# CUDA中使用共享内存对图像进行卷积

因为共享内存比全局内存访问速度快，为提高效率，在程序中尽量使用共享内存。需要注意的是一个block中，线程的最大数量是1024个，所以在数据划分时要注意blockSize的大小，例如，下面的例子中设置的blockSizeo为32×32，如果设置得过大，核函数不会被执行。

使用共享内存，在核函数中需要先声明共享数组，方法是使用extern \_\_shared\_\_，如：

extern \_\_shared\_\_ uchar4 s\_block[];

其大小在外部调用核函数时定义。

然后，需要从改block中使用的全部数据全局内存中拷贝数据到共享内存。例如在使用一个核对图像进行卷积操作时，将图像划分为32×32大小的图像块，每个图像块对应一个block，而如果使用如果3×3大小的核进行卷积，那么每个相邻block之间有1个宽度的重叠区域，每个block运算需要的图像数据量为34×34，因此共享内存大小为34\*34\*sizeof(uchar4)。另外还要注意处理在全局图像边缘处的情况，如在下面的例子中，使用idxClip函数来处理在图像边缘时计算得到的全局图像坐标为负数的情况。下面的例子中，在当前线程序号为(0,0)时，将重叠区域的数据全部拷到共享内存中，而在其它线程中，只需要把对应的元素复制到共享内存中即可。

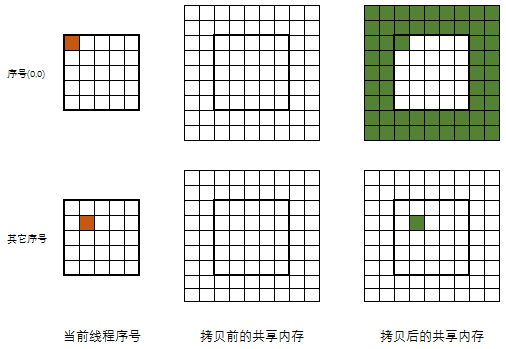


图 1 数据的拷贝过程

为在核函数的输入输出都使用共享内存，可以将共享内存定义为两者总共大小，使用各自的指针指向共享内存中不同的位置。将数据从全局内存中拷贝到共享内存后，对一模块的线程进行同步，方法为调用\_\_syncthreads函数。再进行图像处理操作，图像处理的结果保存到共享内存中，处理完成后，再调用\_\_syncthreads函数进行同步，系统可以通过多个输出数据元素打包在单个存储事务中，来提高写回操作的效率，而不是对每一个输出元素都进行一次全局内存写操作。

