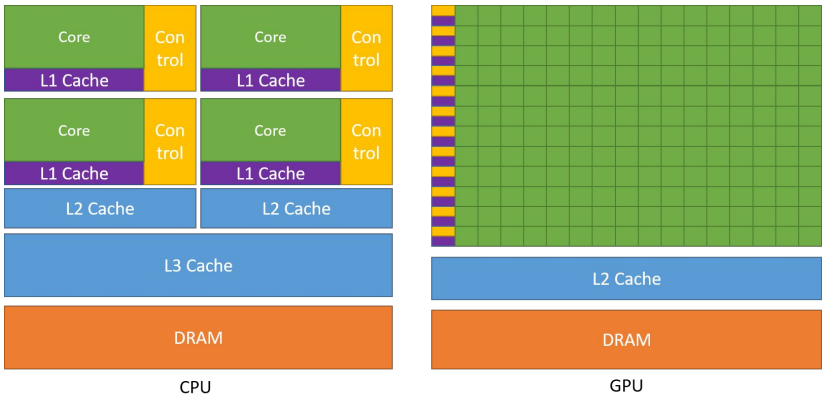
GPU Programming

25.12.03

CPU vs GPU Structure

- CPU는 하나의 쓰레드를 관리하고 실행하는데 수천 Cycle의 instruction이 소요됨. (CPU -> 소수의 쓰레드 사용)
- GPU는 쓰레드 관리가 단순하며 실행에 수 Cycle의 instruction만 소요됨. (GPU -> 대량의 쓰레드 운용 가능)
- CPU는 범용코어로 방대한 량의 instruction을 처리함. 메모리 접근 지연시간을 줄이기 위한 복잡한 캐시 구조 때문에 CPU와 메모리 사이의 대역폭을 늘리는데 한계가 존재함. (CPU -> 메모리 접근 지연시간이 짧으나 대역폭 제한)
- GPU의 경우 코어마다 소수의 특정 코드만 수행하며 되기 때문에 캐시 메모리가 간단하여 GPU 코어와 GPU 메모리 사이의 대역폭을 늘리는데 수월함. (GPU -> 메모리 접근 지연시간이 길지만 대역폭이 큼)



Thread Block Abstraction

- Kernel은 하나 Grid로 구성되며 하나의 GPU에 할당되어 실행
- Thread Block은 하나의 SM에 할당되어 실행 (SM은 여러개의 Thread Block 실행 가능)
- Thread는 하나의 Core 에 할당되어 실행
- Thread Block은 최대 1024개의 Thread로 구성가능

	Executed on the:	Only callable from the:
`__device__ float DeviceFunc()`	device	device
`__global__ void KernelFunc()`	device	host
`__host__ float HostFunc()`	host	host

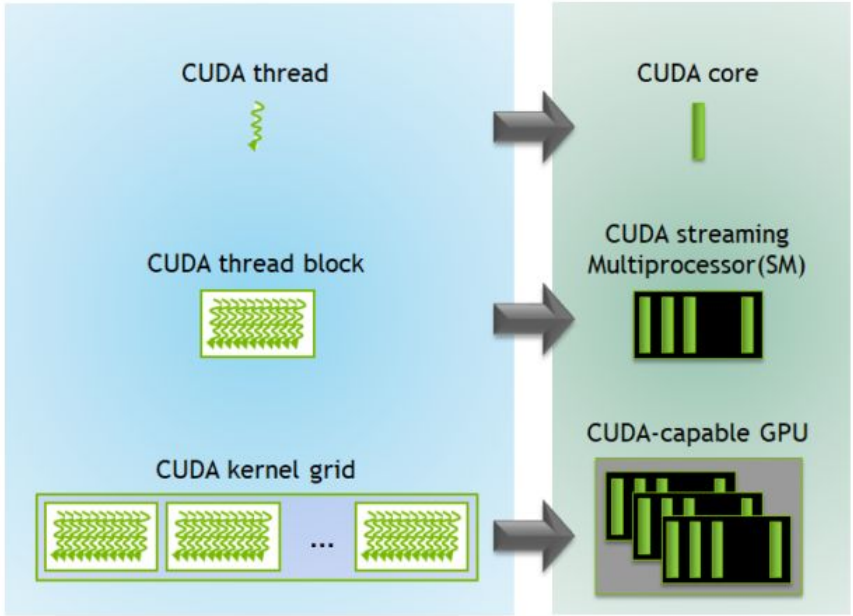
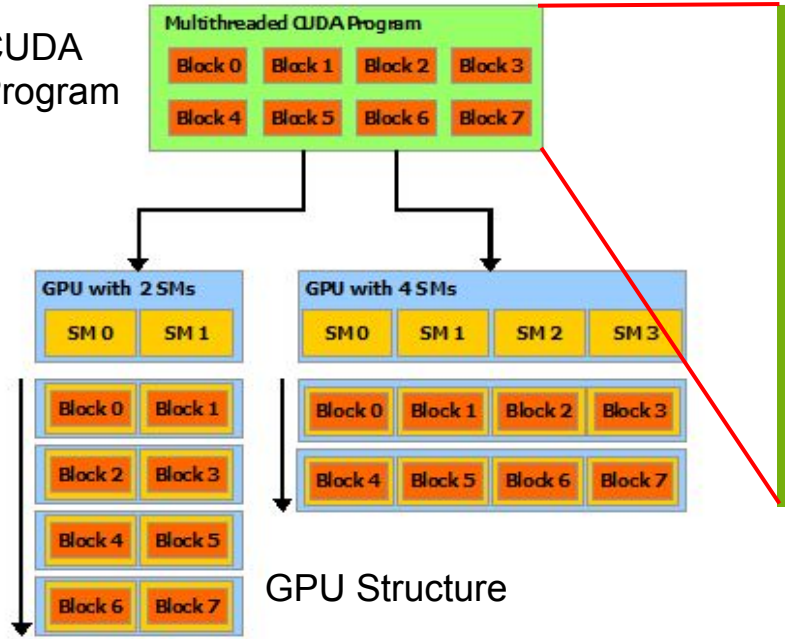


Figure 3. Kernel execution on GPU.

