key points

For convolution, pay attention on memory access is quiet important. Here, using the constant memory for storing the filter kernel can greatly speed up its efficiency with the function __constant__. As a constant variable, its accessing speed is faster than global variable.

Then, data can be transferred form host to device with: cudaMemcpyToSymbol(dest, src, size).

Convolution with tiling:

Halo + tile = reginal data, tile = output data, where halo = filter radius, so that those over tile's data can be accessed form local memory instead of global memory.

The arithmetic-to-global memory access ratio for the tiled kernel is

$$\frac{OUT_TILE_DIM^2 * (2 * FILTER_RADIUS + 1)^2 * 2}{(OUT_TILE_DIM + 2 * FILTER_RADIUS)^2 * 4}.$$

Pay attention on telling difference between Valid convolution(smaller), same convolution(same) and full convolution(larger)!

1

1.1

51, calculated form 40+6+5=51.

1.2 8

,21,13,20, same as 1.

1.3

1.3.1

original number.

1.3.2

All move to left for a number.

1.3.3

All move to right for a number.

1.3.4

~~calculate the difference between the current number~~ computes the central difference, which approximates the first derivative of the signal.

1.3.5

The average of the nearby number.

1.4

1.4.1

$\frac{M-1}{2}$ M-1

1.4.2

$N \cdot M$

1.4.3

~~$(N-(M-1))\cdot M+2\cdot\sum_{i=1}^{\frac{M-1}{2}}(i+1)$~~

get it from $(N-(M-1))\times M-\frac{(M-1)(M+1)}{4}$

which actually is $N\times M-\frac{(M-1)(M+1)}{4}$

1.5

1.5.1

$(N+M-1)^2-N^2$

1.5.2

~~$M^2\cdot N^2$~~

$$[(N+M-1)^2-N^2]\times M^2$$

1.5.3

~~$N^2\cdot N^2-\sum_{i=1}^{\frac{M-1}{2}}[(N+i)^2-N^2]$~~

$$N^2\times M^2$$

1.6

1.6.1

$(N_1+M_1-1)(N_2+M_2-1)-N_1\cdot N_2$

1.6.2

~~$M_1M_2\cdot N_1N_2$~~ $(N_1+M_1-1)\times(N_2+M_2-1)\times(M_1\times M_2)$

1.6.3

~~$M_1M_2\cdot N_1N_2-\sum_{i=1}^{\frac{M_1-1}{2}}\sum_{i=1}^{\frac{M_2-1}{2}}[[(N_i+i)(M_2+j)-N_1N_2]]$~~ $N_1\times N_2\times(M_1\times M_2)$

1.7

1.7.1

$\lceil N/T\rceil^2$

1.7.2

$(T+M-1)^2$

1.7.3

$4*(T+M-1)^2$

1.7.4

$\lceil N/T\rceil^2$,$(T+M-1+halo)^2$,$4*(T+M-1+halo)^2$

1.8

```
__global__ void convolution_3D_basic_kernel(const float *N, const float *F,
     float *P, int r, int width, int height, int depth) {
    int outCol = blockIdx.x*blockDim.x + threadIdx.x;
    int outRow = blockIdx.y*blockDim.y + threadIdx.y;
    int outDep = blockIdx.z*blockDim.z + threadIdx.z;
    float Pvalue = 0.0f;
    // Ensure the thread is within the output bounds
    if (outCol < width && outRow < height && outDep < depth) {
        for (int fRow = 0; fRow < 2*r+1; fRow++){
            for (int fCol = 0; fCol < 2*r+1; fCol++){
                for (int fDep = 0; fDep < 2*r+1; fDep++){
                    int inRow = outRow - r + fRow;
                    int inCol = outCol - r + fCol;
                    int inDep = outDep - r + fDep;
                    if (inRow >= 0 && inRow < height &&
                                    inCol >= 0 &&   inCol < width &&
                                    inDep >= 0 && inDep < depth)
                                    {
                        int filterIdx = fRow * (2*r+1)*(2*r+1)+fCol*(2*r
                              +1)+ fDep;
                        Pvalue += F[filterIdx] * N[inRow*width*depth    +
                            inCol*depth+inDep];
                    }
                }
            }
        }
        int outputIdx = outRow * width * depth + outCol * depth +   outDep;
        P[outputIdx] = Pvalue;
    }
}
```

Attention: F and P is actually a pointer(a 1D array), so use single indices for accessing it.

