

# SAR - SAR 시뮬레이션 RCS 활용

좋아요. \*\*1번(최소 변경 패치) → 2번(커널 fused 최적화)\*\*를 하나의 흐름으로 병합해서 드릴게요.

아래 코드는 “당신이 올린 코드 구조를 최대한 유지”하면서 **가장 큰 병목 3개**를 동시에 줄이는 방향입니다.

---

## 전체 목표(병합 요약)

### 현재 병목

1. `generate_phase_history()`에서 ( $N_{targets} \times nf$ ) `exp` 행렬을 매 `pulse` 생성
2. `generate_phase_history()`에서 GPU→CPU(`asnumpy`) 매 `pulse` 왕복
3. `backprojection_image()`에서 `pulse`마다 풀 프레임 `sqrt+interp+exp` 임시배열 생성

### 개선 방향(Industry + GPU 최적화)

- (A) GPU↔CPU 왕복 제거: 신호는 끝까지 GPU에 두고 마지막에만 내리기
- (B) `phase history`는 chunk streaming: targets를 chunk로 쪼개 `acc[nf]`에 누적 (임시행렬 최소화)
- (C) RC는 batched FFT: `ifft`를 pulse loop 안에서 하지 말고 한 번에
- (D) BP는 “참고 코드 방식” 유지하되,
  - 1차: ROI + range gating(선택)로 연산량 절감
  - 2차: 보간+위상+누적을 CuPy RawKernel 1개로 fused (임시 배열 제거)

## 1+2번 병합 코드: “최소 변경 + fused 커널” 패치

아래를 클래스에 그대로 추가/교체하세요. (`self.xp` 가 `cupy` 면 GPU, `numpy` 면 CPU)

### (1) Phase history: GPU 스트리밍 + 왕복 제거

```
python

import numpy as np
from scipy.constants import speed_of_light, pi

def _is_cupy(xp):
    return xp.__name__ == "cupy"
```

```

def generate_phase_history_optimized(self, use_farfield=False, chunk_targets=200000):
    """
    개선점:
    - signal_freq를 GPU에 유지 (asnumpy 제거)
    - targets를 chunk로 잘라 acc[nf]에 누적 (Ntargets×nf 임시행렬 피크 완화)
    - traj, freq, targets, amp를 미리 GPU로 올림
    """

    xp = self.xp
    use_cupy = self.use_cupy and _is_cupy(xp)

    # 출력도 xp로 생성 (GPU 유지)
    signal_freq = xp.zeros((int(self.na), int(self.nf)), dtype=xp.complex128)

    # amp (rcs)
    if self.rcs is None:
        amp = xp.ones((self.targets.shape[0],), dtype=xp.complex128)
    else:
        amp = xp.asarray(self.rcs, dtype=xp.complex128)

    # 상수
    kc = xp.asarray(1j * 4.0 * pi / speed_of_light, dtype=xp.complex128) # j*4π/c
    traj = self.get_sensor_positions()
    self.sensor_xyz = traj

    # traj/freq/targets/ref_point는 한 번만 GPU로
    traj_gpu = xp.asarray(traj, dtype=xp.float64)
    ref_point_gpu = xp.asarray(self.ref_point, dtype=xp.float64)      # (3,)
    targets_gpu = xp.asarray(self.targets, dtype=xp.float64)          # (Nt,3)
    freq_gpu = xp.asarray(self.freq_space, dtype=xp.float64)          # (nf,)

    Nt = targets_gpu.shape[0]
    nf = int(self.nf)

    print(f"[PH] pulses / nf / Nt = {self.na} / {self.nf} / {Nt}")

    for ia in range(int(self.na)):
        tx = traj_gpu[ia]  # (3,
        rx = traj_gpu[ia]

        # reference range (one-way)
        if use_farfield:

