Image convolution with CUDA

Lecture

Alexey Abramov
abramov _at_ physik3.gwdg.de

Georg-August University, Bernstein Center for Computational Neuroscience,
III Physikalisches Institut, Göttingen, Germany

Convolution (definition)

A **convolution** is an integral that expresses the amount of overlap of one function **g** as it is shifted over another function **f**:

$$(f * g)(i) = \int f(n) g(i-n) dn$$
$$= \int g(n) f(i-n) dn$$

In discrete terms it can be written as:

$$(f * g)(i) = \sum_n f(n) g(i-n)$$

For two dimensions:

$$(f * g)(i,j) = \sum_n \sum_m f(n,m) g(i-n, j-m)$$

Convolution in image processing tasks

Convolution filtering can be used for a wide range of image processing tasks. Many types of blur filters or edge detectors use convolutions.

Original image

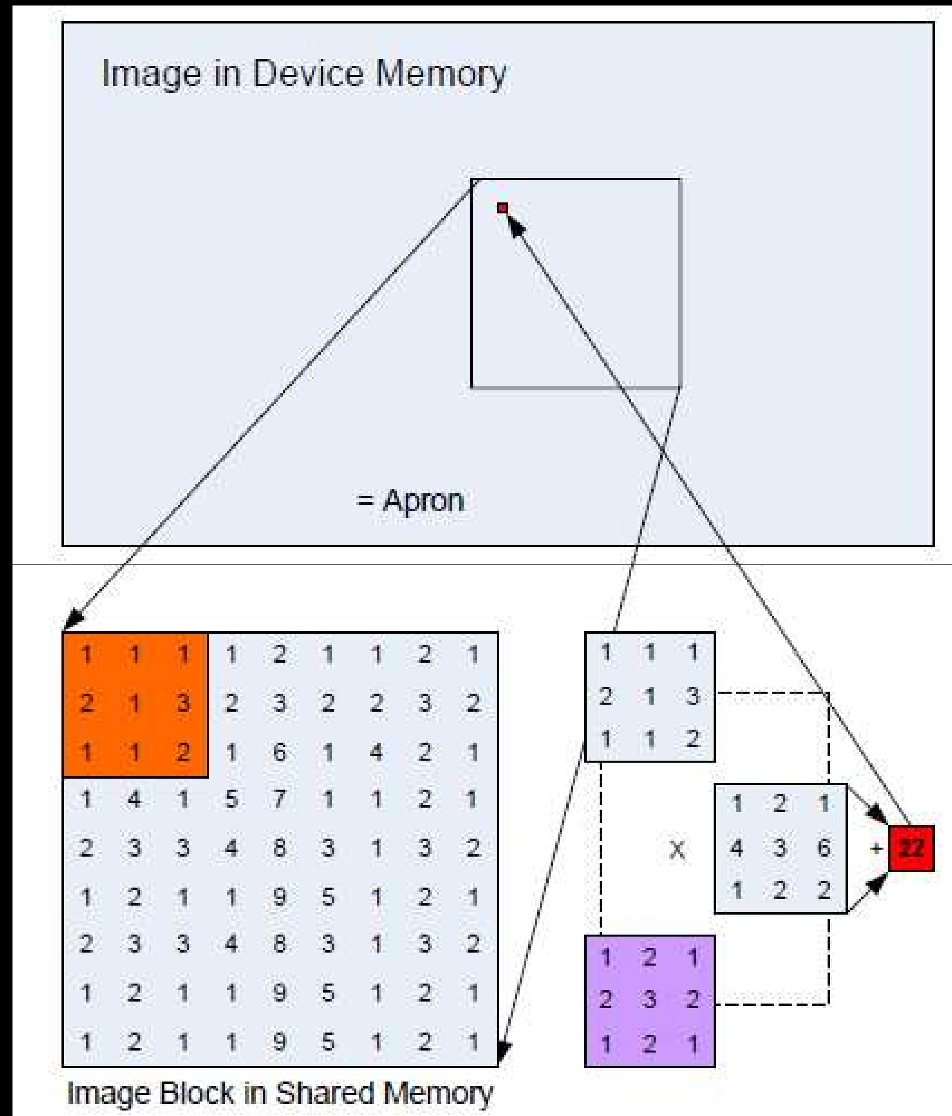
Blur convolution applied to the original image

In the context of image processing a convolution filter is just the scalar product of the filter weights with the input pixels within a window surrounding each of the output pixels.

A naive convolution algorithm

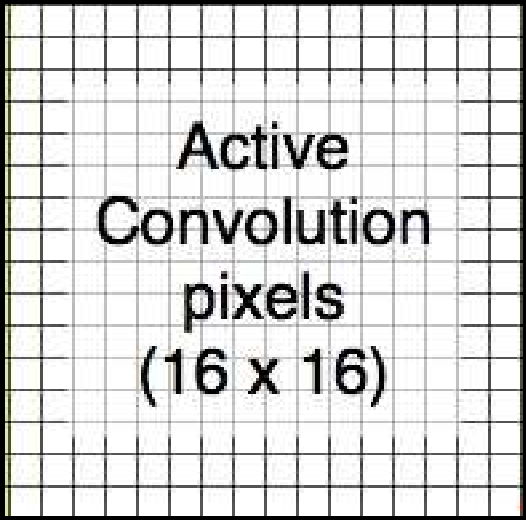
The scalar product is a parallel operation that is well suited to computation on highly parallel hardware such as the GPU.

To process and compute an output pixel (**red**), a region of the input image (**orange**) is multiplied element-wise with the filter kernel (**purple**) and then the results are summed.



A naive convolution algorithm (CPU version)

```
#include <cutil.h>
#include <cuda_runtime.h>
#include <cutil_inline.h>

#include <QtGui/QImage>
#include <QtGui/QColor>

#include <math.h>

#define KERNEL_RADIUS 8
#define KERNEL_LENGTH (2 * KERNEL_RADIUS + 1)

#define BLOCK_W 16
#define BLOCK_H 16
```



A naive convolution algorithm (CPU version)

```
int main(int argc, char **argv){

    // read an input image
    QString filename ("./Lena.png");
    QImage img (filename);

    int width = img.width();
    int height = img.height();

    // separate color components
    float *pR = new float [width * height];
    float *pG = new float [width * height];
    float *pB = new float [width * height];
    float *pBuffer = new float [width * height];

    bzero(pR, width * height * sizeof (float) );
    bzero(pG, width * height * sizeof (float) );
    bzero(pB, width * height * sizeof (float) );
    bzero(pBuffer, width * height * sizeof (float) );
```

Original image

Original image

pR

pG

pB

```
for(int i = 0; i < height; ++i){
    for(int j = 0; j < width; ++j){

        QRgb rgb = img.pixel(j,i);

        *(pR + i*width +j) = (float) qRed(rgb);
        *(pG + i*width +j) = (float) qGreen(rgb);
        *(pB + i*width +j) = (float) qBlue(rgb);


    }
}
```

Original image

After convolution

$$G(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2}{2\sigma^2}}$$

x – the distance from the origin in the horizontal axis

σ – the standard deviation of the Gaussian distribution

Values from the distribution are used to build a convolution matrix which is applied to the original image. Each pixel's new value is set to a weighted average of that pixel's neighborhood. The original pixel's value receives the heaviest weight (having the highest Gaussian value) and neighboring pixels receive smaller weights as their distance to the original pixel increases.

Gaussian convolution kernel is a symmetric function, so the row and column filters are identical. Applying a Gaussian blur has the effect of reducing the image's high-frequency components: a Gaussian blur is thus a low pass filter.

