ARIA-CTR on Apple M1 GPU

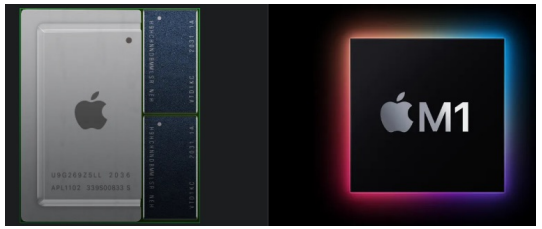
https://youtu.be/B5balooqc24

한성대학교 HANSUNG UNIVERSITY

CryptoCraft LAB

1. Apple M1 GPU

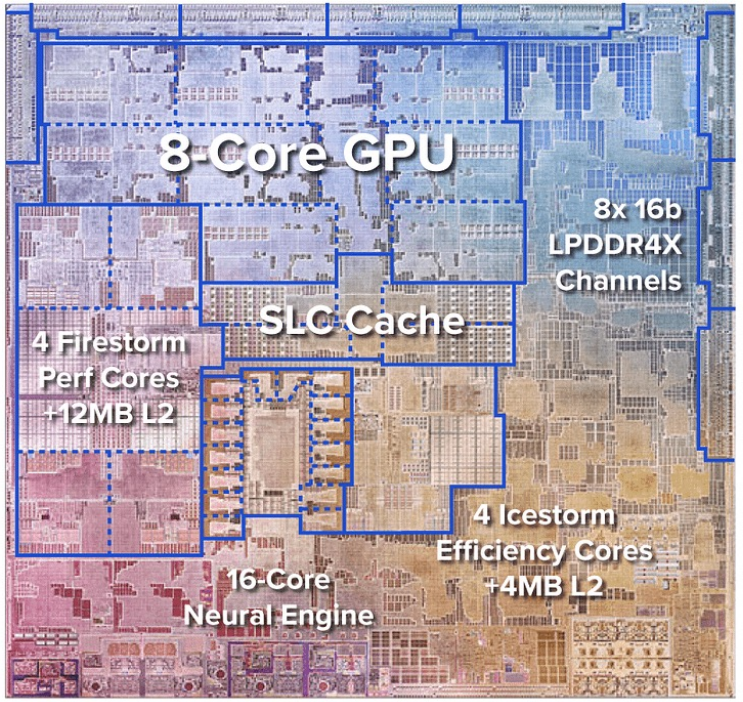
- ARM 기반의 Apple에서 자체 개발한 설계 칩
  - 2020년 M1 출시로 Intel에서 자체 설계인 ARM 기반 칩으로 전환
  - SoC 구조: CPU, GPU, Nural Engine, 메모리 컨트롤러 등을 하나의 칩에 통합
  - 통합 메모리 아키텍처(Unified Memory Architecure, UMA)
    - CPU와 GPU가 동일한 메모리 풀을 공유



