Cryptography Acceleration in a RISC-V GPGPU

Austin Adams*†
Pulkit Gupta*
aja@gatech.edu
pgupta91@gatech.edu
Georgia Institute of Technology
Atlanta, Georgia, USA

Blaise Tine
btine3@gatech.edu
Georgia Institute of Technology
Atlanta, Georgia, USA

Hyesoon Kim
hyesoon@cc.gatech.edu
Georgia Institute of Technology
Atlanta, Georgia, USA

ABSTRACT

AES and the SHA family are popular cryptographic algorithms for symmetric encryption and hashing, respectively. Highly parallel use cases for calling both AES and SHA exist, making hardware-accelerated execution on a GPGPU appealing. We extend an existing GPGPU with a cryptography execute unit that will accelerate key elements of these algorithms. We implement a subset of the RISC-V cryptography extensions draft specification on the Vortex GPGPU, seeing 1.6× speedup for SHA-256 and 6.6× speedup for AES-256 on average over pure software implementations on Vortex.

1 INTRODUCTION

A cryptographic accelerator for SHA-256 and AES-256 could be applicable in a handful of use-cases. Indeed, x86 already provides AES and SHA instructions designed to accelerate these workloads [18, 19]. SHA acceleration is highly applicable to cryptanalysis, and specifically for finding SHA collisions [32]. AES acceleration could be applied to full disk encryption and high-throughput encrypted file servers. Together, for example, they could be used for a secure, reliable file distribution system on a public facing network. With the traditional CPU approach, the bandwidth is bottlenecked by the limited number of threads; however on a GPGPU, many files can be operated on in parallel. Also, depending on the cipher mode, a single file can be operated on in parallel. As Vortex is seemingly the first of its kind in the world of open-source GPGPUs that utilize the RISC-V architecture, we will extend this GPGPU with the proposed RISC-V cryptography ISA extension draft to determine the viability of such an accelerator for increasing throughput and bandwidth for AES encryption/decryption and SHA hashing [15, 36].

From here, Section 2 will cover an introduction to SHA-256 and AES-256, followed by a history of cryptographic acceleration on GPGPUs and in RISC-V. Section 3 will detail our approach to the hardware implementation of the specific SHA-256, AES-256, and bit manipulation instructions we have chosen to implement. Section 4 details the assembly intrinsics and algorithm modifications that have been performed to support hardware acceleration for AES-256 and SHA-256.

In Section 5, we show the performance and area differences and provide some insight and analysis as to the results we find. Based on these results, we recommend AES acceleration where workloads require it, but hesitate to encourage implementation of SHA acceleration or bit manipulation instructions as the gains they provide are not comparable to the frequency increase of CPUs compared to physical GPUs. Finally, in Section 6, we briefly outline future work.

*Both authors contributed equally to this research
†Corresponding author

2 BACKGROUND AND RELATED WORK

2.1 Secure Hash Algorithm 2 (SHA-2)

Each algorithm in the SHA-2 family takes in a message and produces a digest (or hash) [29]. The family includes SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224, and SHA-512/256, but in this paper we focus on SHA-256, which produces a 256-bit digest for a given message.

Equations 1 through 6 show some operators required for SHA-256 [29]. x, y, z represent 32-bit words; $\text{ROTR}^n(x)$ rotates a word x right n bits; and $\text{SHR}^n(x)$ shifts a word x right n bits.

$$\text{Ch}(x,y,z) = (x \wedge y) \oplus (\neg x \wedge z) \tag{1}$$

$$\text{Maj}(x,y,z) = (x \wedge y) \oplus (\neg x \wedge z) \oplus (y \wedge z) \tag{2}$$

$$\Sigma_0(x) = \text{ROTR}^2(x) \oplus \text{ROTR}^{13}(x) \oplus \text{ROTR}^{22}(x) \tag{3}$$

$$\Sigma_1(x) = \text{ROTR}^6(x) \oplus \text{ROTR}^{11}(x) \oplus \text{ROTR}^{25}(x) \tag{4}$$

$$\sigma_0(x) = \text{ROTR}^7(x) \oplus \text{ROTR}^{18}(x) \oplus \text{SHR}^3(x) \tag{5}$$

$$\sigma_1(x) = \text{ROTR}^{17}(x) \oplus \text{ROTR}^{19}(x) \oplus \text{SHR}^{10}(x) \tag{6}$$

For each 512-bit message block, the SHA-256 algorithm invokes Ch and Maj 64 times each, Σ_0 and Σ_1 64 times each, and σ_0 and σ_1 48 times each [29].

2.2 Advanced Encryption Standard (AES)

AES is a symmetric block cipher which operates on 128-bit data blocks and key sizes of 128, 192, or 256 bits [14]. In this paper, we focus on "AES-256," which means AES with a 256-bit key. A full description of AES remains out of the scope of this paper (see the specification for details [14]), but we include a high level description of in order to explain our optimizations.