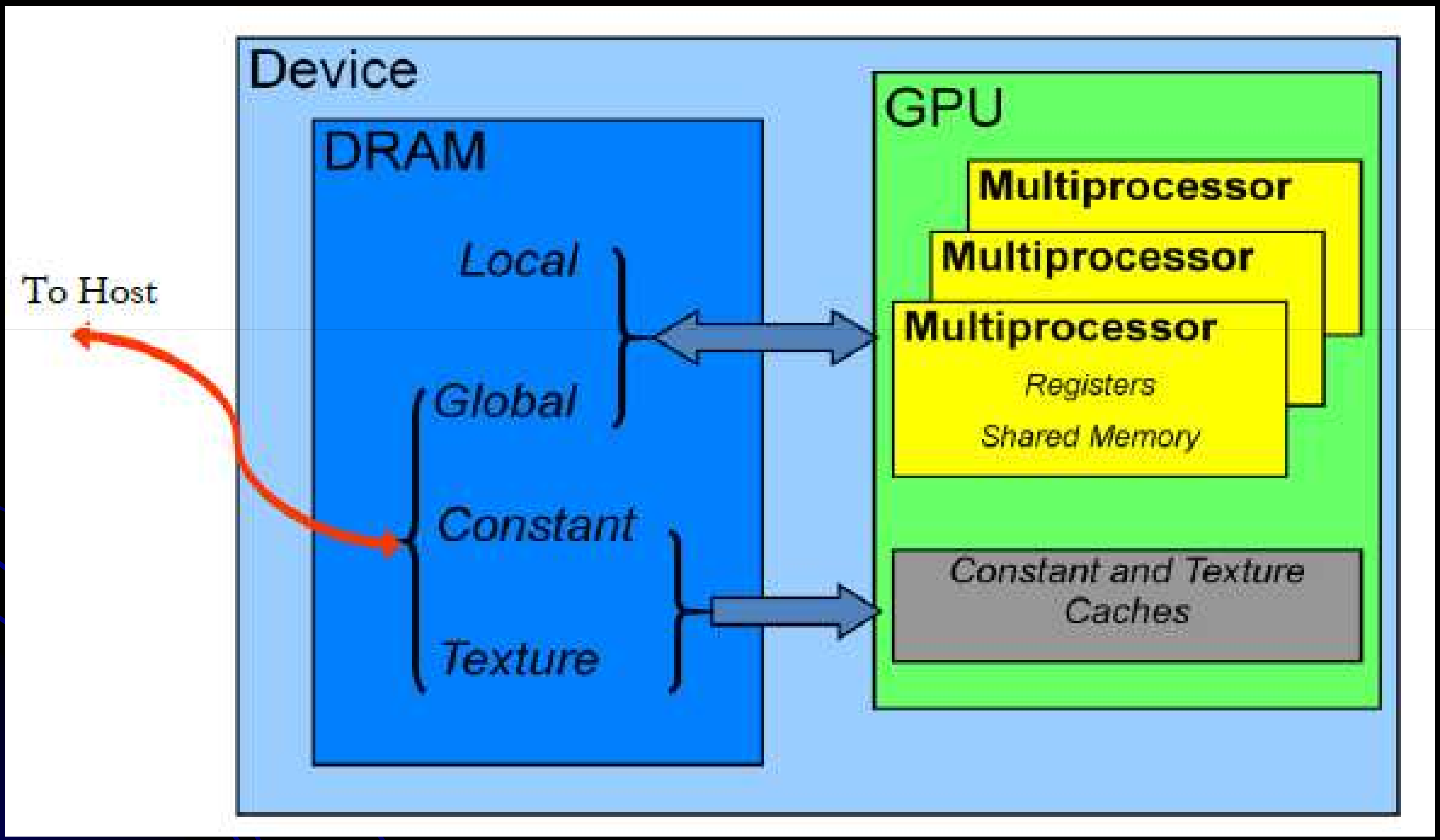
```
// convolution kernel
float *pKernel = new float [KERNEL_LENGTH];
bzero(pKernel, KERNEL_LENGTH * sizeof (float) );

// kernel initialization
float kernelSum = 0;

for(int i = 0; i < KERNEL_LENGTH; ++i){

    float dist = (float)(i - KERNEL_RADIUS) / (float) KERNEL_RADIUS;
    pKernel[i] = expf(- dist * dist / 2);
    kernelSum += pKernel[i];
}

for(int i = 0; i < KERNEL_LENGTH; ++i)
    pKernel [i] /= kernelSum;
```

```
// run very simple convolution on CPU
convolutionCPU(pBuffer, pR, pKernel, width, height);
memcpy(pR, pBuffer, width*height*sizeof (float) );

convolutionCPU(pBuffer, pG, pKernel, width, height);
memcpy(pG, pBuffer, width*height*sizeof (float) );

convolutionCPU(pBuffer, pB, pKernel, width, height);
memcpy(pB, pBuffer, width*height*sizeof (float) );


// build an output image to see the final result
QImage out_img(width,height,QImage::Format_RGB32);

for(int i = 0; i < height; ++i){
    for(int j = 0; j < width; ++j){

        QColor clr((int)*(pR + i*width +j), (int)*(pG + i*width +j), (int)*(pB + i*width +j));
        out_img.setPixel(j,i,clr.rgb());

    }
}
// Free the memory
...
```

```
void convolutionCPU(float *dst, float *src, float *kernel, int width, int height){

    for(int y = 0; y < height; y++){
      for(int x = 0; x < width; x++){

        float sum = 0;
        float value = 0;

        for(int i = -KERNEL_RADIUS; i <= KERNEL_RADIUS; ++i){
          for(int j = -KERNEL_RADIUS; j <= KERNEL_RADIUS; ++j){

            int c_y = y + i;
            int c_x = x + j;

            if (c_x < 0 || c_x > (width-1) || c_y < 0 || (c_y > (height-1)))
              value = 0;
            else
              value = *(src + c_y*width + c_x);

            sum += value * kernel[KERNEL_RADIUS + i] * kernel[KERNEL_RADIUS + j];
          }
        }

        *(dst + y*width + x) = sum;
      }
    }
  }
```

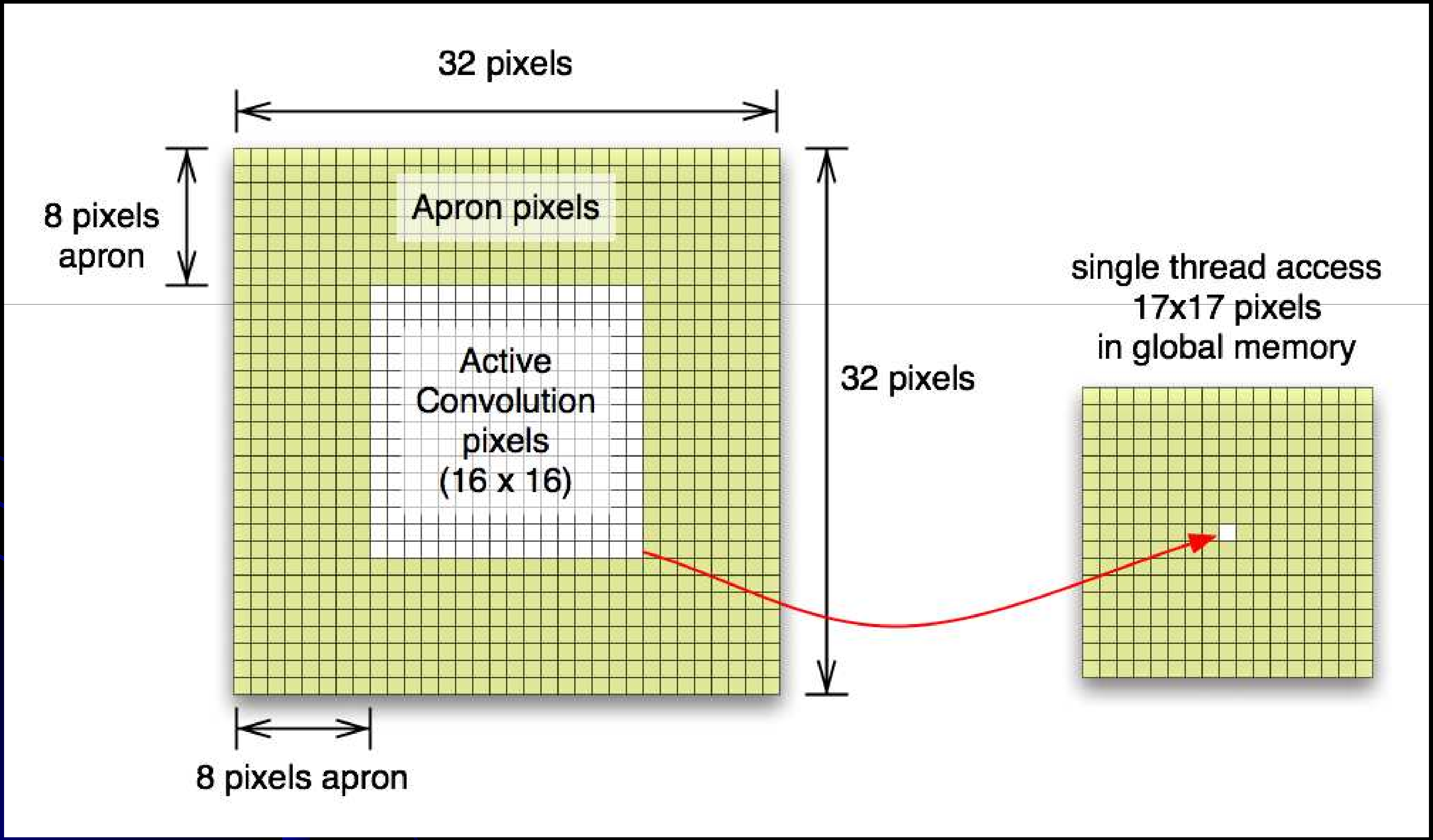
Device memory spaces

CUDA devices use several memory spaces, which have different characteristics that reflect their distinct usage in CUDA applications.

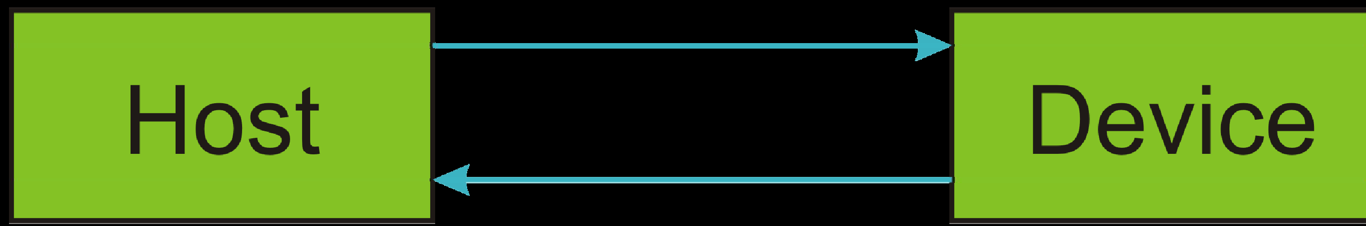


A convolution algorithm (the most naive approach)

The most naive approach is to use global memory to send data to device and each thread accesses it to compute convolution kernel. There are no idle threads since total number of threads invoked is the same as total number of pixels.



