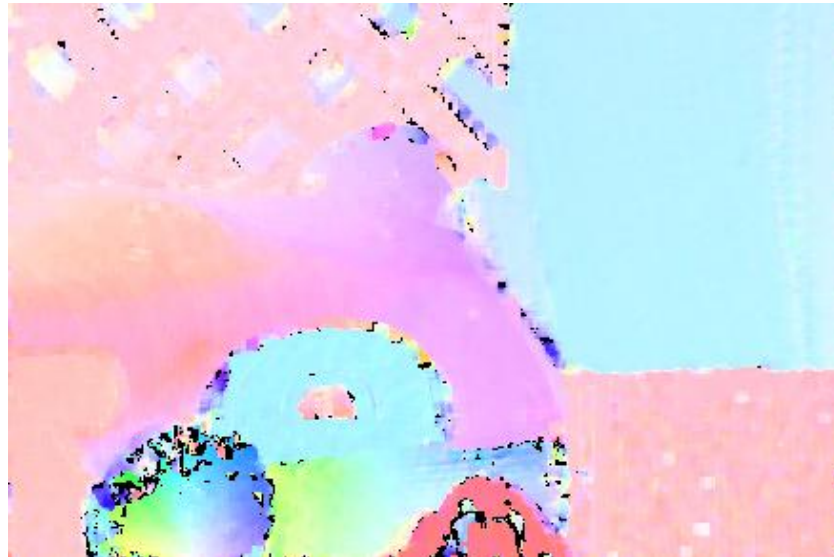


# Optical Flow Computation on Compute Unified Device Architecture



# [ Agenda ]

---

- Questions
- Background
- Optical flow?
- Algorithm description
- Implementation
- Result
- Possible future work

# [ Questions ]

---

- In which way can the algorithm (Lucas-Kanade) be parallelized?
- Name at least three methods implementing the sobel operator
- Name at least three advantages with using the texture memory

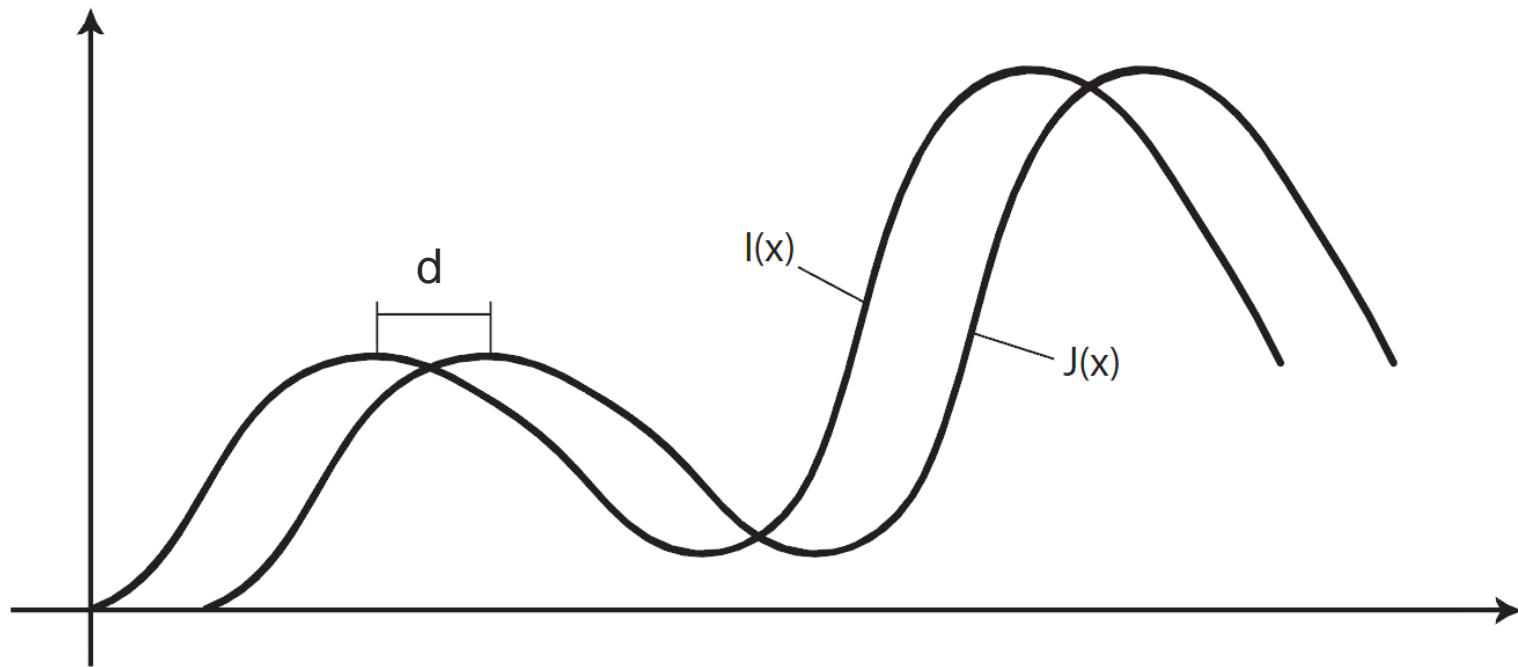
# [ Background ]



