① 将 DM 数组拷贝到 cuda device 上.

```
// Allocate DMs and copy to device
dm=(float *) malloc(sizeof(float)*ndm);
for (idm=0;idm<ndm;idm++)
  dm[idm]=dm_start+(float) idm*dm_step;
checkCudaErrors(cudaMalloc((void **) &ddm,sizeof(float)*ndm));
checkCudaErrors(cudaMemcpy(ddm,dm,sizeof(float)*ndm,cudaMemcpyHostToDevice));
```

② CUFFT 库设置个 complex-to complex 一维傅里叶变换.计划

nbin

100×nsub

```
// Generate FFT plan (batch in-place forward FFT)
idist=nbin;  odist=nbin;  iembed=nbin;  oembed=nbin;  istride=1;  ostride=1;
checkCudaErrors(cufftPlanMany(&ftc2cf,1,&nbin,&iembed,istride,idist,&oembed,ostride,odist,CUFFT_C2C,nfft*nsub));

// Generate FFT plan (batch in-place backward FFT)
idist=mbin;  odist=mbin;  iembed=mbin;  oembed=mbin;  istride=1;  ostride=1;
checkCudaErrors(cufftPlanMany(&ftc2cb,1,&mbin,&iembed,istride,idist,&oembed,ostride,odist,CUFFT_C2C,nchan*nfft*nsub));
```

计划执行傅里叶个数

8×100×nsub

指针　FFT长度　输入数据的尺寸和布局　输出

③ 计算 chirp 函数的数值.

```
// Compute chirp
blocksize.x=32; blocksize.y=32; blocksize.z=1;
gridsize.x=nsub/blocksize.x+1; gridsize.y=nchan/blocksize.y+1; gridsize.z=ndm/blocksize.z+1;
compute_chirp<<<gridsize,blocksize>>>(h5.fcen,nsub*h5.bwchan,ddm,nchan,nbin,nsub,ndm,dc);
```

整体带宽　nbin×nsub×ndm

↓ for iblock=0→nblock

数据的文件命名及读取

④ h5buf[i] → 分配 sizeof(char)*nsamp*nsub

dh5buf[i] → 分配GPU上内存

```
// Loop over input file contents
for (iblock=0;;iblock++) {
  // Read block
  startclock=clock();
  for (i=0;i<4;i++)
    nread=fread(h5buf[i],sizeof(char),nsamp*nsub,rawfile[i])/nsub;
  if (nread==0)
    break;
  printf("Block: %d: Read %d MB in %.2f s\n",iblock,sizeof(char)*nread*nsub*4/(1<<20),(float) (clock()-startclock)/CLOCKS_PER_SEC);

  // Copy buffers to device
  startclock=clock();
  for (i=0;i<4;i++)
    checkCudaErrors(cudaMemcpy(dh5buf[i],h5buf[i],sizeof(char)*nread*nsub,cudaMemcpyHostToDevice));
```

指向读取数据的缓存区地址.

→ nsamp 存储在 nread 中

文件指针.

→ 循环读取输入文件的内容，输出有关读取数据的信息.

→ 将主机上的缓冲数据复制到 CUDA.

⑤ unpack data and padd data 数据解压+填充.

```
246   // Unpack data and padd data
247   blocksize.x=32; blocksize.y=32; blocksize.z=1;
248   gridsize.x=nbin/blocksize.x+1; gridsize.y=nfft/blocksize.y+1; gridsize.z=nsub/blocksize.z+1;
249   unpack_and_padd<<<gridsize,blocksize>>>(dh5buf[0],dh5buf[1],dh5buf[2],dh5buf[3],nread,nbin,nfft,nsub,noverlap,cp1p,cp2p);
```

↓ 32×32

⑥ 进行前向傅里叶(时域→频域)

时 → 频
输入数据 / 输出数据

```
251   // Perform FFTs
252   checkCudaErrors(cufftExecC2C(ftc2cf,(cufftComplex *) cp1p,(cufftComplex *) cp1p,CUFFT_FORWARD));
253   checkCudaErrors(cufftExecC2C(ftc2cf,(cufftComplex *) cp2p,(cufftComplex *) cp2p,CUFFT_FORWARD));
```

⑦ swap-spectrum-halves 交换频谱中的两半.

```
255   dim3 blocksize
      n halves for large FFTs
256   blocksize.x=32; blocksize.y=32; blocksize.z=1;
257   gridsize.x=nbin/blocksize.x+1; gridsize.y=nfft*nsub/blocksize.y+1; gridsize.z=1;
258   swap_spectrum_halves<<<gridsize,blocksize>>>(cp1p,cp2p,nbin,nfft*nsub);
```

↑ 32×32

↓ for idm → ndm.