```

```

los = tx - ref_point_gpu
los = los / (xp.linalg.norm(los) + 1e-12) # (3,
# far-field에서는  $dR = \text{dot}(x-\text{ref}, los)$  형태이므로  $R_{\text{ref}}$ 는 0으로 둘도 됨(이미  $\text{ref}$  빼고 있음)
# 하지만 " $\Delta R$  정의"를 일관되게 하려면  $(x-\text{ref})$ 로 쓰는 게 중요.

else:
    Rref_tx = xp.linalg.norm(ref_point_gpu - tx)
    Rref_rx = xp.linalg.norm(ref_point_gpu - rx)
    Rref = 0.5 * (Rref_tx + Rref_rx)

    # 누적(acc) (nf,)
    acc = xp.zeros((nf,), dtype=xp.complex128)

    # targets chunk streaming
    for i0 in range(0, Nt, int(chunk_targets)):
        i1 = min(i0 + int(chunk_targets), Nt)

        tg = targets_gpu[i0:i1]      # ( $N_c, 3$ )
        ag = amp[i0:i1]             # ( $N_c,$ )

        if use_farfield:
            d = tg - ref_point_gpu[None, :]           # ( $N_c, 3$ )
            dR = xp.sum(d * los[None, :], axis=1)       # ( $N_c,$ )
        else:
            Rtx = xp.linalg.norm(tg - tx[None, :], axis=1)
            Rrx = xp.linalg.norm(tg - rx[None, :], axis=1)
            R_oneway = 0.5 * (Rtx + Rrx)
            dR = R_oneway - Rref                  # ( $N_c,$ )

        # phase: ( $N_c, nf$ ) 는 여전히 생기지만 chunk로 제한됨
        phase = xp.exp(kc * (dR[:, None] * freq_gpu[None, :]))
        acc += xp.sum(ag[:, None] * phase, axis=0)

    signal_freq[ia, :] = acc

    if (ia % max(1, int(self.na)//10)) == 0:
        print(f"[PH] {ia+1}/{self.na}")

return signal_freq, freq_gpu, ref_point_gpu

```

## ✓ 효과:

- `asnumpy()` 제거 → **GPU↔CPU 왕복 사라짐**
- `phase` 임시 배열이 **chunk 크기로 제한** → 메모리 피크/allocator 부담 감소

- traj/asarray 매 반복 제거 → 잔병목 감소

## (2) Window + Range compression: “batched”로 한 번에

python

```
def postprocess_gpu(self, signal_gpu):
    """
    window를 CPU(np)에서 만들지 말고 xp에서 만들어서 곱 (GPU 유지)
    separable window: w_az[:,None] * w_f[None,:]
    """
    xp = self.xp
    na, nf = signal_gpu.shape

    if self.window_type == "Hanning":
        w_az = xp.asarray(np.hanning(na), dtype=xp.float64)
        w_f = xp.asarray(np.hanning(nf), dtype=xp.float64)
    elif self.window_type == "Hamming":
        w_az = xp.asarray(np.hamming(na), dtype=xp.float64)
        w_f = xp.asarray(np.hamming(nf), dtype=xp.float64)
    else:
        w_az = xp.ones((na,), dtype=xp.float64)
        w_f = xp.ones((nf,), dtype=xp.float64)

    return signal_gpu * (w_az[:, None] * w_f[None, :])
```

```
def range_compress_batched(self, signal_gpu):
    """
    개선점:
    - pulse loop 안에서 ifft 하지 말고, batched ifft 한 방에
    - range_window도 생성해서 반환
    """
    xp = self.xp
```

```
df = float(self.freq_space[1] - self.freq_space[0])
range_extent = speed_of_light / (2.0 * df)

# range_window: (-extent/2 .. +extent/2)
range_window = xp.linspace(-0.5 * range_extent, 0.5 * range_extent, int(self.fft_length),
                           dtype=xp.float64)
```

```

# batched IFFT along frequency axis=1
rc = xp.fft.fftshift(xp.fft.ifft(signal_gpu, n=int(self.fft_length), axis=1), axes=1)

return rc, range_window

```

✓ 효과:

- cuFFT batched를 활용 → RC 시간이 크게 줄어듦
- RC 결과 `rc(na, fft_length)` 를 만들어두면 BP에서 IFFT 반복 제거 가능

### (3) BP: 1차(최소 변경) 버전 + 2차(fused 커널) 버전

#### 3-1) 최소 변경 BP: rc를 입력으로 받고 pulse loop 유지

(기존 `backprojection_image`에서 펄스마다 IFFT를 제거하고, 보간+exp+누적만 남김)

python

```

def backprojection_image_from_rc(self, rc, range_window, nx=500, ny=500, z0=0.0,
range_gate=None):
    """
    rc: (na, fft_length) complex (fftshift된 range profile)
    range_window: (fft_length,) meters
    range_gate: (rmin, rmax) in meters (optional) # ΔR gate
    """

    xp = self.xp
    use_cupy = self.use_cupy and _is_cupy(xp)

    # image grid
    xi = xp.linspace(-0.5 * self.x_span, 0.5 * self.x_span, nx, dtype=xp.float64)
    yi = xp.linspace(-0.5 * self.y_span, 0.5 * self.y_span, ny, dtype=xp.float64)
    X, Y = xp.meshgrid(xi, yi)
    Z = xp.full_like(X, float(z0), dtype=xp.float64)

    bp_image = xp.zeros_like(X, dtype=xp.complex128)

    term = (1j * 4.0 * pi * float(self.f0) / speed_of_light) # carrier phase term

    sensor_xyz = xp.asarray(self.sensor_xyz, dtype=xp.float64)
    r0 = float(self.radius) # reference range_center (one-way)