Transfer of original RGB components



Transfer of modified RGB components (convolution results)

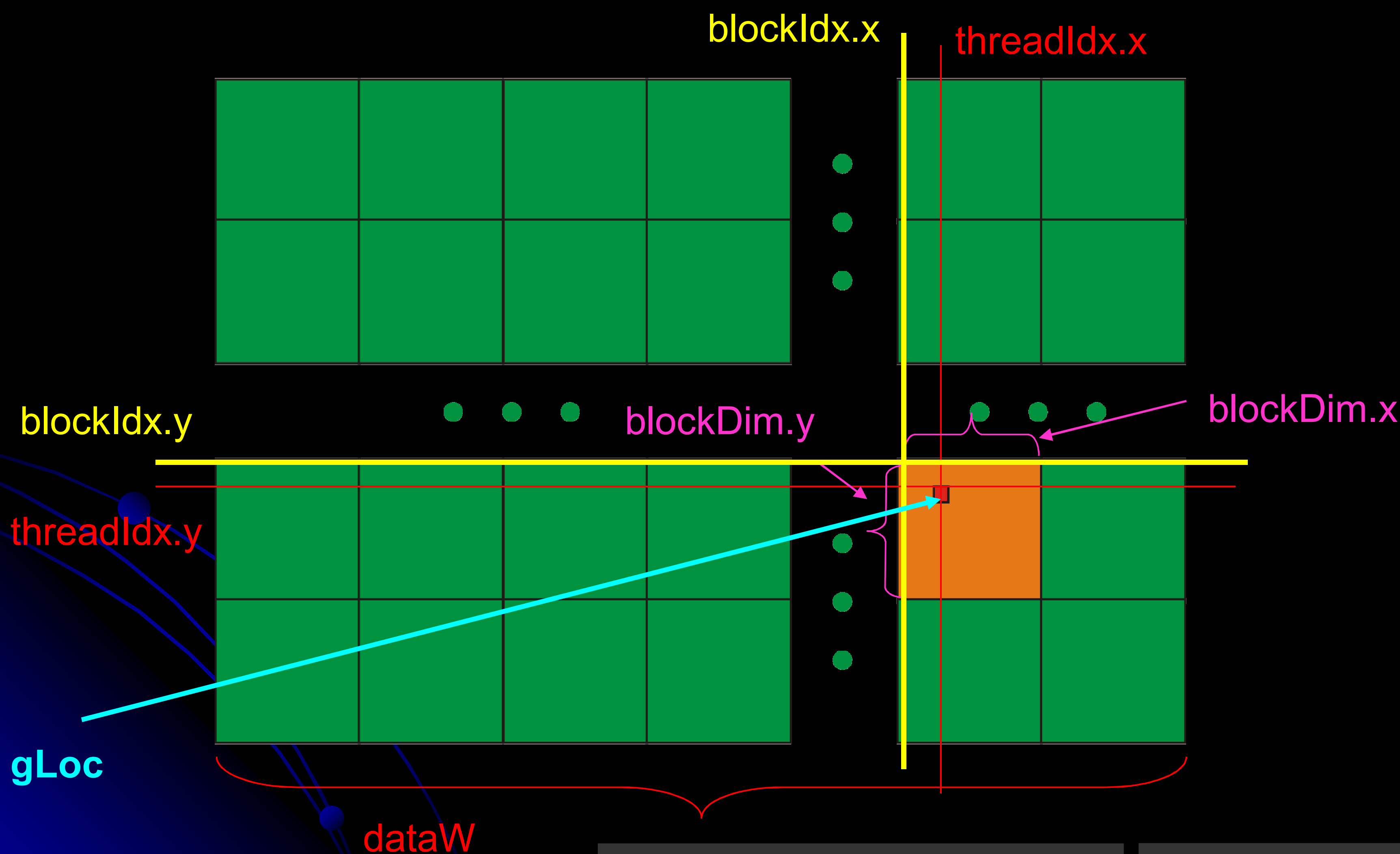
```
// run very simple convolution on GPU
convolutionGPU(pBuffer, pR, pKernel, width, height);
memcpy(pR, pBuffer, width*height*sizeof (float) );

convolutionGPU(pBuffer, pG, pKernel, width, height);
memcpy(pG, pBuffer, width*height*sizeof (float) );

convolutionGPU(pBuffer, pB, pKernel, width, height);
memcpy(pB, pBuffer, width*height*sizeof (float) );
```

On the GPU now!

```
void convolutionGPU(float *h_Data_out, float *h_Data_in, float *d_Kernel, int w, int h){

    float *d_Buffer_in = 0;
    float *d_Buffer_out = 0;

    cudaMalloc( (void **)&d_Buffer_in, w*h*sizeof(float));
    cudaMalloc( (void **)&d_Buffer_out, w*h*sizeof(float));

    // copy convolution kernel to the constant GPU memory
    cudaMemcpyToSymbol((const char*)d_KernelDev, d_Kernel,
                                   KERNEL_LENGTH*sizeof(float));

    int gridY = h / BLOCK_H;
    int gridX = w / BLOCK_W;

    dim3 blocks(BLOCK_W, BLOCK_H);
    dim3 grids(gridX, gridY);

    cudaMemcpy(d_Buffer_in, h_Data_in, w*h*sizeof(float), cudaMemcpyHostToDevice);

    unsigned int hTimer;
    cutCreateTimer(&hTimer);
    cutResetTimer(hTimer);
    cutStartTimer(hTimer);
```

```
convolutionGPU_kernel<<<grids, blocks>>>(d_Buffer_out, d_Buffer_in, w, h);

cudaThreadSynchronize();
cutStopTimer(hTimer);
double gpuTime = cutGetTimerValue(hTimer);

std::cout << "Simple convolution on GPU, time = " << gpuTime << " ms" << std::endl;

cudaMemcpy(h_Data_out, d_Buffer_out, w*h*sizeof(float), cudaMemcpyDeviceToHost);

// Cleanup
cudaFree(d_Buffer_in);
cudaFree(d_Buffer_out);


__global__ void convolutionGPU_kernel(float *d_Result, float *d_Data, int dataW, int dataH){

    // global memory address of the current thread in the whole grid
    const int gLoc = threadIdx.x + blockIdx.x * blockDim.x + threadIdx.y * dataW +
                         blockIdx.y * blockDim.y * dataW;
```

```
// global memory address of the current thread in the whole grid
const int gLoc = threadIdx.x + blockIdx.x * blockDim.x + threadIdx.y * dataW +
                 blockIdx.y * blockDim.y * dataW;
```

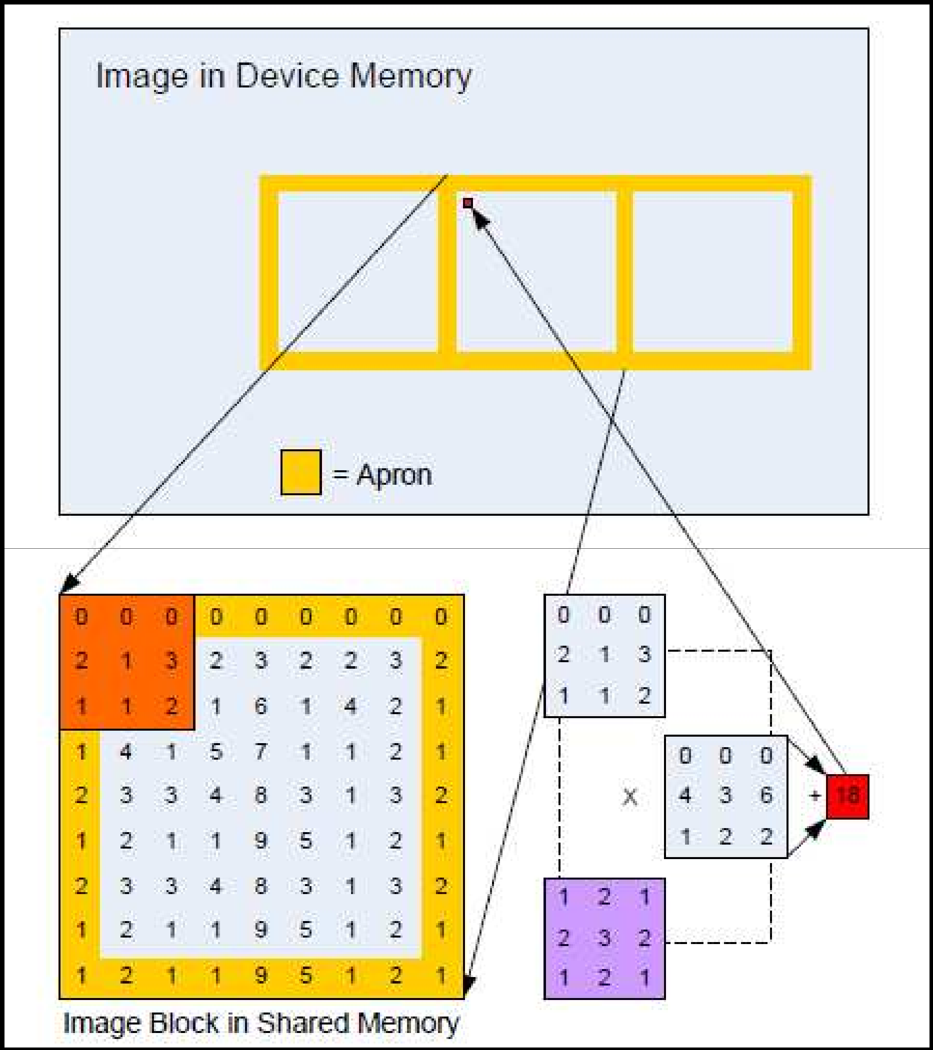


```
float sum = 0;
float value = 0;

// image coordinates of the current thread
int x = threadIdx.x + blockIdx.x * blockDim.x;
int y = threadIdx.y + blockIdx.y * blockDim.y;

for(int i = -KERNEL_RADIUS; i <= KERNEL_RADIUS; ++i){
   for(int j = -KERNEL_RADIUS; j <= KERNEL_RADIUS; ++j){

      int c_y = y + i;
      int c_x = x + j;

      // check boundaries
      if( c_x < 0 || c_x > (dataW-1) || c_y < 0 || c_y > (dataH-1) )
         value = 0;
      else
         value = *(d_Data + c_y*dataW + c_x);

      sum += value * d_KernelDev[KERNEL_RADIUS + i] *
                d_KernelDev[KERNEL_RADIUS + j];
   }
}

d_Result[gLoc] = sum;
```

Shared memory and the apron

For any reasonable kernel size, the pixels at the edge of the shared memory array will depend on pixels not in shared memory. Around the image block within a thread block, there is an **apron** of pixels of the width of the kernel radius that is required in order to filter the image block.

Thus, each thread block must load into shared memory the pixels to be filtered and the apron pixels. The apron of one block overlaps with adjacent blocks.



