시공간 병렬화를 통한
비디오 업 스케일링 성능 분석

빅데이터 처리 2 조
조원: 오태영, 장원덕, 최태빈, 박혜성

서론

비디오 업스케일링은 낮은 해상도의 비디오를 더 높은 해상도로 변환하여 더 큰 화면에서 선 명하게 재생할 수 있도록 하는 기술로, 영화, 스트리밍 서비스, 게임 등 다양한 멀티미디어 응 용 분야에서 중요한 역할을 한다. 최근 산업에서는 주로 딥러닝과 AI 기반의 첨단 기술을 활 용하여 높은 품질의 업스케일링 결과를 구현하고 있다. 본 프로젝트는 이러한 첨단 기술 대신, 빅데이터 처리 수업에서 학습한 멀티프로세싱과 병렬 처리 기법의 성능을 분석하는 데 중점을 둔다. 이를 위해 상대적으로 단순한 선형 보간법을 활용하여 비디오 업스케일링을 구현하고, 병렬 처리 기법이 성능에 미치는 영향을 평가한다. 특히, 비디오 업스케일링 과정에서 두 가지 병렬 처리 방식을 비교한다. 첫 번째는 공간 분할 방식으로, 단일 프레임을 여러 조각으로 분할하여 병렬로 처리하는 방법이다. 두 번째는 시간 분할 방식으로, 비디오를 여러 프레임 단위로 나눠 각 프레임을 독립적으로 병렬 처리하는 방법이다. 이러한 접근 방식은 병렬 처리 자원의 활용도를 높이고 처리 시간을 단축할 수 있다. 본 프로젝트의 목표는 선형 보간법 기반의 업스케일링에 공간 분할 및 분할 방식을 적용한 후, 각 방식의 성능을 정량적으로 비교하는 것이다. 이를 통해 단순 알고리즘에서도 멀티프 로세싱을 효과적으로 활용할 수 있는 가능성을 탐구하고, 병렬 처리 기법이 비디오 업스케일 링 성능에 미치는 영향을 확인하고자 한다.

TASK

해상도: 480 x 270, 영상 길이: 10 초, FPS: 30 의 샘플 영상을 2 배 업스케일링 하여

960 x 540 해상도의 영상으로 변환한다.

480 x 270

960 x 540

(차이를 극명하게 보이기 위해 확대한 이미지 입니다.)

구현

FrameUpSampler

한 프레임에 대해서 업샘플링을 수행하는 FrameUpSampler 클래스를 공간 병렬화 유뮤에 따라 두가지 타입으로 나누어 구현하였다.

1. 병렬화를 적용하지 않은 선형보간 Frame Up Sampler

```
    # (중략)
    src_h, src_w = source_image.shape[:2]
    dest_h, dest_w = int(src_h * scale_factor), int(src_w * scale_factor)

    # 보간 작업 수행
    slope_h = src_h / dest_h
    intercept_h = (slope_h - 1) / 2
    slope_w = src_w / dest_w
    intercept_w = (slope_w - 1) / 2

    dest_image = np.zeros((dest_h, dest_w, 3))  # 출력 이미지 초기화
    source_image = np.pad(source_image, pad_width=((1, 1), (1, 1), (0, 0)), mode='edge')  # 패딩 추가

    for y_dest in range(dest_h):
        for x_dest in range(dest_w):
            x_src = slope_w * x_dest + intercept_w
            y_src = slope_h * y_dest + intercept_h

            x0, y0 = int(x_src), int(y_src)
            x1, y1 = x0 + 1, y0 + 1

            wx = x_src - x0
            wy = y_src - y0

            v1 = source_image[y0, x0]
            v2 = source_image[y0, x1]
            v3 = source_image[y1, x0]
            v4 = source_image[y1, x1]

            dest_image[y_dest, x_dest] = (
                (1 - wx) * (1 - wy) * v1 +
                wx * (1 - wy) * v2 +
                (1 - wx) * wy * v3 +
                wx * wy * v4
            )

    return dest_image.astype(np.uint8)
```

인접 픽셀의 값을 가중합하여 새로운 픽셀의 값을 결정하는 기본적인 선형 보간 방법을 사용하였다. 병렬화 없이 이중 반복문을 사용하였기 때문에 많은 실행시간을 필요로 할 것이라고 예측할 수 있다.