1.9

```
#define MAX_FILTER_SIZE //based on the maximum expected 'r'.
__constant__ float F[MAX_FILTER_SIZE];
__global__ void convolution_3D_const_mem_kernel(const float *N, float *P,
    int r, int width, int height, int depth){
    int outCol = blockIdx.x*blockDim.x + threadIdx.x;
    int outRow = blockIdx.y*blockDim.y + threadIdx.y;
    int outDep = blockIdx.z*blockDim.z + threadIdx.z;
    float Pvalue = 0.0f;
    if (outCol < width && outRow < height && outDep < depth){
        for (int fRow = 0; fRow < 2*r+1; fRow++){
            for (int fCol = 0; fCol < 2*r+1; fCol++){
                for (int fDep = 0; fDep < 2*r+1; fDep++){
                    int inRow = outRow - r + fRow;
                    int inCol = outCol - r + fCol;
                    int inDep = outDep - r + fDep;
                    if (inRow >= 0 && inRow < height &&
                                    inCol >= 0 &&   inCol < width &&
                                    inDep >= 0 && inDep < depth)
                                    {
                        int filterIdx = fRow * (2*r+1)*(2*r+1)+fCol*(2*r
                              +1)+ fDep;
                        Pvalue += F[filterIdx] * N[inRow*width*depth    +
                            inCol*depth+inDep];
                    }
                }
            }
        }
        P[outRow*width*depth+outCol*depth+outDep] = Pvalue;
    }

}
```

Compared to last problem, the variable F changed into a const variable

const VS __constant__:

when data is large and doesn't fit into constant memory, access patterns are non-uniform, or different threads access different data shall use the function const. While __constant__ has faster access when all threads use the same filter coefficients, reduced global memory bandwidth. But it is limited by the size of constant memory and mot suitable for very large filters.

1.10

```
#define IN_TILE_DIM 32
#define OUT_TILE_DIM ((IN_TILE_DIM) - 2*(FILTER_RADIUS))
__constant__ float F_c[2*FILTER_RADIUS+1][2*FILTER_RADIUS+1][2*
    FILTER_RADIUS+1];
__global__ void convolution_tiled_3D_const_mem_kernel(const float *N, float
     *P, int width, int height, int depth) {
    int col = blockIdx.x*OUT_TILE_DIM + threadIdx.x - FILTER_RADIUS;
    int row = blockIdx.y*OUT_TILE_DIM + threadIdx.y - FILTER_RADIUS;
    int dep = blockIdx.z*OUT_TILE_DIM + threadIdx.z - FILTER_RADIUS;
    __shared__ float N_s[IN_TILE_DIM][IN_TILE_DIM][IN_TILE_DIM];
    if (col < width && row < height && dep < depth){
        N_s[threadIdx.z][threadIdx.y][threadIdx.x] = N[row*width*depth+col*
            depth+dep];
    }else{
        N_s[threadIdx.z][threadIdx.y][threadIdx.x] = 0;
    }
    __syncthreads();
    int tileCol = threadIdx.x - FILTER_RADIUS;
    int tileRow = threadIdx.y - FILTER_RADIUS;
    int tileDep = threadIdx.z - FILTER_RADIUS;
    if (col < width && row < height && dep < depth){
        if (tileCol >=0 && tileCol < OUT_TILE_DIM && tileRow >=0 && tileRow
            <OUT_TILE_DIM && tileDep >=0 && tileDep < OUT_TILE_DIM){
            float Pvalue = 0.0f;
            for (int fRow = 0; fRow < 2*FILTER_RADIUS+1; fRow++){
                for (int fCol = 0; fCol < 2*FILTER_RADIUS+1; fCol++){
                    for (int fDep = 0; fDep < 2*FILTER_RADIUS+1; fDep++){
                        Pvalue += F_c[fRow][fCol][fDep]*N_s[tileRow+fRow][
                            tileCol+fCol][tileDep+fDep];
                    }
                }
            }
            P[row*width*depth+col*depth+dep] = Pvalue;
        }
    }
}
```

The use effect of N_s:

Act as original data for a faster reading speed.