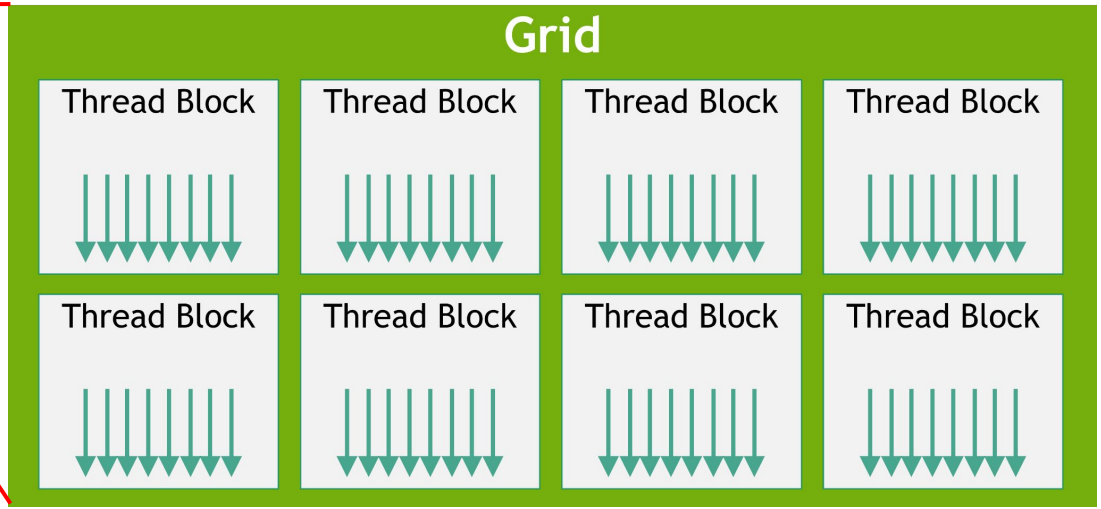
Thread Block Abstraction

사용자는 GPU의 가용 물리 Processor 개수에 상관없이 프로그래밍 가능하며, 각 Thread Block은 런타임에 자동으로 SM(Streaming Multiprocessor)들에 할당되어 실행됨. (하나의 SM이 하나의 Thread Block을 실행)

CUDA Program



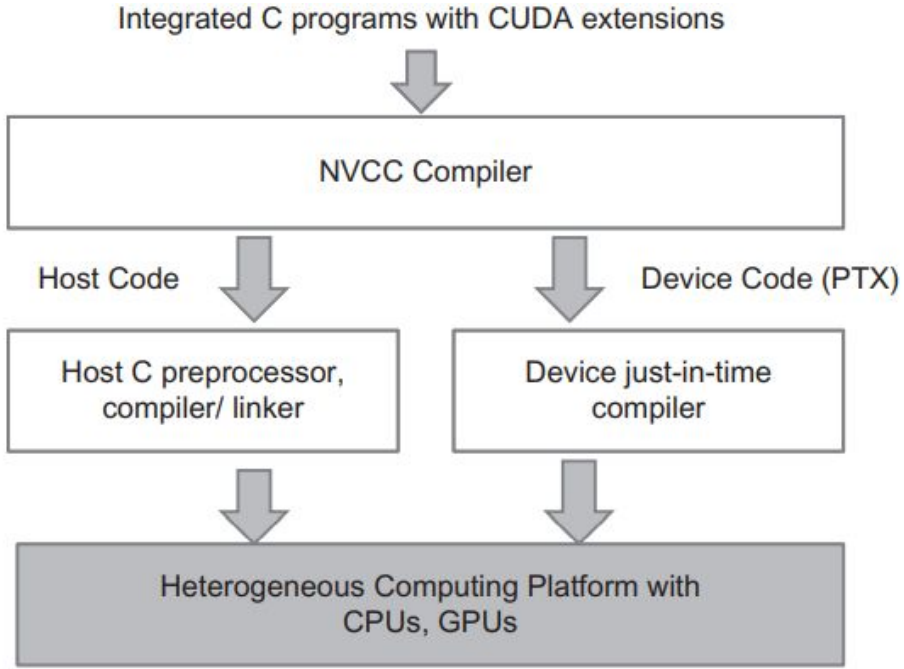
GPU Structure



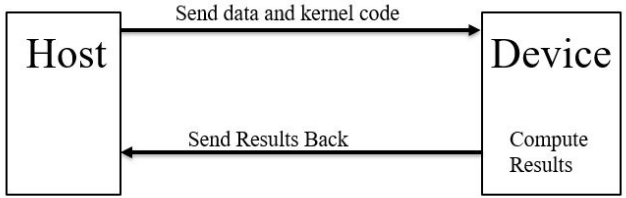
Thread Blocks

CUDA Build Process

NVCC (NVIDIA C Compiler)