Listing 1: AES cipher pseudocode [14]

```
1  Cipher(byte in[16], byte out[16],
2         byte keysched[16*15])
3  begin
4      byte state[4,4]
5      state = in
6      AddRoundKey(state, keysched[0, 16*3])
7
8      for round = 1 step 1 to 14
```

```
9           SubBytes(state)
10          ShiftRows(state)
11          if round < 14
12              MixColumns(state)
13          AddRoundKey(state,
14                      keysched[round*16,
15                               (round+1)*16-1])
16      end for
17
18      out = state
19  end
```

For context, Listing 1 shows the pseudocode for the AES cipher. The state, input, and output are 4-by-4 column-major matrices of bytes. Descriptions of AES subroutines used in the cipher follow:

- `SubBytes`: Replace each byte in the input word/state according to the S-Box, a predefined non-linear substitution table.
- `AddRoundKey`: XOR each column of the state with its corresponding key in the key schedule (explained below).
- `ShiftRows`: Cyclically left-rotate the bytes in each row of the state. The offset of rotation is the zero-indexed row number.
- `MixColumns`: Replace each entry in each column with a function of the entries in the same column, where the function consists of predefined shifts and XORs.

Note that with the exception of `AddRoundKey`, all four aforementioned subroutines have complementary `Inv*` versions used in the inverse cipher. `InvSubBytes`, for example, replaces bytes according to an inverse S-Box.

A key expansion step which runs before the cipher generates the key schedule (`keysched` in Listing 1) from the 256-bit key. The key schedule contains a separate 4-byte key for each column for each cipher round; all subsequent cipher invocations use the same key schedule for a given cipher key. Notably, for AES-256, key expansion makes 13 calls to `SubWord`, which applies `SubBytes` to all four bytes in an operand, and 7 calls to `RotWord`, which performs an left byte rotation on its operand.

2.3 Block Cipher Modes for AES

Practically, using block ciphers such as AES requires choosing a block cipher mode. Block ciphers modes recommended by NIST include Electronic Code Book (ECB), Cipher Block Chaining (CBC), and Counter (CTR) [13]. ECB runs the cipher on each 16-byte input block independently, which allows parallelizing encryption or decryption across many threads at a block granularity; however, ECB fails to conceal plaintext patterns and is vulnerable to replay attacks [31].

CBC avoids these weaknesses first by using a pseudorandom initialization vector, and second by XORing the previous block's ciphertext with the current block's plaintext before encrypting. CBC decryption thus involves XORing the previous block ciphertext with the output of the inverse cipher on the current block [31]. Although CBC decryption still allows for block-level parallelism, CBC encryption introduces an unfortunate data dependency between adjacent data blocks, requiring encryption to be performed serially.

CTR, however, permits full block-level parallelism while still providing better security than ECB. For both encryption and decryption, it runs the forward AES cipher on an initially-pseudorandom 16-byte counter that increments for every block, XORing the result with the input block [31]. Note this makes the encryption and decryption routines identical.

2.4 GPUs as Cryptographic Accelerators

Cook et al. published the earliest work on accelerating cryptography with GPUs [11]. The authors accelerated both stream and block ciphers using OpenGL with the goal of achieving acceleration with hardware found in many consumer systems, rather than more obscure specialized cryptography-specific hardware. However, even the authors' sophisticated strategies for executing portions of stream and block ciphers (namely XORs and S-Boxes) on GPU graphics pipelines could not overcome the inherent hardware and API limitations, and performance suffered, with their optimized C AES implementation performing nearly double the throughput of their GPU-based implementation. Harrison and Waldron compare strategies for improving the performance, but none surpass AES on a CPU [20].

Shortly after Nvidia released CUDA, Manavski improved on this work by writing an optimized CUDA kernel that outperformed implementations of AES running on CPUs by nearly 20 times [25]. Later, Gilger et al. implemented an open-source OpenSSL engine that GPU-accelerates a variety of block ciphers, including AES, by up to 10 times over CPU-based implementations [17].

Researchers have used GPUs for cryptographic hash functions as well. To generate pseudorandom noise, Tzeng and Wei implemented MD5 on a GPU using OpenGL shaders, achieving both high performance and high scores on statistical tests [34]. Later, researchers implemented CUDA kernels for MD5, achieving high throughput [21, 24]. Today, a major usage of cryptographic hash functions on GPUs is cryptocurrency mining, at which GPGPUs excel [4, 22]. However, despite the usefulness of GPUs for cryptography, we failed to find examples of GPUs equipped with cryptography hardware acceleration.

2.5 Cryptographic Acceleration and RISC-V

Stoffelen published the first paper on RISC-V cryptography optimizations, providing open-source, optimized 32-bit RISC-V software implementations of various cryptographic algorithms, including two different approaches for AES implementation: table-based and bitslice-based [33]. The former involves effecting rounds of AES using lookups into 4 KiB tables, which may lead to vulnerability to timing attacks depending on cache configuration [12]. The latter, on the other hand, slices a given bit of every state byte across eight registers and resists timing attacks by performing the S-Box using bitwise instructions on these bitvectors [23]. Stoffelen estimates the performance benefits of instructions in possible RISC-V extensions, for instance finding that a rotation instruction would improve bitsliced AES performance by 7%.

For the purpose of accelerating the signature scheme XMSS, Wang et al. implemented a discrete SHA-256 accelerator that communicates with a RISC-V CPU [35]. While their work yields an impressive 3.8x speedup for SHA-256 on an FPGA, Fritzmann et

al. note that the remote accelerator strategy used by Wang et al. causes expensive data transfers to and from the accelerator, needs "large buffers to store the input and output" of the accelerator, and suffers from overall inflexibility [16].

