Argon2 on GPU

https://youtu.be/3m6W0HxG-ZU

1. 해시 함수

- 해시 함수
  - 해시 함수는 입력 데이터를 고정된 길이의 고유한 문자열(해시값)로 변환하는 알고리즘

- 해시 함수의 특징
  - 동일한 입력 값에 대해 항상 같은 해시 값 생성(무결성)
    - 입력 값이 조금만 달라도 완전히 다른 해시 값을 생성

- 해시 함수의 용도
  - 데이터 베이스에서 데이터 검색을 빠르게 수행하기 위해 인덱싱에 사용
  - 비밀번호 관리를 안전하게 하기 위하여 사용
  - 블록체인 기술에서 응용

2. 비밀번호 해시 함수

- 비밀번호 해시 함수
  - 특별한 성질을 갖춘 해시 함수로 사용자 비밀번호의 안전한 저장과 검증을 위해 사용

- 일반 해시 함수와의 차이점
  - 해시 계산에 더 많은 시간이 걸리도록 설계
    - GPU/CPU에 의한 크래킹 공격에 대한 저항력을 높이기 위해서
  - 솔트(Salt) 사용을 통해서 공격자가 미리 계산한 레인보우 테이블을 사용한 공격을 기본적으로 방어 가능
  - 고용량의 메모리를 사용하여 계산되도록 설계되어 효과적인 해시 공격을 늦추거나 비용을 증가시킴

- 대표적인 비밀번호 해시 함수
  - Bcrypt / Scrypt / Arogn2

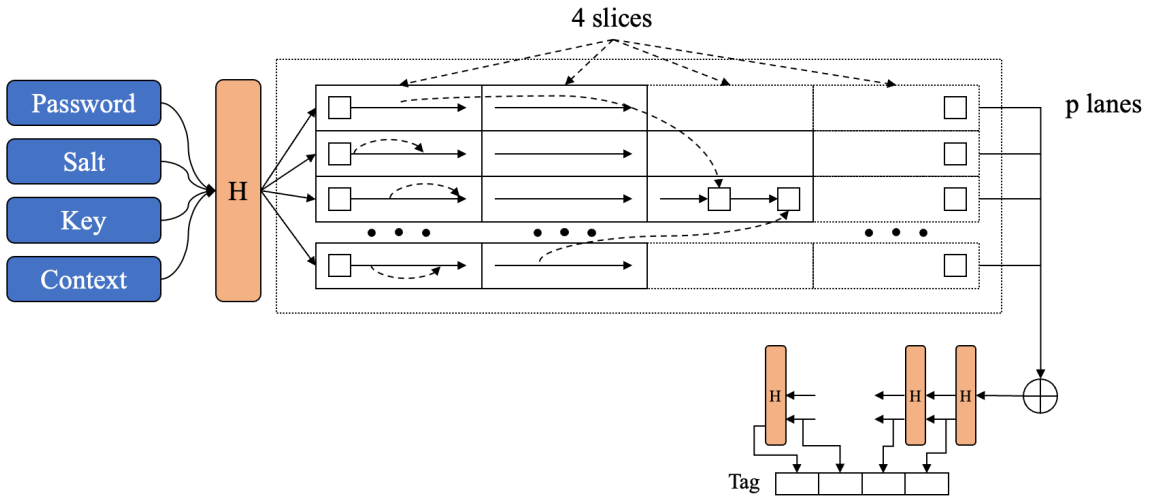
3. Argon2

- Argon2는 비밀번호 해시 함수로 2015년 Password Hashing Competition에서 최종 우승한 암호로 3개의 버전을 지원
  - Argon2d / Argno2i / Argon2id

- Argon2i
  - 데이터 독립적인 메모리 액세스를 사용하여 구성, 부채널 공격으로부터 안전
- Argon2d
  - 데이터 종속적인 메모리 액세스를 사용하여 구성, 대량의 메모리 요구 사항으로 GPU를 활용한 공격에 저항력이 높음
- Argon2id
  - Argon2d와 Argon2i의 결합으로 하이브리드 버전

