

Microprocessor Application

Project

Team Plane Optimizers

박세훈

천승현

김희진

박효정

프로젝트

- ARM Assembly로 RGBA 이미지 데이터를 처리
- P2를 통해 R/G/B plane 구조를 만들고, P3에서 word load 및 loop 최적화 적용 후 성능 비교 분석
- P1, P2, P3의 동일 output 검증
- 2차 발표 후 3차 발표 개선 사항 적용

활용 도구

Keil MDK_Lite
μVision V5.26.2.0

ARM Assembly 코드 작성,
빌드 및 디버그 수행

Memory
Window

input/output 주소의 hex 값 확인
및 결과 검증

t1 측정값

수행 시간 비교를 위한 기준값

Instruction
분석

loop body 기준 동적 명령어 수와
분기 횟수 계산

프로젝트 구현함수

- F1 : Brightness
- F2 : Binary Image
- F3 : Blur Filter

프로젝트 수행 방식

- P1 : Baseline
 - 원본 RGBA 구조에서 F1/F2/F3 직접 구현
→ $F1 + F2 + F3$
- P2 : Memory Relocation
 - A 제거 후 R/G/B plane으로 재배치
→ $\text{Memory relocation} + F1 + F2 + F3$
- P3 : 최적화
 - R/G/B plane 구조에서 F1, F2, F3 명령어 최적화
→ $\text{Memory relocation} + F1(\text{최적화}) + F2(\text{최적화}) + F3(\text{최적화})$

3차 발표 보완 사항

개선 사항	기존	3차 보완
결과 검증	첫 pixel / 첫 byte 중심	전체 output byte 비교 hex → png
정량 검증	일부 memory 값 확인	Mismatch, MSE, PSNR 지표 활용
구조 개선	각 함수별 relocation 반복	relocation 1회 후 F1/F2/F3 재사용
성능 비교	개별 비교	통합 비교

추가 최적화: 독립 실행 구조 → 통합 실행 구조

핵심 변화

구분	기존 구조	통합 구조
실행 방식	F1/F2/F3 각각 독립 실행	F1 → F2 → F3 연속 실행
Relocation	함수마다 반복 수행	한 번 생성 후 재사용
R/G/B Plane	매번 새로 생성	최초 1회만 생성

기대 효과

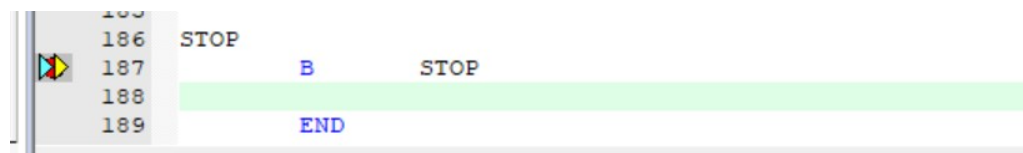
항목	기대 효과
수행 시간	통합구조를 통한 전체 처리 시간 감소
명령어수	중복 relocation 제거를 통한 수행량 감소
Relocation 횟수	3회 → 1회로 감소하는 구조적 최적화

PSNR, 이미지 추출 방법

1. Keil에서 image_rgba.hex를 메모리에 Load

```
MAP 0x40000000, 0x40130000 READ WRITE  
MAP 0x41000000, 0x41060000 READ WRITE  
LOAD C:\imgdump\image_rgba.hex
```

2. P2 / P3 통합 코드를 STOP까지 실행



3. Output memory를 hex 파일로 저장

```
SAVE C:\imgdump\P3_F1.hex 0x41049000, 0x4105007F  
SAVE C:\imgdump\P3_F2.hex 0x41051000, 0x410514AF  
SAVE C:\imgdump\P3_F3.hex 0x41052000, 0x41053C1F
```

4. Python으로 결과 비교

- hex → PNG 복원
- byte mismatch 확인
- MSE(평균 제곱 오차) / PSNR[$10\log_{10}(255^2/\text{MSE})$] 계산
- Difference image 생성



```
C:\Users\COM2102104>python C:\wimgdump\analyze_p3_p4.py
```

```
[F1] P3 vs P4  
total bytes : 28800  
mismatch bytes : 0  
MSE : 0.0  
PSNR : inf  
first mismatch : -
```

```
[F2] P3 vs P4  
total bytes : 1200  
mismatch bytes : 0  
MSE : 0.0  
PSNR : inf  
first mismatch : -
```

```
[F3] P3 vs P4  
total bytes : 7200  
mismatch bytes : 0  
MSE : 0.0  
PSNR : inf  
first mismatch : -
```