Recently, Marshall et al. evaluated multiple robust approaches for AES acceleration in RISC-V [28]. Based on an analysis of performance and hardware complexity, for 32-bit RISC-V systems, the authors recommend a "hardware-assisted T-tables" ISA extension originally proposed by Saarinen [30] which behaves similar to the popular table-based strategy, except computing table entries in hardware at runtime instead of looking them up in memory. The draft specification for RISC-V cryptography extensions uses this strategy for 32-bit RISC-V, defining the following instructions [36]:

- `aes32esi rt, rs2, bs`:
  To encrypt, perform `SubBytes` on the $\texttt{bs}^{\text{th}}$ byte (where `0 <= bs <= 3`) of the column contained in register `rs2`. XOR result into `rt`. Carefully choosing the source and destination registers (columns) allows effectively performing `ShiftRows`.
- `aes32esmi rt, rs2, bs`:
  Same as the previous instruction, except also allow for `MixColumns` by performing shifts as needed; this way, after XORing all shifted source column values into the destination column register, we have effectively performed `MixColumns`. This is a separate instruction from `aes32esi` because `MixColumns` is not used on the last AES round.
- `aes32dsi rt, rs2, bs`:
  Same as `aes32esi` except in reverse, for decryption.
- `aes32dsmi rt, rs2, bs`:
  Same as `aes32esmi` except in reverse, for decryption.

For SHA-256, the draft RISC-V cryptography extension specification also defines four new instructions corresponding to the four SHA-256 sigma subroutines mentioned in Section 2.1 [36]. Each consists solely of bitwise operations on a single operand, with `sha256sum0`, `sha256sum1`, `sha256sig0`, and `sha256sig1` performing Σ_0, Σ_1, σ_0, and σ_1 respectively. The specification does not implement Ch or Maj (Equations 1 and 2 in Section 2.1) as instructions, saying "as ternary functions they are too expensive in terms of opcode space" [36].

In addition to the instructions we have mentioned, the draft specification contains instructions for gathering entropy, AES on 64-bit RISC-V, other functions in the SHA-2 family, the SM3 and SM4 cryptographic algorithms, and some bitwise instructions useful for cryptography. All its bitwise instructions overlap with those already defined in "Bitmanip," an overlapping draft specification for RISC-V bit manipulation instructions [6]. These bit manipulation instructions include `rori rd, rs1, imm`, which cyclically rotates the bits in register `rs1` according to the immediate value `imm`, placing the result in `rd`.

2.6 RISC-V on a GPGPU

There is a "V" (Vector) proposal for vector extensions to RISC-V [5]. However, this provides only a limited SIMD execution model best for CPUs. Collage designed a RISC-V GPGPU with a SIMT execution model but only implemented a limited proof-of-concept [10]. In this paper, we use Vortex, a complete GPGPU using the RISC-V architecture [15]. It implements RV32IMF, that is, 32-bit RISC-V with the base integer instruction set, the multiplication extension, and the floating point extension.

3 HARDWARE IMPLEMENTATION

We have implemented a subset of the version 0.9 draft of the specification for cryptographic extensions to RISC-V [36] in Vortex. Our subset consists of the AES-specific instructions `aes32esi`, `aes32esmi`, `aes32dsi`, and `aes32dsmi`; the SHA-256–specific instructions `sha256sum0`, `sha256sum1`, `sha256sig0`, and `sha256sig1`; and the bit rotation instruction `rori`. Please see Section 2.5 for functional details on these instructions.

To support these instructions in Vortex, we added a new cryptography execution unit, shown in Figure 1, to each Vortex core and adjusted the decode and execute pipeline stages to direct instructions to it. Unfortunately, version 0.9 of the draft specification uses an encoding of the AES instructions that interprets the `rs1` field in the standard RISC-V R-type instruction format as the destination register instead of `rd`, complicating our decoding logic. This inconsistent design choice was intended to save opcode space by using `rd` as a future opcode extension [36] but we find it puzzling. The specification editor has stated he plans to revert to the three-operand design originally proposed by Saarinen [30] in a future version [26], but we chose to follow the current draft specification nonetheless.

We built the AES portion of our cryptography execution unit from the Verilog reference implementation of the draft specification [27]. For S-Boxes, it uses the lightweight scheme proposed by Boyar and Peralta [9], which consists of three layers: a separate outer layer for each of the forward and inverse S-Boxes, a shared middle layer, and an outer layer again separated for forward and inverse. The result is only 128 gates, 16 deep, for the forward S-Box and 127 gates, also 16 deep, for the inverse S-Box [9], but we duplicate this hardware for every thread. To avoid stretching the cycle time of Vortex, we pipelined the S-Box, adding a buffer between the first two layers and the outer layer, as seen in Figure 1.

We have synthesized our modified Vortex design and programmed it on an Arria 10 FPGA, generally maintaining the frequency of the original Vortex design (more details in Section 5.4). We have posted our implementation publicly on GitHub [3].

4 SOFTWARE IMPLEMENTATION

4.1 Pure Software Implementations

To measure the speedup offered by the native instructions we implemented for AES and SHA, we wrote pure software Vortex kernels for SHA-256 and AES-256[1]. The SHA implementation is based off a naïve reading of the specification [29], and the AES implementation uses the lookup table strategy mentioned in Section 2.5, except with a single lookup table as proposed by Daemen and Rijmen [12]. Our software table-based strategy showed an average 1.35× speedup over our original naïve software implementation that it replaced.