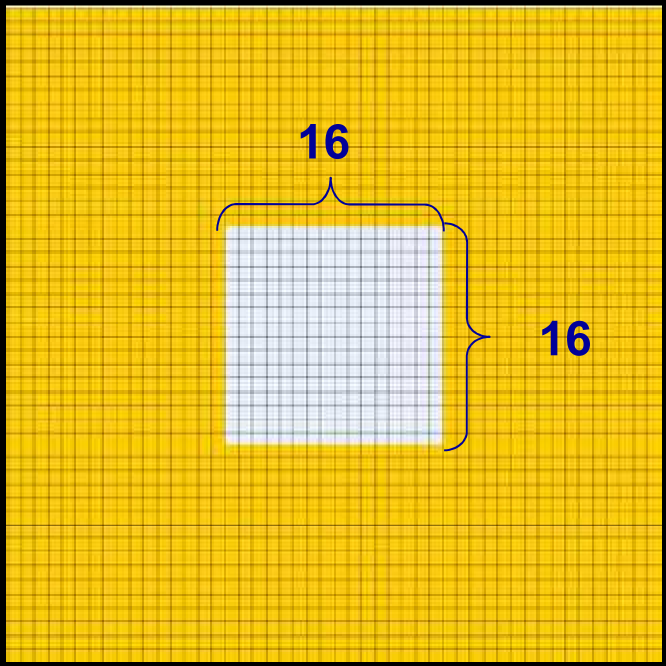
Avoid idle threads

If one thread is used for each pixel loaded into shared memory, then the threads loading the apron pixels will be idle during the filter computation. As the radius of the filter increases, the percentage of idle threads increases.

This wastes much of the available parallelism, and with the limited amount of shared memory available, the waste for large radius kernels can be quite high.

Image block 16 x 16

Radius of the kernel = 16



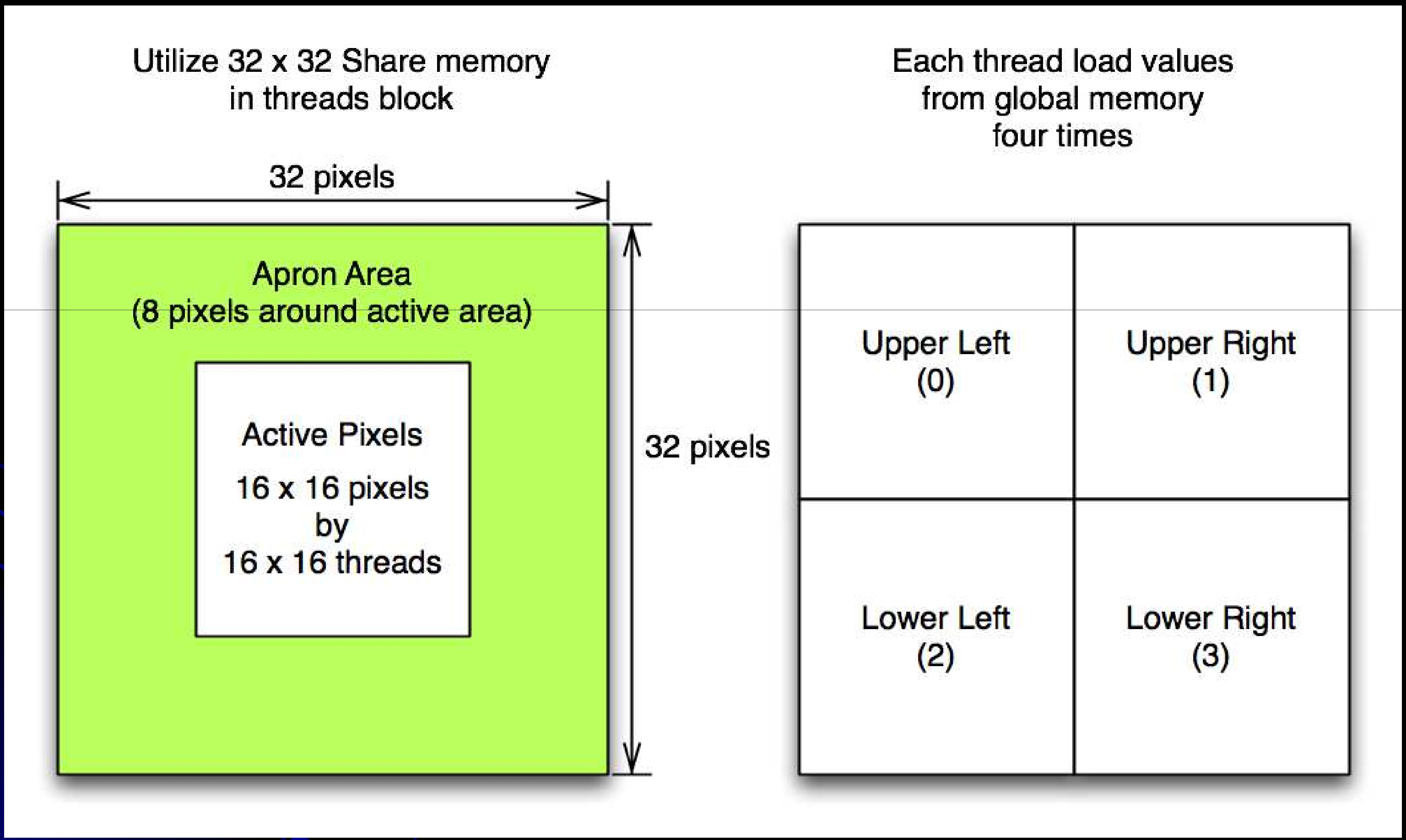
1 pixel – 4 bytes

1 block – 9216 bytes

This is more than half of the available 16KB shared memory per multiprocessor on the G80 GPU.

Shared memory

Shared memory model for naive approach: each thread in block loads 4 values from the global memory. Therefore, total shared memory size is 4 times bigger than active convolution pixels area.



```
__global__ void convolutionGPU_shMem_kernel(float *d_Result, float *d_Data, int dataW, int
    dataH){

      // shared memory for the thread (active pixels + apron)
      __shared__ float s_Data[BLOCK_W + KERNEL_RADIUS * 2][BLOCK_W +
                                          KERNEL_RADIUS * 2];

      // thread indices; every thread loads four pixels
      int tx = threadIdx.x;
      int ty = threadIdx.y;

      // global memory address of the current thread in the whole grid
      const int gLoc = tx + blockIdx.x * blockDim.x + ty * dataW +
                            blockIdx.y * blockDim.y * dataW;

      // original image coordinates of the current thread
      int x0 = tx + blockIdx.x * blockDim.x;
      int y0 = ty + blockIdx.y * blockDim.y;

      // upper left
      int x = x0 - KERNEL_RADIUS;
      int y = y0 - KERNEL_RADIUS;
```

```
if( x < 0 || y < 0)
    s_Data[ty][tx] = 0;
else
    s_Data[ty][tx] = *(d_Data + gLoc - KERNEL_RADIUS – dataW * KERNEL_RADIUS);

// upper right
x = x0 + KERNEL_RADIUS;
y = y0 - KERNEL_RADIUS;

if( x > (dataW-1) || y < 0)
    s_Data[ty][tx + blockDim.x] = 0;
else
    s_Data[ty][tx + blockDim.x] = *(d_Data + gLoc + KERNEL_RADIUS –
                                    dataW * KERNEL_RADIUS);

// lower left
x = x0 - KERNEL_RADIUS;
y = y0 + KERNEL_RADIUS;

if( x < 0 || y > (dataH-1) )
    s_Data[ty + blockDim.y][tx] = 0;
else
    s_Data[ty + blockDim.y][tx] = *(d_Data + gLoc - KERNEL_RADIUS +
                                    dataW * KERNEL_RADIUS);
```

```
    // lower right
    x = x0 + KERNEL_RADIUS;
    y = y0 + KERNEL_RADIUS;

    if( x > (dataW-1) || y > (dataH-1) )
        s_Data[ty + blockDim.y][tx + blockDim.x] = 0;
    else
        s_Data[ty + blockDim.y][tx + blockDim.x] = *(d_Data + gLoc + KERNEL_RADIUS +
                                                      dataW * KERNEL_RADIUS);

    __syncthreads();

    // convolution itself
    float sum = 0;

    // index of the current thread in an active pixels area
    x = KERNEL_RADIUS + tx;
    y = KERNEL_RADIUS + ty;

    for(int i = -KERNEL_RADIUS; i <= KERNEL_RADIUS; ++i)
        for(int j = -KERNEL_RADIUS; j <= KERNEL_RADIUS; ++j)
            sum += s_Data[y + i][x + j] * d_KernelDev[KERNEL_RADIUS + i] *
                       d_KernelDev[KERNEL_RADIUS + j];

    d_Result[gLoc] = sum;
}
```