⑧ 对频域上的cp1p和cp2p两组与chirp函数相乘

```
263   // Perform complex multiplication of FFT'ed data with chirp
264   blocksize.x=32; blocksize.y=32; blocksize.z=1;
265   gridsize.x=nbin*nsub/blocksize.x+1; gridsize.y=nfft/blocksize.y+1; gridsize.z=1;
266   PointwiseComplexMultiply<<<gridsize,blocksize>>>(cp1p,dc,cp1,nbin*nsub,nfft,idm,1.0/(float) nbin);
267   PointwiseComplexMultiply<<<gridsize,blocksize>>>(cp2p,dc,cp2,nbin*nsub,nfft,idm,1.0/(float) nbin);
```

[nbin] [nsub] [ndm]

```
533 // Pointwise complex multiplication (and scaling)
534 static __global__ void PointwiseComplexMultiply(cufftComplex *a,cufftComplex *b,cufftComplex *c,int nx,int ny,int l,float scale)
535 {
536   int i,j,k;
537   i=blockIdx.x*blockDim.x+threadIdx.x; → nbin*nsub
538   j=blockIdx.y*blockDim.y+threadIdx.y; → nfft
539
540   if (i<nx && j<ny) {
541     k=i+nx*j;
542     c[k]=ComplexScale(ComplexMul(a[k],b[i+nx*l]),scale);
543   }
544 }
```

idm.

复数缩放. → { c.x = R.x × scale ; c.y = R.y × scale.

复数乘法 = { z1 = a+bi ; z2 = c+di } z1×z2 = (ac−bd) + (ad+bc)i.
x y

{ z1 = a[k] ; z2 = b[i+nx*l] } { x = z1.x × z2.x − z1.y × z2.y ; y = z1.x × z2.y − z1.y × z2.x

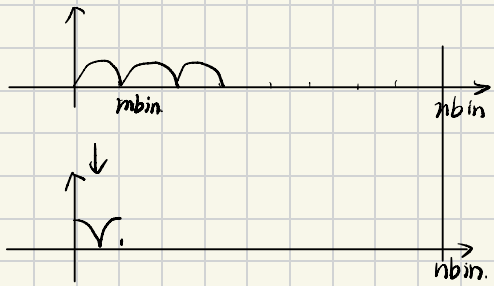
⑨ 再次 swap_spectrum_halves 交换数据

```
269     // Swap spectrum halves for small FFTs
270     blocksize.x=32; blocksize.y=32; blocksize.z=1;
271     gridsize.x=mbin/blocksize.x+1; gridsize.y=nchan*nfft*nsub/blocksize.y+1; gridsize.z=1;
272     swap_spectrum_halves<<<gridsize,blocksize>>>(cp1,cp2,mbin,nchan*nfft*nsub);
273
```

[mbin][nchan*nfft*nsub]

cp1: [nbin][nfft][nsub]

[mbin][nchan]



⑩ 进行 IFFT，将 cp1，cp2 → 时域信息

```
274     // Perform FFTs
275     checkCudaErrors(cufftExecC2C(ftc2cb,(cufftComplex *) cp1,(cufftComplex *) cp1,CUFFT_INVERSE));
276     checkCudaErrors(cufftExecC2C(ftc2cb,(cufftComplex *) cp2,(cufftComplex *) cp2,CUFFT_INVERSE));
```

⑪ 核函数 transpose_unpadd_and_detect. 对数据进行转置和去填充.

```
278     // Detect data
279     blocksize.x=32; blocksize.y=32; blocksize.z=1;
280     gridsize.x=mbin/blocksize.x+1; gridsize.y=nchan/blocksize.y+1; gridsize.z=nfft/blocksize.z+1;
281     transpose_unpadd_and_detect<<<gridsize,blocksize>>>(cp1,cp2,mbin,nchan,nfft,nsub,noverlap/nchan,nread/nchan,dfbuf);
```