For AES, we implemented the ECB, CBC, and CTR cipher modes. Due to the data dependence highlighted in Section 2.3, CBC encryption executes serially in a single thread; all other kernels evenly spread work across all available threads.

[1]To simplify debugging, we first tested our code for these algorithms on a CPU. We have posted this code publicly as well [2].

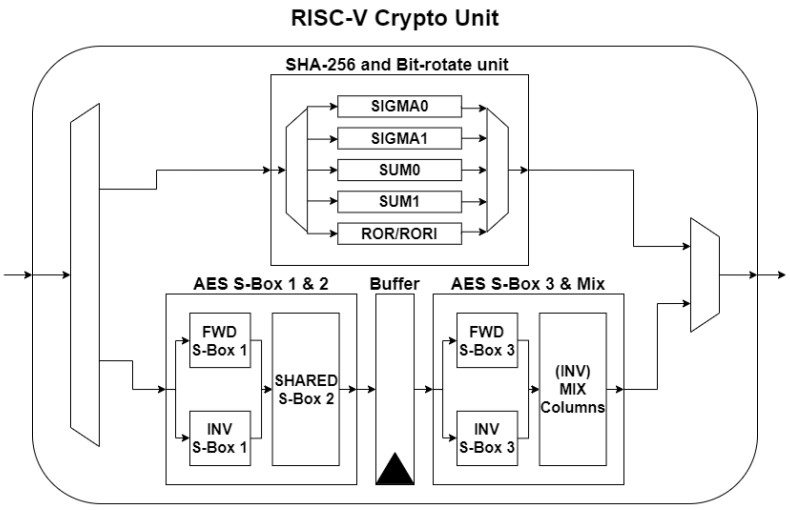

Figure 1: Cryptography execution unit added to Vortex

4.2 Accelerating SHA-256

In addition to the software implementation, to test the new SHA-256 instructions, we created a "native" kernel for SHA-256. The implementation is straightforward: for the sigma functions listed in Section 2.1, we invoke the four new SHA-specific instructions instead of our software versions in C. To insert the instructions into generated code without making compiler modifications, we used the `asm` syntax provided by the GNU C compiler [1]. To get an idea of the impact of the new SHA-256–specific instructions compared to more generic cryptography-friendly instructions, we also implemented a "hybrid" SHA kernel that incorporates `rori` into the software sigma functions instead of replacing them entirely.

4.3 Accelerating AES

To evaluate the new AES instructions, we wrote a "native" AES kernel that uses the four new AES instructions. Listing 2 shows our assembly for an AES encryption round, which consists of four load instructions followed by sixteen `aes32esmi` instructions, one for each byte of the state. We assume register `a0` initially contains the address of the entry in the key schedule for that round, registers `s0-s3` contain the current state columns, and registers `t0-t3` will hold the new state columns. These instructions can effectively replace lines 9-15 in Listing 1.

Listing 2: AES round assembly [28]

```
1  lw t0, 0(a0)
2  lw t1, 4(a0)
3  lw t2, 8(a0)
4  lw t3, 12(a0)
5  aes32esmi t0, s0, 0
6  aes32esmi t0, s1, 1
7  aes32esmi t0, s2, 2
8  aes32esmi t0, s3, 3
9  aes32esmi t1, s1, 0
10 aes32esmi t1, s2, 1
11 aes32esmi t1, s3, 2
12 aes32esmi t1, s0, 3
13 aes32esmi t2, s2, 0
14 aes32esmi t2, s3, 1
15 aes32esmi t2, s0, 2
16 aes32esmi t2, s1, 3
17 aes32esmi t3, s3, 0
18 aes32esmi t3, s0, 1
19 aes32esmi t3, s1, 2
20 aes32esmi t3, s2, 3
```

The AES instructions listed in Section 2.5 were intended for use in the cipher and inverse cipher, but we have used them to accelerate parts of the key expansion as well. To avoid having to store the S-Box in memory for the `SubWord` calls the key expansion routine makes, we wrote a function that invokes `aes32esi` four times, once for each byte of the word.

More subtly, the equivalent inverse cipher explained in Section 5.3.5 of [14], which the `aes32dsi` and `aes32dsmi` instructions implement [28], requires 13 new `InvMixColumns` invocations to be added into the key expansion routine. We cannot use `aes32dsmi` on its own for `InvMixColumns` as it also performs `InvSubBytes`. As a workaround, we first perform sixteen additional `aes32esi` instructions to perform `SubBytes`, which the sixteen `aes32dsmi` instructions then undo via `InvSubBytes`, resulting in the required `InvMixColumns` operation.

Finally, we created a "hybrid" AES kernel that uses only `rori` to accelerate the 7 calls to `RotWord` in key expansion and none of the AES-specific instructions.