```

```

# pulse loop
for p in range(int(self.na)):
    xyz = sensor_xyz[p]
    prof = rc[p] # (fft_length,)

    # range_image = |sensor - pixel| - r0
    rng = xp.sqrt((xyz[0]-X)**2 + (xyz[1]-Y)**2 + (xyz[2]-Z)**2) - r0

    if range_gate is not None:
        rmin, rmax = range_gate
        mask = (rng >= rmin) & (rng <= rmax)
    else:
        mask = None

    # 보간(당신의 interp1d_gpu_uniform 사용)
    f = interp1d_gpu_uniform(range_window, prof, xp=xp, kind="linear", fill_value=0.0)
    samp = f(rng)

    if mask is not None:
        samp = samp * mask.astype(xp.float64)

    bp_image += samp * xp.exp(term * rng)

    if ((p+1) % 10) == 0:
        print(f"[BP] {p+1}/{self.na}")

if use_cupy:
    return xp.asarray(bp_image), xp.asarray(xi), xp.asarray(yi)
return bp_image, xi, yi

```

### ✓ 효과:

- pulse마다 IFFT 제거
- window/rc 모두 GPU 유지
- (옵션) range\_gate로 연산량/노이즈 크게 감소

## 3-2) fused 커널 BP: “보간+carrier exp+누적”을 한 커널로

여기서부터가 2번(커널 fused)입니다.

핵심은 `rng / interp / exp / add` 를 하나의 CUDA 커널에서 처리하여 임시 배열을 없애는 것.

⚠ 주의: CuPy RawKernel은 복소 atomic add가 불편해서 pulse를 하나씩 처리(커널 호출)하면서 이미지에 누적하는 방식이 가장 단순하고 안정적입니다.  
(그래도 “임시 배열 제거” 효과가 큽니다.)

python

```
def backprojection_image_fused(self, rc, range_window, nx=500, ny=500, z0=0.0,
range_gate=None):
    """
    CuPy 전용 fused BP:
    - per-pulse로 커널 호출 (이미지 누적)
    - 커널 내부에서: range 계산 + uniform linear interp + exp + add
    """

    xp = self.xp
    if not (_is_cupy(xp) and self.use_cupy):
        # cupy가 아니면 최소 변경 버전으로
        return self.backprojection_image_from_rc(rc, range_window, nx, ny, z0, range_gate)

import cupy as cp

# grid
xi = cp.linspace(-0.5 * self.x_span, 0.5 * self.x_span, nx, dtype=cp.float64)
yi = cp.linspace(-0.5 * self.y_span, 0.5 * self.y_span, ny, dtype=cp.float64)
X, Y = cp.meshgrid(xi, yi)
Z = cp.full_like(X, float(z0), dtype=cp.float64)

img = cp.zeros((ny, nx), dtype=cp.complex128)

sensor_xyz = cp.asarray(self.sensor_xyz, dtype=cp.float64)
r0 = float(self.radius)

# range_window uniform params
x0 = float(cp.asarray(range_window[0]))
dx = float(cp.asarray(range_window[1] - range_window[0]))
M = int(range_window.shape[0])

# carrier
term_re = 0.0
term_im = float(4.0 * pi * float(self.f0) / speed_of_light) # exp(j*term_im*rng)

# gate
```

```

if range_gate is None:
    rmin = -1e30
    rmax = +1e30
else:
    rmin, rmax = map(float, range_gate)

kernel_code = r"""
extern "C" __global__
void bp_fused(
    const complex<double>* __restrict__ prof, // (M,
    const double* __restrict__ X,           // (ny*nx,)
    const double* __restrict__ Y,
    const double* __restrict__ Z,
    complex<double>* __restrict__ img,      // (ny*nx)
    const int n,
    const double sx, const double sy, const double sz,
    const double r0,
    const double x0, const double dx, const int M,
    const double term_im,
    const double rmin, const double rmax
){
    int idx = (int)(blockDim.x * blockIdx.x + threadIdx.x);
    if(idx >= n) return;

    double x = X[idx];
    double y = Y[idx];
    double z = Z[idx];

    // one-way range to pixel
    double rx = x - sx;
    double ry = y - sy;
    double rz = z - sz;
    double R = sqrt(rx*rx + ry*ry + rz*rz) - r0;

    // range gate
    if(R < rmin || R > rmax) return;

    // uniform linear interpolation
    double t = (R - x0) / dx;
    long i0 = (long)floor(t);
    double a = t - (double)i0;
}
"""

