
FOUNDRY.
imagination engineered

Introduction to *Blinkscript*.

What is *Blinkscript*

- Allows NUKE users to implement their own nodes in a very efficient manner
- The *Blinkscript* node runs a *Blink kernel* over every pixel in the output image
- *Blink kernel*
  - Similar to a C++ class with some added specific parameters
  - Implements processing for one pixel
- NUKE translates, compiles and runs the code on-the-fly on the GPU (CUDA or OpenCL)
- Processing is performed in parallel for each destination pixel
- Can revert to CPU if no GPU is available (code is translated to SIMD instructions)

Blink Kernel

- Example: inverting the pixel value

Kernel name

Kernel type

ImageComputationKernel<eComponentWise>
ImageComputationKernel<ePixelWise>

Kernel definition

```
kernel InvertKernel : ImageComputationKernel<eComponentWise>
{
  Image<eRead, eAccessPoint, eEdgeClamped> src; // the input image

  Image<eWrite> dst; // the output image

  void process() {

    // Read the input image
    float input = src();

    // Write the result to the output image
    dst() = 1.0f - input;
  }
};
```

Kernel input image(s)

Kernel output image

Kernel processing method

Blink Kernel properties

- Kernel types:
  - **ImageComputationKernel<eComponentWise>** Same processing on every pixel and every channel
  - **ImageComputationKernel<ePixelWise>** Same processing on every pixel, different on the channels

- Input image access types:
  - **eAccessPoint** The process method can only read the pixel at the kernel location
  - **eAccessRandom** The process method can also access any pixel other pixel

- Edge processing types:
  - **eEdgeClamped** The nearest edge pixel value is extended outside the boundaries
  - **eEdgeConstant** A constant value is used outside the image boundaries

Pixel Wise Processing

- Example: Averaging the RGB channel values of each pixel

```
kernel AvgColorKernel : ImageComputationKernel<ePixelWise>
{
  Image<eRead, eAccessPoint, eEdgeClamped> src; // the input image

  Image<eWrite> dst; // the output image

  void process() {

    // Read the input image
    SampleType(src) input = src();

    float avg = (input.x + input.y + input.z) / 3.0f;

    // Write the result to the output image
    dst() = float4(avg, avg, avg, 1.0f);
  }
};
```

Local variables

- Example: Averaging the RGB channel values of each pixel with a weighting vector

```
kernel AvgColorWKernel : ImageComputationKernel<ePixelWise>
{
  Image<eRead, eAccessPoint, eEdgeClamped> src; // the input image
  Image<eWrite> dst; // the output image

  // local variables are shared by pixels
  local:
    float3 w;

  // Init avoid repeating the assignments in the processes
  void init() {
    w.x = 0.7f;
    w.y = 0.2f;
    w.z = 0.1;
  }

   void process() {

    // Read the input image
    SampleType(src) input = src();

    float avg = (w.x * input.x + w.y * input.y + w.z * input.z);

    // Write the result to the output image
    dst() = float4(avg, avg, avg, 1.0f);
  }

};
```

Use parameters

- Example: Averaging the RGB channel values of each pixel with a weighting vector given as a user parameter

```
kernel AvgColorUWKernel : ImageComputationKernel<ePixelWise>
{
  Image<eRead, eAccessPoint, eEdgeClamped> src; // the input image
  Image<eWrite> dst; // the output image

  param:
   float wr;
   float wg;
   float wb;

  // Defines user parameters
  void define() {
    defineParam(wr, "Weight red", 1.0f);
    defineParam(wg, "Weight green", 1.0f);
    defineParam(wb, "Weight blue", 1.0f);
  }

   void process() {

    // Read the input image
    SampleType(src) input = src();

    float normalisation = wr + wg + wb;
    float avg = (wr  * input.x + wg * input.y + wb  * input.z) / normalisation;

    // Write the result to the output image
    dst() = float4(avg, avg, avg, 1.0f);
  }
};
```