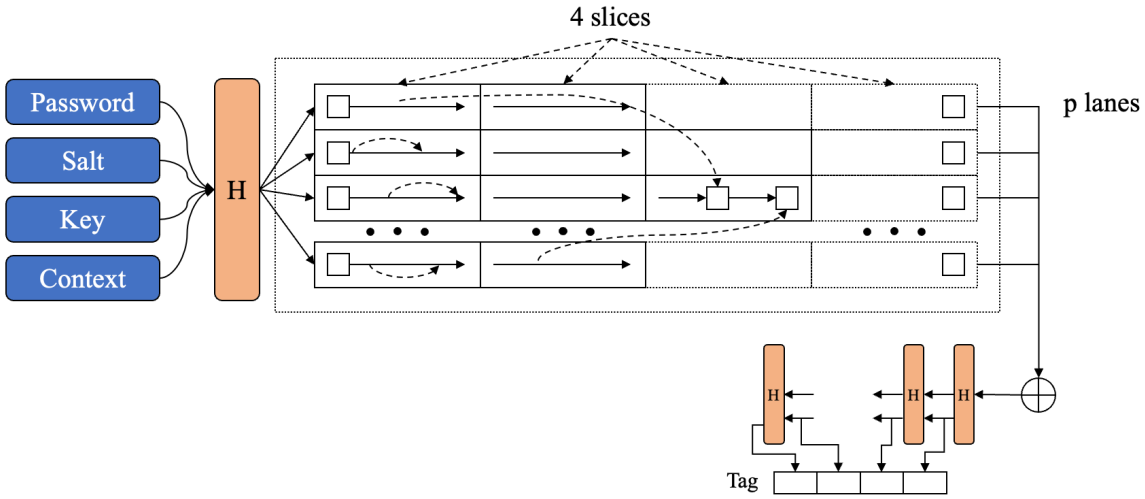
3. Argon2

- Argon2 해싱 과정
  - Initialization – 비밀번호, 솔트, 키 등 여러 파라미터를 통해서 초기 블록을 생성
    - 블록 생성 시 blake2b 해시 함수 사용
  - Block Filling – 초기 블록을 기반으로 설정된 메모리 크기까지 블록을 채움
    - 이전의 블록이 다음 블록에 영향을 주기 때문에 이 과정은 데이터 종속적임
    - Argon2i / Argon2d 에 따라서 다른 접근 방식을 따름
  - Final Block Creation 그리고 Hash Generation
    - 마지막 메모리 블록의 값을 압축하고 최종 해시 값을 계산하는 과정 진행



3. Argon2

- Argon2 해싱 과정
  - Initialization
  - Block Filling
  - Final Block Creation / Hash Generation

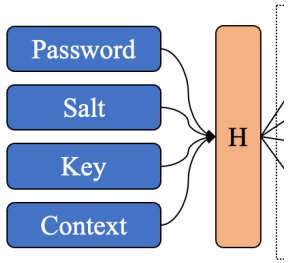



```
    for (std::size_t i = 0; i < batchSize; i++) {

        const void* pw;
        std::size_t pwLength;
        pwGen.nextPassword(pw, pwLength);
        std::string pw2;
        pw2.resize(pwLength);
        for (std::size_t i = 0; i < pwLength; i++) {
            pw2[i] = i+2;
        }
        unit.setPassword(i, pw, pwLength);
    }
    clock_type::time_point checkpt1 = clock_type::now();

    unit.beginProcessing();
    unit.endProcessing();

    clock_type::time_point checkpt2 = clock_type::now();
    for (std::size_t i = 0; i < batchSize; i++) {
        uint8_t buffer[HASH_LENGTH];
        unit.getHash(i, buffer);
    }
    clock_type::time_point checkpt3 = clock_type::now();
```

3. Argon2 – 기존 구현 코드 분석

- Argon2 해싱 과정
  - Initialization – 비밀번호, 솔트, 키 등 여러 파라미터를 통해서 초기 블록을 생성
    - 블록 생성 시 blake2b 해시 함수 사용

```
void ProcessingUnit::setPassword(std::size_t index, const void* pw,
    std::size_t pwSize)
{
    std::size_t size = params->getLanes() * 2 * ARGON2_BLOCK_SIZE;
    auto buffer = std::unique_ptr<uint8_t[]>(new uint8_t[size]);
    params->fillFirstBlocks(buffer.get(), pw, pwSize,
        programContext->getArgon2Type(),
        programContext->getArgon2Version());

    runner.writeInputMemory(index, buffer.get());
}
```