# [ Background ]

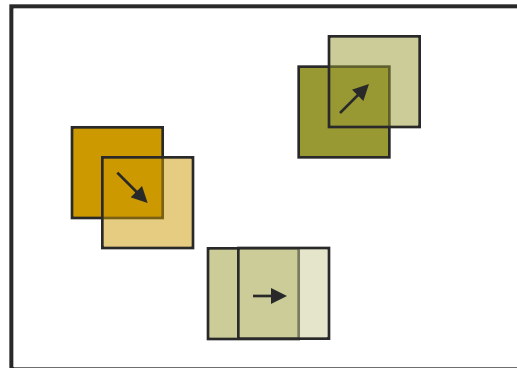
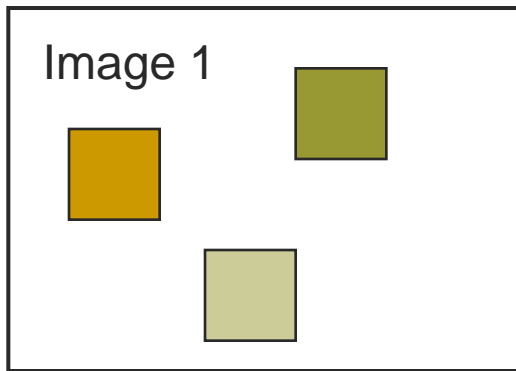


# [ Optical flow? ]



One-dimensional case, the disparity  $d$  is unknown

# [ Optical flow? ]



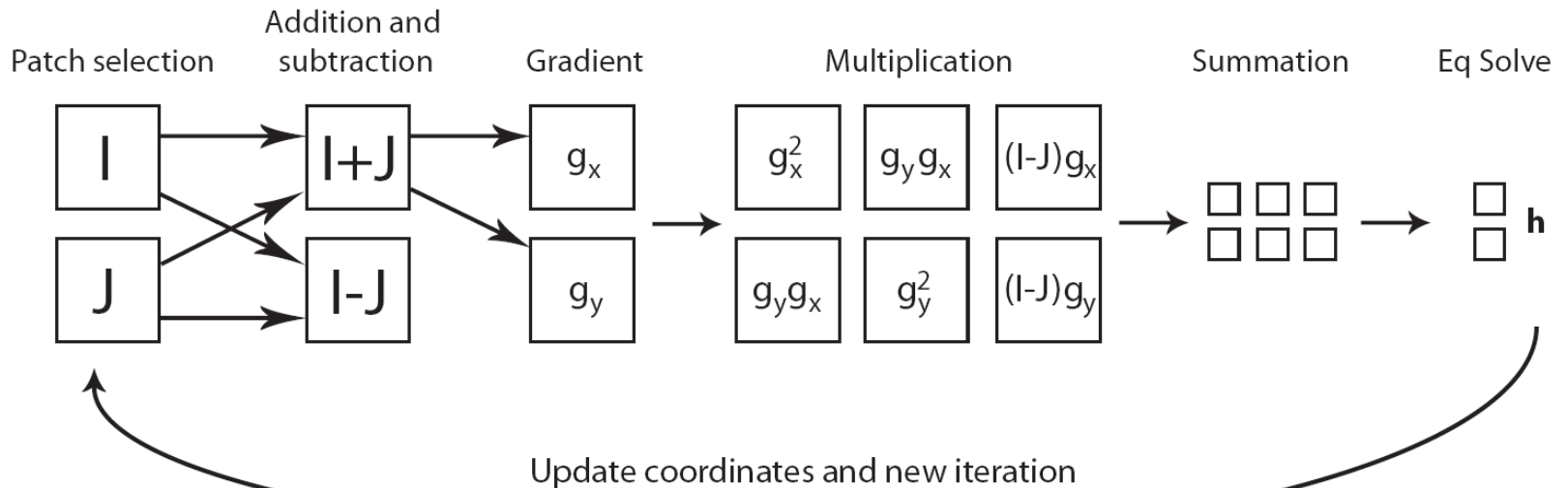
# [ Algorithm, Lucas-Kanade ]