Output 차이에 따른 Difference Image 변화

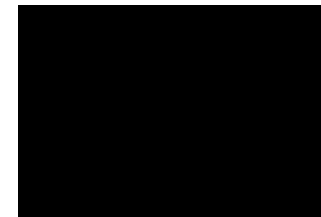
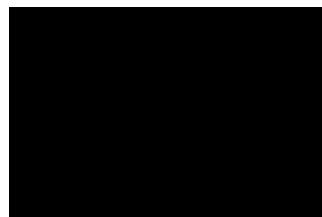
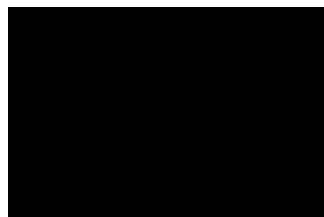
상태

F1 Output

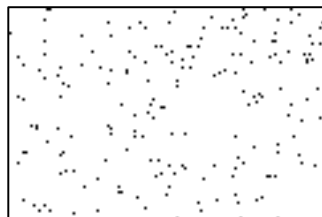
F2 Output

F3 Output

정상 일치 시



값이 다를 시



Summary

- Difference Image는 두 output의 pixel 차이를 시각화한 결과
- 정상 일치 시: Mismatch = 0, MSE = 0, PSNR = ∞
→ Difference Image가 검정색으로 나타남
- 불일치 시: Mismatch > 0, MSE > 0, PSNR은 유한한 값
→ Difference Image에 흰색 영역 또는 점이 나타남
- 실제 정확성 검증은 각 F1, F2, F3끼리 동일 함수 output 간 비교로 수행

상태	Mismatch	MSE	PSNR	Difference Image
정상 일치	0	0	∞	검정색
불일치	0보다 큼 ex) 1192	0보다 큼 ex) 32390.58	유한값 ex) 3.0266	흰색 영역 /점 발생

P1

기본 이미지 처리 함수 구현

원본 RGBA 데이터 기반 F1~F3 구현 및 기본 동작 검증

R	G	B	A
R	G	B	A

P1 목표와 실행 기준

핵심 목표

- 원본 RGBA image data 기반 F1~F3 기본 함수 구현
- Load → Process → Store 구조로 각 output 저장 검증
- 이후 P2/P3 결과 비교를 위한 baseline output 생성

P1 함수별 실행 기준

함수	처리 기능	입력 구조	Output 주소
P1-F1	Brightness	32-bit RGBA	0x41000000
P1-F2	Binary image	32-bit RGBA	0x41020000
P1-F3	Blur filter	4 RGBA pixels	0x41030000

P1은 Baseline 단계로,
원본 RGBA 구조에서 R/G/B 값을 직접 읽어 처리하고
Alpha 값은 연산에서 제외하였다.

P1-F1 Brightness : 원본 RGBA에서 밝기 조절

연산방식

32-bit RGBA → 24-bit RGB

R/G/B + 0x40

0xFF 초과 시 0xFF로 saturation

Alpha는 연산 제외

연산식

$R' = \min(R + 64, 255)$

$G' = \min(G + 64, 255)$

$B' = \min(B + 64, 255)$

Output 검출

Component	입력 값	연산	출력 값
R	0x78	$0x78 + 0x40$	0xB8
G	0x5E	$0x5E + 0x40$	0x9E
B	0x5C	$0x5C + 0x40$	0x9C
A	0xBD	연산 제외	-

Memory 1

Address: 0x41000000

```
0x41000000: B8 9E 9C B7 B3 63 D2 FF 5B 61
0x4100002A: FF FF FF FF FF FF F5 BD 57 73
0x41000054: FF 72 FF EE F1 F6 F4 FF 58 FF
0x4100007E: FF CB AD 7E BA FF 8D 53 FF 46
0x410000A8: E3 A9 DA FF 99 BF 6B 7F AB C6
```

F1 Baseline output 검출

P1-F2 Binary: Grayscale → 1-bit Result → Byte Packing

연산방식

RGBA → Grayscale → Binary

$\text{Gray} = (\max(R,G,B) + \min(R,G,B)) / 2$

Gray > 127이면 1, 아니면 0

8 pixel 결과를 1 byte로 packing

첫 4 Pixel 계산 예시

Pixel	max	min	Gray	Binary
0	0x78	0x5C	0x6A	0
1	0x77	0x23	0x4D	0
2	0xEC	0x1B	0x83	1
3	0xB2	0x08	0x5D	0

첫 4 pixel binary = 0 0 1 0

P1-F2는 각 pixel의 RGB 값으로 grayscale을 계산한 뒤, 8개 binary 결과를 하나의 byte로 packing한다.

