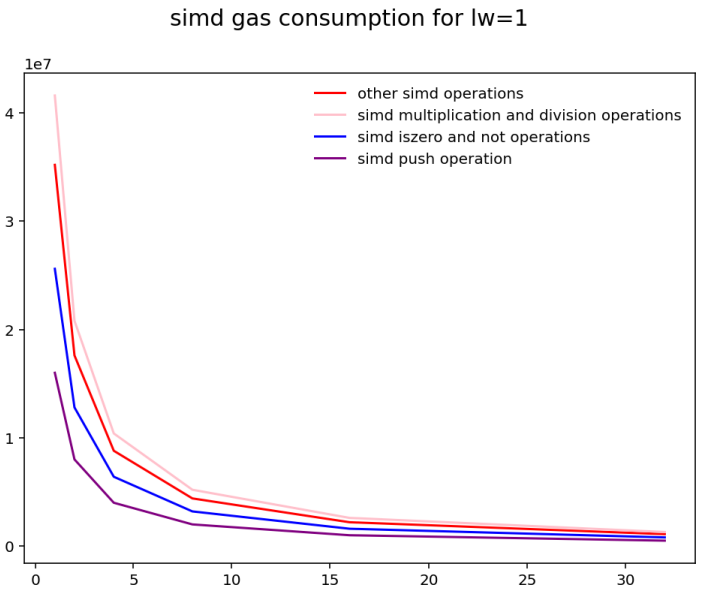

Figure B.1. LC versus gas cost.

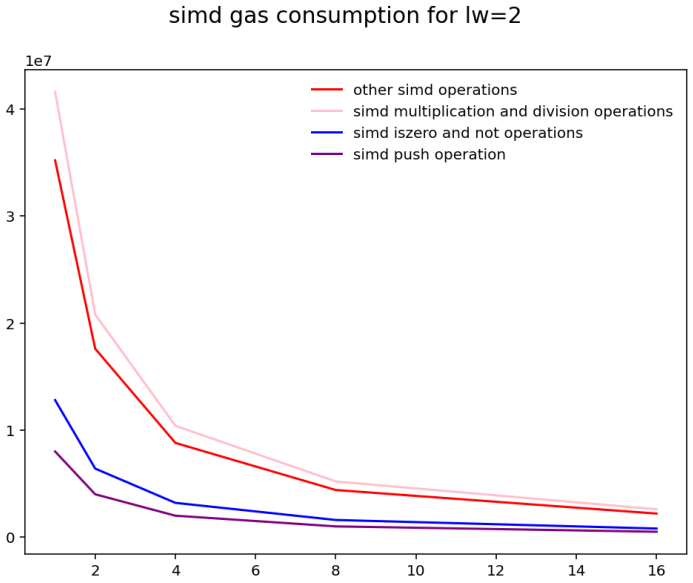

Figure B.2. LC versus gas cost.

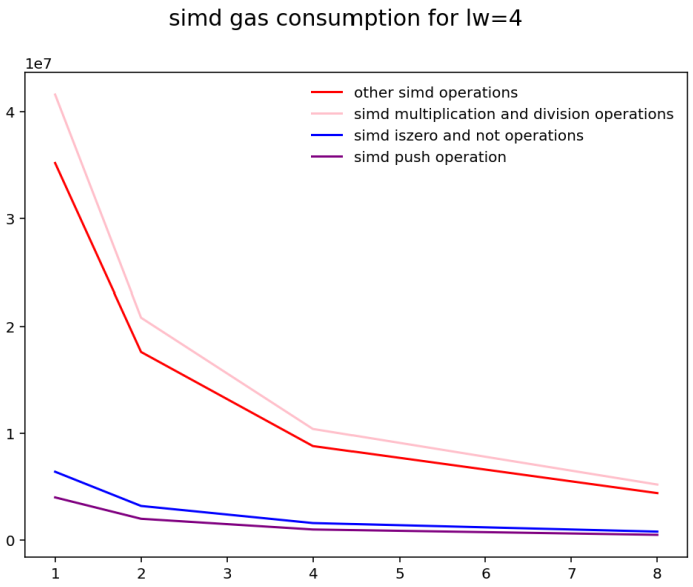

Figure B.3. LC versus gas cost.

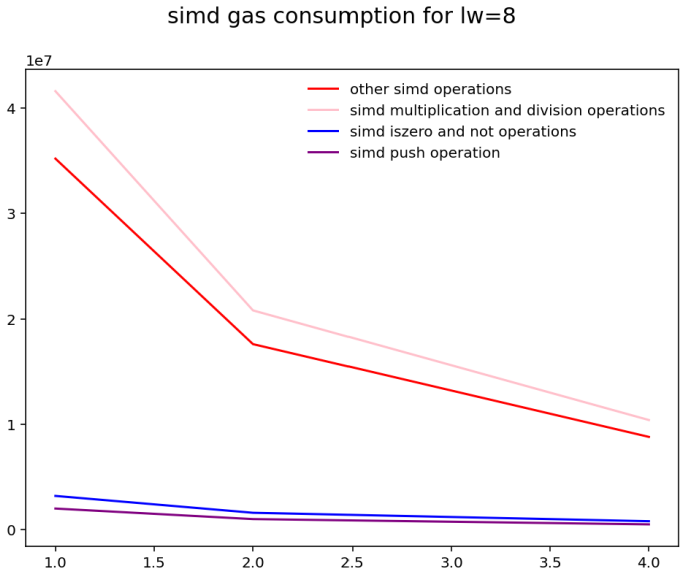

Figure B.4. LC versus gas cost.