- Uses spatial gradient
- Movements comes only from translation
- Iterative
- Local, many image patches
- Good in real-time applications

# [ Algorithm, Lucas-Kanade ]

$$\underbrace{\begin{bmatrix} \sum \sum_w g_x^2 & \sum \sum_w g_x g_y \\ \sum \sum_w g_y g_x & \sum \sum_w g_y^2 \end{bmatrix}}_{\mathbf{Z}} \underbrace{\begin{bmatrix} d_x \\ d_y \end{bmatrix}}_{\mathbf{d}} = 2 \underbrace{\begin{bmatrix} \sum \sum_w (I - J) g_x \\ \sum \sum_w (I - J) g_y \end{bmatrix}}_{\mathbf{e}}$$

# [ Implementation, algorithm ]



# [ Lucas-Kanade on the GPU ]

- Every step within an iteration need input from the previous one
- The result from one iteration will be input for the next iteration
- But there are many image patches which can be calculated in parallel

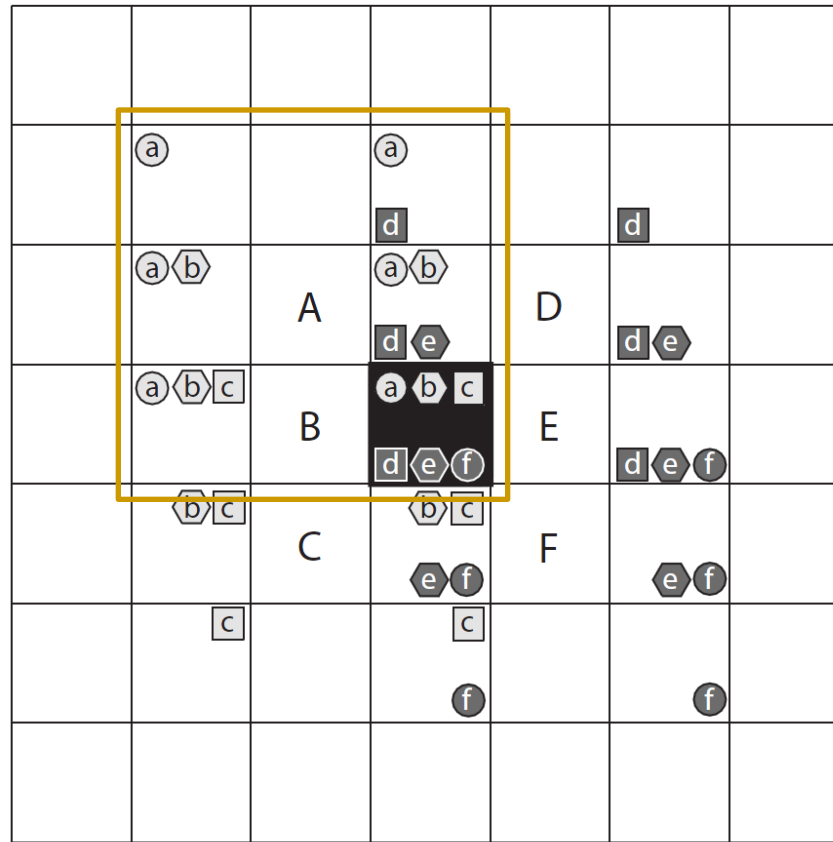
# [ Lucas-Kanade on the GPU ]