|  |
| --- |
| #include "cuda\_runtime.h"  #include "device\_launch\_parameters.h"  #include <stdio.h>  #define TX 32  #define TY 32  #define RAD 2  //将n限制在0至255区域  \_\_device\_\_  unsigned char clip(int n) { return n > 255 ? 255 : (n < 0 ? 0 : n); }  //将idx限制在0至idxMax  \_\_device\_\_  int idxClip(int idx, int idxMax) { return idx > idxMax ? (idxMax - 1) : (idx < 0 ? 0 : idx); }  //获取第row行第col列在宽为width，高为height的二维图像中所在的序号  \_\_device\_\_  int flatten(int col, int row, int width, int height)  {  return idxClip(col, width) + idxClip(row, height) \* width;  }  \_\_device\_\_ int2 share2global(int2 s\_pos, int2 start\_g\_pos, int pad)  {  int2 g\_pos;  g\_pos.x = s\_pos.x - pad + start\_g\_pos.x;  g\_pos.y = s\_pos.y - pad + start\_g\_pos.y;  return g\_pos;  }  \_\_global\_\_  void blurKernel(uchar4 \*d\_in, uchar4 \*d\_out, float \*d\_filter, int w, int h)  {  int c = threadIdx.x + blockDim.x \* blockIdx.x;  int r = threadIdx.y + blockDim.y \* blockIdx.y;  if ((c >= w) || (r >= h))return;  int i = flatten(c, r, w, h);  int s\_c = threadIdx.x + RAD;  int s\_r = threadIdx.y + RAD;  int s\_w = blockDim.x + 2 \* RAD;  int s\_h = blockDim.y + 2 \* RAD;  int s\_i = flatten(s\_c, s\_r, s\_w, s\_h);  int fltSz = 2 \* RAD + 1;  extern \_\_shared\_\_ uchar4 s\_block[];  uchar4 \* s\_in = s\_block;  uchar4 \* s\_out = &s\_block[s\_w \* s\_h];  //对于一般的像素  s\_in[s\_i] = d\_in[i];  //对于block边缘像素（重叠）  if (threadIdx.x == 0 && threadIdx.y == 0)  {  for (int s\_y = 0; s\_y < s\_h; s\_y++)  {  for (int s\_x = 0; s\_x < s\_w; s\_x++)  {  if (s\_x < RAD || s\_y < RAD || s\_x >= RAD + blockDim.x || s\_y >= RAD + blockDim.y)  {  int2 s\_pos, start\_pos;  s\_pos.x = s\_x;  s\_pos.y = s\_y;  start\_pos.x = blockDim.x \* blockIdx.x;  start\_pos.y = blockDim.y \* blockIdx.y;  int2 g\_pos = share2global(s\_pos, start\_pos, RAD);  s\_in[flatten(s\_pos.x, s\_pos.y, s\_w, s\_h)] = d\_in[flatten(g\_pos.x, g\_pos.y, w, h)];  }  }  }  }  \_\_syncthreads();    float rgb[3] = { 0.f,0.f,0.f };  for (int rd = -RAD; rd <= RAD; rd++)  {  for (int cd = -RAD; cd <= RAD; cd++)  {  int s\_imgIdx = flatten(s\_c + cd, s\_r + rd, s\_w, s\_h);  int fltIdx = flatten(RAD + cd, RAD + rd, fltSz, fltSz);  uchar4 color = s\_in[s\_imgIdx];  float weight = d\_filter[fltIdx];  rgb[0] += weight\*color.x;  rgb[1] += weight\*color.y;  rgb[2] += weight\*color.z;  }  }  s\_out[i].x = clip(rgb[0]);  s\_out[i].y = clip(rgb[1]);  s\_out[i].z = clip(rgb[2]);  \_\_syncthreads();  d\_out[i] = s\_out[i];    }  void blurParallel(uchar4 \*arr, int w, int h)  {  int fltSz = 2 \* RAD + 1;  float filter[25] = { 0.04f,0.04f,0.04f,0.04f,0.04f,  0.04f,0.04f,0.04f,0.04f,0.04f,  0.04f,0.04f,0.04f,0.04f,0.04f,  0.04f,0.04f,0.04f,0.04f,0.04f,  0.04f,0.04f,0.04f,0.04f,0.04f  };  uchar4 \*d\_in = 0, \*d\_out = 0;  float \*d\_filter = 0;  cudaSetDevice(0);  //输入图像  cudaMalloc(&d\_in, w \* h \* sizeof(uchar4));  cudaMemcpy(d\_in, arr, w \* h \* sizeof(uchar4), cudaMemcpyHostToDevice);  //输出图像  cudaMalloc(&d\_out, w \* h \* sizeof(uchar4));  //滤波器  cudaMalloc(&d\_filter, fltSz \* fltSz \* sizeof(float));  cudaMemcpy(d\_filter, filter, fltSz \* fltSz \* sizeof(float), cudaMemcpyHostToDevice);  //数据划分  dim3 blockSize(TX, TY);  dim3 gridSize((w + TX - 1) / TX, (h + TY - 1) / TY);  const size\_t smSz = (TX + 2 \* RAD)\*(TY + 2 \* RAD)\*sizeof(uchar4);  //执行内核  blurKernel <<<gridSize, blockSize, smSz >>>(d\_in, d\_out, d\_filter, w, h);  //拷回数据  cudaMemcpy(arr, d\_out, w\*h\*sizeof(uchar4), cudaMemcpyDeviceToHost);  //释放内存  cudaFree(d\_in);  cudaFree(d\_out);  cudaFree(d\_filter);  }  #include <opencv2\opencv.hpp>  using namespace std;  using namespace cv;  void main()  {  Mat image = imread("flower.jpg");  if (image.data)  {  cvtColor(image, image, CV\_RGB2RGBA);  blurParallel(image.ptr<uchar4>(0), image.cols, image.rows);  imwrite("blur.jpg", image);  }  } |

图 2 卷积结果