[mbin][nchan] [nfft]

nbin

并在 dfbuf 中存储 cp1 和 cp2 模的平方和 stokeI

⑫ 核函数 compute_block_sums. 在 nchan 下分 msum=1024 个 mblock.

```
283     // Compute block sums for redigitization
284     blocksize.x=32; blocksize.y=32; blocksize.z=1;
285     gridsize.x=mchan/blocksize.x+1; gridsize.y=mblock/blocksize.y+1; gridsize.z=1;
286     compute_block_sums<<<gridsize,blocksize>>>(dfbuf,mchan,mblock,msum,bs1,bs2);
```

每个 block 求和和平方和

[mchan][mblock]

msamp/msum → nsamp/nchan/1024

⑬ 核函数 compute_channel_statistics 来计算每个 channel 的统计值（平均值、标准差）

```
288     // Compute channel stats
289     blocksize.x=32; blocksize.y=1; blocksize.z=1;
290     gridsize.x=mchan/blocksize.x+1; gridsize.y=1; gridsize.z=1;
291     compute_channel_statistics<<<gridsize,blocksize>>>(mchan,mblock,msum,bs1,bs2,zavg,zstd);
```

mchan　nsub*nchan　nsamp/msum　[mchan]

⑭ 对滤波器组进行分段重新数字化为 8 位

```
293     // Redigitize data to 8bits
294     blocksize.x=32; blocksize.y=32; blocksize.z=1;
295     gridsize.x=mchan/blocksize.x+1; gridsize.y=mblock/blocksize.y+1; gridsize.z=1;
296     if (ndec==1)
297   redigitize<<<gridsize,blocksize>>>(dfbuf,mchan,mblock,msum,zavg,zstd,3.0,5.0,dcbuf);
298     else
299   decimate_and_redigitize<<<gridsize,blocksize>>>(dfbuf,ndec,mchan,mblock,msum,zavg,zstd,3.0,5.0,dcbuf);
```

[mchan][mblock]

⑮ 将 GPU 上的 dcbuf 指次复制到 CPU 上的 cbuf 缓冲区

字节数 = [msamp][mchan/ndec]

```
299   decimate_and_redigitize<<<gridsize,blocksize>>>(dfbuf,ndec,mchan,mblock,msum,zavg,zstd,3.0,5.0,dcbuf);
300
301     // Copy buffer to host
302     checkCudaErrors(cudaMemcpy(cbuf,dcbuf,sizeof(unsigned char)*msamp*mchan/ndec,cudaMemcpyDeviceToHost));
303
304     // Write buffer
305     fwrite(cbuf,sizeof(char),nread*nsub/ndec,outfile[idm]);
306   }
307   printf("Processed %d DMs in %.2f s\n",ndm,(float) (clock()-startclock)/CLOCKS_PER_SEC);
308 }
```

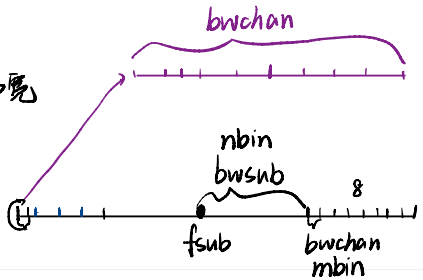
写出到

compute_chirp 核函数

```
__global__ void compute_chirp(double fcen,double bw,float *dm,int nchan,int nbin,int nsub,int ndm,cufftComplex *c)
{
  int ibin,ichan,isub,idm,mbin,idx;
  double s,rt,t,f,fsub,fchan,bwchan,bwsub;

  // Number of channels per subband
  mbin=nbin/nchan;

  // Subband bandwidth
  bwsub=bw/nsub;

  // Channel bandwidth
  bwchan=bw/(nchan*nsub);

  // Indices of input data
  isub=blockIdx.x*blockDim.x+threadIdx.x;
  ichan=blockIdx.y*blockDim.y+threadIdx.y;
  idm=blockIdx.z*blockDim.z+threadIdx.z;

  // Keep in range
  if (isub<nsub && ichan<nchan && idm<ndm) {
    // Main constant
    s=2.0*M_PI*dm[idm]/DMCONSTANT;

    // Frequencies
    fsub=fcen-0.5*bw+bw*(float) isub/(float) nsub+0.5*bw/(float) nsub;
    fchan=fsub-0.5*bwsub+bwsub*(float) ichan/(float) nchan+0.5*bwsub/(float) nchan;

    // Loop over bins in channel
    for (ibin=0;ibin<mbin;ibin++) {
      // Bin frequency
      f=-0.5*bwchan+bwchan*(float) ibin/(float) mbin+0.5*bwchan/(float) mbin;

      // Phase delay
      rt=-f*f*s/((fchan+f)*fchan*fchan);

      // Taper
      t=1.0/sqrt(1.0+pow((f/(0.47*bwchan)),80));

      // Index
      idx=ibin+ichan*mbin+isub*mbin*nchan+idm*nsub*mbin*nchan;

      // Chirp
      c[idx].x=cos(rt)*t;
      c[idx].y=sin(rt)*t;
    }
  }
```

一个子带的带宽.