Separable filters

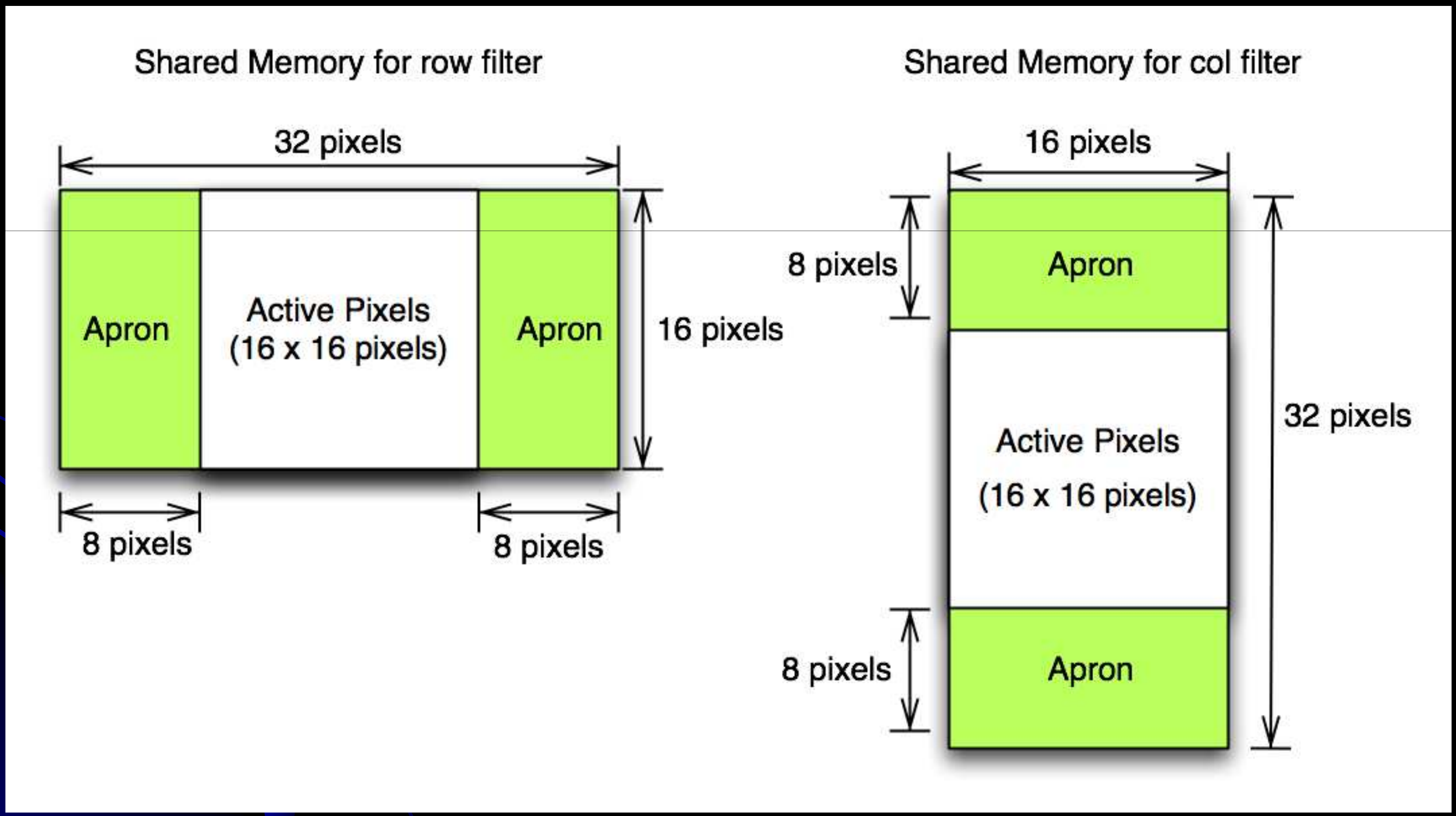
Generally, a two-dimensional convolution filter requires **n*m** multiplications for each output pixel, where **n** and **m** are the width and height of the filter kernel. **Separable filters** are a special type of filter that can be expressed as the composition of two one-dimensional filters, one on the rows on the image, and one on the columns.

Applying $\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$ to the data is the same as applying $\begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} + \begin{bmatrix} -1 & 0 & 1 \end{bmatrix}$

Separable filter requires only **n+m** multiplications for each output pixel. Separable filters have the benefit of offering more flexibility in the implementation and in addition reducing the arithmetic complexity and bandwidth usage of the computation for each data point.

Filter separation

Basically two separate convolutions are applied. The first one is row-wise and the second one is column-wise from the first result data (apply column convolution over row-wised filtered data). This also reduces some of conditional statements and total number of apron pixels, since vertical apron in row-convolution kernel and horizontal apron in column-convolution kernel do not need to be considered.



```
__global__ void convolutionRowGPU_kernel(float *d_Result, float *d_Data, int dataW, int
                                                          dataH){

    int ty = threadIdx.y;
    int tx = threadIdx.x;

    // each thread loads two values from global memory into shared mem
    __shared__ float s_Data[BLOCK_H][BLOCK_W + 2 * KERNEL_RADIUS];

    // global memory address of the current thread in the whole grid
    const int gLoc = tx + blockIdx.x * blockDim.x + ty * dataW +
                     blockIdx.y * blockDim.y * dataW;

    // original image based coordinate
    const int x0 = tx + blockIdx.x * blockDim.x;

    // case1: left
    int x = x0 - KERNEL_RADIUS;

    if ( x < 0 )
      s_Data[ty][tx] = 0;
    else
      s_Data[ty][tx] = d_Data[gLoc - KERNEL_RADIUS];
```

```
// case2: right
x = x0 + KERNEL_RADIUS;

if ( x > dataW-1 )
    s_Data[ty][tx + blockDim.x] = 0;
else
    s_Data[ty][tx + blockDim.x] = d_Data[gLoc + KERNEL_RADIUS];

__syncthreads();

// convolution
float sum = 0;

x = KERNEL_RADIUS + tx;

for (int i = -KERNEL_RADIUS; i <= KERNEL_RADIUS; i++)
    sum += s_Data[ty][x+i] * d_KernelDev[KERNEL_RADIUS + i];

d_Result[gLoc] = sum;

}
```

```
__global__ void convolutionColGPU_kernel(float *d_Result, float *d_Data, int dataW, int
                                                 dataH){

    int ty = threadIdx.y;
    int tx = threadIdx.x;

    // each thread loads two values from global memory into shared mem
    __shared__ float s_Data[BLOCK_H + KERNEL_RADIUS * 2][BLOCK_W];

    // global memory address of the current thread in the whole grid
    const int gLoc = tx + blockIdx.x * blockDim.x + ty * dataW +
                     blockIdx.y * blockDim.y * dataW;

    // original image based coordinate
    const int y0 = ty + blockIdx.y * blockDim.y;

    // case1: upper
    int y = y0 - KERNEL_RADIUS;

    if ( y < 0 )
      s_Data[ty][tx] = 0;
    else
      s_Data[ty][tx] = d_Data[gLoc – dataW * KERNEL_RADIUS];
```

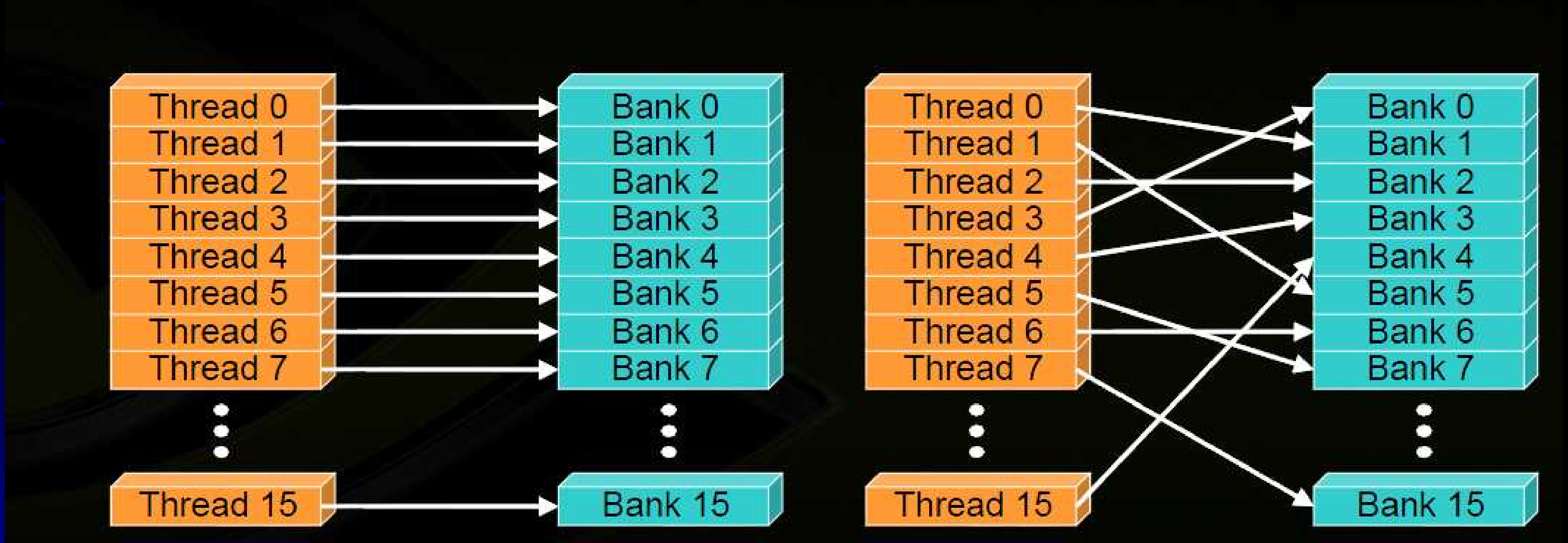
```
// case2: lower
 y = y0 + KERNEL_RADIUS;

 if ( y > dataH-1 )
     s_Data[ty + blockDim.y][tx] = 0;
 else
     s_Data[ty + blockDim.y][tx] = d_Data[gLoc + dataW * KERNEL_RADIUS];

 __syncthreads();

 // convolution
 float sum = 0;

 y = KERNEL_RADIUS + ty;

 for (int i = -KERNEL_RADIUS; i <= KERNEL_RADIUS; i++)
     sum += s_Data[y+i][tx] * d_KernelDev[KERNEL_RADIUS + i];

 d_Result[gLoc] = sum;

}
```

Shared memory and Bank conflicts

Shared memory is divided into equally sized memory modules (**banks**) that can be accessed simultaneously. Therefore, any memory load or store of **n** addresses that spans **n** distinct memory banks can be serviced simultaneously. However, if multiple addresses of a memory request map to the same memory bank, the accesses are serialized.