CUDA Host-device Model & Memory Management

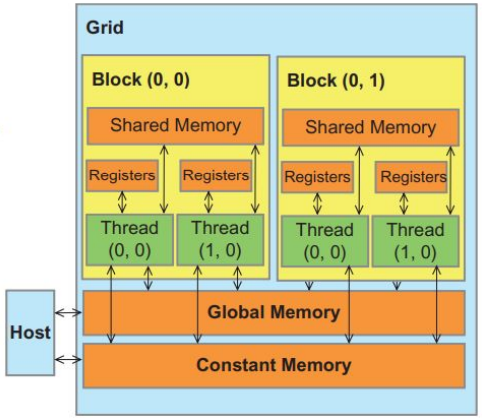


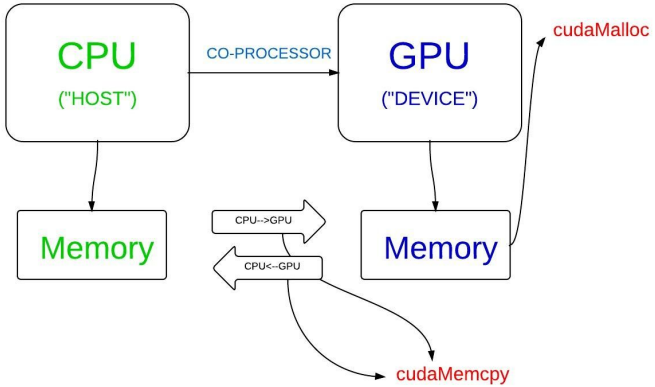
Device code can:

- R/W per-thread registers
- R/W per-thread local memory
- R/W per-block shared memory
- R/W per-grid global memory
- Read only per-grid constant memory

Host code can

- Transfer data to/from per grid global and constant memories





Variable declaration	Memory	Scope	Lifetime
Automatic variables other than arrays	Register	Thread	Kernel
Automatic array variables	Local	Thread	Kernel
`__device__ __shared__ int SharedVar;`	Shared	Block	Kernel
`__device__ int GlobalVar;`	Global	Grid	Application
`__device__ __constant__ int ConstVar;`	Constant	Grid	Application

```
cudaMalloc(void** devPtr, size_t size )
cudaMemcpy(void* dst, const void* src, size_t count, cudaMemcpyKind kind)
cudaFree (void* devPtr)
```

Device의 global 메모리는 속도가 느리기 때문에 적절한 타입의 변수를 사용하여 적절한 메모리를 사용하도록 설계가 필요함

CUDA Memory Access Optimization Technique

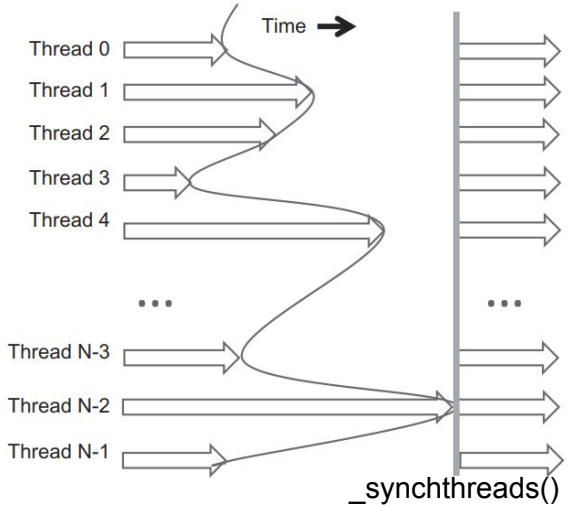
- Tiling
  - Device Global Memory의 데이터를 Shared Memory로 옮겨 처리하는 기법
  - 전역 메모리 Access 횟수를 효과적으로 줄일 수 있음.
  - 전역 메모리 Access 횟수 최소화하는 것 만으로도 벤치마크 상 수배의 속도 향상 가능

- 메모리 병합기법
  - 현대 메모리는 대역폭을 높이기 위해 한번의 Access에 인접한 위치의 메모리 주소를 동시에 접근 할 수 있게 설계됨.
  - 메모리 접근시 인접한 주소를 함께 접근하면 메모리 Access시 instruction이 병합되어 한번의 memory cycle에 처리 될 수 있도록 최적화 됨.

Barrier Synchronization

Barrier Synchronization: 모든 대상이 특정 코드를 실행할 때까지 대기함.

- Host-device Sync
  - cudaDeviceSynchronize(): device에서 현재 실행중인 kernel이 모두 종료될때까지 host의 코드 실행 대기
- Thread Sync
  - _syncthreads(): Thread Block 내의 모든 쓰레드가 해당 코드를 실행할때 까지 쓰레드 실행 대기