2. SIMD 를 이용해 공간 병렬화를 구현한 선형보간 Frame Up Sampler

위의 보간 방식은 각 픽셀에 대한 연산이 개별적으로 수행되었기 때문에 알고리즘의 시간 복잡도가 높았고, 메모리 접근 효율이 낮아 CPU 캐시 활용도가 떨어지는 문제가 있을 것이라고 예상된다. 이를 해결하기 위해 SIMD 기법을 활용하여 벡터화를 도입하였고, 공간 병렬화를 통해 전체 연산을 병렬적으로 수행하도록 하였다.

벡터화 도입

SIMD 기법을 적용한 가장 큰 변화는 이중 루프를 제거하고, 모든 계산을 배열 기반으로 병렬 처리한 점이다. 목적지 이미지의 모든 픽셀 좌표를 np.mgrid 를 사용해 한 번에 계산하고, 소스 좌표 (x_src, y_src)와 보간 가중치 (wx, wy)를 벡터화된 방식으로 처리하였다. 벡터화를 통해 모든 픽셀에 대해 독립적인 계산을 동시에 수행할 수 있다.

```
# 목적지 좌표 계산
y_dest, x_dest = np.mgrid[0:dest_h, 0:dest_w]

# 소스 좌표 계산
x_src = slope_w * x_dest + intercept_w
y_src = slope_h * y_dest + intercept_h
```

보간 가중치 및 소스 픽셀 값 벡터화

보간에 필요한 네 개의 소스 픽셀 값 (v1, v2, v3, v4)을 배열 연산을 활용하여 동시에 가져오고 처리한다. wx 와 wy 는 각 소스 좌표와 목적지 좌표의 차이를 이용해 계산된다. 이를 통해 각 픽셀 연산을 독립적으로 수행하던 기존 방식과 달리, 전체 이미지를 하나의 연산 단위로 묶어 병렬적으로 처리할 수 있게 되었다.

```
# 소스 픽셀 좌표와 가중치 계산
x0 = np.floor(x_src).astype(int)
y0 = np.floor(y_src).astype(int)
x1 = x0 + 1
y1 = y0 + 1

wx = (x_src - x0)[..., None]
wy = (y_src - y0)[..., None]

# 소스 픽셀 값 가져오기
v1 = source_image[y0, x0]
v2 = source_image[y0, x1]
v3 = source_image[y1, x0]
v4 = source_image[y1, x1]
```

이러한 방식으로 벡터화된 연산은 병렬 처리의 장점인 CPU 캐시 효율성을 높이고, 메모리 접근 패턴을 최적화하여 성능을 향상시킨다. 이 방식은 기존 방식에서 사용된 이중 루프에 비해 훨씬 더 효율적이며, 멀티 코어 환경에서도 성능 개선 효과를 기대할 수 있다.

Video Up Sampler

위에서 정의한 FrameUpSampler 를 이용하여 비디오에 대한 업샘플링을 수행하는 VideoUpSampler 를 구현하였다. 시간 병렬화 유무 또는 시간 병렬화 방식에 따라 세가지 타입으로 나누어 구현하였다.

1. 병렬화를 적용하지 않은 Video Up Sampler

```
    cap.set(cv2.CAP_PROP_POS_FRAMES, 0)
    while cap.isOpened():
        ret, frame = cap.read()
        if not ret:
            break

        enhanced_frame = self.frame_up_sampler.up_sample_frame(frame, scale_factor)
        out.write(enhanced_frame)
```

단순히 반복문으로 모든 프레임을 읽어와서 앞프레임부터 순차적으로 업샘플링을 수행한다.

2. 정적 시간 병렬화를 적용한 Video Up Sampler

비디오 업샘플링 작업을 효율적으로 수행하기 위해 멀티 프로세싱을 활용한 시간적 병렬화 설계를 적용하였다. 이 클래스는 비디오를 프레임 단위로 나누어 각 프레임을 병렬로 처리하며, 이를 통해 다수의 CPU 코어를 활용하여 처리 속도를 극대화한다. 프레임 간 처리가 독립적이라는점을 기반으로 작업을 분산하고, 데이터 공유를 최적화하기 위해 공유 메모리를 사용한다.