- Patch selection
  - Addition and subtraction
  - Gradient
  - Multiplication
  - Summation
  - Equation solver
- Kernel 1
- Kernel 2
- Kernel 3
- 
- ```
graph LR; A[Addition and subtraction] --> K1[Kernel 1]; B[Gradient] --> K1; C[Multiplication] --> K2[Kernel 2]; D[Summation] --> K2; E[Equation solver] --> K3[Kernel 3];
```

# [ Gradient example, sobel ]

- Straight forward through global memory access
- Using shared memory
- Using texture memory

# [ Gradient example, sobel ]

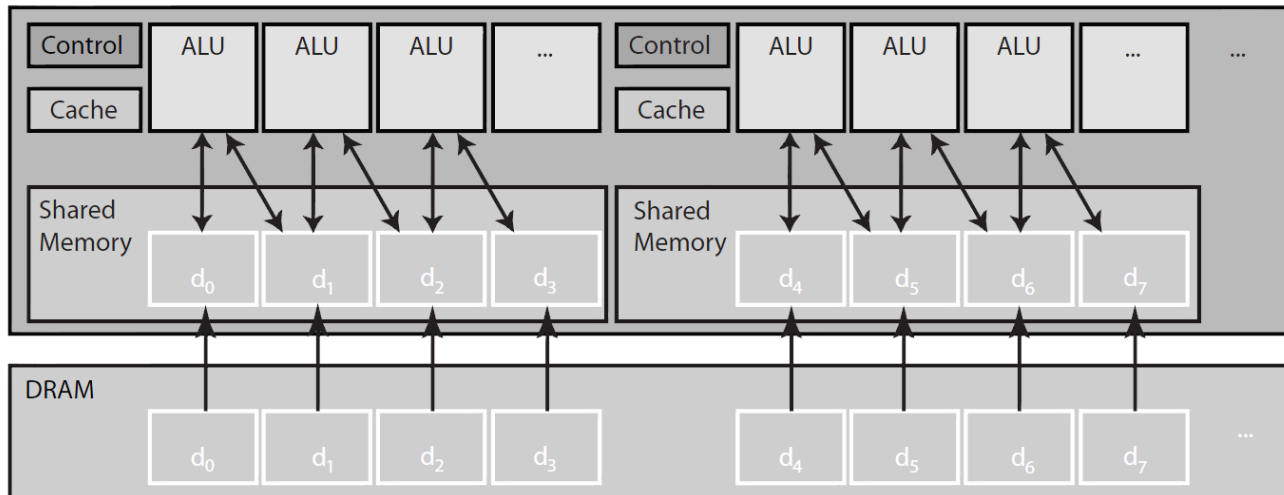


# [ Sobel with global memory ]

- The same value has to be read six times
- Global memory is slow
- = waste of bandwidth
  