P1-F2 Output 검증 : 첫 Packed Byte 확인

첫 8 pixel binary

Pixel 0~7

0 0 1 0 0 1 1 1

bit7 → bit0 packing

0010 0111 = 0x27

Memory 1

Address: 0x41020000

0x41020000: 27 A3 7D C5 3E

0x41020025: FF 9E ED 4E CE

0x4102004A: 30 04 C8 CF 97

구분	내용
Output address	0x41020000
Output format	1-bit binary, 8 pixel → 1 byte
첫 output byte	0x27

F2 Baseline output 검출

P1-F3 Blur : 4 Pixel 평균 필터 처리

연산방식

4개의 32-bit RGBA pixel 입력

각 pixel에서 R/G/B 값만 사용

R, G, B 성분별 평균 계산

소수점 발생 시 버림 처리

1개의 24-bit RGB blur output 생성

연산식

Blur R = $\text{avg}(R0, R1, R2, R3)$

Blur G = $\text{avg}(G0, G1, G2, G3)$

Blur B = $\text{avg}(B0, B1, B2, B3)$

첫 Blur output 계산

Component	입력 값	평균 계산	출력
R	78, 77, 92, 21	$418 / 4 = 104$	0x68
G	5E, 73, EC, 08	$453 / 4 = 113$	0x71
B	5C, 23, 1B, B2	$332 / 4 = 83$	0x53

Memory 1

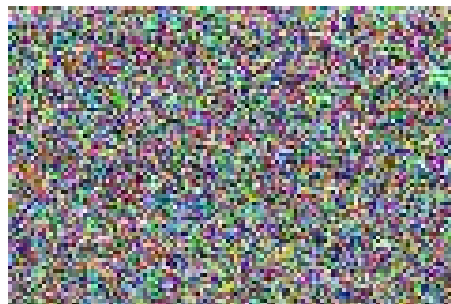
Address: 0x41030000

```
0x41030000: 68 71 53 B4 8D A7
0x41030020: AC 99 B7 83 C8 8A
0x4103004A: 72 86 64 80 56 6E
```

F3 Baseline output 검출

P1 수행 결과: 원본 이미지 변화 확인

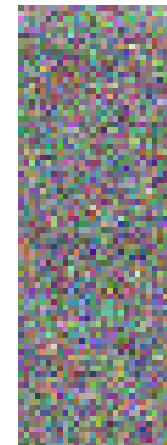
[Original]



[F1 Brightness]
[F2 Binary]



[F3 Blur]



P1 output은 이후 P2/P3 결과 검증을 위한 baseline으로 사용
→ 전체 byte 비교, MSE, PSNR 기준으로 동일성 확인

P2

Memory Relocation

RGBA에서 Alpha를 제외하고 R/G/B를 각각 연속된 plane으로 재배치



P2 Memory Relocation이 필요한 이유

단순히 Alpha만 제거하면 RGB가 3-byte 단위로 반복되어 word 단위 접근에서 성분이 섞임.

Original RGBA



A만 제거한 RGB



4-byte word 접근 시



문제 핵심

- F1~F3는 RGB만 사용하고
Alpha는 연산에 사용하지 않음
- A만 제거한 RGBRGB... 구조는
3-byte stride라 4-byte 정렬이 깨짐
- LDR 기반 4-byte 접근 시
[R G B R], [G B R G]처럼 섞임

결론

A 제거보다 더 중요한 것은
같은 색상 성분을 연속 배치하는 것

P2 : Memory Relocation 기반 이미지 처리

원본 RGBA Memory

Input base : 0x40000036



Memory Relocation
A 제거 + 색상 성분별 연속 배치

A는 F1~F3 연산에 사용되지 않음

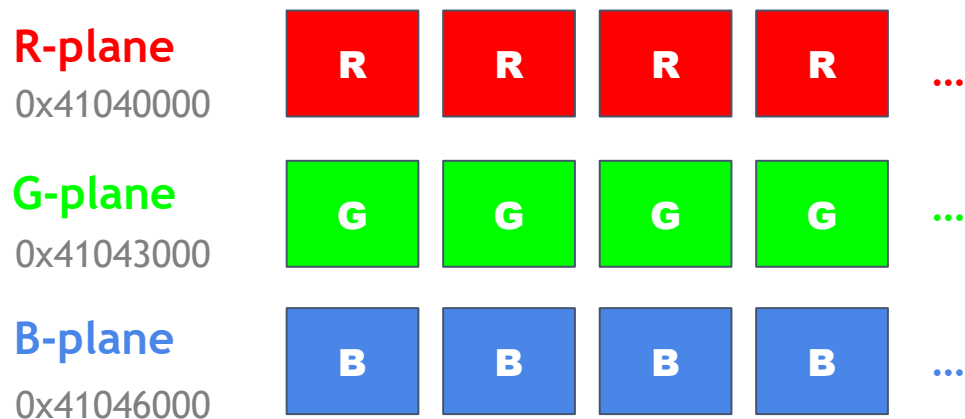
P2 Plane-based Processing

재배치된 R/G/B plane을 입력으로 사용해 P1과 동일한 연산을 수행

F1	Brightness : R/G/B 각각 +64, 255 초과 시 saturation
F2	Binary : max/min grayscale -> 8 pixels = 1 byte packing
F3	Blur : 같은 색상 성분 4개 평균 -> 24 bit RGB