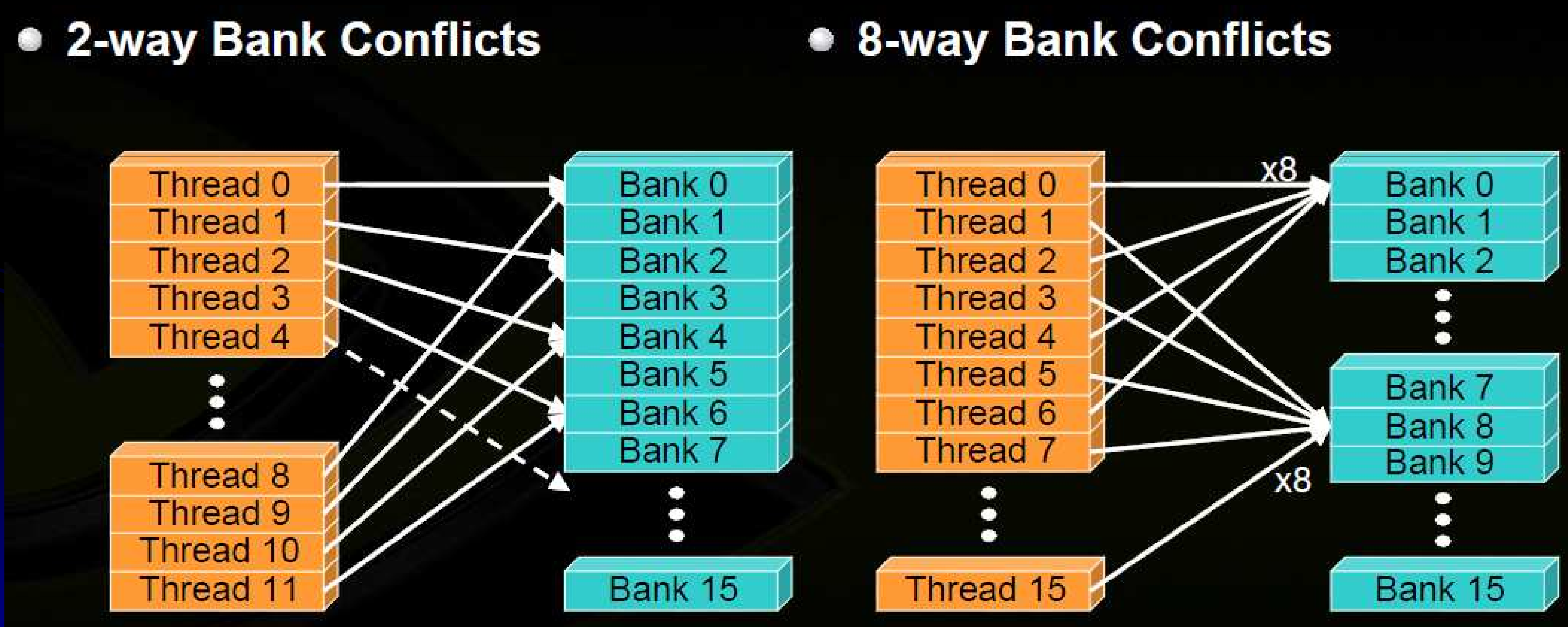
Access to shared memory should be designed to avoid serializing requests due to bank conflicts.

- No Bank Conflicts
- No Bank Conflicts



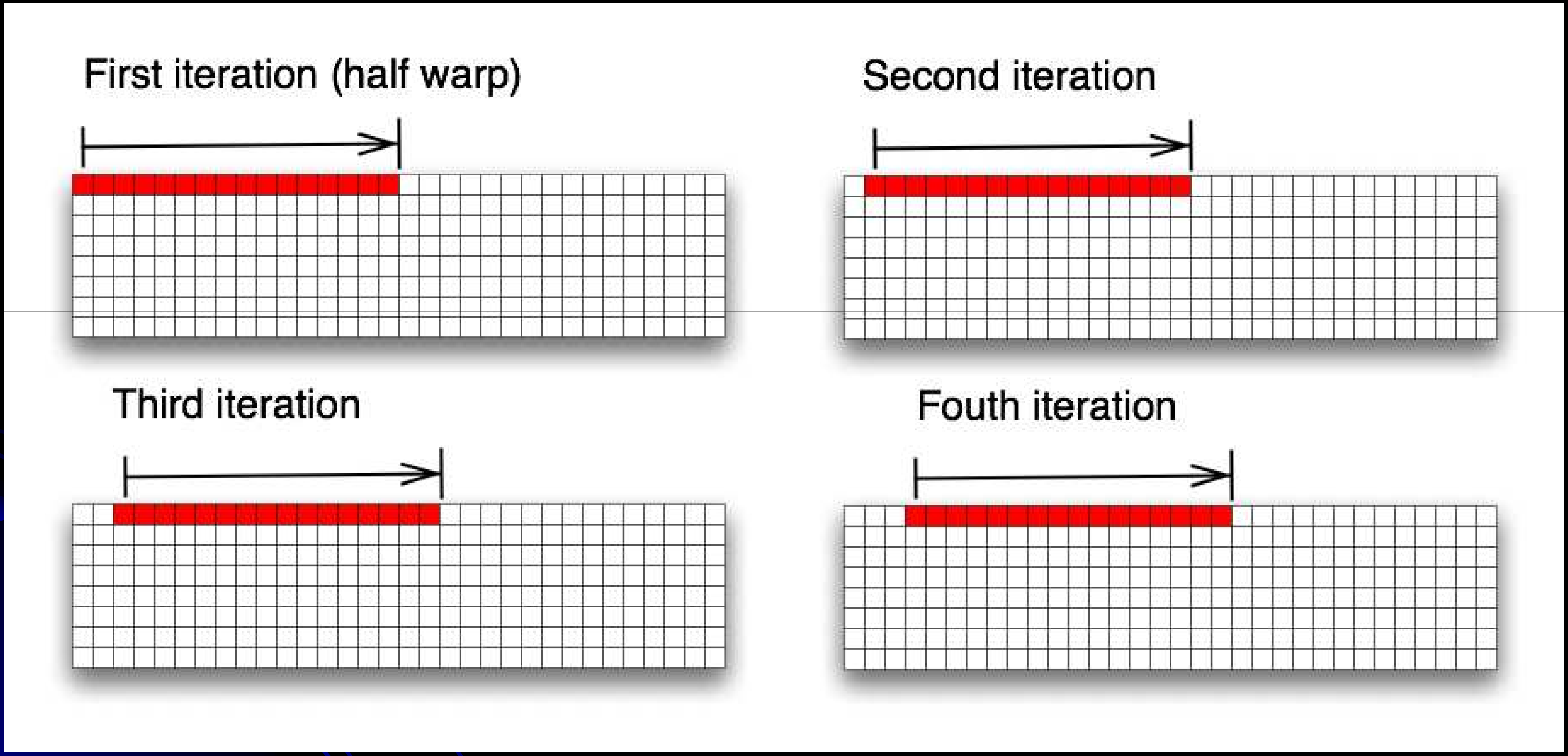
Shared memory and Bank conflicts

- 16 banks
- Successive 32-bit words belong to different banks
- Shared memory accesses are per 16-threads (half-warp)
- If **n** threads (out of 16) access the same bank, **n** accesses are executed serially



Reorganize shared memory

1D shared memory access pattern for a row filter. The first four iterations of the convolution computation. Red area is indicating values accessed by half warp threads.



```
__global__ void convolutionRowGPU_optimized_kernel(float *d_Result, float *d_Data, int
                                                            dataW, int dataH){

     int ty = threadIdx.y;
     int tx = threadIdx.x;

     // shared memory represented here by 1D array
     // each thread loads two values from global memory into shared mem
     __shared__ float s_Data[BLOCK_H * (BLOCK_W + 2 * KERNEL_RADIUS)];

     // global memory address of the current thread in the whole grid
     const int gLoc = tx + blockIdx.x * blockDim.x + ty * dataW +
                         blockIdx.y * blockDim.y * dataW;

     // original image based coordinate
     const int x0 = tx + blockIdx.x * blockDim.x;
     const int shift = ty * (BLOCK_W + 2 * KERNEL_RADIUS);

     // case1: left
     int x = x0 - KERNEL_RADIUS;

     if ( x < 0 )
        s_Data[tx + shift] = 0;
     else
        s_Data[tx + shift] = d_Data[gLoc - KERNEL_RADIUS];
```

```
// case2: right
 x = x0 + KERNEL_RADIUS;

 if ( x > dataW-1 )
     s_Data[tx + blockDim.x + shift] = 0;
 else
     s_Data[tx + blockDim.x + shift] = d_Data[gLoc + KERNEL_RADIUS];

 __syncthreads();

 // convolution itself
 float sum = 0;

 x = KERNEL_RADIUS + tx;

 for (int i = -KERNEL_RADIUS; i <= KERNEL_RADIUS; i++)
     sum += s_Data[x + i + shift] * d_KernelDev[KERNEL_RADIUS + i];

 d_Result[gLoc] = sum;
}
```