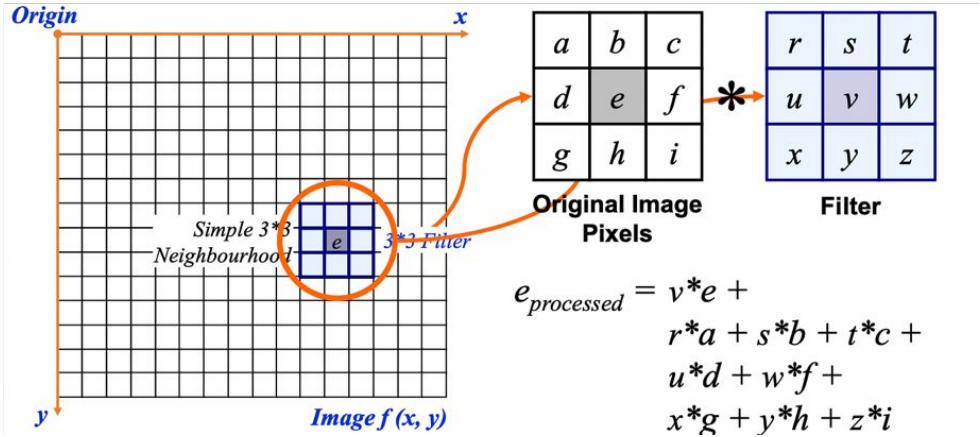
Race Condition Handling

- atomicAdd(): 해당 메모리 위치에 대한 atomic add 수행

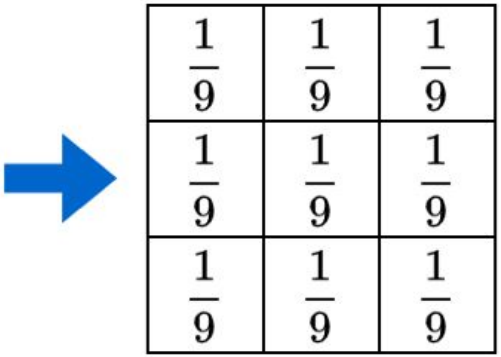
Spatial Filtering

- **Spatial Filtering**
  - Spatial filtering is defined as the process of applying a filter (kernel) to an image by sliding it over a neighborhood of pixels, performing element-wise multiplication, and summing the results.

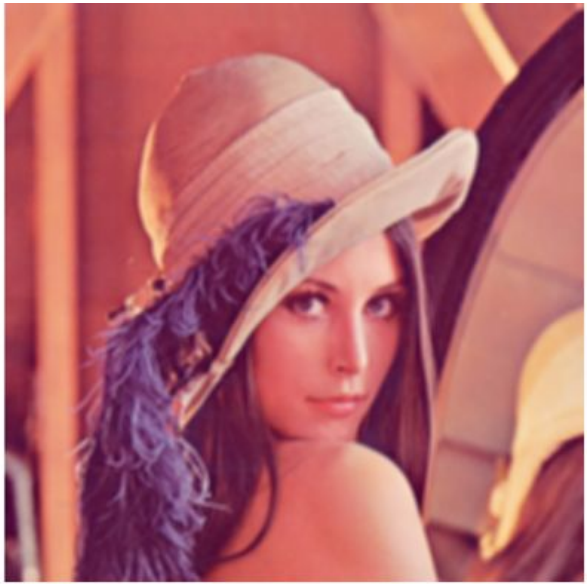
$$g(x,y) = \sum_{i=-k}^{k} \sum_{j=-k}^{k} f(x+i, y+j) \cdot h(i,j)$$



Spatial Filtering



$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{9}$
$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{9}$
$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{9}$



```cpp
// gaussian_blur_naive.cu

#include <cstdio>
// 3x3 Gaussian kernel (1/16 normalization은 나중에 곱해줌)
__constant__ float d_gaussKernel[9];

#define CHECK_CUDA(call)                                                \
    do {                                                                \
        cudaError_t err = call;                                         \
        if (err != cudaSuccess) {                                       \
            fprintf(stderr, "CUDA Error %s (code %d), line %d\n",       \
                    cudaGetErrorString(err), err, __LINE__);            \
            exit(EXIT_FAILURE);                                         \
        }                                                               \
    } while (0)
```

```cpp
__global__ void gaussianBlurNaive(const float* input, float* output,
                                  int width, int height)
{
    int x = blockIdx.x * blockDim.x + threadIdx.x; // column
    int y = blockIdx.y * blockDim.y + threadIdx.y; // row

    if (x >= width || y >= height) return;

    // 경계 처리: 이미지 밖은 그냥 0으로 본다고 가정 (zero padding)
    float sum = 0.0f;

    for (int ky = -1; ky <= 1; ++ky) {
        for (int kx = -1; kx <= 1; ++kx) {
            int ix = x + kx;
            int iy = y + ky;

            if (ix >= 0 && ix < width && iy >= 0 && iy < height) {
                float pixel = input[iy * width + ix];
                int kIndex = (ky + 1) * 3 + (kx + 1);
                float k = d_gaussKernel[kIndex];
                sum += k * pixel;
            }
        }
    }

    // 1/16로 정규화
    output[y * width + x] = sum / 16.0f;
}
```

```cpp
#include <cstdio>
#include <vector>

extern __constant__ float d_gaussKernel[9];
__global__ void gaussianBlurNaive(const float* input, float* output,
                                  int width, int height);

int main() {
    int width = 1024;
    int height = 768;

    // 호스트 메모리 (예: 0~1 정규화된 그레이스케일)
    std::vector<float> h_input(width * height, 0.0f);
    std::vector<float> h_output(width * height, 0.0f);

    // 테스트용으로 중앙에 흰 점 하나
    h_input[(height/2) * width + (width/2)] = 1.0f;
    // Gaussian kernel 값
    float h_kernel[9] = {
        1, 2, 1,
        2, 4, 2,
        1, 2, 1
    };

    // constant memory에 kernel 업로드
    CHECK_CUDA(cudaMemcpyToSymbol(d_gaussKernel, h_kernel,
                                  9 * sizeof(float)));

    float *d_input = nullptr, *d_output = nullptr;
    CHECK_CUDA(cudaMalloc(&d_input,  width * height * sizeof(float)));
    CHECK_CUDA(cudaMalloc(&d_output, width * height * sizeof(float)));

    CHECK_CUDA(cudaMemcpy(d_input, h_input.data(),
                          width * height * sizeof(float),
                          cudaMemcpyHostToDevice));
    dim3 block(16, 16);
    dim3 grid( (width + block.x - 1) / block.x,
               (height + block.y - 1) / block.y );

    gaussianBlurNaive<<<grid, block>>>(d_input, d_output, width, height);
    CHECK_CUDA(cudaDeviceSynchronize());

    CHECK_CUDA(cudaMemcpy(h_output.data(), d_output,
                          width * height * sizeof(float),
                          cudaMemcpyDeviceToHost));

    // 여기서 h_output을 파일로 저장하거나 시각화
    printf("Naive blur finished.\n");

    cudaFree(d_input);
    cudaFree(d_output);
    return 0;
}
```