R/G/B Plane 구조

각 Plane = 9,600 byte, plane 간격 = 0x3000



결과 검증 : P1 output = P2 output

함수	Total Byte	Mismatch	MSE	PSNR	First mismatch
F1	28800	0	0	inf	-
F2	1200	0	0	inf	-
F3	7200	0	0	inf	-

P3

P2 기반 이미지 처리시 메모리 최적화

최적화 이후 F1~F3 구현 및 P2 결과와 동일성 검증

P3 Memory 주소

공통 메모리 주소:

입력 RGBA 시작 주소: 0x40000036

R-plane 시작 주소: 0x41040000

G-plane 시작 주소: 0x41043000

B-plane 시작 주소: 0x41046000

최적화

P2:

LDRB r7, [r0], #1

ADD r0, r0, #1

P3:

LDRB r7, [r0], #2

수행 후 메모리 주소:

P3-F1 결과 주소: 0x41049000 (B8 9E 9C B7 B3 63 D2 FF ...)

P3-F2 결과 주소: 0x41051000 (27 A3 7D C5 3E F1 A6 DA ...)

P3-F3 결과 주소: 0x41052000 (68 71 53 B4 8D A7 91 64 ...)

Memory 1	
Address:	0x41049000
0x41049000:	B8 9E 9C B7 B3 63 D2 FF 5B 61 48 F2 B7 CF 98
0x41049033:	73 FF FF FF F6 AD FF 75 C6 C9 FF FF FF 8F FF
0x41049066:	F4 FC 87 FF FF AC ED 8B FF 71 FF FB FF 95 FF

Memory 1	
Address:	0x41051000
0x41051000:	27 A3 7D C5 3E F1 A6 DA 65 67 27 EF FF FE BD
0x4105102D:	FF 5D CD 77 DF BB 70 1D FF FC 9E AF CF E7 E4
0x4105105A:	64 FF 1B 7F B0 21 F7 6F 64 D4 04 87 26 D6 5F

Memory 1	
Address:	0x41052000
0x41052000:	68 71 53 B4 8D A7 91 64 70 A5 8B B9 A1
0x41052030:	91 62 50 94 49 81 95 98 AA 8A 98 7E 71
0x41052060:	A1 A9 68 82 B6 8E 8C 99 68 C9 87 BD 50

P3-F1 Brightness 최적화

순서	코드	역할
① 반복 횟수 감소	LDR r4, =2400	9600개 pixel을 4개씩 처리하여 loop를 9600회 → 2400회로 감소
② Word Load 적용	LDR r5,[r0],#4LDR r6,[r1],#4LDR r7,[r2],#4	R,G,B 각각 4개 pixel을 한 번에 읽어옴
③ Shift로 Byte 분리	MOV r8,r5,LSR #8AND r8,r8,#0xFF	Word 안에 들어있는 4개의 pixel 값을 다시 분리
④ 상수 레지스터화	MOV r11,#64	밝기 증가값 64를 레지스터에 저장 후 재사용
⑤ Saturation 유지	CMP r8,#255MOVHI r8,#255	255 초과 방지
⑥ 결과 저장	STRB r8,[r3],#1	밝기 조절된 값 저장

P3-F1 Brightness 핵심 최적화

1. Word Load

P2	P3
LDRB r5,[r0],#1	LDR r5,[r0],#4
LDRB r6,[r1],#1	LDR r6,[r1],#4
LDRB r7,[r2],#1	LDR r7,[r2],#4

2. Loop Unrolling

P2	P3
LDR r4, =9600	LDR r4, =2400
F1_LOOP	F1_LOOP
...	; Pixel0
SUBS r4, r4, #1	...
BNE F1_LOOP	; Pixel3
	...
	SUBS r4, r4, #1
	BNE F1_LOOP

P2(memory relocation + baseline) Output



P3(최적화) Output



Difference Image



결과 : Mismatch = 0으로 동일함

P3-F2 Binary 최적화

순서	코드	역할
① Relocation Alpha skip 개선	LDRB r7,[r0],#2	B를 읽으면서 Alpha까지 같이 건너뛰어 ADD r0,r0,#1 제거
② 내부 loop 제거	; Pixel 0 ~ ; Pixel 7	P3의 INNER_LOOP 8회 반복을 없애고 8개 pixel 코드를 직접 펼침
③ Word Load 적용	LDR r11,[r0],#4 LDR r12,[r1],#4 LDR r14,[r2],#4	R,G,B 각각 4개 pixel을 한 번에 읽음
④ Shift로 Byte 분리	MOV r5,r11,LSR #8 AND r5,r5,#0xFF	Word 안의 pixel 값을 1byte씩 분리
⑤ LSR #1 제거	ADD r8,r8,r9 CMP r8,#255	$(\text{max} + \text{min}) / 2 > 127$ 을 $\text{max} + \text{min} > 255$ 로 바꿔 나누기 제거
⑥ Bit Packing 방식 변경	ORRHI r10,r10,#0x80 ORRHI r10,r10,#0x40	각 pixel 위치에 맞는 bit를 직접 설정