5 EVALUATION AND ANALYSIS

To evaluate the speedup provided by our implementation, we executed our AES-256 and SHA-256 kernels on our modified Vortex programmed onto an Intel Arria 10 GX 1150 FPGA. We fed the SHA-256 and AES-256 kernels 1 MiB and 2 MiB (respectively)

Table 1: Cycle and Instruction Counts in Our Experiments

Algorithm	Configuration	Instructions (×1000)	Cycles (×1000)
SHA-256	Software	84335	2137
SHA-256	Hybrid	64642	1707
SHA-256	Native	49503	1331
AES ECB encrypt	Software	451057	22269
AES ECB decrypt	Software	450046	21878
AES ECB encrypt	Native	64741	2469
AES ECB decrypt	Native	62422	2727
AES CBC encrypt	Software	7235197	2798349
AES CBC decrypt	Software	451754	22596
AES CBC encrypt	Native	1012945	398312
AES CBC decrypt	Native	64521	4534
AES CTR encrypt	Software	456585	14979
AES CTR decrypt	Software	456585	14985
AES CTR encrypt	Native	70159	2883
AES CTR decrypt	Native	70165	2767

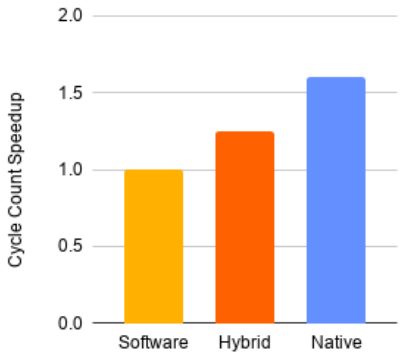

Figure 2: SHA-256 cycle count speedup, with Software normalized to 1

of CPU-generated pseudorandom data, which the kernels spread evenly across 256 threads, with 4 warps of 4 threads on each of the 16 cores.

5.1 SHA-256 Results

Figure 2 shows the results of our SHA-256 experiments, in which we assigned each of 256 threads two messages to hash, each 2 KiB of CPU-generated pseudorandom data. The hybrid implementation, which uses the `rori` instruction in a software implementation of the sigma functions mentioned in Section 2.1, offers a 1.25× speedup over our pure software implementation. The native instructions for the sigma functions provide a 1.30× speedup over hybrid, giving them an overall 1.60× speedup over software.

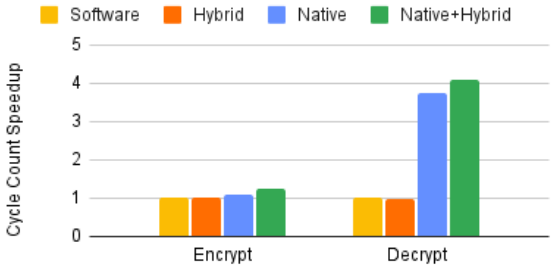

Figure 3: AES-256 key expansion speedup, with software normalized to 1

5.2 AES Key Expansion Results

We evaluate acceleration of AES key schedule generation separately from AES cipher acceleration since it occurs only once for all cipher calls using the same key. Figure 3 shows our results. For key expansion: "Software" uses only our C code and no new instructions; "Hybrid" uses `rori` for the `RotWord` calls; "Native" uses our new AES instructions for `SubWord` and `InvMixColumns` as described in Section 4.3; and "Native+Hybrid" combines "Native" and "Hybrid."

Key expansion for decryption with native instructions clearly shows the strongest speedup. We attribute this to the cost of the software implementation of `InvMixColumns`, which the AES Equivalent Inverse Cipher adds to key expansion, as described in Section 4.3.

5.3 AES-256 Results

For each combination of software and native implementations with block cipher modes ECB, CBC, and CTR, we tasked 256 threads with either decrypting or encrypting 512 16-byte blocks each of CPU-generated pseudorandom data. We have plotted our results in Figure 4.

As mentioned in Section 4.1, AES encryption in the CBC mode is serialized across blocks and must be performed in a single Vortex thread. Thus, the cycle count for encryption with CBC is 125.7 to 161.3 times larger than the cycle count for ECB encryption, as shown in Table 1; we believe the 256 threads used in ECB versus the 1 thread used in CBC encryption explain this.

In Figure 4, we see that CTR achieves a 5.2× and 5.4× speedup for encryption and decryption, respectively (as mentioned in Section 2.3, both are identical). By contrast, the serialized workload for the CBC mode extends the runtime so severely that even the respective 7.0× and 5.0× speedups for encryption and decryption cannot compensate.

5.4 Physical Characteristics

In Table 2, we see that our modified GPGPU still fits in the Intel Arria 10 GX 1150 FPGA we use for the baseline Vortex model. For small core counts, our modification decreases the frequency by a marginal amount, but with larger core counts, the frequency decreases further. This is likely due to synthesis or place and route issues that may be mitigated by optimizing our initial design. We were able to naïvely reduce the area cost of the `rori` instruction

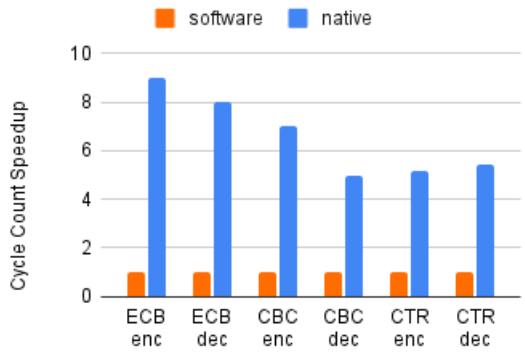

Figure 4: AES-256 cycle count speedup, with software normalized to 1

Table 2: Characteristics on FPGA (427,200 ALMs)

Configuration	Core(s)	Area Usage (%)	Frequency (MHz)
Baseline	1	12.86	220
+ Crypto Unit	1	13.12	218
Baseline	4	26.48	213
+ Crypto Unit	4	27.82	208
Baseline	16	80.24	192
+ Crypto Unit	16	85.78	177

to allow it to fit in the area with the large core count, and expect there may be easy gains elsewhere as well.