```

```

if(i0 < 0 || i0 >= (M-1)) return;

complex<double> s0 = prof[i0];
complex<double> s1 = prof[i0+1];
complex<double> samp = (1.0 - a) * s0 + a * s1;

// carrier exp(j*term_im*R)
double ph = term_im * R;
double c = cos(ph);
double s = sin(ph);
complex<double> ej = complex<double>(c, s);

img[idx] += samp * ej;
}

"""

mod = cp.RawModule(code=kernel_code, options=("-std=c++11"), name_expressions=
['bp_fused'])
bp_k = mod.get_function('bp_fused')

# flatten pointers
Xf = X.ravel()
Yf = Y.ravel()
Zf = Z.ravel()
imgf = img.ravel()
n = imgf.size

threads = 256
blocks = (n + threads - 1) // threads

for p in range(int(self.na)):
    prof = rc[p] # (M,)
    sx, sy, sz = map(float, cp.asarray(sensor_xyz[p]))
    bp_k((blocks,), (threads,),
          (prof, Xf, Yf, Zf, imgf,
           np.int32(n),
           np.float64(sx), np.float64(sy), np.float64(sz),
           np.float64(r0),
           np.float64(x0), np.float64(dx), np.int32(M),
           np.float64(term_im),
           np.float64(rmin), np.float64(rmax)
          )))

```

```

if ((p+1) % 10) == 0:
    print(f"[BP-fused] {p+1}/{self.na}")

img = imgf.reshape((ny, nx))
return cp.asarray(img), cp.asarray(xi), cp.asarray(yi)

```

### ✓ 효과:

- `rng` 배열, `samp` 배열, `exp(term*rng)` 배열 같은 **큰 임시 배열이 사라짐**
- 보간+위상+누적이 **커널 1번**에서 처리 → GPU 메모리 트래픽 크게 감소
- pulse 루프는 남지만, \*\*각 pulse에서 하는 일이 “커널 호출 1번”\*\*이라 훨씬 낫습니다.

## 실행 흐름(병합된 권장 파이프라인)

python

```

# 1) phase history (GPU 유지 + streaming)
sig, freq_gpu, ref_gpu = sim.generate_phase_history_optimized(use_farfield=False,
chunk_targets=200000)

# 2) window (GPU)
sig_w = sim.postprocess_gpu(sig)

# 3) range compression batched (GPU)
rc, range_window = sim.range_compress_batched(sig_w)

# 4) BP (1차: 최소 변경 or 2차: fused)
img, xi, yi = sim.backprojection_image_fused(
    rc=rc, range_window=range_window,
    nx=500, ny=500,
    z0=0.0,
    range_gate=(-30.0, 30.0) # 예: ΔR 계이팅(|E|). 없으면 None
)

```

## 추가로 “Industry standard”에서 거의 무조건 넣는 것 2개 (시간 확 줄어듦)

### (a) Range gating은 꼭 쓰세요

지금 BP는 모든 픽셀이 모든 range\_window를 보간할 수 있는 것처럼 계산합니다.

하지만 실제로 object가 있을 거리 대역이 좁아요.

- object 중심 예상  $\Delta R$ 를  $rng\theta$  라고 하면
- $range\_gate = (rng\theta - w, rng\theta + w)$  로 W를 수 m~수십 m로 제한

→ BP 시간 수배~수십배 감소 + clutter 억제 효과

## (b) ROI BP (전체 500×500 대신 관심 영역만)

coarse 이미지(대충)로 target 위치를 찾고, 그 주변만 고해상도로 BP합니다.

실제로 위성/군용에서도 이 하이브리드가 흔합니다.

---

## 요약: 당신이 요청한 “1+2 병합”이 정확히 해결하는 것

- 왕복 제거(GPU→CPU→GPU) ✓
  - phase history 임시행렬 폭발 완화 ✓
  - RC batched로 묶기 ✓
  - BP에서 임시배열 제거(보간+위상+누적 fused) ✓
- 

원하시면, 지금 코드의 `interp1d_gpu_uniform` 를 완전히 제거하고(커널이 대체함),  
또 `**bistatic( Tx/Rx 분리 )**`까지 fused 커널에 포함한 버전도 바로 확장해드릴게요.