시간적 병렬화

비디오 업샘플링은 각 프레임을 독립적으로 처리할 수 있기 때문에, 전체 비디오를 프레임 단위로 나누어 여러 개의 chunk 로 분할하고, 각 chunk 를 별도의 프로세스에서 처리하도록 설계되었다. 이를 통해 비디오 업샘플링 작업을 멀티 프로세싱으로 병렬 처리할 수 있다. 각 프로세스는 자신에게 할당된 범위의 프레임을 처리하며, 작업 범위는 전체 프레임 수를 프로세스 수로 나누어 균등하게 분배된다.

```
# 프레임을 여러 chunk로 분할
chunk_size = total_frames // self.num_workers
for i in range(self.num_workers):
    start_idx = i * chunk_size
    end_idx = (i + 1) * chunk_size if i < self.num_workers - 1 else total_frames
```

공유 메모리 사용

멀티 프로세싱을 사용하여 비디오의 각 프레임을 병렬로 처리하는 과정에서, 프로세스 간 공유 메모리를 활용하여 데이터 전송의 오버헤드를 줄였다. 비디오의 원본 프레임과 업샘플링된 결과 프레임은 모두 공유 메모리에 저장되며, 각 프로세스는 이 메모리에 접근하여 필요한 데이터를 읽고 결과를 기록한다. 이를 통해 프로세스 간 데이터를 전달하는 속도를 최적화하고, 성능 저하를 방지할 수 있다.

```
# 공유 메모리 생성 및 연결
shm = SharedMemory(create=True,
size=int(np.prod(original_shape) * first_frame.itemsize))
input_frames = np.ndarray(original_shape, dtype=np.uint8, buffer=shm.buf)

# 업샘플링된 결과를 공유 메모리에 기록
shm_result = SharedMemory(create=True,
size=int(np.prod(result_shape) * first_frame.itemsize))
result_frames = np.ndarray(result_shape, dtype=np.uint8, buffer=shm_result.buf)
```

병렬 처리 및 데이터 저장

각 프로세스는 _process_frame_worker 라는 메서드를 통해 자신에게 할당된 프레임 범위를 처리한다. 모든 프로세스가 작업을 완료한 후, 결과는 다시 공유 메모리에서 읽어 비디오 파일로 저장된다. 이렇게 병렬화된 작업은 비디오 업샘플링의 처리 시간을 크게 단축시킬 것으로 기대된다.

```
# 병렬로 프레임 처리
process = mp.Process(target=self._process_frame_worker,
args=(shm.name, original_shape, scale_factor,
result_shape, shm_result.name, start_idx, end_idx))
```

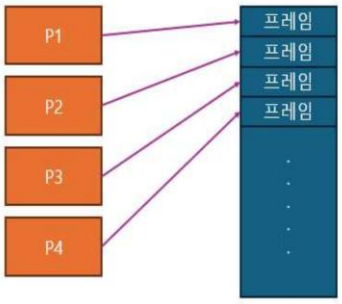
결과적으로, TimeParallelVideoUpSampler 는 멀티 프로세싱을 활용해 CPU 리소스를 최대한 활용하며, 공유 메모리를 통해 데이터 접근을 최적화하였다. 이를 통해 비디오 업샘플링 과정에서 처리 속도를 크게 향상시킬 수 있을 것으로 기대할 수 있다.

3. 실시간 처리를 위한 동적 시간 병렬화를 적용한 Video Up Sampler

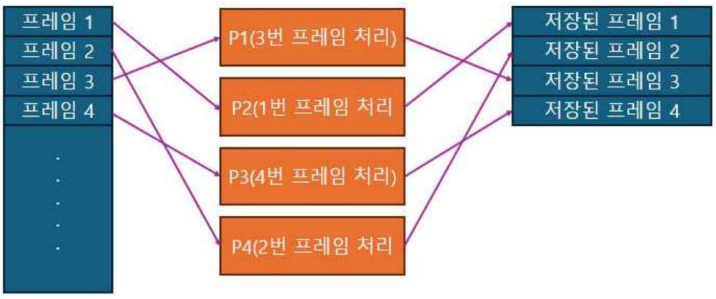
그런데 위에서의 방법처럼 동영상을 여러 개의 chunk 로 미리 분할한 후 처리하는 방식으로 전체 동영상이 주어졌을 때 빠르게 처리하도록 할 수는 있지만 실시간으로 비디오를 처리하고 보여주는 속도를 향상하는 방법으로는 사용할 수 없다. 따라서 실시간으로 비디오를 처리할 수 있도록 하는 추가적인 Vido Up Sampler 를 구현하였다.