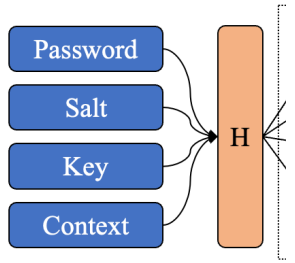


3. Argon2 – 기존 구현 코드 분석

- Argon2 해싱 과정
  - Initialization – 비밀번호, 솔트, 키 등 여러 파라미터를 통해서 초기 블록을 생성
    - 블록 생성 시 blake2b 해시 함수 사용

```
std::uint8_t initHash[ARGON2_PREHASH_SEED_LENGTH];
initialHash(initHash, pwd, pwdLen, type, version);
```

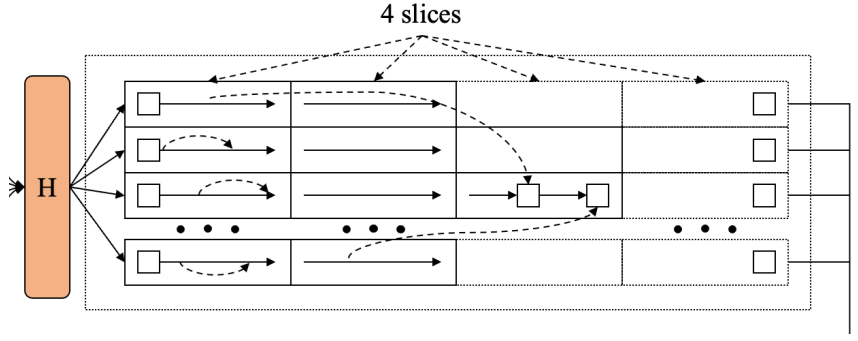
```
    store32(initHash + ARGON2_PREHASH_DIGEST_LENGTH, 1);
    for (std::uint32_t l = 0; l < lanes; l++) {
        store32(initHash + ARGON2_PREHASH_DIGEST_LENGTH + 4, l);
        digestLong(bmemory, ARGON2_BLOCK_SIZE, initHash, sizeof(initHash));

def DEBUG
        std::fprintf(stderr, "Initial block 1 for lane %u: {\n", (unsigned)l);
        for (std::size_t i = 0; i < ARGON2_BLOCK_SIZE / 8; i++) {
            std::fprintf(stderr, "  0x");
            for (std::size_t k = 0; k < 8; k++) {
                std::fprintf(stderr, "%02x", (unsigned)bmemory[i * 8 + 7 - k]);
            }
            std::fprintf(stderr, "UL,\n");
        }
        std::fprintf(stderr, "}\n");
dif

        bmemory += ARGON2_BLOCK_SIZE;
    }
```



3. Argon2

- Argon2 해싱 과정
  - Block Filling – 초기 블록을 기반으로 설정된 메모리 크기까지 블록을 채움
    - 이전의 블록이 다음 블록에 영향을 주기 때문에 이 과정은 데이터 종속적임
    - Argon2i / Argon2d 에 따라서 다른 접근 방식을 따름

```
for (uint32_t offset = start_offset; offset < segment_blocks; ++offset) {
    argon2_step<type, version>(
        memory, mem_curr, &prev, &tmp, &addr, shuffle_buf,
        lanes, segment_blocks, thread, &thread_input,
        lane, pass, slice, offset);

    mem_curr += lanes;
}
```



3. Argon2

```
if (type == ARGON2_I || (type == ARGON2_ID && pass == 0 &&
    slice < ARGON2_SYNC_POINTS / 2)) {
    uint32_t addr_index = offset % ARGON2_QWORDS_IN_BLOCK;
    if (addr_index == 0) {
        if (thread == 6) {
            ++* thread_input;
        }
        next_addresses(addr, tmp, *thread_input, thread, shuffle_buf);
    }

    uint32_t thr = addr_index % THREADS_PER_LANE;
    uint32_t idx = addr_index / THREADS_PER_LANE;

    uint64_t v = block_th_get(addr, idx);
    v = u64_shuffle(v, thr, thread, shuffle_buf);
    ref_index = u64_lo(v);
    ref_lane = u64_hi(v);
}
```