5.5 Implementation Recommendations

In regards to a physical implementation of this work for applications such as the use-case mentioned in Section 1, we believe that the AES encryption speedup and the throughput gains of GPUs make it highly advisable to utilize a hardware accelerator for cryptography in a GPGPU. Additionally, for systems operating on many files at once with a longer latency requirement, CBC encryption is still viable, and on systems where files are accessed less often and with shorter latency requirements, CTR mode is advisable. Also, a system targeting only CTR could save hardware by implementing only AES encryption instructions, as CTR decryption does not use the inverse cipher (see Section 2.3).

The limited speedup in SHA-256 workloads is not as convincing since dedicated SHA-256 accelerators have significantly better speedup and throughput over our implementation [35]. The marginal performance improvement of the rotate instruction in SHA and AES key schedule generation leave the hardware cost of `rori` undesirable as well, unless other workloads are able to better utilize it.

While it may be tempting to implement the draft specification in its entirety, thereby including many more instructions than the ones we have implemented, the increase in area and complexity are likely warranted mainly for CPU implementations. However, if a subset of the specification is known to be applicable to a popular workload, then the small increase in area and logical complexity is not insurmountable. In our case of only supporting AES, SHA-256, and `rori` instructions, the area increase and frequency decrease are within reasonable tolerances of the unmodified GPGPU. In an area constrained environment, our estimates in Table 3 show that running the FPGA at the maximum supported frequency (192 MHz), using only four crypto-accelerated cores would exceed the performance of sixteen unmodified cores. Based on this, we believe that the crypto modification can be made on a subset of the cores on a Vortex GPGPU, leaving the rest unmodified. If we add use 4 crypto cores and 12 unmodified cores in the GPGPU, the area should only increase by 1.5% while allowing for accelerated performance of crypto tasks when necessary.

Table 3: Estimates for AES CTR encryption execution time speedup over 16 non-accelerated cores, on various crypto-accelerated core counts

Cores	Speedup at Supported Freq.	Speedup at 192 MHz
1	0.37	0.32
2	0.73	0.65
3	1.08	0.97
4	1.42	1.30
5	1.75	1.62
6	2.07	1.95
7	2.39	2.27
8	2.69	2.60
9	2.98	2.92
10	3.27	3.25
11	3.55	3.57
12	3.81	3.90
13	4.07	4.22
14	4.32	4.54
15	4.56	4.87
16	4.79	5.19

6 FUTURE WORK

Future work should consider the throughput of our work compared to other GPGPUs and CPUs, with and without native instructions such as AES-NI or Intel SHA Extensions in x86 [18, 19]. Additionally, we attempted to optimize our software implementations, in particular for AES, but we do not use advanced software optimization methods such as those proposed by Bertoni et al. [8] or Bernstein and Schwabe [7]. Future work should determine if applying these advanced software strategies reduces the speedup provided by the instructions we implemented. On the hardware side, future work could analyze whether our implementation is vulnerable to hardware attack vectors such as timing attacks, and also conduct a more robust analysis of the impacts of our design on the 15nm Vortex chip as described by Elsabbagh et al. [15].

REFERENCES

[1] 2021. Extended Asm (Using the GNU Compiler Collection (GCC)), Version 12.0.0. https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html.

[2] Austin Adams. 2021. GitHub: AES-256 and SHA-256 Implementations. https://github.com/ausbin/vortex-crypto-algos.

[3] Austin Adams and Pulkit Gupta. 2021. GitHub: Cryptography Instructions for Vortex. https://github.com/ausbin/vortex/tree/crypto.

[4] Mahdi Kh. Alkaeed, Zaid Alamro, Muhammed Samir Al-Ali, Hasan Abbas Al-Mohammed, and Khaled M. Khan. 2020. Highlight on Cryptocurrencies Mining with CPUs and GPUs and their Benefits Based on their Characteristics. In *2020 IEEE 10th International Conference on System Engineering and Technology (ICSET)*. 67–72. https://doi.org/10.1109/ICSET51301.2020.9265386 ISSN: 2470-640X.

[5] Alon Amid, Krste Asanovic, Allen Baum, Alex Bradbury, Tony Brewer, Chris Celio, Aliaksei Chapyzhenka, Silviu Chiricescu, Ken Dockser, Bob Dreyer, Roger Espasa, Sean Halle, John Hauser, David Horner, Bruce Hoult, Bill Huffman, Nicholas Knight, Constantine Korikov, Ben Korpan, Hanna Kruppe, Yunsup Lee, Guy Lemieux, Grigorios Magklis, Filip Moc, Rich Newell, Albert Ou, David Patterson, Colin Schmidt, Alex Solomatnikov, Steve Wallach, Andrew Waterman, and Jim Wilson. 2021. RISC-V Vector Extension. https://github.com/riscv/riscv-v-spec/releases/tag/v0.10.