一个子带下一个通道的带宽



子带的中频.

nchan对应的中频.

?

$$t=\frac{1}{\sqrt{1+\left[f/(0.47\cdot bwchan)\right]^{80}}}$$

c[idx] = c[idm] c[ichan] c[imbin] c[isub]

unpack and padd 核函数

nbin

```
598 // Unpack the input buffer and generate complex timeseries. The output
599 // timeseries are padded with noverlap samples on either side for the
600 // convolution.
601 __global__ void unpack_and_padd(char *dbuf0,char *dbuf1,char *dbuf2,char *dbuf3,int nsamp,int nbin,int nfft,int nsub,int noverlap,cufftComplex *cp1,cufftComplex *cp2)
602 {
603   int64_t ibin,ifft,isamp,isub,idx1,idx2;
604
605   // Indices of input data
606   ibin=blockIdx.x*blockDim.x+threadIdx.x;
607   ifft=blockIdx.y*blockDim.y+threadIdx.y;
608   isub=blockIdx.z*blockDim.z+threadIdx.z;
609
610   // Only compute valid threads
611   if (ibin<nbin && ifft<nfft && isub<nsub) {
612     idx1=ibin+nbin*isub+nsub*nbin*ifft;
613     isamp=ibin+(nbin-2*noverlap)*ifft-noverlap;
614     idx2=isub+nsub*isamp;
615     if (isamp<0 || isamp>=nsamp) {
616       cp1[idx1].x=0.0;
617       cp1[idx1].y=0.0;
618       cp2[idx1].x=0.0;
619       cp2[idx1].y=0.0;
620     } else {
621       cp1[idx1].x=(float) dbuf0[idx2];
622       cp1[idx1].y=(float) dbuf1[idx2];
623       cp2[idx1].x=(float) dbuf2[idx2];
624       cp2[idx1].y=(float) dbuf3[idx2];
625     }
626   }
627
628   return;
629 }
```

2048　　cp1p　　cp2p

dbuf　nsamp*nsub = nfft*nvalid

cp1p, cp2p = nbin * nfft * nsub

idx2 = isub + nsub(ibin + (nbin-2*noverlap)*ifft - noverlap)

ifft=1　isamp = ibin + nvalid - noverlap

transpose_unpadd_and_detect 核函数

[nsamp][nsub]

```
665 // After the segmented FFT the data is in a cube of nbin by nchan by
666 // nfft, where nbin and nfft are the time indices. Here we rearrange
667 // the 3D data cube into a 2D array of frequency and time, while also
668 // removing the overlap regions and detecting (generating Stokes I).
669 __global__ void transpose_unpadd_and_detect(cufftComplex *cp1,cufftComplex *cp2,int nbin,int nchan,int nfft,int nsub,int noverlap,int nsamp,float *fbuf)
670 {
671   int64_t ibin,ichan,ifft,isub,isamp,idx1,idx2;
672
673   ibin=blockIdx.x*blockDim.x+threadIdx.x;
674   ichan=blockIdx.y*blockDim.y+threadIdx.y;
675   ifft=blockIdx.z*blockDim.z+threadIdx.z;
676   if (ibin<nbin && ichan<nchan && ifft<nfft) {
677     // Loop over subbands
678     for (isub=0;isub<nsub;isub++) {
679       // Padded array index
680       //      idx1=ibin+nbin*isub+nsub*nbin*(ichan+nchan*ifft);
681       idx1=ibin+ichan*nbin+(nsub-isub-1)*nbin*nchan+ifft*nbin*nchan*nsub;
682
683       // Time index
684       isamp=ibin+(nbin-2*noverlap)*ifft-noverlap;
685
686       // Output array index
687       idx2=(nchan-ichan-1)+isub*nchan+nsub*nchan*isamp;
688
689       // Select data points from valid region
690       if (ibin>=noverlap && ibin<=nbin-noverlap && isamp>=0 && isamp<nsamp)
691   fbuf[idx2]=cp1[idx1].x*cp1[idx1].x+cp1[idx1].y*cp1[idx1].y+cp2[idx1].x*cp2[idx1].x+cp2[idx1].y*cp2[idx1].y;
692     }
693   }
694
695   return;
696 }
```

noverlap/nchan　　dfbuf

cp1　　cp2　　nbin

nread/nchan
↓
nsamp

[nbin][nfft][nsub]