구체적인 알고리즘은 다음과 같다.

프로세스끼리 서로 경쟁해서 다음 프레임을 확보한 후 처리한다. 이 경우 원자성을 보장해야 하기 때문에 lock 이 사용된다.



위 그림의 예시처럼 프로세스를 4 개로 가정하면 프로세스 1~4 번이 각 프레임의 원자성을 보장하면서 작업한다. 하지만 동영상 출력 시 프레임이 순서대로 write 되어야 한다는 문제점이 남아있다. 프로세스가 프레임을 처리하고 write 할 때, 프레임의 순서가 보장된다고 확신할 수 없다.



따라서 이를 해결하기 위해 위 그림과 같이 별도의 index 를 저장하도록 구현하였다. 이를 통해 이미지 처리를 완료한 프레임이 순서를 보장받으면서 write 할 수 있다.

테스트

테스트 환경

AMD Ryzen 5 3500 6-core Processor

Base speed:	3.60 GHz
Sockets:	1
Cores:	6
Logical processors:	6
Virtualization:	Enabled
L1 cache:	384 KB
L2 cache:	3.0 MB
L3 cache:	16.0 MB

테스트 시나리오

시나리오 1: 공간적 병렬화 x, 시간적 병렬화 x

시나리오 2: 공간적 병렬화 o, 시간적 병렬화 x

시나리오 3: 공간적 병렬화 o, 시간적 병렬화(정적) o (2, 4, 8, 16 개 프로세스 테스트)

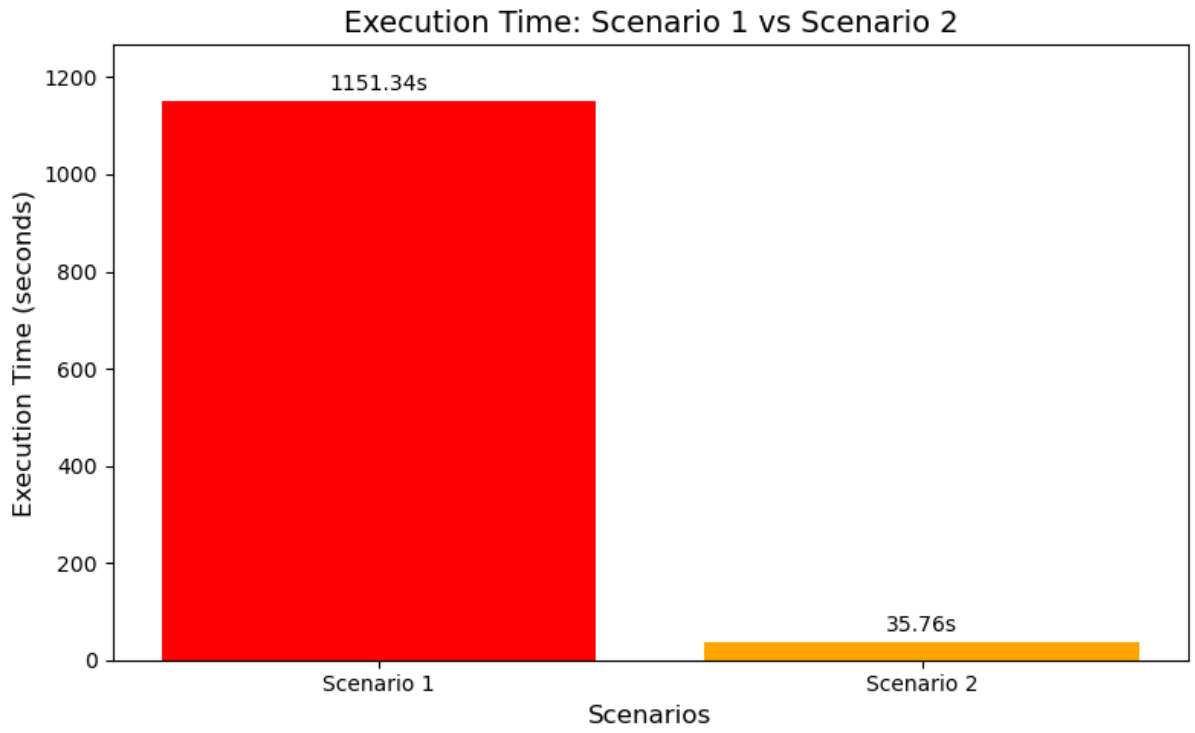
시나리오 4: 공간적 병렬화 o, 시간적 병렬화(동적) o (2, 4, 8, 16 개 프로세스 테스트)

