SAR 시뮬레이션 RCS 활용

01/22/2026 18:01:09

아래 코드는 “**phase history 생성(na×nf)** → window → **pulse마다 RC(IFFT)** → **픽셀마다 보간+carrier 위상 곱 + 합산(BP)**” 흐름이고, 병목은 거의 확실하게 3군데입니다. 결론부터 말하면 **BP 루프(펄스×픽셀) + RC를 펄스마다 따로 하는 부분 + phase history에서 거대한 (Ntargets×nf) exp 생성**이 가장 큽니다.

아래에 **(1) 병목 분석**, **(2) industry standard 관점의 구조 변경**, **(3) GPU 최적화 방향(지금 코드에 맞춘 수정 포인트)** 순서로 정리할게요.

---

1) 어디가 병목인가

A. `generate_phase_history()` 병목

병목 A1: `phase = exp( ... (dR[:,None] * freq[None,:]) )` (Ntargets×nf)

- `targets`가 많거나 `nf`가 크면, 매 pulse마다 **(Ntargets × nf)** 크기의 복소 행렬을 생성합니다.
- 이건 연산량도 크지만, **GPU 메모리 대역폭**이 더 큰 병목이 됩니다.
- 또한 `phase`라는 큰 임시 배열을 매 pulse 생성/파괴 → allocator/GC 부담도 큼.

병목 A2: 매 pulse마다 `tx = xp.asarray(traj[ia])` 같은 잦은 작은 전송/형변환

- 작은 배열을 자주 `asarray` 하는 건 GPU에서 은근히 비싸요.
  (특히 Python 루프 안에서 반복)

병목 A3: `asnumpy(signal_pf)` 매 pulse (GPU→CPU 복사)

- `signal_freq`를 CPU numpy로 만들고, 매 pulse마다 GPU 결과를 CPU로 빼오고 있음.
- 이건 **PCIe 왕복**이라 가장 나쁜 패턴 중 하나.
- 뒤에서 `backprojection_image()`가 다시 `signal_gpu = xp.asarray(signal)`로 GPU로 올리니…
  GPU→CPU→GPU 왕복이 추가로 들어가요 (대형 병목).

✅ 결론: `signal_freq`는 **끝까지 GPU에 두고**, 최종 결과만 CPU로 내려야 합니다.

B. `postprocess()` 병목

- `coefficients = np.outer(...)`는 CPU에서 만들어서 곱함 → signal이 GPU면 또 변환 필요.
- 이 자체가 지배 병목은 아니지만, **불필요한 CPU 경유**가 문제.

✅ 결론: window도 GPU에서 만들고 곱하세요.

C. `backprojection_image()` 병목 (가장 큼)

병목 C1: pulse loop + 풀 프레임 보간/위상/합산

- 현재 구조는 **pulse마다 전체 픽셀(nx×ny)에 대해**:
  - `range_image = sqrt(...)` : 픽셀당 sqrt (비쌈)
  - `f(range_image)` : 보간(메모리 랜덤 접근)
  - `exp(term*range_image)` : 픽셀당 복소 exp (비쌈)
  - `bp_image += ...` : 누적

연산량 대략:

$$O(N_{\text{pulse}} \times N_{\text{pix}})$$

500×500=250k 픽셀, pulse 1000이면 2.5e8 픽셀 연산입니다.
여기에 sqrt+exp+interp가 붙으니 GPU도 오래 걸립니다.

병목 C2: 매 pulse마다 IFFT 수행

```
range_profile = fftshift(ifft(signal_gpu[index, :], fft_length))
```

- na번 IFFT(길이 fft_length)를 매번 호출
- cuFFT가 빠르긴 해도 na가 크면 누적 비용 큼
- 더 큰 문제: **펄스 루프 안에서 IFFT → 보간 → BP**가 섞여 있어서 파이프라이닝/배치가 어렵습니다.

병목 C3: interp1d_gpu_uniform가 “진짜 GPU 커널”인지

- 이 함수가 python 레벨 루프가 있거나, gather가 비효율이면 보간이 큰 병목이 됩니다.
- 보간은 텍스처/LDG(읽기 전용 캐시)나 커스텀 커널로 최적화하는 게 정석입니다.

2) Industry standard 관점에서 “구조”를 어떻게 바꾸나

BP를 완전히 버리라는 뜻이 아니라, **BP를 “ROI/정밀화 단계”로 내리는 게 표준**입니다.

(1) 표준 접근: “coarse FFT-SAR → ROI BP refine” (Hybrid)

- 1차 영상은 Range–Doppler / ω–k / Chirp scaling 같은 FFT 기반으로 뽑고
- 관심 영역(타겟 근방)만 BP로 고정밀 재구성

✅ 장점: 전체 시간 10×~100× 감소
✅ 군/위성 SAR 쪽에서 실제로 흔한 파이프라인

(2) BP를 유지하되: “RC를 전 펄스 배치로 먼저 만들고”, BP는 보간+누적만

지금은 pulse 루프 안에서 IFFT를 돌리는데,

- 먼저 `rc = ifft(signal, axis=1)`를 **한 방에 배치로 수행** (cuFFT batched)
- 그 다음 BP에서 pulse마다 `rc[p]`를 읽어 보간/누적

✅ batched FFT가 훨씬 잘 최적화됩니다.