1. INNER_LOOP 제거

P2	P3
MOV r11, #8	; Pixel 0
INNER_LOOP	...
...	; Pixel 1
SUBS r11, r11, #1	...
BNE INNER_LOOP	; Pixel 7
	...

2. Word Load

P2	P3
LDRB r5,[r0],#1	LDR r11,[r0],#4
LDRB r6,[r1],#1	LDR r12,[r1],#4
LDRB r7,[r2],#1	LDR r14,[r2],#4

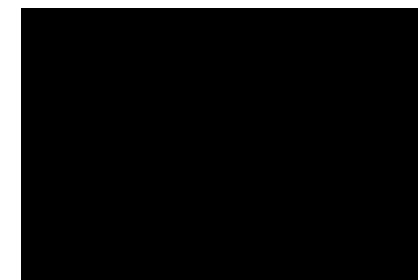
P2(memory relocation + baseline) Output



P3(최적화) Output



Difference Image



결과 : Mismatch = 0으로 동일함

P3-F3 Blur 최적화

순서	P3 코드	최적화 역할
① Alpha skip 명령어 제거	LDRB r7,[r0],#2	B를 읽으면서 Alpha까지 같이 건너뛰어 ADD 제거
② Load 구조 변경	LDR r5,[r0],#4	4개 pixel을 1byte씩 읽던 구조를 4byte 한 번에 읽는 구조로 변경
③ Byte 분리 방식 추가	MOV r7,r5,LSR #8 AND r7,r7,#0xFF	Word Load로 읽은 4byte를 각각의 pixel 값으로 분리
④ 평균 계산 레지스터 구조 변경	MOV r6,r6,LSR #2	Word에서 분리한 값을 합산한 뒤 평균 계산하도록 레지스터 사용 변경

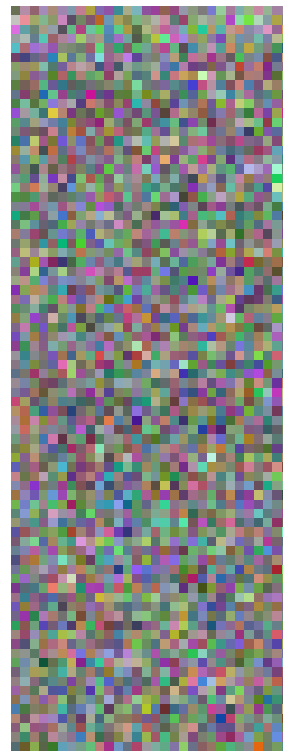
1. Load 구조 변경

P2
LDRB r5,[r0],#1
LDRB r6,[r0],#1
LDRB r6,[r0],#1
LDRB r6,[r0],#1

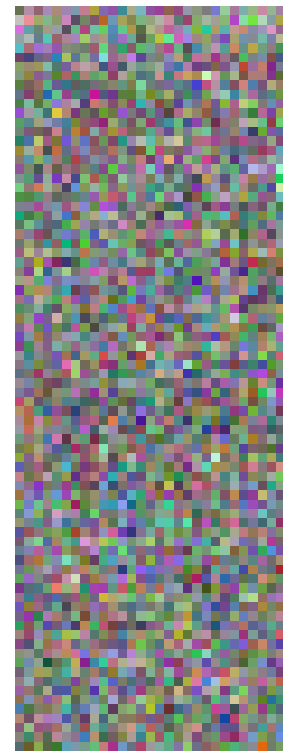
P3
LDR r5,[r0],#4

함수	Total Byte	Mismatch	MSE	PSNR	First mismatch
F1	28800	0	0	inf	-
F2	1200	0	0	inf	-
F3	7200	0	0	inf	-

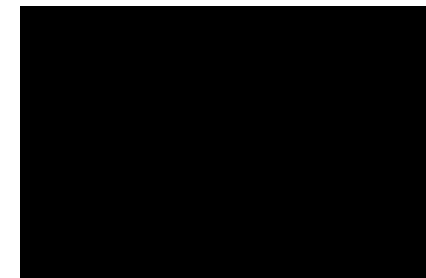
P2 Output



P3 Output



Difference Image



결과 : Mismatch = 0으로 동일함

최적화 전/후 성능 비교 분석

ARM 명령어 기반 이미지 처리 최적화
P1 · P2 · P3 수행 시간 및 명령어 관점 비교

성능 비교 :순수 이미지 처리부 수행 시간 비교



연산방식

P1: 전체 수행 시간 사용

P2: 전체 수행 시간 - 메모리 재배치 시간

P3: 전체 수행 시간 - 메모리 재배치 시간

Memory relocation 시간

P2 방식:

LDRB r7, [r0], #1

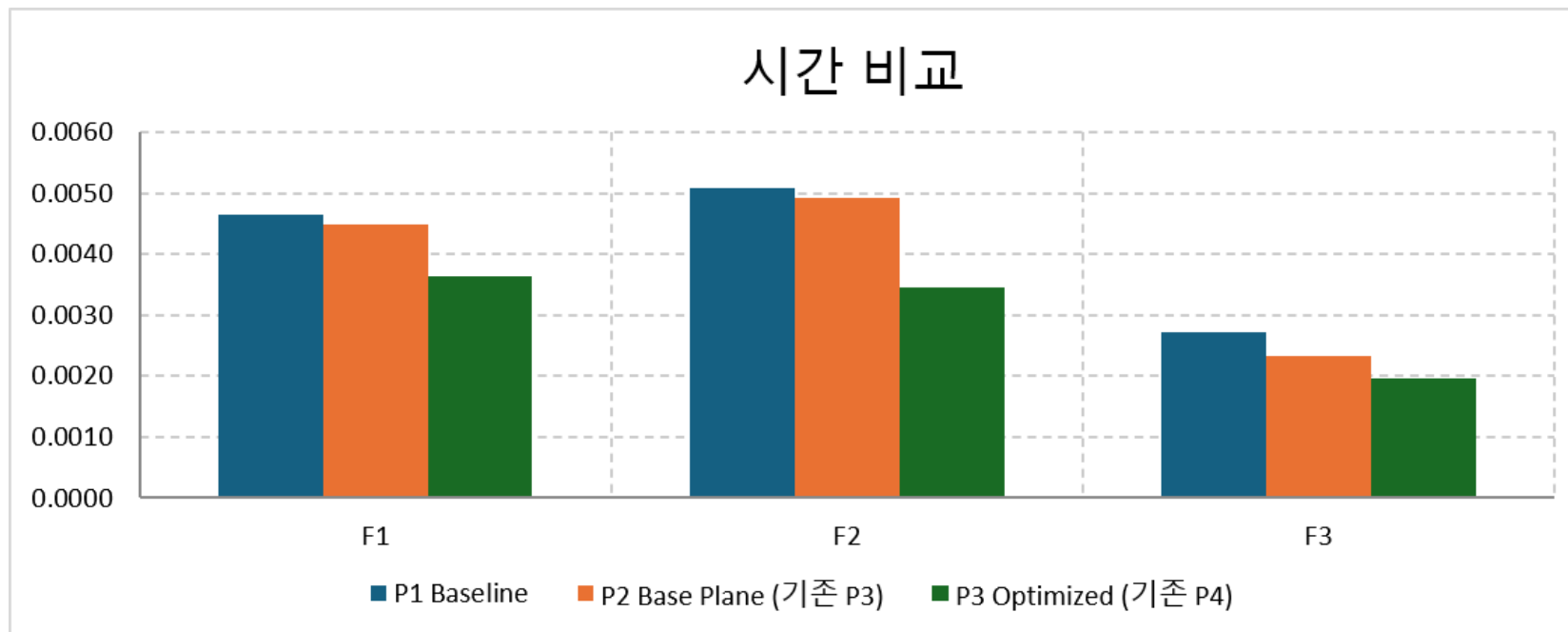
ADD r0, r0, #1

P3 방식:

LDRB r7, [r0], #2

구분	메모리 재배치 시간
P2	0.00320015 sec
P3	0.00304015 sec

성능 비교 : 개별 F1,F2,F3 순수 이미지 처리부 수행 시간 비교



Function	P1 Baseline	P2 Base Plane	P3 Optimized	P1→P2 개선율	P2→P3 개선율	P1→P3 개선율
F1	0.00464005	0.00448018	0.00364020	3.45%	18.75%	21.55%
F2	0.00508005	0.00492018	0.00346018	3.15%	29.67%	31.89%
F3	0.00272005	0.00232018	0.00196018	14.70%	15.52%	27.94%