테스트 결과

	실행시간 (초)
시나리오 1	1151.34
시나리오 2	35.76
시나리오 3	25.21, 19.96, 20.3, 21.26
시나리오 4	90.39, 89.88, 89.43, 90.21

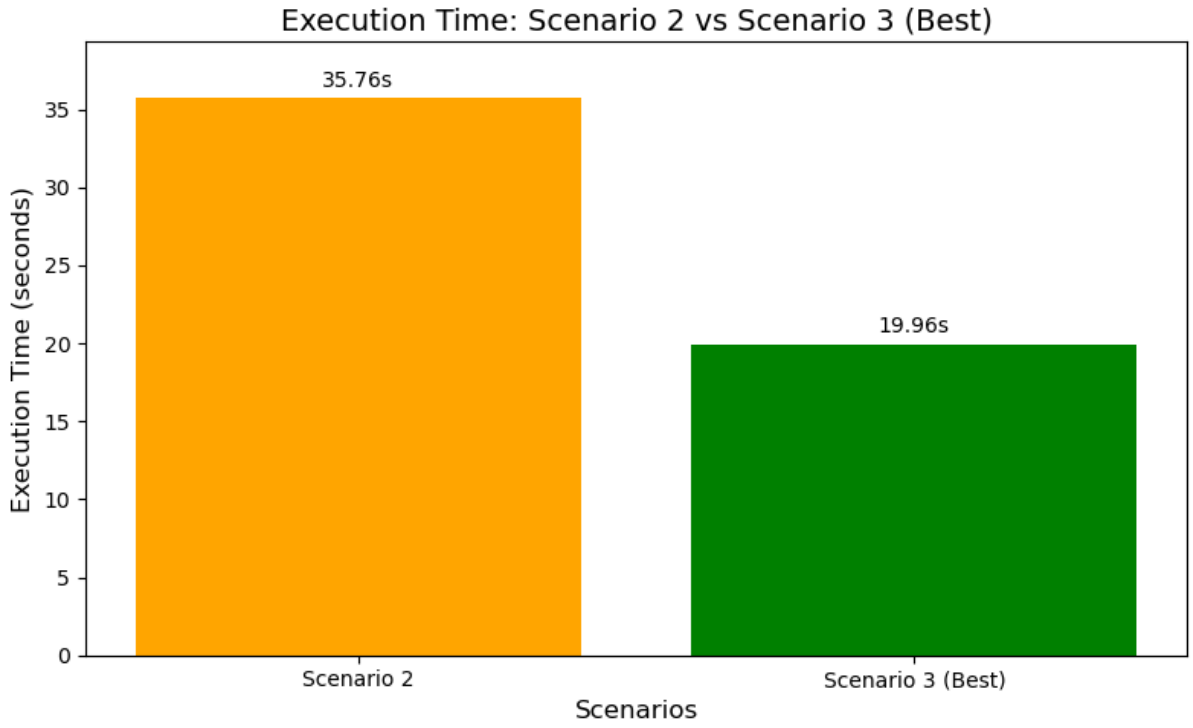
결과 분석

1. 시나리오 1 vs 시나리오 2 - 공간적 병렬화에 따른 성능 개선 효과



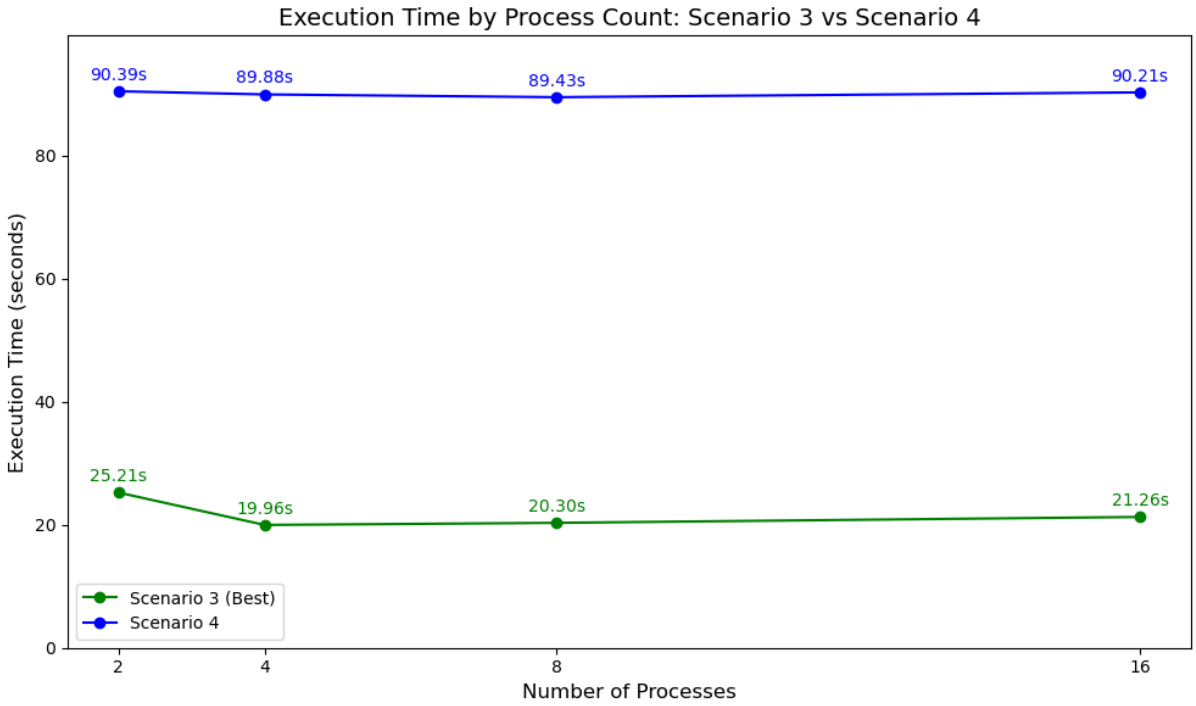
시나리오 1(공간적 병렬화 미적용)에서는 1151.34 초가 소요된 반면, 시나리오 2(공간적 병렬화 적용)에서는 35.76 초로 실행 시간이 대폭 단축되었다. SIMD 기반의 공간 병렬화를 적용하여 각 픽셀 연산을 벡터화하고, CPU 코어의 병렬 처리 기능을 최적화함으로써 병렬 처리를 구현했다. 이 방식은 CPU 캐시 효율성을 높이고, 메모리 접근 패턴을 최적화하여 연산 속도를 크게 개선했다. 이론적으로 실행 시간은 픽셀 수(960 × 540)에 비례해 1/518,400 수준으로 단축될 것으로 기대되었으나, 실제 실행 시간은 이보다 훨씬 컸다. 그 이유는 SIMD 와 같은 병렬화 기술이 병렬 연산 자체의 오버헤드(초기화, 데이터 배치, 캐시 관리 등)를 발생시키기 때문이다. 또한, CPU 의 병렬 레지스터에 들어갈 수 있는 크기에는 한계가 있어 내부적으로 완전히 한번에 모든 연산을 처리할 수는 없다. 대신, 반복 회수를 줄이는 정도의 최적화 효과만을 얻을 수 있었다. 이러한 요소들이 실제 실행 시간을 예측보다 더 길게 만든 주요 원인이다.