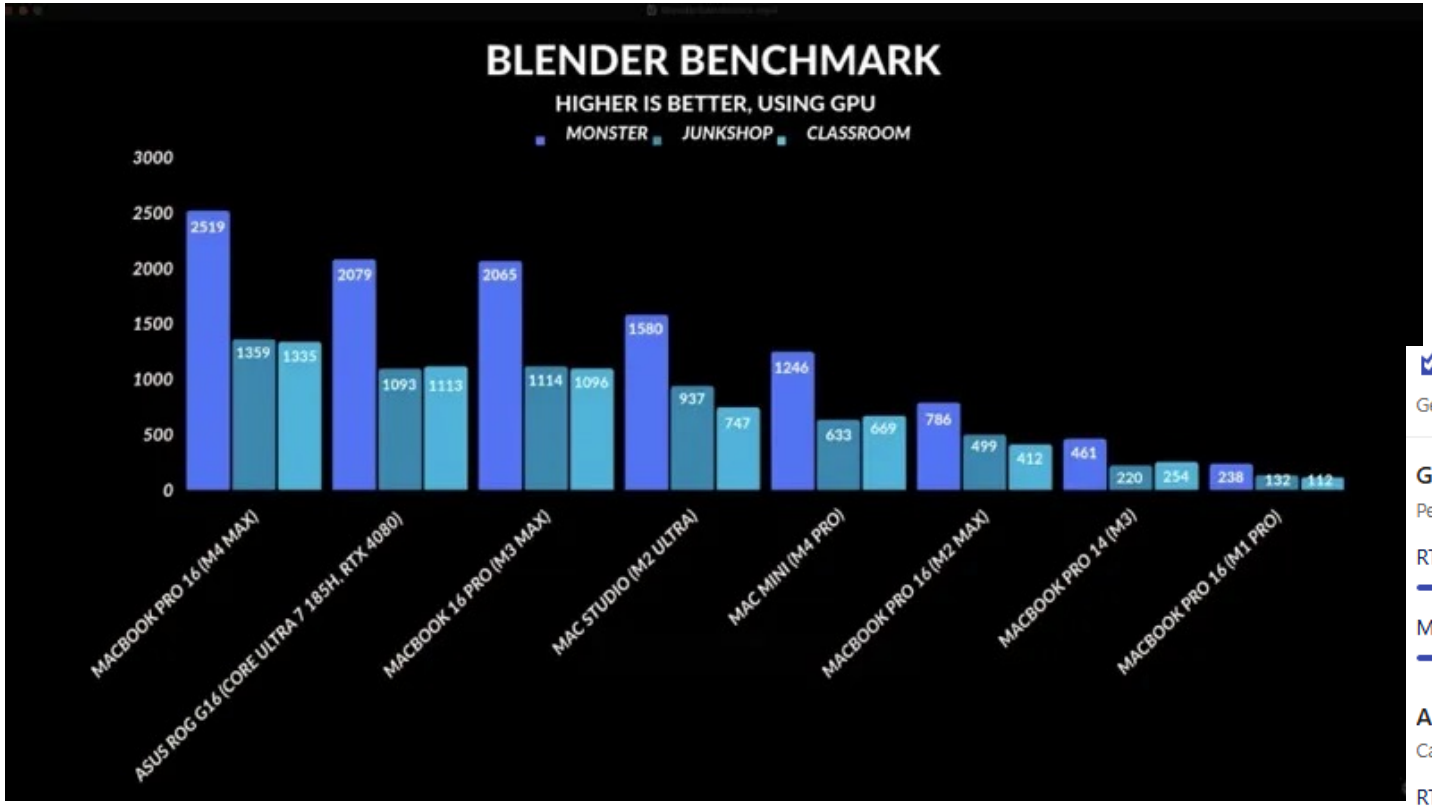
  - M시리즈 라인업
    - M1(2020) – 8코어 CPU, 최대 8코어 GPU
    - M1 Pro/Max(2021) – 성능 향상 및 GPU 코어 확장
    - M2(2022), M3(2023), M4(2024)로 성능 향상이 된 칩을 매년 공개

1. Apple M1 GPU

- CPU
  - 8코어: 4개 성능 코어 + 4개 효율 코어
- GPU
  - 최대 8코어 통합
  - 128개 실행 유닛
  - Metal Performance shaders 지원으로 GPU 컴퓨팅 최적화
    - 행렬, 컨볼류션, 정렬, 리덕션과 같은 연산을 제공
- 메모리 시스템
  - 통합 메모리: 최대 16GB LPDDR4X
  - 높은 메모리 대역폭: 68.25GB/s
  - CPU와 GPU간 제로 카피 데이터 공유



2. Apple M1 GPU 성능 비교



Review

General comparison of performance in games, applications, power efficiency, and other metrics

Gaming

Performance in DirectX, OpenCL, and Vulkan games

RTX 3060 Laptop	29
M1 GPU (8-core)	8

Workstation

Perf. in 3D modeling, video editing and rendering apps

RTX 3060 Laptop	35
M1 GPU (8-core)	21

AI/ML

Capabilities for machine learning and AI-related tasks

RTX 3060 Laptop	26
M1 GPU (8-core)	7

Energy Efficiency

Power consumption efficiency in different scenarios

RTX 3060 Laptop	48
M1 GPU (8-core)	90

NanoReview Final Score

Overall video card score

RTX 3060 Laptop	33
M1 GPU (8-core)	25

2. Apple M1 GPU 성능 비교

스페이스 블랙

M4 MAX

16코어 CPU
40코어 GPU
48GB 통합 메모리
1TB SSD 저장 장치[1]

16코어 Neural Engine

41.0cm Liquid Retina XDR 디스플레이[2]

Thunderbolt 5 포트 3개, HDMI 포트, SDXC 카드 슬롯, 헤드폰 잭, MagSafe 3 포트

Touch ID 탑재형 Magic Keyboard

Force Touch 트랙패드

140W USB-C 전원 어댑터

₩5,990,000

ASUS ROG STRIX G16 G614JZR-N4120 14세대 인텔 i9 RTX4080 TGP 175W 16인치 게이밍 노트북

2% ~~3,899,000원~~ **3,789,000원**

무료배송

Review

General comparison of performance in games, applications, power efficiency, and other metrics

Gaming
Performance in DirectX, OpenCL, and Vulkan games

RTX 4080 Laptop	57
M4 Max GPU (40-core)	39

Workstation
Perf. in 3D modeling, video editing and rendering apps

RTX 4080 Laptop	55
M4 Max GPU (40-core)	48

AI/ML
Capabilities for machine learning and AI-related tasks

RTX 4080 Laptop	48
M4 Max GPU (40-core)	46

Energy Efficiency
Power consumption efficiency in different scenarios

RTX 4080 Laptop	46
M4 Max GPU (40-core)	84

NanoReview Final Score
Overall video card score

RTX 4080 Laptop	54
M4 Max GPU (40-core)	50

3. Metal Shading Language

- 2014년에 Swift와 함께 공개된 언어
- Apple 생태계에서 자체적으로 자신들만의 언어를 개발한 것으로 보임.
- Swift는 범용적인 개발 언어로 사용된다면, Metal은 GPU를 활용하기 위해서 개발된 언어?