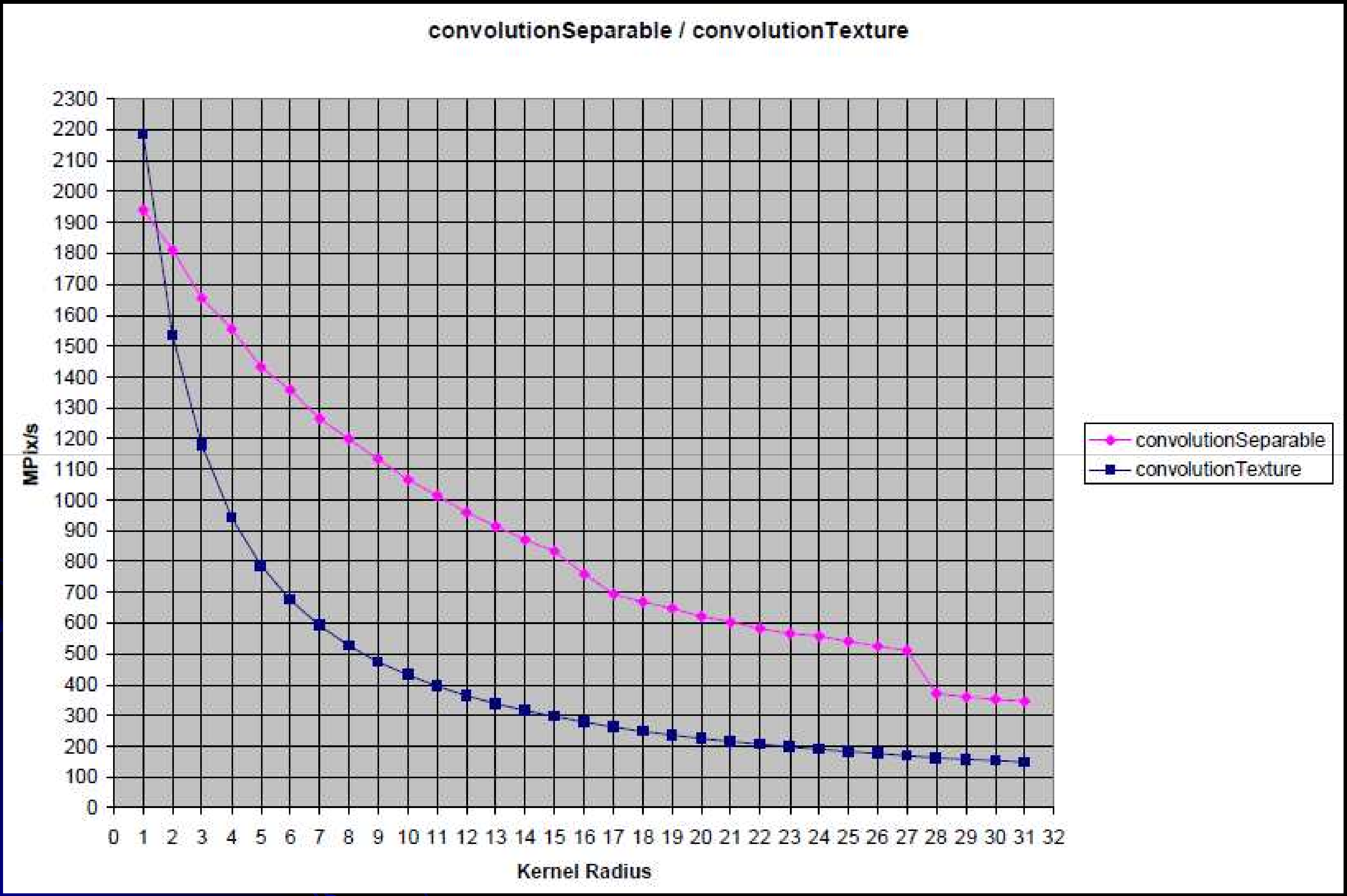
```
__global__ void convolutionColGPU_optimized_kernel(float *d_Result, float *d_Data, int
                                                                     dataW, int dataH){

    int ty = threadIdx.y;
    int tx = threadIdx.x;

    // shared memory represented here by 1D array
    // each thread loads two values from global memory into shared mem
    __shared__ float s_Data[BLOCK_W * (BLOCK_H + KERNEL_RADIUS * 2)];

    // global mem address of this thread
    const int gLoc = tx + blockIdx.x * blockDim.x + ty * dataW +
                           blockIdx.y * blockDim.y * dataW;

    // original image based coordinate
    const int y0 = ty + blockIdx.y * blockDim.y;
    const int shift = ty * (BLOCK_W);

    // case1: upper
    int y = y0 - KERNEL_RADIUS;

    if ( y < 0 )
       s_Data[tx + shift] = 0;
    else
       s_Data[tx + shift] = d_Data[ gLoc – dataW * KERNEL_RADIUS];
```

```
// case2: lower
y = y0 + KERNEL_RADIUS;
const int shift1 = shift + blockDim.y * BLOCK_W;

if ( y > dataH-1 )
    s_Data[tx + shift1] = 0;
else
    s_Data[tx + shift1] = d_Data[gLoc + dataW * KERNEL_RADIUS];

__syncthreads();

// convolution
float sum = 0;

for (int i = 0; i <= KERNEL_RADIUS*2; i++)
    sum += s_Data[tx + (ty + i) * BLOCK_W] * d_KernelDev[i];

d_Result[gLoc] = sum;

}
```

Runtime and speedups

	400 x 400 pixels		2000 x 2000 pixels	
Naive version (CPU)	91 ms		2330 ms	
Naive version (GPU)	27 ms	(3.3)	693 ms	(3.3)
Shared memory	26.9 ms	(1.003)	663 ms	(1.05)
Separable convolution	3 ms	(8.9)	72 ms	(9.2)
Optimized separable convolution	1.6 ms	(1.8)	38 ms	(1.9)

Time performance

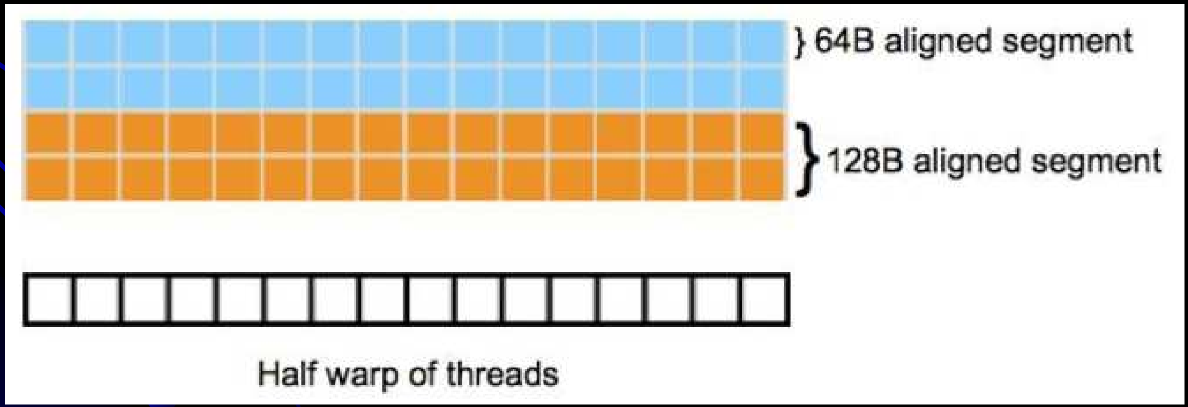


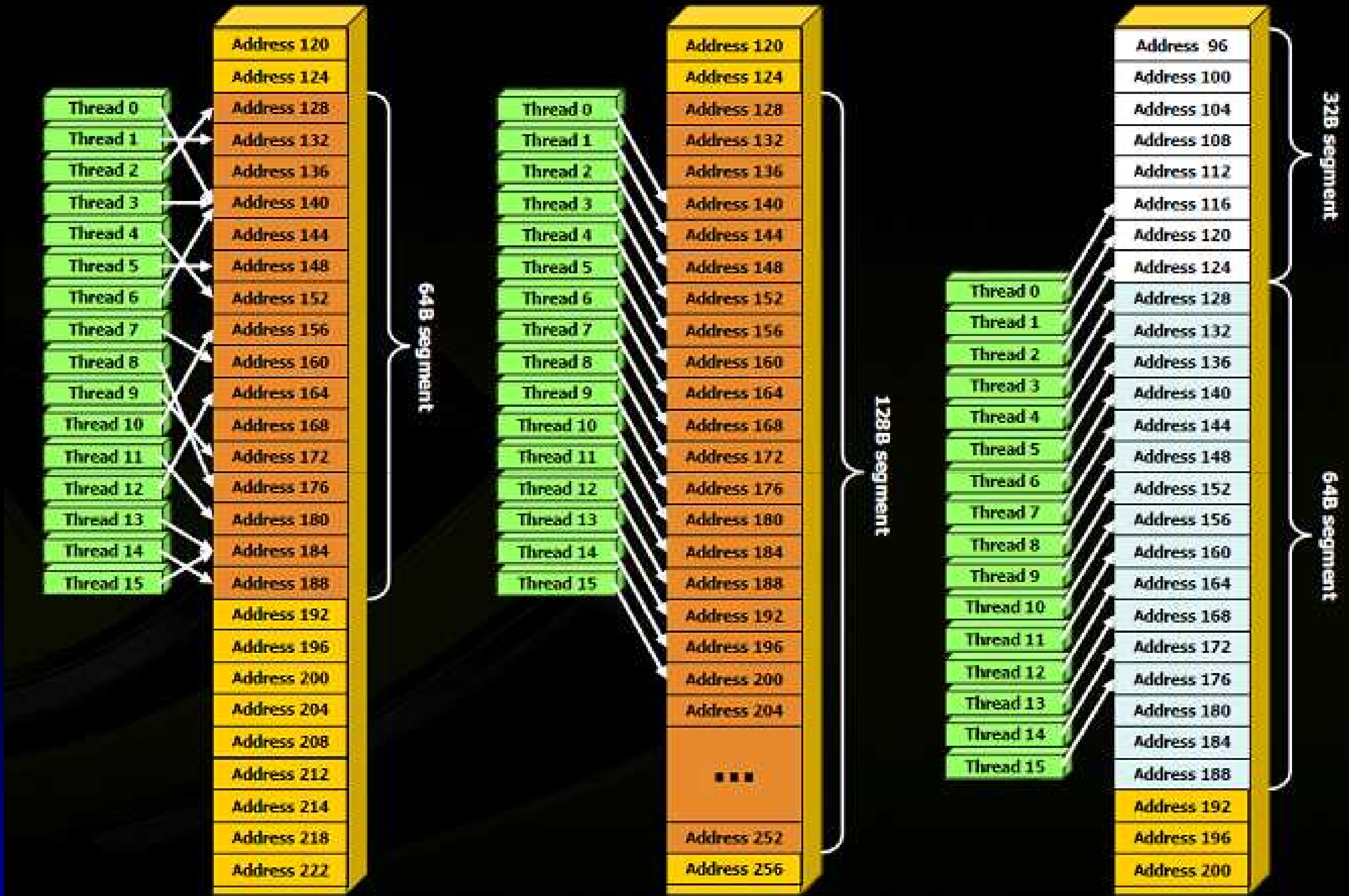
Coalesced access to Global memory

Global memory access by all threads in the half-warp of a block can be coalesced into efficient memory transactions on a G80 architectures when:

- Threads access 32-, 64-, 128-bit data types.
- All 16 words of the transaction must lie in the same segment of size equal to the memory transaction size (or twice the memory transaction size when accessing 128-bit words). This implies that the starting address and alignment are important.
- Threads must access words in sequence.