CUDA 컴파일

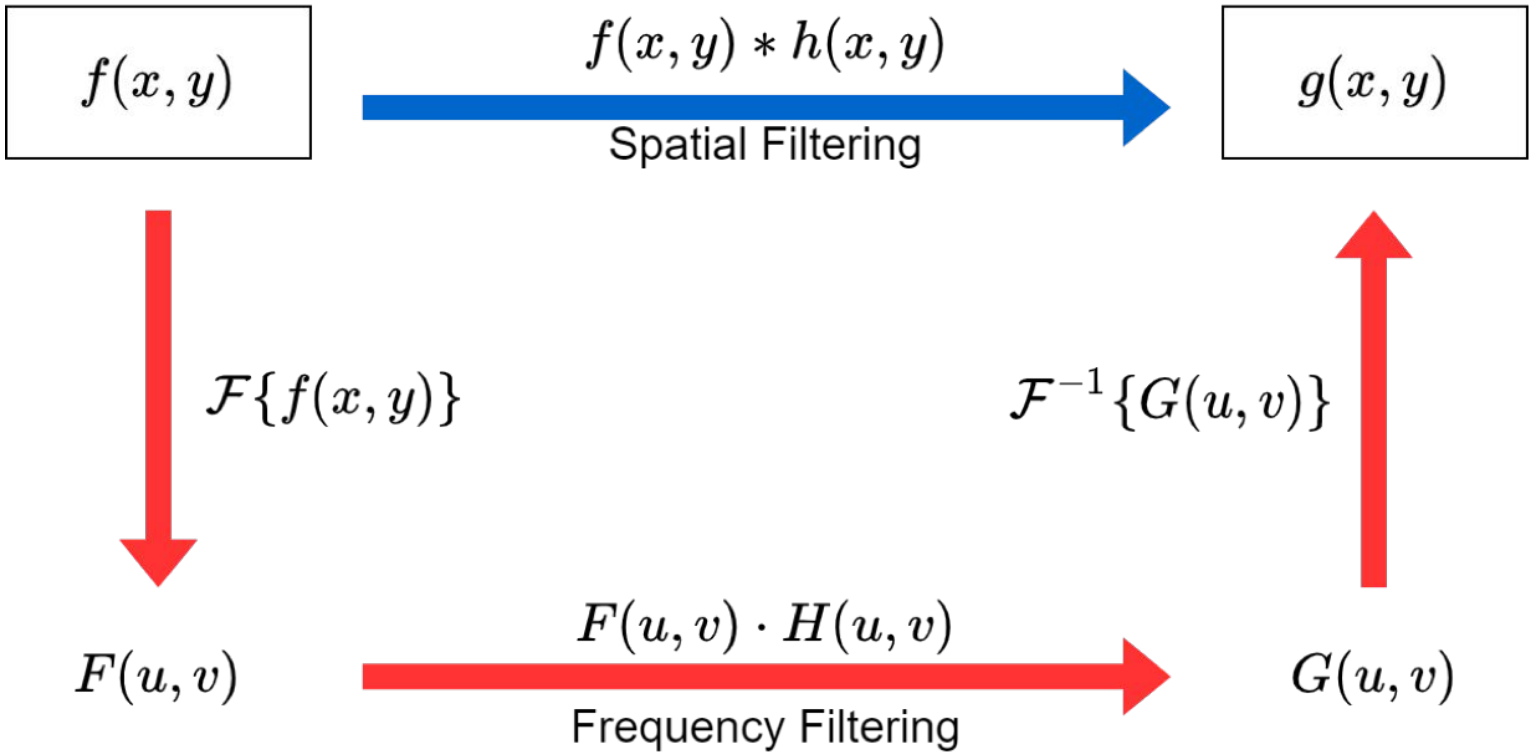
nvcc -o blur_naive gaussian_blur_naive.cu main_naive.cpp

```cpp
template<int BLOCK_W, int BLOCK_H>
__global__ void gaussianBlurTiled(const float* input, float* output,
                                  int width, int height)
{
    // 타일 크기 + halo (좌우/상하 한 줄씩)
    __shared__ float tile[BLOCK_H + 2][BLOCK_W + 2];

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

    int x = blockIdx.x * BLOCK_W + tx; // global x
    int y = blockIdx.y * BLOCK_H + ty; // global y

    // shared memory 내에서 자신의 "중심" 위치
    int sx = tx + 1; // halo 때문에 +1
    int sy = ty + 1;

    // 1) 먼저 중심 부분 로드
    float value = 0.0f;
    if (x >= 0 && x < width && y >= 0 && y < height) {
        value = input[y * width + x];
    }
    tile[sy][sx] = value;

    // 2) halo 로드
    // 왼쪽
    if (tx == 0) {
        int nx = x - 1;
        int ny = y;
        float v = 0.0f;
        if (nx >= 0 && nx < width && ny >= 0 && ny < height) {
            v = input[ny * width + nx];
        }
        tile[sy][sx - 1] = v;
    }

    // 오른쪽
    if (tx == BLOCK_W - 1) {
        int nx = x + 1;
        int ny = y;
        float v = 0.0f;
        if (nx >= 0 && nx < width && ny >= 0 && ny < height) {
            v = input[ny * width + nx];
        }
        tile[sy][sx + 1] = v;
    }
```