Access Random

- Example: Box filter

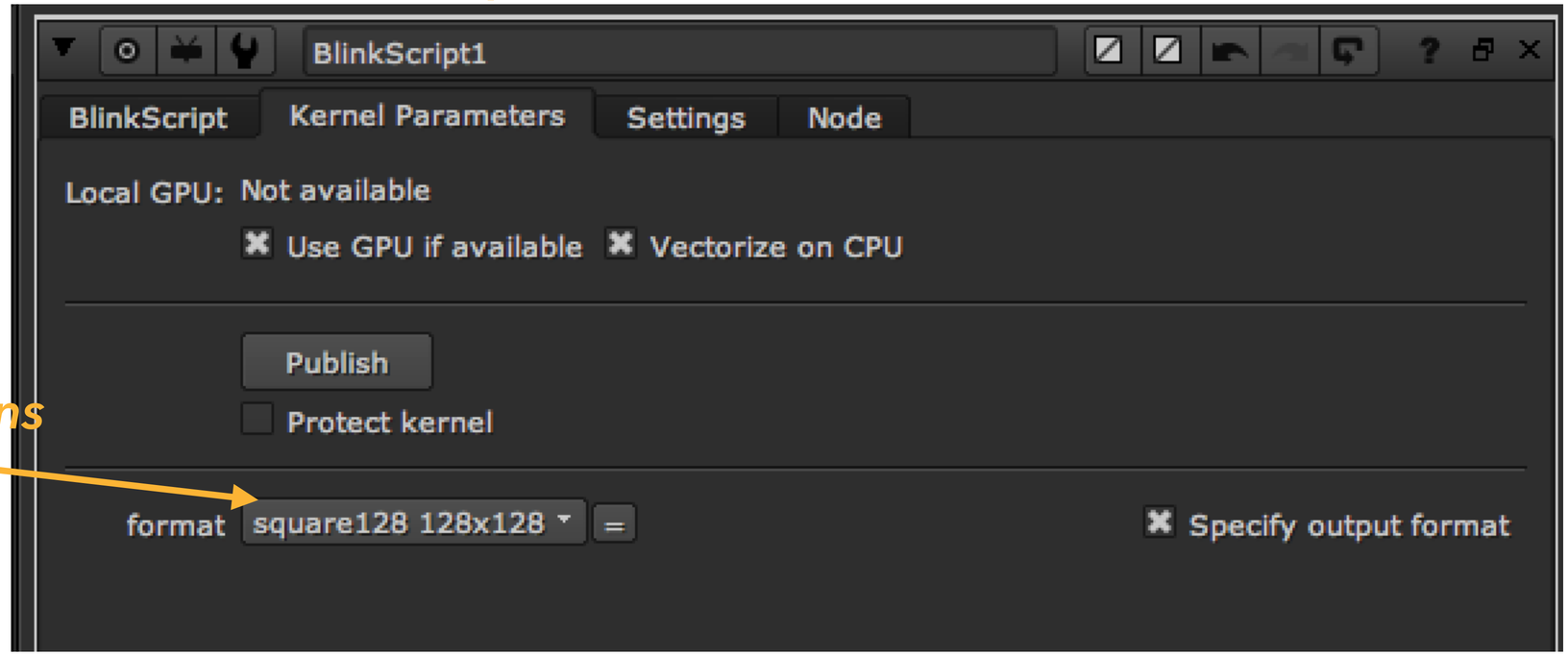
```
kernel BoxFilterKernel : ImageComputationKernel<eComponentWise>
{
  Image<eRead, eAccessRandom, eEdgeClamped> src; // the input image
  Image<eWrite> dst; // the output image

  param:
    int size;

  local:
    int hSize; // half size
    float normalisation; // Number of pixels in the box

  // Defines user parameters
  void define() {
    defineParam(size, "Box size", 3);
  }

  void init() {
    hSize = size / 2;
    normalisation = float(size * size);
  }

   // pos gives the position of the kernel in the output image
   void process(int2 pos) {

    // Read the input image
    float sum = 0;

    for(int k = -hSize; k <= hSize; k++) {
      for(int l = -hSize; l <= hSize; l++) {
        sum += src(pos.x + l, pos.y + k);
      }
    }

    // Write the result to the output image
    dst() = sum / normalisation;
  }
};
```