[6] Jacob Bachmeyer, Allen Baum, Ari Ben, Alex Bradbury, Steven Braeger, Rogier Brussee, Michael Clark, Ken Dockser, Paul Donahue, Dennis Ferguson, Fabian Giesen, John Hauser, Robert Henry, Bruce Hoult, Po-wei Huang, Ben Marshall, Rex McCrary, Lee Moore, Jiri Moravec, Samuel Neves, Markus Oberhumer, Christopher Olson, Nils Pipenbrinck, Joseph Rahmeh, Xue Saw, Tommy Thorn, Avishai Tvila, Andrew Waterman, Thomas Wicki, and Claire Wolf. 2021. RISC-V Bitmanip Extension. https://github.com/riscv/riscv-bitmanip/releases/tag/v0.93.

[7] Daniel J. Bernstein and Peter Schwabe. 2008. New AES Software Speed Records. In *Progress in Cryptology - INDOCRYPT 2008 (Lecture Notes in Computer Science)*, Dipanwita Roy Chowdhury, Vincent Rijmen, and Abhijit Das (Eds.). Springer, Berlin, Heidelberg, 322–336. https://doi.org/10.1007/978-3-540-89754-5_25

[8] Guido Bertoni, Luca Breveglieri, Pasqualina Fragneto, Marco Macchetti, and Stefano Marchesin. 2003. Efficient Software Implementation of AES on 32-Bit Platforms. In *Cryptographic Hardware and Embedded Systems - CHES 2002 (Lecture Notes in Computer Science)*. Springer, Berlin, Heidelberg, 159–171. https://doi.org/10.1007/3-540-36400-5_13

[9] Joan Boyar and René Peralta. 2012. A Small Depth-16 Circuit for the AES S-Box. In *Information Security and Privacy Research (IFIP Advances in Information and Communication Technology)*, Dimitris Gritzalis, Steven Furnell, and Marianthi Theoharidou (Eds.). Springer, Berlin, Heidelberg, 287–298. https://doi.org/10.1007/978-3-642-30436-1_24

[10] Caroline Collange. 2017. Simty: generalized SIMT execution on RISC-V. In *CARRV 2017: First Workshop on Computer Architecture Research with RISC-V*. https://hal.inria.fr/hal-01622208

[11] Debra L. Cook, John Ioannidis, Angelos D. Keromytis, and Jake Luck. 2005. CryptoGraphics: Secret Key Cryptography Using Graphics Cards. In *Topics in Cryptology – CT-RSA 2005 (Lecture Notes in Computer Science)*, Alfred Menezes (Ed.). Springer, Berlin, Heidelberg, 334–350. https://doi.org/10.1007/978-3-540-30574-3_23

[12] Joan Daemen and Vincent Rijmen. 2002. *The Design of Rijndael*. Springer Berlin Heidelberg, Berlin, Heidelberg. https://doi.org/10.1007/978-3-662-04722-4

[13] Morris Dworkin. 2001. *Recommendation for Block Cipher Modes of Operation: Methods and Techniques*. Technical Report NIST Special Publication (SP) 800-38A. National Institute of Standards and Technology. https://doi.org/10.6028/NIST.SP.800-38A

[14] Morris J. Dworkin, Elaine B. Barker, James R. Nechvatal, James Foti, Lawrence E. Bassham, E. Roback, and James F. Dray Jr. 2001. Advanced Encryption Standard (AES). (Nov. 2001). https://www.nist.gov/publications/advanced-encryption-standard-aes Last Modified: 2021-03-01T01:03-05:00.

[15] Fares Elsabbagh, Blaise Tine, Priyadarshini Roshan, Ethan Lyons, Euna Kim, Da Eun Shim, Lingjun Zhu, Sung Kyu Lim, and Hyesoon Kim. 2020. Vortex: OpenCL Compatible RISC-V GPGPU. *arXiv:2002.12151 [cs]* (Feb. 2020). http://arxiv.org/abs/2002.12151 arXiv: 2002.12151.

[16] Tim Fritzmann, Georg Sigl, and Johanna Sepúlveda. 2020. RISQ-V: Tightly Coupled RISC-V Accelerators for Post-Quantum Cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems* (Aug. 2020), 239–280. https://doi.org/10.13154/tches.v2020.i4.239-280

[17] Johannes Gilger, Johannes Barnickel, and Ulrike Meyer. 2012. GPU-Acceleration of Block Ciphers in the OpenSSL Cryptographic Library. In *Information Security (Lecture Notes in Computer Science)*, Dieter Gollmann and Felix C. Freiling (Eds.). Springer, Berlin, Heidelberg, 338–353. https://doi.org/10.1007/978-3-642-33383-5_21

[18] Shay Gueron. 2010. *Intel Advanced Encrypt Standard (AES) New Instruction Set*. Technical Report.