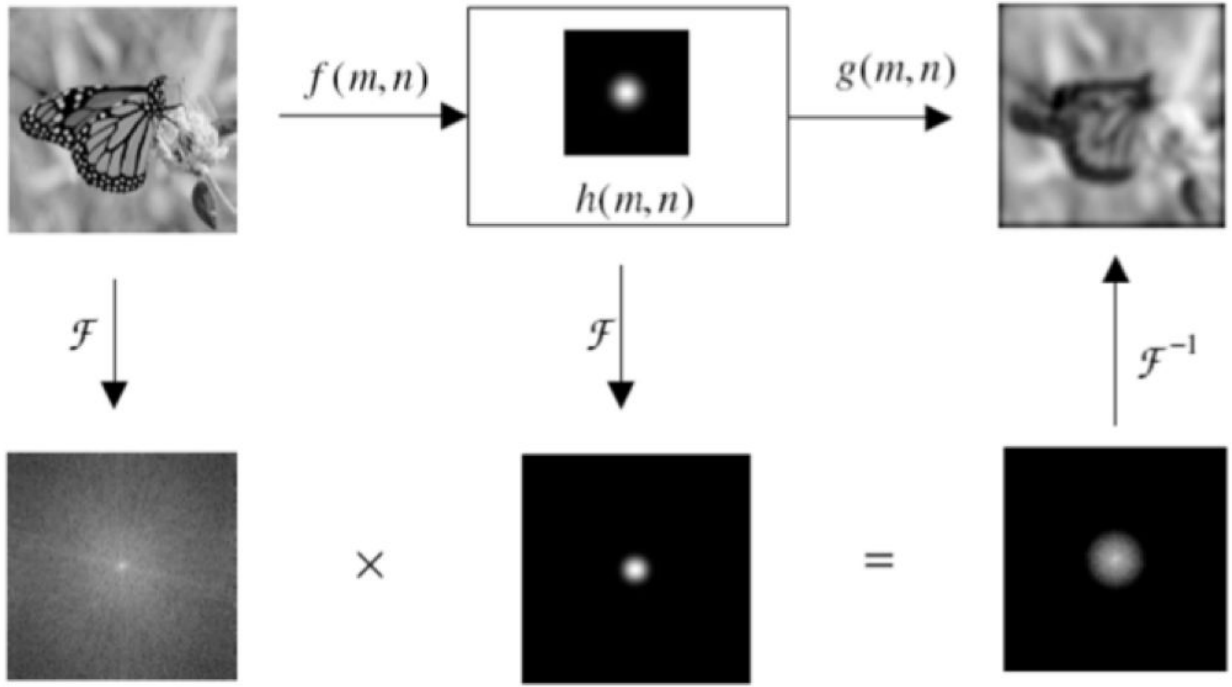
```
    // *** Barrier synchronization ***
    __syncthreads();

    // 3) 이제 shared memory에서 3x3 convolution 수행
    if (x < width && y < height) {
        float sum = 0.0f;

        for (int ky = -1; ky <= 1; ++ky) {
            for (int kx = -1; kx <= 1; ++kx) {
                int kIndex = (ky + 1) * 3 + (kx + 1);
                float k = d_gaussKernel[kIndex];

                float pixel = tile[sy + ky][sx + kx];
                sum += k * pixel;
            }
        }

        output[y * width + x] = sum / 16.0f;
    }
}
```

```
    dim3 block(BLOCK_W, BLOCK_H);
    dim3 grid( (width + BLOCK_W - 1) / BLOCK_W,
               (height + BLOCK_H - 1) / BLOCK_H );

    gaussianBlurTiled<BLOCK_W, BLOCK_H><<<grid, block>>>(
        d_input, d_output, width, height);
    CHECK_CUDA(cudaDeviceSynchronize());

    CHECK_CUDA(cudaMemcpy(h_output.data(), d_output,
                          width * height * sizeof(float),
                          cudaMemcpyDeviceToHost));

    printf("Tiled blur finished.\n");

    cudaFree(d_input);
    cudaFree(d_output);
    return 0;
}
```

Why do we need to know spatial filtering



Why do we need to know spatial filtering



Spatial Filtering in 2D Image

- **Smoothing**
  - Averaging Filter
  - Weighted Averaging Filter(Gaussian Filter)