- Metal의 경우 C++ 기반으로 되어 있기 때문에, 장벽이 너무 높지 않음
- 이번 개발도 Swift와 Metal을 활용해서 구현함.

4. UMA의 장점

- SoC 구조로 CPU와 GPU가 통합 메모리를 사용함.
  - 따라서 메모리 복사 필요가 없음

- CUDA에서도 Unified Memory를 제공하고 있는데, 이는 실제로는 다른 메모리를 사용하지만 소프트웨어 적으로 자동으로 매칭해주는 것이라고 함.

5. 동작 과정

- GPU에서 연산될 커널 함수는 Metal로 작성 -> Swift에서 API로 호출하여 연산

```
1. 초기화 단계
   ├── Metal 디바이스 생성
   ├── 라이브러리 로드
   ├── 커널 함수 가져오기
   └── 컴퓨트 파이프라인 상태 생성

2. 데이터 준비 단계
   ├── 입력 버퍼 생성
   ├── 출력 버퍼 생성
   └── 데이터 복사

3. 실행 단계
   ├── 명령 큐 생성
   ├── 명령 버퍼 생성
   ├── 컴퓨트 인코더 생성
   ├── 파이프라인 상태 설정
   ├── 버퍼 바인딩
   ├── 스레드 그룹 설정
   ├── 디스패치 실행
   └── 인코더 종료

4. 완료 및 결과 처리
   ├── 명령 실행
   ├── 완료 대기
   └── 결과 데이터 추출
```

```swift
// M1 GPU 디바이스 가져오기
guard let device = MTLCreateSystemDefaultDevice() else {
    print("Metal을 지원하는 디바이스를 찾을 수 없습니다.")
    return nil
}
```

```swift
// Metal 셰이더 라이브러리 로드
do {
    let bundle = Bundle.main
    let libraryURL = bundle.url(forResource: "aria_device", withExtension: "metallib")
    if let libraryURL = libraryURL {
        self.library = try device.makeLibrary(URL: libraryURL)
    } else {w
        // 소스 코드에서 직접 컴파일
        let shaderSource = try String(contentsOfFile: "aria_device.metal", encoding: .utf8)
        self.library = try device.makeLibrary(source: shaderSource, options: nil)
```

```swift
// 명령 큐 생성
guard let commandQueue = device.makeCommandQueue() else {
    print("명령 큐를 생성할 수 없습니다.")
    return nil
}
```

5. 동작 과정

- GPU에서 연산될 커널 함수는 Metal로 작성 -> Swift에서 API로 호출하여 연산

```
1. 초기화 단계
   ├── Metal 디바이스 생성
   ├── 라이브러리 로드
   ├── 커널 함수 가져오기
   └── 컴퓨트 파이프라인 상태 생성

2. 데이터 준비 단계
   ├── 입력 버퍼 생성
   ├── 출력 버퍼 생성
   └── 데이터 복사

3. 실행 단계
   ├── 명령 큐 생성
   ├── 명령 버퍼 생성
   ├── 컴퓨트 인코더 생성
   ├── 파이프라인 상태 설정
   ├── 버퍼 바인딩
   ├── 스레드 그룹 설정
   ├── 디스패치 실행
   └── 인코더 종료

4. 완료 및 결과 처리
   ├── 명령 실행
   ├── 완료 대기
   └── 결과 데이터 추출
```

```swift
// GPU 버퍼 생성
guard let bufferA = device.makeBuffer(bytes: ct2, length: 524288 * blocksize, options: []),
    let bufferB = device.makeBuffer(bytes: plain128, length: blocksize, options: []),
    let bufferC = device.makeBuffer(bytes: rk2, length: rk2.count, options: []),
    let sboxBuffer = device.makeBuffer(bytes: X21_S21, length: X21_S21.count * MemoryLayout<UInt32>.size, options: []) else {
    print("GPU 버퍼를 생성할 수 없습니다.")
    return
}
```

```swift
// 커널 함수 가져오기
guard let kernelFunction = library.makeFunction(name: "aria_ctr_shared_kernel") else {
    print("aria_ctr_shared_kernel 커널 함수를 찾을 수 없습니다.")
    return
}
```

```swift
// 컴퓨트 인코더 설정
computeEncoder.setComputePipelineState(pipelineState)
computeEncoder.setBuffer(bufferA, offset: 0, index: 0) // ciphertext
computeEncoder.setBuffer(bufferB, offset: 0, index: 1) // plaintext
computeEncoder.setBuffer(bufferC, offset: 0, index: 2) // roundkeys
computeEncoder.setBuffer(sboxBuffer, offset: 0, index: 3) // sbox
```