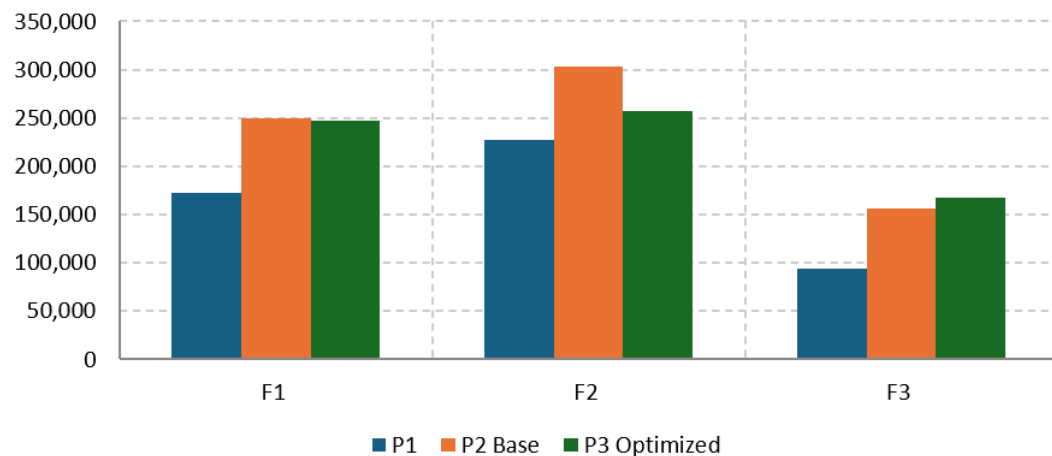
명령어 수 비교: 기존 개별 실행 기준

Relocation 포함 전체 수행량						
합수	P1 Inst.	P2 Base Inst.	P3 Opt. Inst.	P1 Branch	P2 Branch	P3 Branch
F1	172,800	249,600	247,200	9,600	19,200	12,000
F2	226,800	303,600	256,800	10,800	20,400	10,800
F3	93,600	156,000	168,000	2,400	12,000	12,000
Relocation 제외 이미지 처리부						
합수	P1 Inst.	P2 Base Inst.	P3 Opt. Inst.	P1 Branch	P2 Branch	P3 Branch
F1	172,800	163,200	170,400	9,600	9,600	2,400
F2	226,800	217,200	180,000	10,800	10,800	1,200
F3	93,600	69,600	91,200	2,400	2,400	2,400
Load/Store 보충 분석						
합수	P1 Load	P2 Base Load	P3 Opt. Load	감소율	P1/P2/P3 Store	의미
F1	28,800	28,800	7,200	75%	28,800 / 28,800 / 28,800	output store는 동일
F2	28,800	28,800	7,200	75%	1,200 / 1,200 / 1,200	input load만 감소
F3	28,800	28,800	7,200	75%	7,200 / 7,200 / 7,200	word load 효과

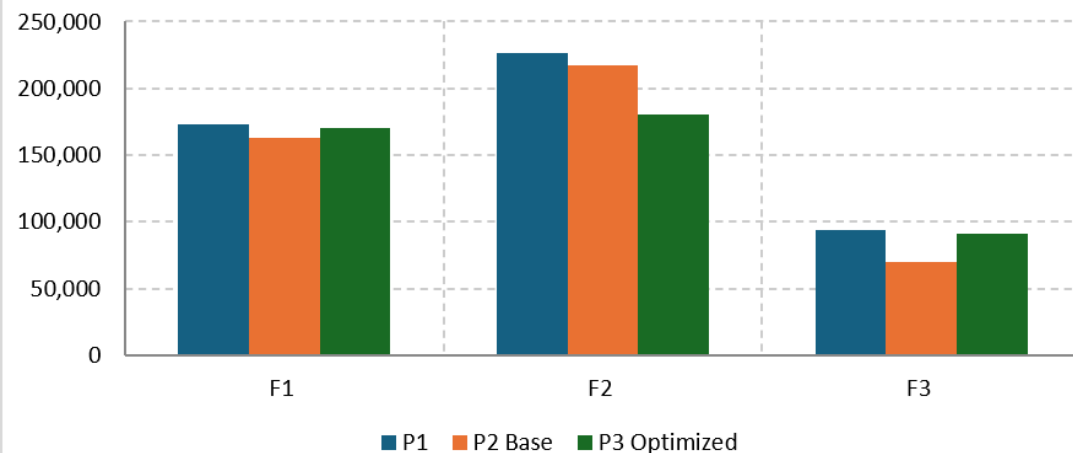
핵심 요약

- Relocation 포함 시 전체 명령어 수 증가
- 이미지 처리부만 보면 P3 최적화 효과 확인
- Input load 75% 감소, Output store 동일
- 한계: F1/F2/F3마다 Relocation 반복