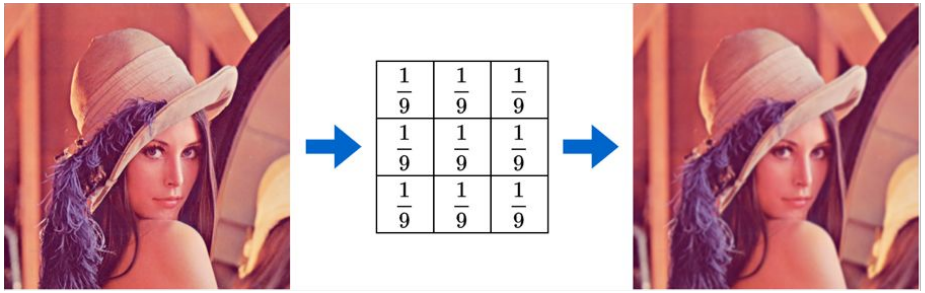
- **Edge Extraction**
  - Prewitt Filter
  - Sobel Filter
  - Second Deviation Filter(Laplacian Filter)

- **Sharpening**
  - Sharpening Filter

⋮

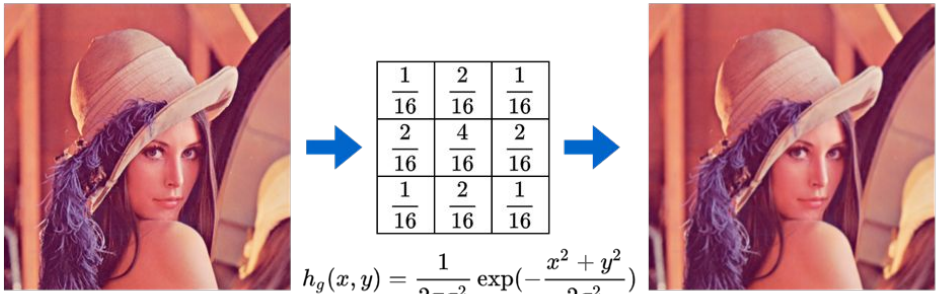
Smoothing Filter

- **Averaging Filter**



$$\begin{bmatrix} \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \end{bmatrix}$$

- **Weighted Averaging Filter**



$$\begin{bmatrix} \frac{1}{16} & \frac{2}{16} & \frac{1}{16} \\ \frac{2}{16} & \frac{4}{16} & \frac{2}{16} \\ \frac{1}{16} & \frac{2}{16} & \frac{1}{16} \end{bmatrix}$$

$$h_g(x,y) = \frac{1}{2\pi\sigma^2} \exp(-\frac{x^2+y^2}{2\sigma^2})$$

Edge Extraction Filter

- **Spatial filtering with derivative operations**

$$f'(x) = \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x} \text{ (for continuous signals)}$$

$$\rightarrow f'(x) \approx f(x+1) - f(x-1) \text{ (for discrete signals)}$$

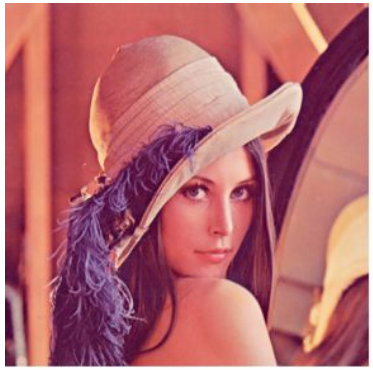
0	0	0
0	−1	1
0	0	0

0	0	0
−1	1	0
0	0	0

0	0	0
$-\frac{1}{2}$	0	$\frac{1}{2}$
0	0	0

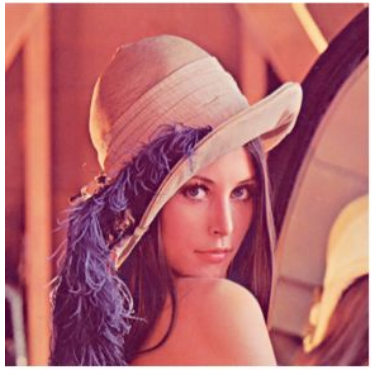
Example of derivative filter

Edge Extraction Filter



X-Axis

−1	0	1
−2	0	2
−1	0	1



Y-Axis

1	2	1
0	0	0
−1	−2	−1

Edge Extraction Filter

- **Sobel Filter**

$g_x(x, y)$

$g_y(x, y)$

Gradient Magnitude

$$\sqrt{g_x(x, y)^2 + g_y(x, y)^2}$$

Edge Extraction Filter

- **Second Derivative Filter (Laplacian Filter)**

$$\frac{\partial}{\partial x^2} f(x,y) + \frac{\partial}{\partial y^2} f(x,y)$$

0	0	0
0	−1	1
0	0	0

$\ominus$

0	0	0
−1	1	0
0	0	0

$=$

0	0	0
1	−2	1
0	0	0

X-Axis Second Derivative Filter

0	0	0
1	−2	1
0	0	0

X-Axis

$\oplus$

0	1	0
0	−2	0
0	1	0

Y-Axis

$=$

0	1	0
1	−4	1
0	1	0

Laplacian Filter

Edge Extraction Filter

- **Second Derivative Filter (Laplacian Filter)**

0	1	0
1	−4	1
0	1	0

Sharpening Filter

- **Sharpening Filter**

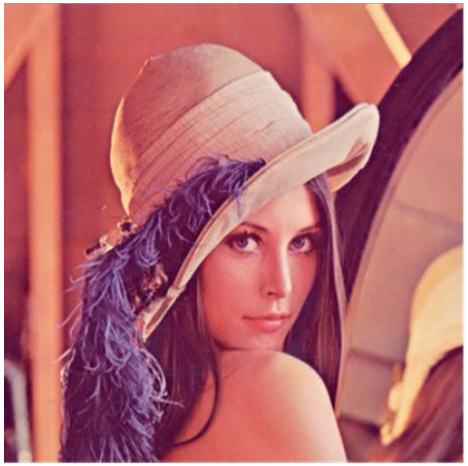
$$\underbrace{\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}}_{\text{Origianl Signal}} \ominus \underbrace{\begin{bmatrix} \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \end{bmatrix}}_{\text{Smooth Filter}} = \underbrace{\begin{bmatrix} -\frac{1}{9} & -\frac{1}{9} & -\frac{1}{9} \\ -\frac{1}{9} & \frac{8}{9} & -\frac{1}{9} \\ -\frac{1}{9} & -\frac{1}{9} & -\frac{1}{9} \end{bmatrix}}_{\text{Emphasis Edge}}$$