니 크기까지 블록을 채움

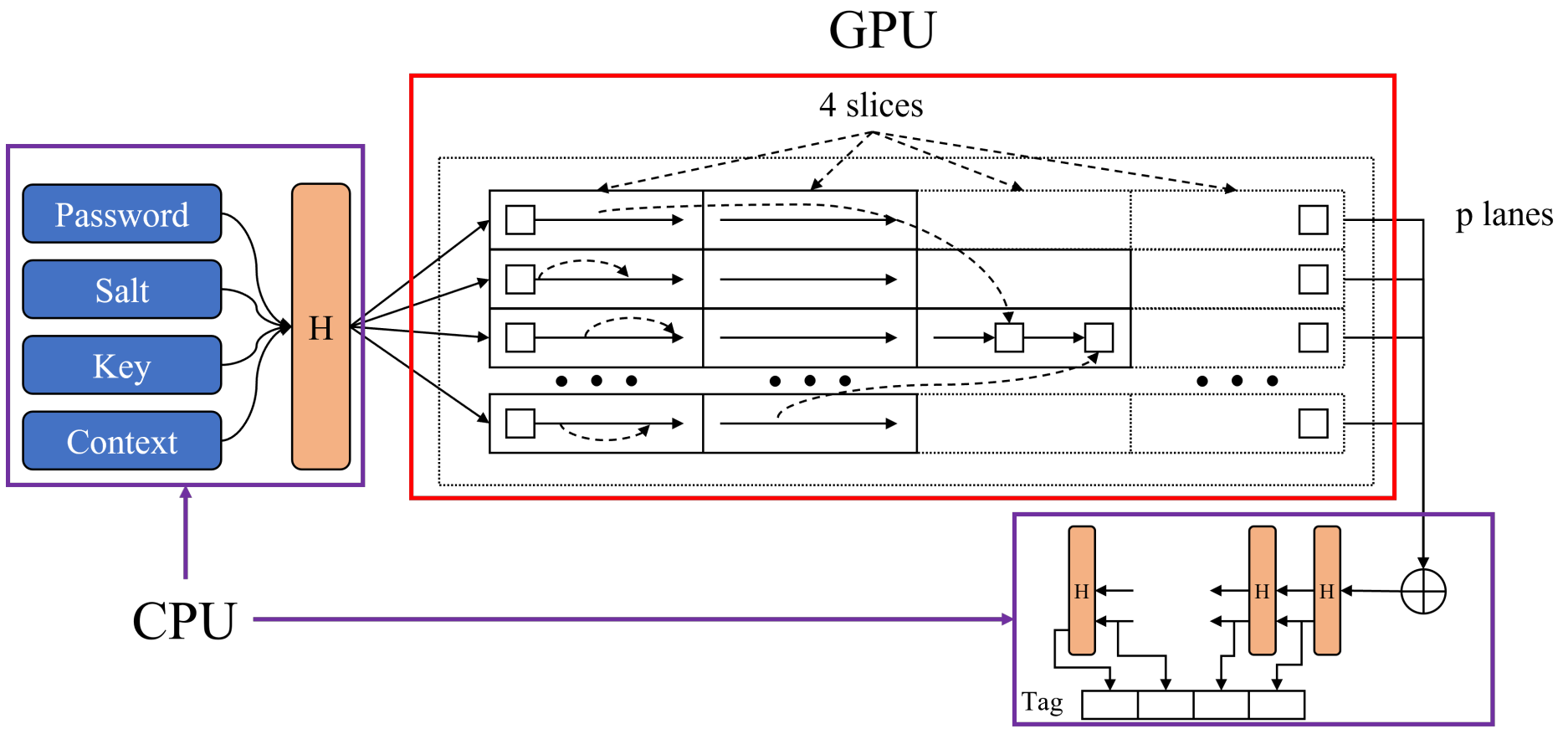
과정은 데이터 종속적임

름

```
else {
    uint64_t v = u64_shuffle(prev->a, 0, thread, shuffle_buf);
    ref_index = u64_lo(v);
    ref_lane = u64_hi(v);
}

compute_ref_pos(lanes, segment_blocks, pass, lane, slice, offset,
    &ref_lane, &ref_index);

argon2_core<version>(memory, mem_curr, prev, tmp, shuffle_buf, lanes,
    thread, pass, ref_index, ref_lane);
```