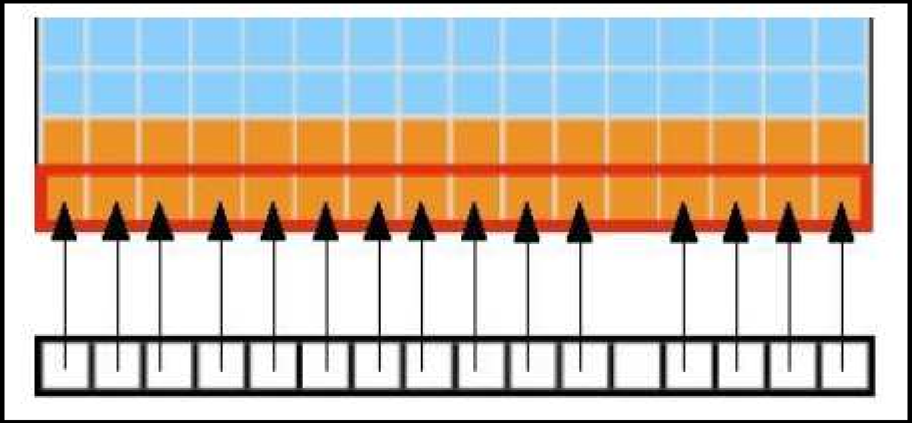
Coalescing of a half-warp of 32-bit words, such as floats (1.x)



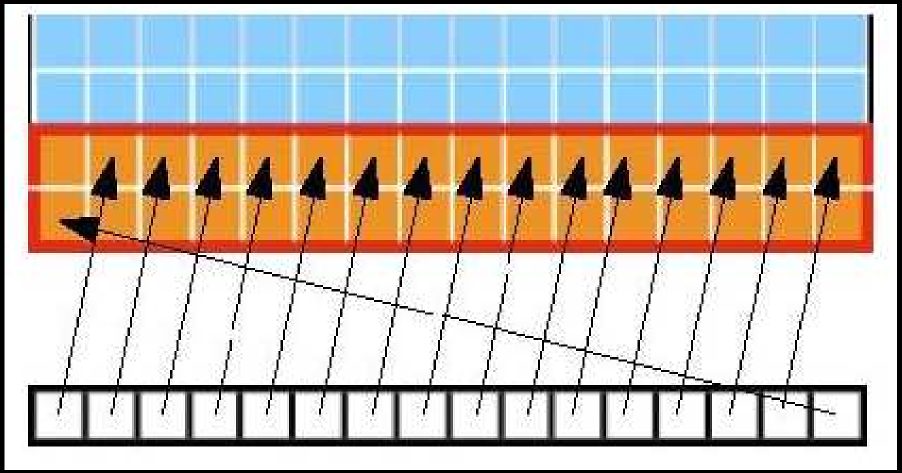

Address 120
Address 124
Address 128
Address 132
Address 136
Address 140
Address 144
Address 148
Address 152
Address 156
Address 160
Address 164
Address 168
Address 172
Address 176
Address 180
Address 184
Address 188
Address 192
Address 196
Address 200
Address 204
Address 208
Address 212
Address 214
Address 218
Address 222
Thread 0
Thread 1
Thread 2
Thread 3
Thread 4
Thread 5
Thread 6
Thread 7
Thread 8
Thread 9
Thread 10
Thread 11
Thread 12
Thread 13
Thread 14
Thread 15
64B segment
Address 120
Address 124
Address 128
Address 132
Address 136
Address 140
Address 144
Address 148
Address 152
Address 156
Address 160
Address 164
Address 168
Address 172
Address 176
Address 180
Address 184
Address 188
Address 192
Address 196
Address 200
Address 204
...
Address 252
Address 256
Thread 0
Thread 1
Thread 2
Thread 3
Thread 4
Thread 5
Thread 6
Thread 7
Thread 8
Thread 9
Thread 10
Thread 11
Thread 12
Thread 13
Thread 14
Thread 15
128B segment
Address 96
Address 100
Address 104
Address 108
Address 112
Address 116
Address 120
Address 124
Address 128
Address 132
Address 136
Address 140
Address 144
Address 148
Address 152
Address 156
Address 160
Address 164
Address 168
Address 172
Address 176
Address 180
Address 184
Address 188
Address 192
Address 196
Address 200
Thread 0
Thread 1
Thread 2
Thread 3
Thread 4
Thread 5
Thread 6
Thread 7
Thread 8
Thread 9
Thread 10
Thread 11
Thread 12
Thread 13
Thread 14
Thread 15
32B segment
64B segment

Coalesced access to Global memory

The simplest case of coalescing: the **k**-th thread accesses the **k**-th word in a segment. Not all threads need to participate. This access results in a single 64B transaction.

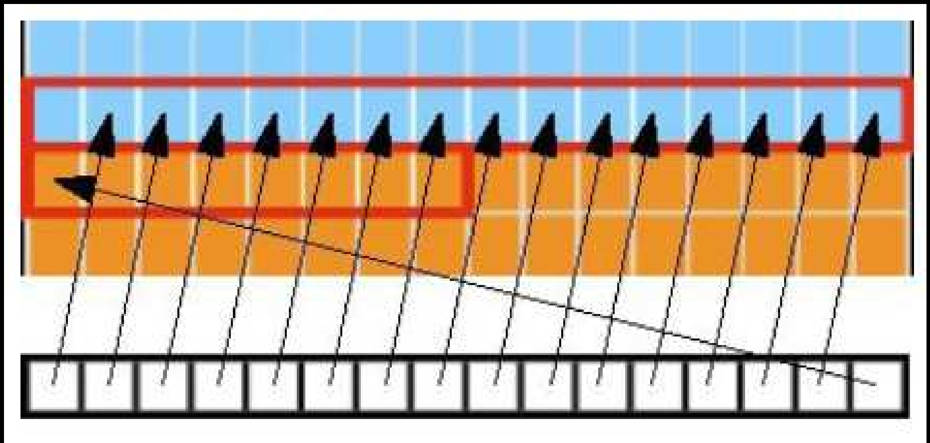


Unaligned sequential addresses that fit within a single 128-byte segment.

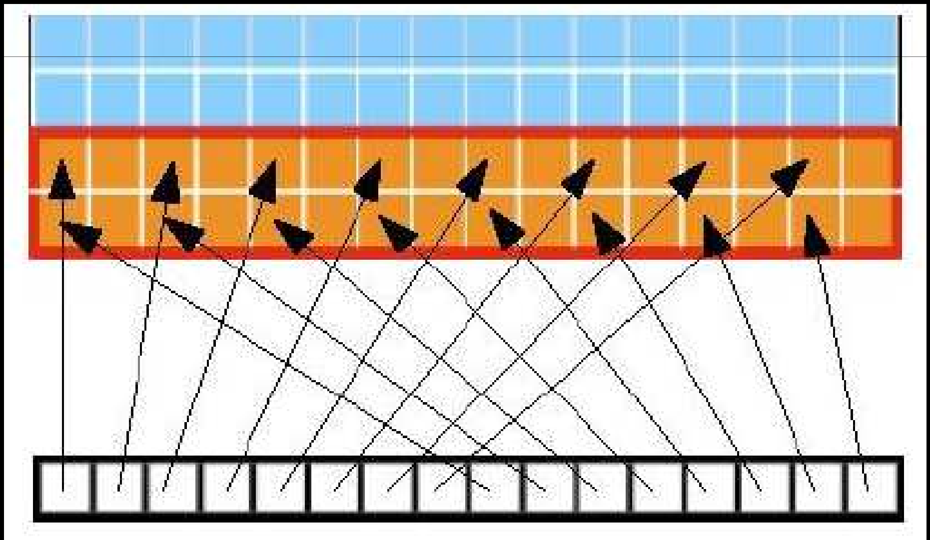


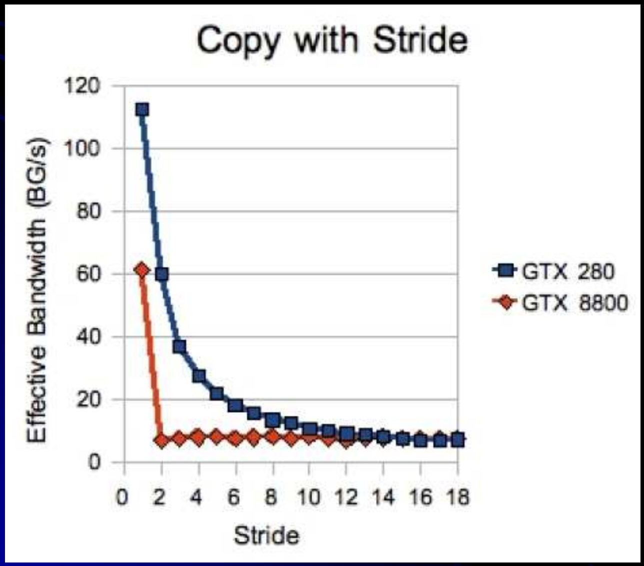
Coalesced access to Global memory

If a half warp accesses memory that is sequential but split across two 128B segments, then two transactions are performed. One 64B transaction and one 32B transaction.



A half-warp accessing memory with a stride of 2.





Unrolling loops

By default, the compiler unrolls small loops with a known trip count. The **#pragma unroll** directive however can be used to control unrolling of any given loop. It must be placed Immediately before the loop and only applies to that loop. It is optionally followed by a number that specifies how many times the loop must be unrolled.

```
# pragma unroll 5
for(int i = 0; i < n; ++i)
```

Fast multiplication

__mul24(x, y) multiplies two 24-bit integer values **x** and **y**. **x** and **y** are 32-bit integers but only the low 24 bits are used to perform the multiplication. **mul24** should only be used when values in **x** and **y** are in the range $[-2^{23}, 2^{23} - 1]$, if **x** and **y** are signed integers and in the range $[0, 2^{24} - 1]$, if **x** and **y** are unsigned integers. If **x** and **y** are not in this range, the multiplication result is implementation-defined.

Fast integer functions can be used for optimizing performance of kernels.

Bibliography

- **Wolfram Mathworld**, “Convolution”, http://mathworld.wolfram.com/Convolution.html
- **WolframMathworld,** “Normal Distribution”, http://mathworld.wolfram.com/NormalDistribution.html
- **DrDobbs,** http://drdobbs.com/
- **Victor Podlozhnyuk,** “Image Convolution with CUDA”, NVIDIA CUDA SDK, convolutionSeparable document
- **NVIDIA CUDA Programming Guide**
- **CUDA C Best Practices Guide**

Thank you for your attention !

QUESTIONS ?

Göttingen, 9.03.2011