$$\underbrace{\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}}_{\text{Origianl Signal}} \oplus k\times \underbrace{\begin{bmatrix} -\frac{1}{9} & -\frac{1}{9} & -\frac{1}{9} \\ -\frac{1}{9} & \frac{8}{9} & -\frac{1}{9} \\ -\frac{1}{9} & -\frac{1}{9} & -\frac{1}{9} \end{bmatrix}}_{\text{Emphasis Edge}} = \underbrace{\begin{bmatrix} -\frac{k}{9} & -\frac{k}{9} & -\frac{k}{9} \\ -\frac{k}{9} & 1+\frac{8}{9}k & -\frac{k}{9} \\ -\frac{k}{9} & -\frac{k}{9} & -\frac{k}{9} \end{bmatrix}}_{\text{Sharpening Filter}}$$

$$\underbrace{\begin{bmatrix} -1 & -1 & -1 \\ -1 & 9 & -1 \\ -1 & -1 & -1 \end{bmatrix}}_{\text{Shapening Filter}(k=9)}$$

Sharpening Filter

- **Sharpening Filter**



−1	−1	−1
−1	9	−1
−1	−1	−1

Implementation of Spatial Filtering (CPU version)

- **Convolution Function**

```python
import numpy as np
import cv2

def convolution2d(image, kernel, kernel_size=3):
    if kernel.shape != (kernel_size, kernel_size):
        raise ValueError("Kernel size does not match kernel shape")

    h, w = image.shape

    pad_size = kernel_size // 2
    padded_image = np.pad(image, pad_size) # zero padding
    output = np.zeros_like(image, dtype=np.float64)

    for i in range(h):
        for j in range(w):
            region = padded_image[i:i + kernel_size, j:j + kernel_size]
            output[i, j] = np.sum(region * kernel)

    return output
```

Implementation of Spatial Filtering (CPU version)

- **Gaussian Smoothing & Sobel Filtering**

```python
def gaussian_smooth(image, kernel_size=3, sigma=1.0):
    kernel = cv2.getGaussianKernel(kernel_size, sigma)
    kernel = kernel @ kernel.T

    return convolution2d(image, kernel, kernel_size).astype(np.uint8)


def sobel_filter(image):
    sobel_x = np.array([[-1, 0, 1],
                        [-2, 0, 2],
                        [-1, 0, 1]], dtype=np.float64)

    sobel_y = np.array([[-1, -2, -1],
                        [0, 0, 0],
                        [1, 2, 1]], dtype=np.float64)

    sobel_x_filtered = convolution2d(image, sobel_x)
    sobel_y_filtered = convolution2d(image, sobel_y)

    sobel_combined = np.sqrt(sobel_x_filtered**2 + sobel_y_filtered**2

    return sobel_combined.astype(np.uint8)
```

Implementation of Spatial Filtering (CPU version)

- **Gaussian Smoothing & Sobel Filtering**

```
22  def gaussian_smooth(image, kernel_size=3, sigma=1.0):
23      kernel = cv2.getGaussianKernel(kernel_size, sigma)
24      kernel = kernel @ kernel.T
25
26      return convolution2d(image, kernel, kernel_size).astype(np.uint8)
27
28
29  def sobel_filter(image):
30      sobel_x = np.array([[-1, 0, 1],
31                          [-2, 0, 2],
32                          [-1, 0, 1]], dtype=np.float64)
33
34      sobel_y = np.array([[-1, -2, -1],
35                          [0, 0, 0],
36                          [1, 2, 1]], dtype=np.float64)
37
38      sobel_x_filtered = convolution2d(image, sobel_x)
39      sobel_y_filtered = convolution2d(image, sobel_y)
40
41      sobel_combined = np.sqrt(sobel_x_filtered**2 + sobel_y_filtered**2
42
43      return sobel_combined.astype(np.uint8)
```

Implementation of Spatial Filtering (CPU version)

- **Custom Kernel**

```
45  def main():
46      img = cv2.imread('target_img.png', cv2.IMREAD_GRAYSCALE)
47
48      gaussian_smoothed_img = gaussian_smooth(img)
49      sobel_filtered_img = sobel_filter(img)
50
51      custom_kernel = np.array([[0, -1, 0],
52                                [-1, 5, -1],
53                                [0, -1, 0]], dtype=np.float64)
54      filtered_img = convolution2d(img, custom_kernel).astype(np.uint8)
55
56      result = cv2.hconcat([img, gaussian_smoothed_img, sobel_filtered_img, filtered_img])
57
58      cv2.imshow("Result", result)
59      cv2.waitKey(0)
60      cv2.destroyAllWindows()
61
62  if __name__ == '__main__':
63      main()
```

Implementation of Spatial Filtering (CPU version)

- **Result**