(3) 반드시 넣는 trick: Range gating (거의 필수)

- 타겟이 있을 법한 range bin만 남기고 BP
- terrain/object 분리면 object range gate가 특히 효과적

예:

- 전체 fft_length=8192인데
- target 예상 구간 ±128 bin만 쓰면
  → 보간과 메모리 접근이 **~32× 감소**

---

3) GPU 최적화 관점에서 “지금 코드”를 어떻게 고치나

3.1 가장 먼저: GPU↔CPU 왕복 제거 (즉시 효과)

generate_phase_history 수정 방향

- `signal_freq`를 **xp.zeros**로 만들고 끝까지 GPU에 둔다.
- `asnumpy`는 마지막에 딱 한 번.

```
signal_freq = self.xp.zeros((int(self.na), int(self.nf)),
dtype=self.xp.complex128)
```

```
...
signal_freq[ia, :] = signal_pf   # GPU에 그대로
...
return signal_freq, freq_gpu, ref_point_gpu
```

그리고 `backprojection_image()`에서

```
signal_gpu = self.xp.asarray(signal)
```

이 줄이 없어지거나 no-op이 됩니다.

✅ 이거 하나만 해도 꽤 줄어듭니다.

3.2 phase history: “targets×freq” 거대 exp 만들지 않기 (chunking/streaming)

지금은 `phase`가 (Ntargets, nf)라서 큰 임시가 생깁니다. industry/GPU 정석은:

- targets를 chunk로 쪼개서
- `acc[nf]`에 누적하는 streaming

이미 이전에 하셨던 구조(CuPy chunk streaming)로 돌아가면 됩니다.

개념:

```
acc = xp.zeros(nf, complex)
for chunk targets:
    dR = ...
    phase = exp(kc * f[None,:] * dR[:,None])
    acc += sum(amp[:,None]*phase, axis=0)
signal_freq[ia] = acc
```

✅ peak 메모리를 낮추고 allocator 압박을 줄입니다.

3.3 BP: “pulse loop 안에서 풀 프레임 sqrt/exp/보간”을 줄이는 3대 방법

(A) ROI/타겟 중심 윈도우로 픽셀 수를 줄이기 (가장 강력)

- 전체 500×500 대신
- 타겟 주변만 128×128 같은 ROI
- 또는 coarse image에서 peak 찾고 adaptive ROI

(B) Range gating (보간 대상 축을 줄이기)

- `range_profile` 전체를 보간하지 말고,
- 관심 range bin 구간만 잘라서 보간

또는 더 나아가:

- 픽셀의 `rng`가 range_window 밖이면 바로 skip (mask)

(C) 커널화: 보간+위상+누적을 "하나의 GPU 커널"로

industry에서 BP를 제대로 빠르게 만들 때는 보통:

- pulse별 range_profile을 텍스처/readonly로 두고
- 픽셀당 range index를 계산
- linear interp
- carrier phase 곱
- 누적
  을 **한 커널**에서 처리합니다.

지금처럼 Python 레벨 루프 + 여러 xp 연산을 쪼개면

- 임시 배열이 많이 생기고
- 메모리 트래픽이 폭발합니다.

CuPy면 `RawKernel` / `ElementwiseKernel`로 합칠 수 있어요.

3.4 "sqrt + exp"의 비용 줄이기

BP에서 제일 비싼 두 연산이 보통:

- 거리 계산의 sqrt
- 복소 exp

최적화 팁:

- **range_image = R - r0**에서 R은 큰 값이고 변화는 작음
  → **테일러 근사/파라볼릭 근사**로 sqrt를 줄이는 기법이 자주 쓰입니다(특히 stripmap, 작은 장면).
- exp는 완전히 없앨 수는 없지만,
  - `term`은 상수이고
  - `range_image`가 제한된 구간이면
    → LUT/근사(또는 sincos) 기반 커널로 튜닝 가능

(다만 이건 정확도 손실 가능성이 있어, 먼저 위의 구조 최적화가 우선입니다.)

---

4) “현실적인 수정 로드맵” (industry 감각)

우선순위대로 하면 효과가 큽니다.

1순위 (즉시, 정확도 영향 없음)

- GPU↔CPU 왕복 제거
- window도 GPU에서 처리
- RC를 batched FFT로 한 방에 수행(가능하면)

2순위 (즉시, 정확도 영향 거의 없음)

- phase history chunk streaming
- BP에서 ROI + range gating 적용

3순위 (큰 개선, 구현 난이도↑)

- BP를 custom CUDA 커널로 (보간+위상+누적 fused)
- 또는 Hybrid FFT-SAR + ROI BP (가장 industry)

---

5) 네 코드에서 “가장 큰 병목 3개”를 한 문장으로

1. **BP에서** `for pulse: full-frame sqrt + interp + exp + accumulate` **가 압도적 병목**
2. **phase history에서** `exp(dR[:,None]*f[None,:])` **의 거대 중간 배열 생성이 병목**
3. `asnumpy` **로 매 pulse GPU→CPU 복사 후 다시 GPU로 올리는 왕복이 심각한 병목**

원하면, 다음 단계로 내가 바로 해줄 수 있는 건 두 가지 중 하나야(질문 안 하고 바로 가능):

1. **지금 코드 형태를 유지하면서**
   - (a) `signal_freq` GPU 유지
   - (b) RC batched로 분리
   - (c) BP에 ROI + range gating 추가
     하는 “최소 변경 패치”를 코드로 써주기

2. **BP를 fused 커널로 바꾸는 방향**(CuPy RawKernel)으로
   `interp + exp + accumulate`를 한 커널로 합친 버전 스켈레톤 제공

어느 쪽을 원하든, 당신이 지금 쓰는 `self.xp`가 cupy인지(=GPU)인 경우를 기준으로 바로 작성해줄게.