4. 구현 기법 제안

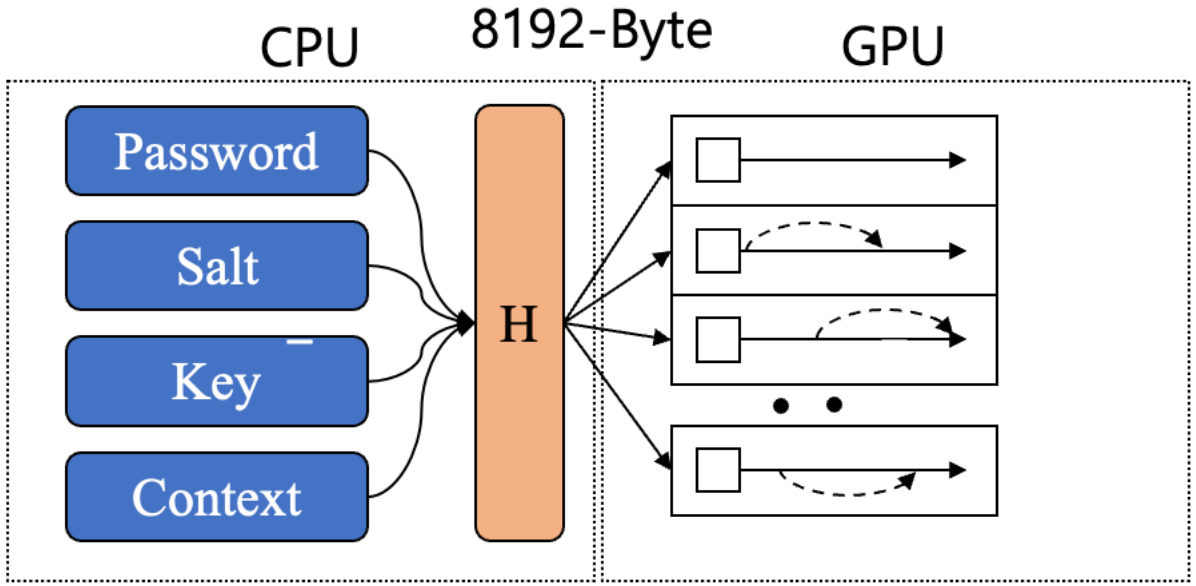
- 기존의 GPU를 활용한 구현
  - 크래킹 공격 가능함을 보여주기 위해 GPU를 활용한 Argon2를 최적화함