[19] Sean Gulley, Vinodh Gopal, Kirk Yap, Wajdi Feghali, Jim Guilford, and Gil Wolrich. 2013. *Intel SHA Extensions*. Technical Report. https://www.intel.com/content/www/us/en/develop/articles/intel-sha-extensions.html

[20] Owen Harrison and John Waldron. 2007. AES Encryption Implementation and Analysis on Commodity Graphics Processing Units. In *Cryptographic Hardware and Embedded Systems - CHES 2007 (Lecture Notes in Computer Science)*, Pascal Paillier and Ingrid Verbauwhede (Eds.). Springer, Berlin, Heidelberg, 209–226. https://doi.org/10.1007/978-3-540-74735-2_15

[21] Guang Hu, Jianhua Ma, and Benxiong Huang. 2009. High Throughput Implementation of MD5 Algorithm on GPU. In *Proceedings of the 4th International Conference on Ubiquitous Information Technologies Applications*. 1–5. https://doi.org/10.1109/ICUT.2009.5405734 ISSN: 1976-0035.

[22] Alexandr Kuznetsov, Kyryl Shekhanin, Andrii Kolhatin, Diana Kovalchuk, Vitalina Babenko, and Iryna Perevozova. 2019. Performance of Hash Algorithms on GPUs for Use in Blockchain. In *2019 IEEE International Conference on Advanced Trends in Information Theory (ATIT)*. 166–170. https://doi.org/10.1109/ATIT49449.2019.9030442

[23] Emilia Käsper and Peter Schwabe. 2009. Faster and Timing-Attack Resistant AES-GCM. In *Cryptographic Hardware and Embedded Systems - CHES 2009 (Lecture Notes in Computer Science)*, Christophe Clavier and Kris Gaj (Eds.). Springer, Berlin, Heidelberg, 1–17. https://doi.org/10.1007/978-3-642-04138-9_1

[24] Changxin Li, Hongwei Wu, Shifeng Chen, Xiaochao Li, and Donghui Guo. 2009. Efficient implementation for MD5-RC4 encryption using GPU with CUDA. In *and Identification in Communication 2009 3rd International Conference on Anti-counterfeiting, Security*. 167–170. https://doi.org/10.1109/ICASID.2009.5276924 ISSN: 2163-5056.

[25] Svetlin A. Manavski. 2007. CUDA Compatible GPU as an Efficient Hardware Accelerator for AES Cryptography. In *2007 IEEE International Conference on Signal Processing and Communications*. 65–68. https://doi.org/10.1109/ICSPC.2007.4728256

[26] Ben Marshall. 2021. Re: Some issues for discussion. https://lists.riscv.org/g/tech-crypto-ext/message/473

[27] Ben Marshall. 2021. RISC-V Crypto RTL. https://github.com/riscv/riscv-crypto/tree/f5db502dd266666a800875b3b5ffa0158d08aae2/rtl.

[28] Ben Marshall, G. Richard Newell, Dan Page, Markku-Juhani O. Saarinen, and Claire Wolf. 2021. The design of scalar AES Instruction Set Extensions for RISC-V. *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2021), 109–136. https://doi.org/10.46586/tches.v2021.i1.109-136

[29] National Institute of Standards and Technology. 2015. *Secure Hash Standard (SHS)*. Technical Report Federal Information Processing Standard (FIPS) 180-4. U.S. Department of Commerce. https://doi.org/10.6028/NIST.FIPS.180-4

[30] Markku-Juhani O. Saarinen. 2020. A Lightweight ISA Extension for AES and SM4. *arXiv:2002.07041 [cs]* (Aug. 2020). http://arxiv.org/abs/2002.07041 arXiv: 2002.07041.

[31] Bruce Schneier. 2015. *Applied Cryptography: Protocols, Algorithms and Source Code in C* (20th edition ed.). Wiley, Indianapolis, IN.

[32] Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov. 2017. The First Collision for Full SHA-1. In *Advances in Cryptology – CRYPTO 2017 (Lecture Notes in Computer Science)*, Jonathan Katz and Hovav Shacham (Eds.). Springer International Publishing, Cham, 570–596. https://doi.org/10.1007/978-3-319-63688-7_19

[33] Ko Stoffelen. 2019. Efficient Cryptography on the RISC-V Architecture. In *Progress in Cryptology – LATINCRYPT 2019 (Lecture Notes in Computer Science)*, Peter Schwabe and Nicolas Thériault (Eds.). Springer International Publishing, Cham, 323–340. https://doi.org/10.1007/978-3-030-30530-7_16

[34] Stanley Tzeng and Li-Yi Wei. 2008. Parallel white noise generation on a GPU via cryptographic hash. In *Proceedings of the 2008 symposium on Interactive 3D graphics and games (I3D '08)*. Association for Computing Machinery, New York, NY, USA, 79–87. https://doi.org/10.1145/1342250.1342263

[35] Wen Wang, Bernhard Jungk, Julian Wälde, Shuwen Deng, Naina Gupta, Jakub Szefer, and Ruben Niederhagen. 2020. XMSS and Embedded Systems. In *Selected Areas in Cryptography – SAC 2019 (Lecture Notes in Computer Science)*, Kenneth G. Paterson and Douglas Stebila (Eds.). Springer International Publishing, Cham, 523–550. https://doi.org/10.1007/978-3-030-38471-5_21

[36] Alexander Zeh, Andy Glew, Barry Spinney, Ben Marshall, Daniel Page, Derek Atkins, Ken Dockser, Markku-Juhani O. Saarinen, Nathan Menhorn, Richard Newell, and Claire Wolf. 2021. RISC-V Cryptographic Extension Proposals Volume I: Scalar & Entropy Source Instructions. https://github.com/riscv/riscv-crypto/releases/tag/v0.9.0-scalar.