- A better way would be to make use of the shared memory

# Sobel with shared memory

- Idea:
- First read the values once from global memory into shared memory

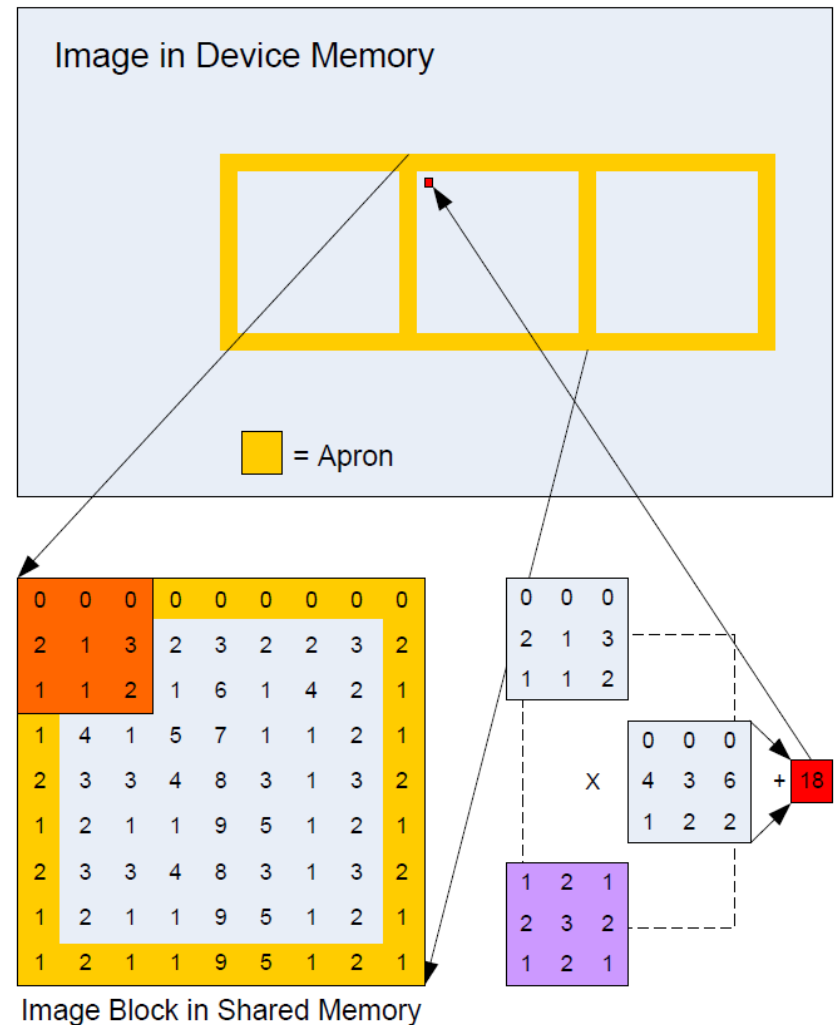


# [ Sobel with shared memory ]

- Global memory ~200-300 cycles
- Shared memory ~1 cycle
- Shared memory 16K
- Maximum of 512 threads / block
- Image has to be divided

# Sobel with shared memory

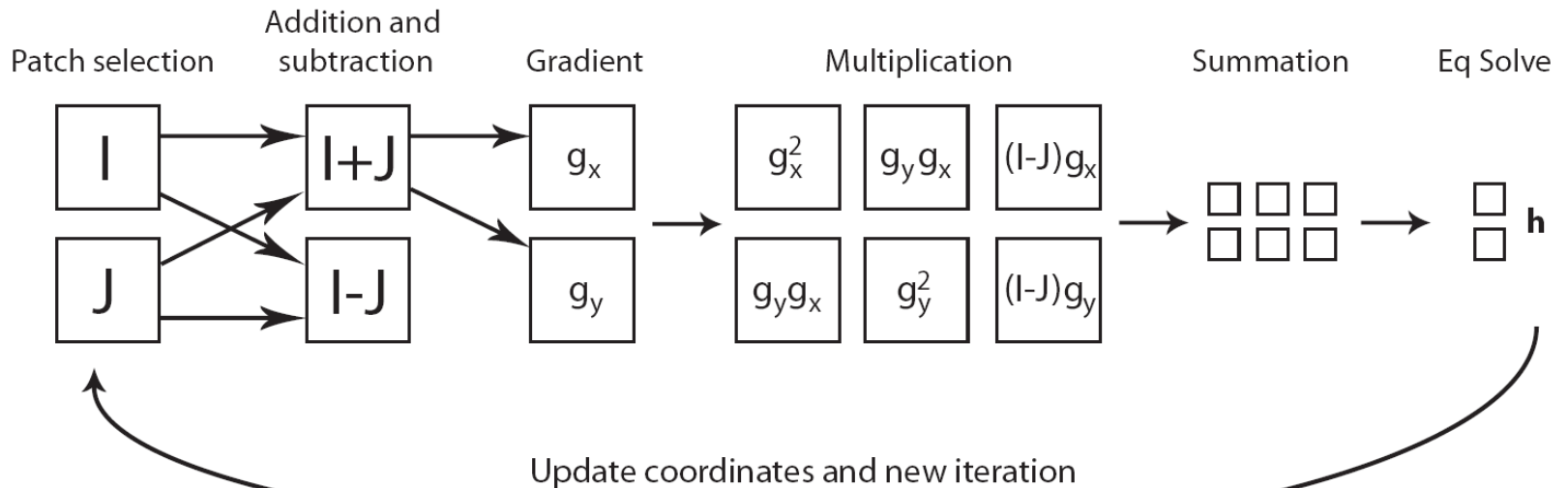
- Border problem, add apron
- Still problem at image border