4. 구현 기법 제안

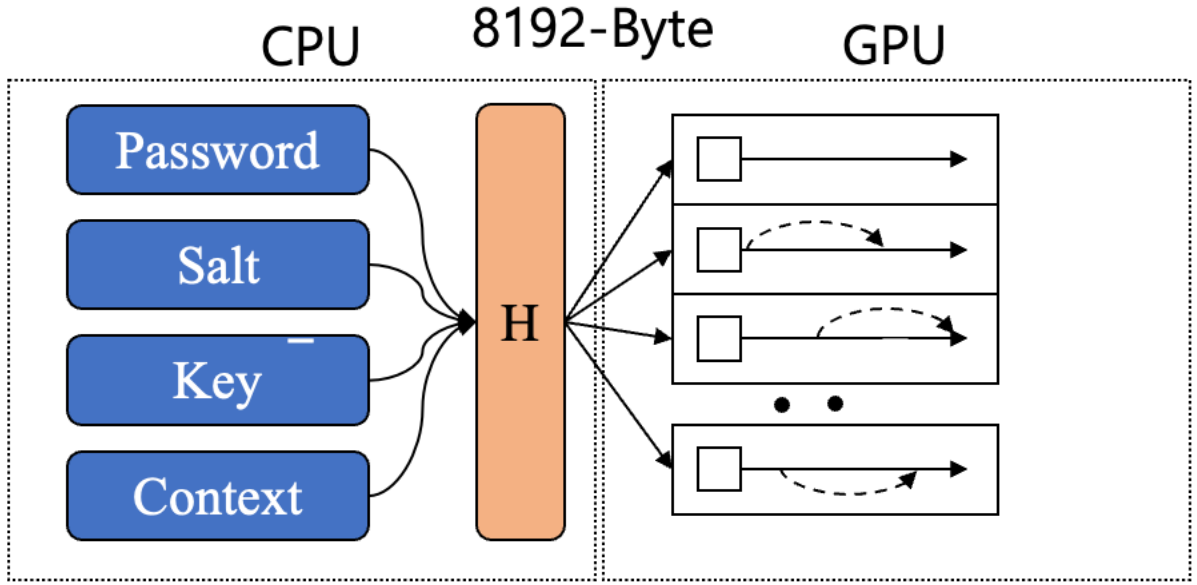
- GPU를 활용한 구현에서는 데이터의 크기가 성능에 영향을 줌
  - Stream, pinned 메모리 사용 등 데이터 복사 비용을 감소시키기 위한 기능을 제공

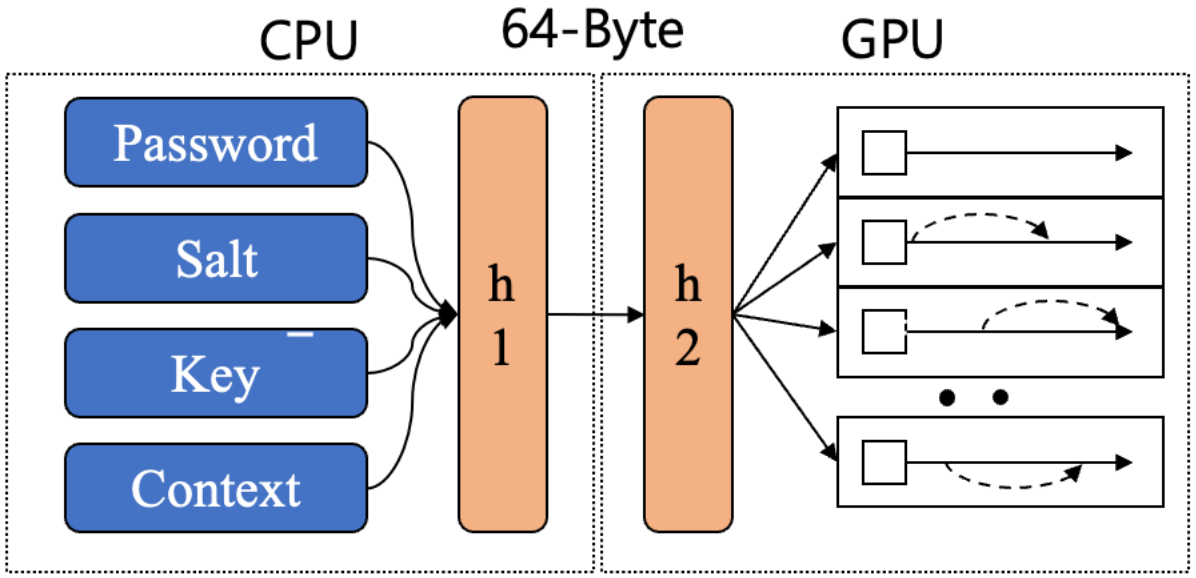
- 기존 구현에서 가장 작은 파라미터를 사용하여 하나의 해시를 생성할 때 8192-byte를 복사하는 것을 확인