[nfft][nbin][nchan][nsub]

模的平方和

swap_spectrum_halves 核函数

```
631 // Since complex-to-complex FFTs put the center frequency at bin zero
632 // in the frequency domain, the two halves of the spectrum need to be
633 // swapped.
634 __global__ void swap_spectrum_halves(cufftComplex *cp1,cufftComplex *cp2,int nx,int ny)
635 {
636   int64_t i,j,k,l,m;
637   cufftComplex tp1,tp2;
638
639   i=blockIdx.x*blockDim.x+threadIdx.x;
640   j=blockIdx.y*blockDim.y+threadIdx.y;
641   if (i<nx/2 && j<ny) {
642     if (i<nx/2)
643       k=i+nx/2;
644     else
645       k=i-nx/2;
646     l=i+nx*j;
647     m=k+nx*j;
648     tp1.x=cp1[l].x;
649     tp1.y=cp1[l].y;
650     tp2.x=cp2[l].x;
651     tp2.y=cp2[l].y;
652     cp1[l].x=cp1[m].x;
653     cp1[l].y=cp1[m].y;
654     cp2[l].x=cp2[m].x;
655     cp2[l].y=cp2[m].y;
656     cp1[m].x=tp1.x;
657     cp1[m].y=tp1.y;
658     cp2[m].x=tp2.x;
659     cp2[m].y=tp2.y;
660   }
661
662   return;
663 }
```

cp1p　　cp2p

↓

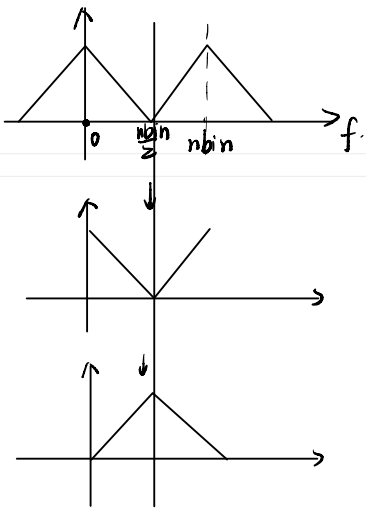
[nbin][nfft][nsub]

nbin.

→

→ 未变换

→ 已变.



复数→复数软平中心

通常在0频率点

compute_block_sums

[nfft][nbin][nchan][nsub]
[nfft][nbin][nsub]
[nsamp][nsub]

[mblock][mchan]
bs1 bs2

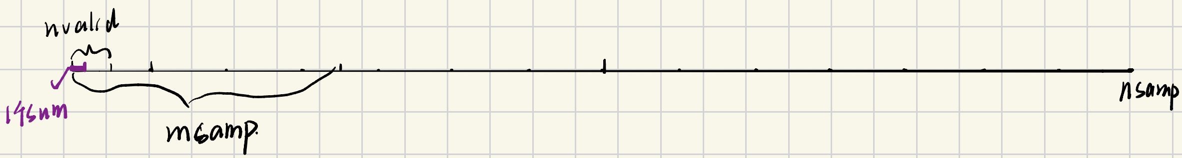
```
// Compute segmented sums for later computation of offset and scale
__global__ void compute_block_sums(float *z,int nchan,int nblock,int nsum,float *bs1,float *bs2)
{
  int64_t ichan,iblock,isum,idx1,idx2;

  ichan=blockIdx.x*blockDim.x+threadIdx.x;
  iblock=blockIdx.y*blockDim.y+threadIdx.y;
  if (ichan<nchan && iblock<nblock) {
    idx1=ichan+nchan*iblock;
    bs1[idx1]=0.0;
    bs2[idx1]=0.0;
    for (isum=0;isum<nsum;isum++) {
      idx2=ichan+nchan*(isum+iblock*nsum);
      bs1[idx1]+=z[idx2];
      bs2[idx1]+=z[idx2]*z[idx2];
    }
  }

  return;
}
```