# [ Sobel with texture memory ]

- Texture memory is cached
- Hardware support for out-of-range
- 2D-textures are not writable

# [ Implementation, algorithm ]



# [ Multiplication and summation ]

- Calculated in the same kernel
- Many methods to optimize summation / reduction, but a lot of data required
- Each patch is not so large, but they are many

# [ Multiplication and summation ]

- Summation is parallelized within each patch in shared memory through multiple threads
- CUDA enables the programmer to use many outputs

# [ Eq solver & new coordinates ]

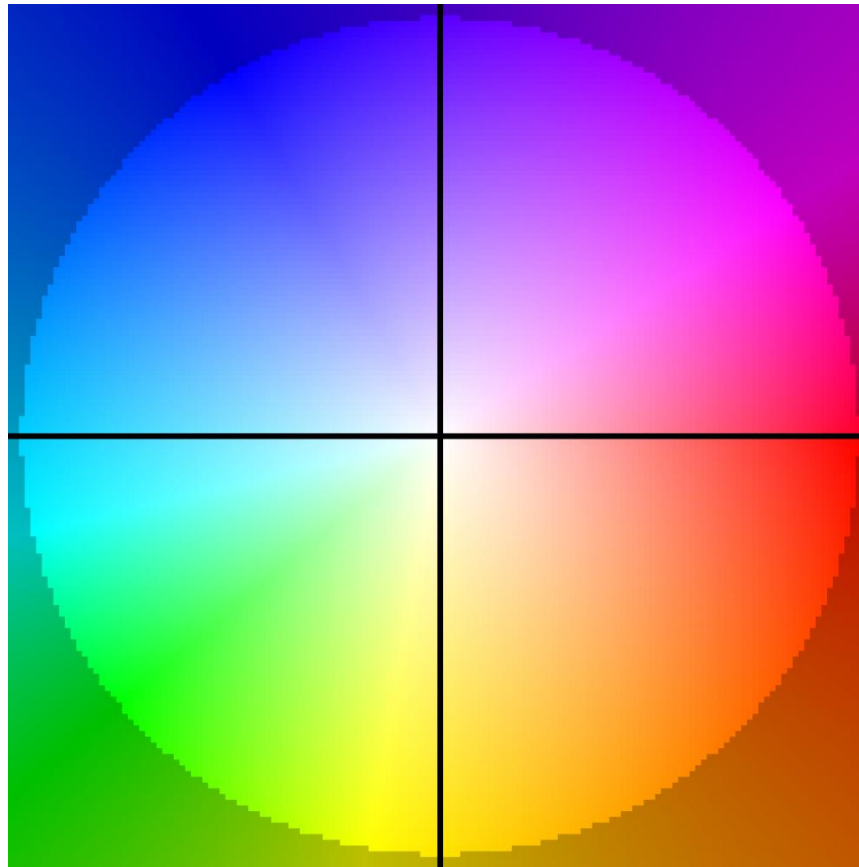
$$\underbrace{\begin{bmatrix} \sum \sum_w g_x^2 & \sum \sum_w g_x g_y \\ \sum \sum_w g_y g_x & \sum \sum_w g_y^2 \end{bmatrix}}_{\mathbf{Z}} \underbrace{\begin{bmatrix} d_x \\ d_y \end{bmatrix}}_{\mathbf{h}} = 2 \underbrace{\begin{bmatrix} \sum \sum_w (I - J) g_x \\ \sum \sum_w (I - J) g_y \end{bmatrix}}_{\mathbf{e}}$$