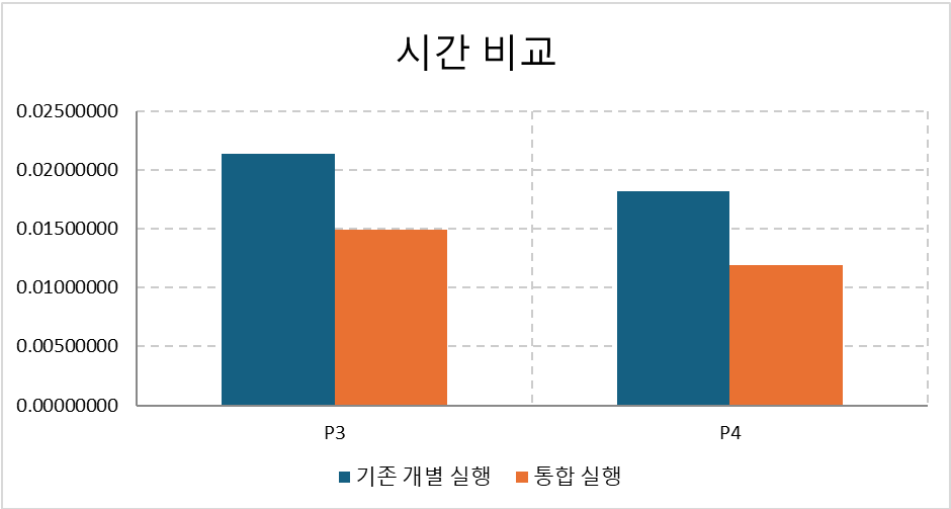
전체 수행량 기준



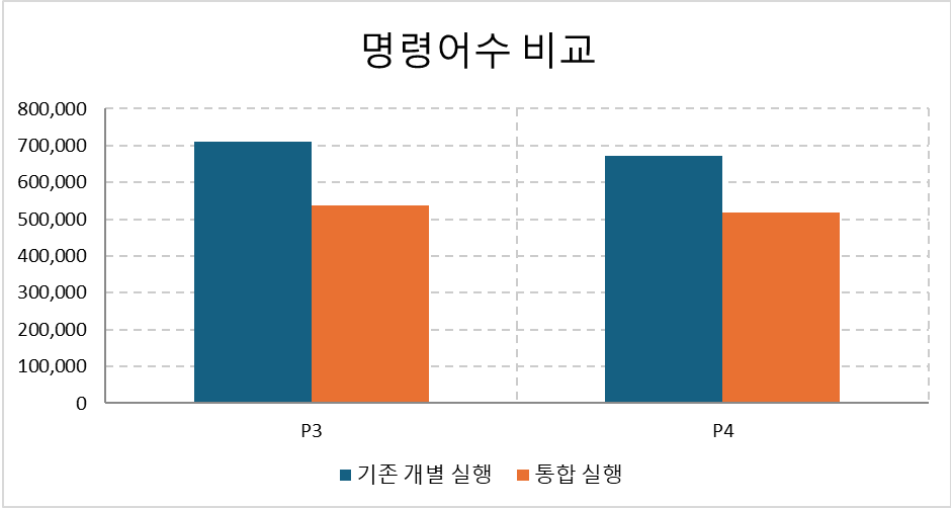
순수 이미지 처리부 기준



구조 비교: 기존 개별 실행 vs Integrated Flow



Structure	기존 개별 실행	통합 실행
P3	0.02132099	0.01492070
P4	0.01818101	0.01194072



Structure	기존 개별 실행	통합 실행
P3	709,200	536,400
P4	672,000	518,400

상대 개선율

비교	Time 개선율	Inst. 개선율
P3	30.02%	24.37%
P4	34.32%	22.86%

Relocation 회수 비용

기존 개별 실행	통합 실행
Relocation + F1	Relocation 1회 + F1 + F2 + F3
Relocation + F2	
Relocation + F3	
>> 총 3회	>> 총 1회

최종 결론 : 정확성 검증과 구조적 최적화 효과

F1 밝기 조절

- 4 pixels/loop 구조 적용
- Word load로 input load 감소
- Saturation 처리는 유지 필요

⇒ branch와 load는 감소했지만,
saturation 비용으로 개선폭은 제한적

F2 Binary 생성

- INNER loop 제거
- OUTER+INNER → OUTER only
- 조건식 단순화

$(\max + \min) / 2 > 127$

→ $\max + \min > 255$

⇒ branch 감소 효과가 가장 큼

※ 한계:

byte-level 검증은 완료했지만,
별도 bit-level 분석은 향후 보완

F3 Blur 필터

- 반복 횟수는 동일
2,400회 유지
- Word load로 input load 감소
- byte 분리 연산 추가

⇒ load 감소 효과는 있으나,
byte 분리 비용까지 고려 필요

핵심: 전체 output은 mismatch 0, MSE 0, PSNR ∞로 동일성을 검증했고,
Integrated Flow를 통해 반복되던 Memory Relocation을 1회로 줄여 전체 수행 구조를 개선하였다.

감사합니다