dfdnf. mchan mblock. msum

[nblock][nchan] → 压缩数据

[nblock][nchan][nsam]

mblock = msamp / nsum = nsamp / nchan / nsum



compute_channel_statistics 核函数.

```
// Compute segmented sums for later computation of offset and scale
__global__ void compute_channel_statistics(int nchan,int nblock,int nsum,float *bs1,float *bs2,float *zavg,float *zstd)
{
  int64_t ichan,iblock,idx1;
  double s1,s2;

  ichan=blockIdx.x*blockDim.x+threadIdx.x;
  if (ichan<nchan) {
    s1=0.0;
    s2=0.0;
    for (iblock=0;iblock<nblock;iblock++) {
      idx1=ichan+nchan*iblock;
      s1+=bs1[idx1];
      s2+=bs2[idx1];
    }
    zavg[ichan]=s1/(float) (nblock*nsum);
    zstd[ichan]=s2/(float) (nblock*nsum)-zavg[ichan]*zavg[ichan];
    zstd[ichan]=sqrt(zstd[ichan]);
  }

  return;
}
```

mchan mblock [mblock][mchan] ↘ ↙ mchan

压缩 [mblock][mchan] → [mchan]

求和

→ 平均值.

→ 方差.

(1chan平方和) − (1chan平均值)²

↓ 标准差

redigitize 核函数.

```
// Redigitize the filterbank to 8 bits in segments
__global__ void redigitize(float *z,int nchan,int nblock,int nsum,float *zavg,float *zstd,float zmin,float zmax,unsigned char *cz)
{
  int64_t ichan,iblock,isum,idx1;
  float zoffset,zscale;

  ichan=blockIdx.x*blockDim.x+threadIdx.x;
  iblock=blockIdx.y*blockDim.y+threadIdx.y;
  if (ichan<nchan && iblock<nblock) {
    zoffset=zavg[ichan]-zmin*zstd[ichan];
    zscale=(zmin+zmax)*zstd[ichan];

    for (isum=0;isum<nsum;isum++) {
      idx1=ichan+nchan*(isum+iblock*nsum);
      z[idx1]-=zoffset;
      z[idx1]*=256.0/zscale;
      cz[idx1]=(unsigned char) z[idx1];
      if (z[idx1]<0.0) cz[idx1]=0;
      if (z[idx1]>255.0) cz[idx1]=255;
    }
  }

  return;
}
```

- z: bfbuf [nsamp][nchan]; nchan: mchan; nblock: mblock; nsum: msum; zavg: mchan; zmin: 3; zmax: 5
- cz: dcbuf [msamp][mchan/ndec]
- zoffset: → 数字化偏移 X-3θ
- zscale: 8 x → 缩放 8×θ
- idx1: [mblock][msum][mchan]
- if (z[idx1]>255.0) …: 8位无符号位

$msamp = nsamp / nchan$

$mblock = msamp / msum$

$mchan = nsub * nchan$

decimate_and_redigitize 核函数.

```
// Decimate and Redigitize the filterbank to 8 bits in segments
__global__ void decimate_and_redigitize(float *z,int ndec,int nchan,int nblock,int nsum,float *zavg,float *zstd,float zmin,float zmax,unsigned char *cz)
{
  int64_t ichan,iblock,isum,idx1,idx2,idec;
  float zoffset,zscale,ztmp;

  ichan=blockIdx.x*blockDim.x+threadIdx.x;
  iblock=blockIdx.y*blockDim.y+threadIdx.y;
  if (ichan<nchan && iblock<nblock) {
    zoffset=zavg[ichan]-zmin*zstd[ichan];
    zscale=(zmin+zmax)*zstd[ichan];

    for (isum=0;isum<nsum;isum+=ndec) {
      idx2=ichan+nchan*(isum/ndec+iblock*nsum/ndec);
      for (idec=0,ztmp=0.0;idec<ndec;idec++) {
  idx1=ichan+nchan*(isum+idec+iblock*nsum);
  ztmp+=z[idx1];
      }
      ztmp/=(float) ndec;
      ztmp-=zoffset;
      ztmp*=256.0/zscale;
      cz[idx2]=(unsigned char) ztmp;
      if (ztmp<0.0) cz[idx2]=0;
      if (ztmp>255.0) cz[idx2]=255;
    }
  }

  return;
}
```

- z: [nsamp][nchan]
- ndec≠1
- ichan: mchan.
- iblock: mblock