$$d_x = 2 \frac{g_y^2 (I - J) g_x - g_x g_y (I - J) g_y}{g_x^2 g_y^2 - (g_x g_y)^2}$$

$$d_y = 2 \frac{g_x^2 (I - J) g_y - g_x g_y (I - J) g_x}{g_x^2 g_y^2 - (g_x g_y)^2}$$

- The disparity will be on a sub-pixel level
- When using texture memory, interpolation will be done for "free"

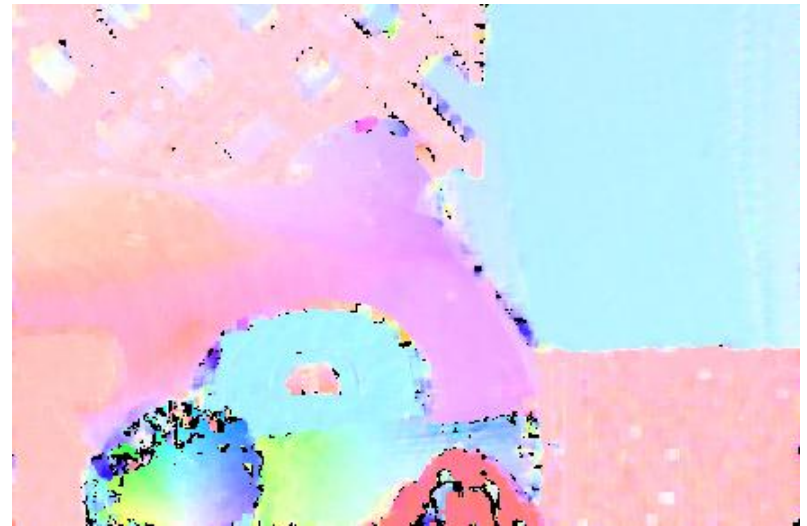
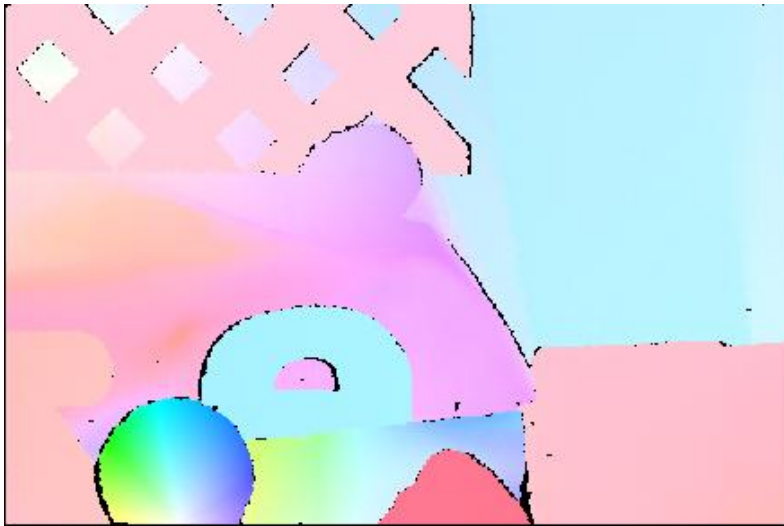
# [ Results, color coding ]



# [ Results ]



# [ Results ]



# [ Possible future work ]

---

- Algorithm requirements:
  - No big movements
  - No rotations or other transformations
- Use scale pyramids
- Affine model