4. 구현 기법 제안

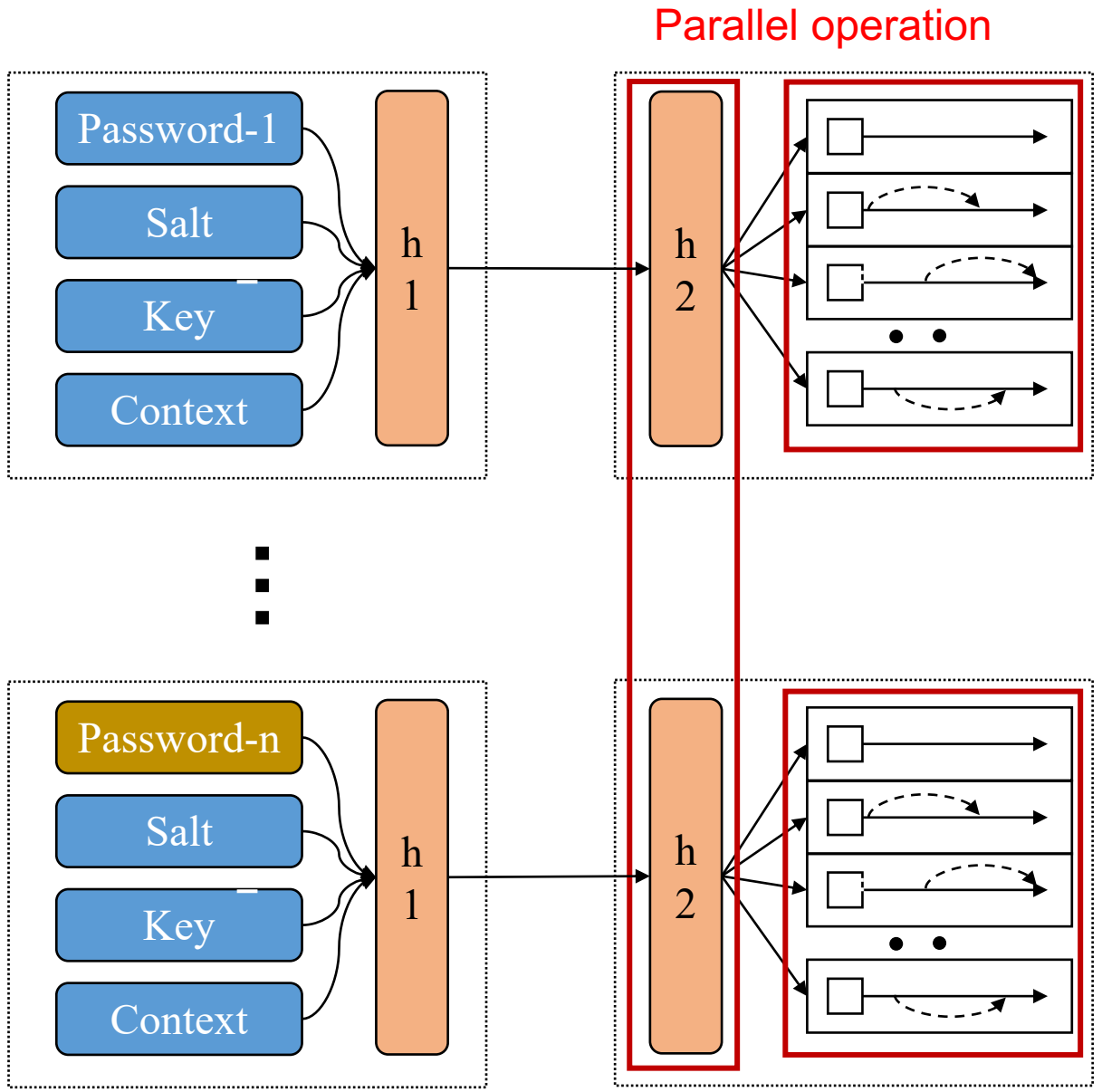
- 기존의 H 함수에서는 Blake2b 해시 함수를 활용
  - 입력된 여러 정보를 기반으로 초기 해시 값 계산 (64-byte) h1
  - 초기 해시 값을 1024-byte로 확장
  - 초기 블록에 저장을 위해서 8192-byte로 확장 h2





4. 구현 기법 제안

- 제안하는 기법을 적용 하면 여러개의 해시함수를 병렬 연산 할 때 복사하는 비용이 감소



감 사 합 니 다