2. 시나리오 2 vs 시나리오 3 - 시간적 병렬화에 따른 성능 개선 효과



시나리오 3 에서는 정적 시간 병렬화를 도입하여 실행 시간이 19.96 초로 감소하였다. 이는 비디오의 각 프레임을 독립적으로 처리할 수 있어 병렬화에 적합했기 때문이다. 또한, 공유 메모리를 활용하여 프로세스 간 통신 오버헤드를 최소화하고, 파일을 합치는 과정에서 발생하는 추가적인 오버헤드를 줄임으로써 전체 실행 시간을 단축시킬 수 있었다. 이러한

최적화 덕분에 4 개의 프로세스를 사용할 때 최적의 성능을 발휘할 수 있었고, 효율적인 자원 활용이 이루어졌다.



3. 시나리오 3 vs 시나리오 4- 경쟁에 따른 실행시간 차이 비교

동적 시간 병렬화를 도입한 시나리오 4 는 실행 시간이 89.43~90.39 초로, 정적 시간 병렬화(19.96 ~25.21 초)에 비해 성능이 떨어졌다. 동적 시간 병렬화는 프로세스 간 경쟁으로 인한 lock 사용, 작업 분배 시 발생하는 원자성 보장, 그리고 실시간 순서 보장 메커니즘으로 인해 추가적인 오버헤드가 발생했기 때문으로 볼 수 있다.

그러나 시나리오 4 는 실시간 처리가 가능하다는 점에서 정적 시간 병렬화와 차별화된다. 각 프로세스가 동적으로 작업을 할당받아 프레임을 처리하며, 프레임 순서를 보장하기 위해 별도의 인덱스 매핑 방식을 도입함으로써 출력 시 데이터 정합성을 유지하였다. 따라서 동적 시간적 병렬화 방식은 실시간 처리와 프레임 순서 보장이라는 요구사항을 충족시켰다는 점에서 그 의의를 가진다.

4. 프로세스 수에 따른 실행시간 변화 추이

정적 시간 병렬화(시나리오 3)의 실행 시간은 프로세스 수가 증가할수록 개선되는 경향을 보였으나, 8 개 이상의 프로세스부터는 감소 효과가 미미해졌다. 이는 프로세스 간 통신, 메모리 접근 충돌, 그리고 OS 스케줄러의 병목 현상이 등에서 기인한다고 예상해 볼 수 있다. 동적 시간 병렬화(시나리오 4)에서 또한 유사한 패턴이 나타났다. 이러한 점은 병렬 처리의 한계를 보여주며, 효율적인 스케줄링 알고리즘이 추가적으로 필요함을 시사한다.

결론

본 프로젝트는 선형 보간법 기반의 비디오 업스케일링 작업에서 공간적 병렬화와 시간적 병렬화가 성능에 미치는 영향을 분석하였다. SIMD 를 활용한 공간적 병렬화는 픽셀 단위 연산을 벡터화함으로써 처리 속도를 약 32 배 개선하였으며, CPU 자원을 효율적으로 활용하는 데 기여했다. 또한, 정적 시간 병렬화를 통해 비디오의 독립적인 프레임 처리를 병렬적으로 수행하여 19.96 초까지 실행 시간을 단축할 수 있었다. 이는 다중 코어 환경에서의 병렬 처리가 비디오 업스케일링에 적합한 기법임을 입증하였다.

한편, 동적 시간 병렬화는 실시간 처리에 적합한 방법으로 구현되었으나, 경쟁과 lock 사용으로 인한 오버헤드로 인해 성능이 정적 시간 병렬화보다 저하되었다. 이는 실시간 처리에서의 병렬화 구현 시 자원 경합을 줄이기 위한 알고리즘 개선이 필요함을 보여준다.

결론적으로, 본 연구는 단순한 선형 보간법에서도 병렬 처리 기법을 활용하여 성능을 극대화할 수 있음을 실증하였으며, 이러한 병렬화 기법이 비디오 업스케일링뿐만 아니라 다른 비디오 프로세싱 작업에도 유용하게 적용될 수 있음을 확인하였다. 향후 연구에서는 lock-free 메커니즘 및 메모리 접근 최적화 기술을 도입하여 실시간 처리와 성능을 동시에 개선할 수 있는 방안을 모색하는 것이 필요하다.