5. 동작 과정

- GPU에서 연산될 커널 함수는 Metal로 작성 -> Swift에서 API로 호출하여 연산

```
1. 초기화 단계
   ├── Metal 디바이스 생성
   ├── 라이브러리 로드
   ├── 커널 함수 가져오기
   └── 컴퓨트 파이프라인 상태 생성

2. 데이터 준비 단계
   ├── 입력 버퍼 생성
   ├── 출력 버퍼 생성
   └── 데이터 복사

3. 실행 단계
   ├── 명령 큐 생성
   ├── 명령 버퍼 생성
   ├── 컴퓨트 인코더 생성
   ├── 파이프라인 상태 설정
   ├── 버퍼 바인딩
   ├── 스레드 그룹 설정
   ├── 디스패치 실행
   └── 인코더 종료

4. 완료 및 결과 처리
   ├── 명령 실행
   ├── 완료 대기
   └── 결과 데이터 추출
```

```swift
// 스레드 그룹 크기 계산
let threadGroupSize = MTLSizeMake(512, 1, 1)  // 스레드 그룹당 512개의 스레드
let threadGroups = MTLSizeMake(1024, 1, 1)    // 1024개의 스레드 그룹

// GPU에서 계산 실행
computeEncoder.dispatchThreadgroups(threadGroups, threadsPerThreadgroup: threadGroupSize)
computeEncoder.endEncoding()

// 명령 버퍼 커밋 및 완료 대기
commandBuffer.commit()
commandBuffer.waitUntilCompleted()
```

5. 동작 과정

```swift
// 커널 함수 가져오기
guard let kernelFunction = library.makeFunction(name: "aria_ctr_shared_kernel") else {
    print("aria_ctr_shared_kernel 커널 함수를 찾을 수 없습니다.")
    return
}
```

```
kernel void aria_ctr_shared_kernel(
    device uchar *ciphertext [[buffer(0)]],
    device const uchar *plaintext [[buffer(1)]],
    device const uchar *roundkeys [[buffer(2)]],
    device const uint *sbox_g [[buffer(3)]],
    // 'tS'는 일반적으로 커널 시작 시 Global Memory에서 Threadgroup Memory로 복사됩니다.
    threadgroup uint *tS [[threadgroup(0)]],
    uint tid [[thread_position_in_grid]]
) {
    uint threadIndex = tid;

    threadgroup uint sbox_s[256];
    for (int i = 0; i < 256; i++) {
        sbox_s[i] = sbox_g[i];
    }
```

6. 메모리 종류

CUDA	Metal
Global	Device
Shared	Threadgroup
__Constant__	Constant
Local memory	thread

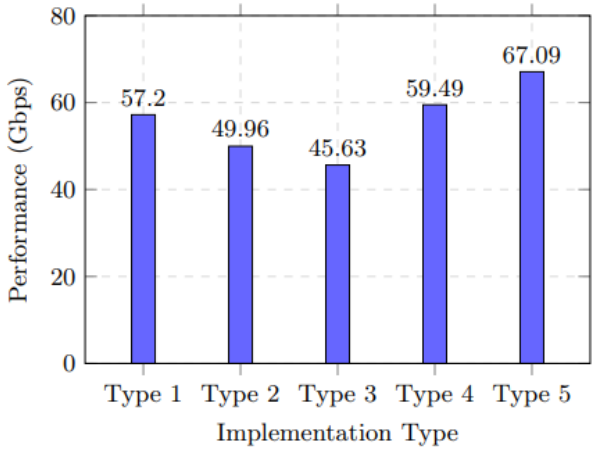
```
uint ptInit_2[4];
thread uchar *pt8 = (thread uchar *)ptInit_2;
```

7. 성능 결과

Type	Kernel Name	S-box	Round Keys
1	device_kernel	device	device
2	threadgroup_kernel	threadgroup	device
3	constant_kernel	constant	device
4	threadgroup_rk_constant_kernel	threadgroup	constant
5	threadgroup_rk_threadgroup_kernel	threadgroup	threadgroup

Table 3. Performance comparison of different memory configurations for ARIA-CTR

Type	S-box	Round Keys	Time (s)	Gbps
Type 1	Device	Device	76.89	57.20
Type 2	Threadgroup	Device	88.03	49.96
Type 3	Constant	Device	96.38	45.63
Type 4	Threadgroup	Threadgroup	73.93	59.49
Type 5	**Threadgroup**	**Constant**	**65.55**	**67.09**



감 사 합 니 다