Reduction

- Example: Half sizing the dimensions

By default the output dimensions = the input dimensions

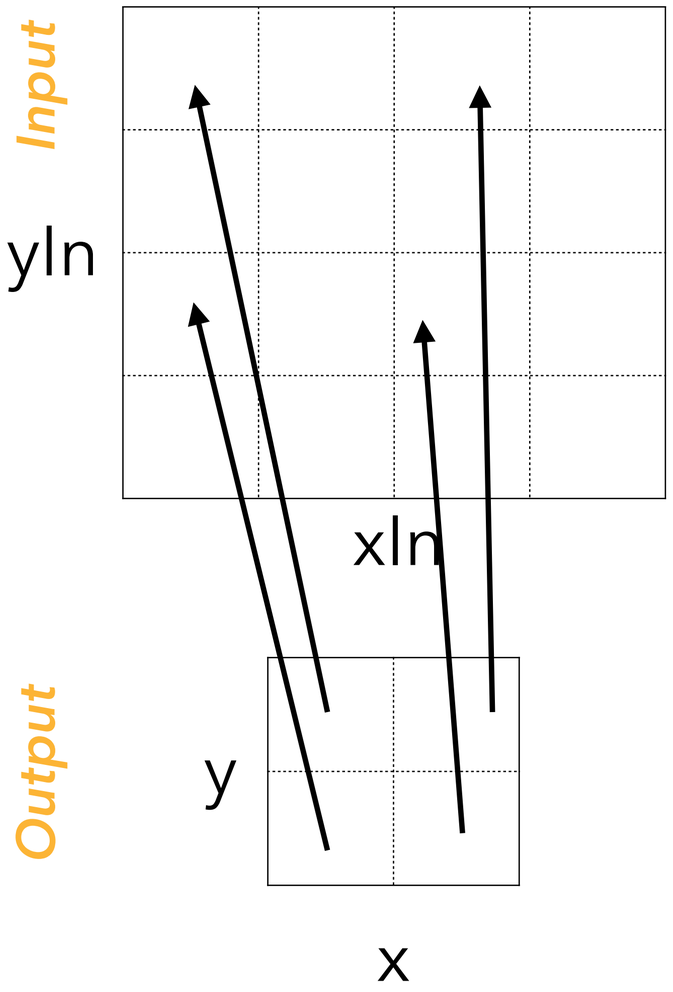
You can override the output dimensions



but remember each process maps to one pixel in output !

Reduction

- Example: Half sizing the dimensions




```
kernel reduceKernel : ImageComputationKernel<eComponentWise>
{
  Image<eRead, eAccessRandom, eEdgeClamped> src; // the input image
  Image<eWrite> dst; // the output image

  void process(int2 pos) {

    // Output position
    int x = pos.x;
    int y = pos.y;

    // Input position
    int xIn = 2*x;
    int yIn = 2*y;

    // Read the input image
    float input = src(xIn, yIn);

    // Write the result to the output image
    dst() = input;
  }
};
```

Multiple inputs

- Example: Time offset, frame difference

A time offset is used as a second input providing the next frame

```
kernel differenceKernel : ImageComputationKernel<eComponentWise>
{
  Image<eRead, eAccessPoint, eEdgeClamped> t0; // the input image t0
  Image<eRead, eAccessPoint, eEdgeClamped> t1; // the input image t1
  Image<eWrite> dst; // the output image

  void process() {
    dst() = t1()-t0();
  }
};
```

Casting

- Example:

```
kernel castKernel : ImageComputationKernel<eComponentWise>
{
  Image<eRead, eAccessPoint, eEdgeClamped> src;
  Image<eWrite> dst; // the output image

  local:
    float scale;

  void init() {
    scale = 1024.0f;
  }

  void process() {
    int srcI = int(src()*scale);
    dst() = float(srcI)  / scale; //! You will introduce quantisation errors
  }
};
```

Built in functions

- https://docs.thefoundry.co.uk/nuke/90/BlinkKernels/Blink.html#functions

```
scalar dot(vec a, vec b);      // Returns the dot product of vector *a* with vector *b*.
vec3 cross(vec3 a, vec3 b);    // Returns the cross product of vector *a* with vector *b* (3-component vector types only).
scalar length(vec a);          // Returns the length of vector *a*.
vec normalize(vec a);          // Returns the result of dividing vector *a* by its length.
```

```
type sin(type a);
type cos(type a);
type tan(type a);
type asin(type a);
type acos(type a);
type atan(type a);
type atan2(type a, type b);
```

```
//Rounding:
type floor(type a);
type ceil(type a);
int_type round(type a);  //Rounds a to the nearest integer
```

```
type exp(type a);
type log(type a);
type log2(type a);
type log10(type a);
```

```
//Powers and square roots
type pow(type a, type b);
type sqrt(type a);
type rsqrt(type a);     //Returns 1 over the square root of a
```

```
//Absolute functions
type fabs(type a);
int_type abs(int_type a);
```

```
scalar median(scalar data[], int size);  //Finds the median value in an array of data of length size
```

```
//Integer and fractional parts
type fmod(type a, type b);
type modf(type a, type *b);
```

```
//Size functions
scalar width();                  //Return the width of the rectangle
scalar height();                 //Return the height of the rectangle
vec    height();                 //Return a vector containing the width and height of the rectangle
```

```
//Sign function
type sign(type a);
```

```
//Min and max functions
type min(type a, type b);
type max(type a, type b);
type clamp(type a, type min, type max);  //Clamp a to be between min and max
```

```
//Reciprocal function
type rcp(type a);
```