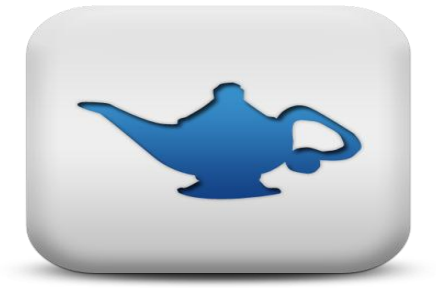
KernelGenius

Programming guide

Performance, Productivity, and Portability
for Computer Vision Programmers



Author: Thierry LEPLEY

TABLE OF CONTENTS

1. Introduction

1.1. Main philosophy

The main motivations behind KernelGenius are:

- Let the programmer focus on algorithmic aspects of its application and not on the optimization and parallelization aspect. This is the way to obtain more productivity and more performance portability across hardware platforms.
- Focuses on computation kernels that are the most crucial and delicate part of the applications to optimize on a parallel accelerator and that can often be reworked by without needing to re-architecture all the application, simplifying then his optimization work.
- Separate concerns for more easily reuse of the kernel library in different usage context and in particular separate the algorithm heart from the management the dataset edges and from the size of the input dataset.
- Simple and images processing friendly programming language in order to allow the tool easily capturing the data access patterns of computation kernels, information that is either hardly and partially retrieved from classical programming languages like C, but that is a fundamental information when optimizing for systems with a complex memory hierarchy.
- Allow the user structuring his computation kernel library and let a tool building the kernel library with a well-specified API that can be called from the main application.
- Propose a set of classical predefined built-in functions, and allow the user programming its own functions in order to cover his application needs, in a way to maintain the possibility of optimization on the target accelerator.

1.2. KernelGenius at a glance

KernelGenius is a compiler that takes as input, expressed in a dedicated simple language, a set of `kernel` sections. Each file (or module) will result into an OpenCL program and each `kernel` section will result into an OpenCL kernel. In this way, the programmer can control directly the structuring of the computation kernel library automatically generated by KernelGenius.

Each kernel is described as a set of *graphs* that describe the functionality of the kernel. Each graph will compute at least one output of the kernel and potentially several. The graph is composed of a set of *nodes* that correspond to the smallest compute granularity of KernelGenius. A graph can be as simple as a single node or composed of numerous nodes.

There are two kinds of nodes in KernelGenius:

- Builtin nodes have a predefined functionality that is configurable for a subset of them. Such nodes can be very simple operations the input data set like an image subtraction or more complex like a convolution filter. In the current version of KernelGenius, the

built-in nodes stay simple, leaving complex algorithms in the hands of the programmer who may want to assemble nodes into a graph for this purpose.

- Programmable nodes can be programmed with *native code sections* (OpenCL-C in our case) in which the programmer will simply have to add some markers to let the KernelGenius retrieve the relevant information without needing parsing and analysing this native code. Such nodes let the programmer control more the final performance and the computation accuracy (for instance, one may not want to rely on OpenCL built-in function for performing the basic math operations like square-root or trigonometric), while keeping the nice abstraction in KernelGenius in term of parallelization and optimization with regard to the accelerator hardware architecture.

The management of node function behaviour at the borders of the data set can be simply described as a property of node.

From the input file, the KernelGenius compiler generates optimized OpenCL kernels optimized for the STHORM many-core. The optimization process for the target architecture is entirely left in the hand of the compiler and the user does not need to know anything about the STHORM architecture. The user can specify the minimum and maximum size of the input data set, what allows the KernelGenius compiler optimizing the generated OpenCL kernel, the memory management being an crucial aspect of the optimization process for this architecture.

2. Kernel Genius language

2.1. Introduction

KernelGenius takes as input the KG language, expected in `.kg` files. This language stays simple, focuses on what fundamental for parallel acceleration and does not intend capturing the full semantics of neither the application nor the computation kernel itself. The KG language then allows embedding 'native C sections' that will not be interpreted or very partially by the KernelGenius compiler. Native code is not interpreted in term of grammar/syntax by the KernelGenius compiler, but as it will be discussed later in the chapter, the user must nevertheless place some markers to let the KernelGenius extract the relevant data access pattern information. This philosophy is already used by compiler generator tools like ANTLR and has been found very relevant for KernelGenius

In this section, we provide an overview of the KernelGenius language syntax, following the classical BNF way.

2.2. Program definition

The *program* corresponds to a `.kg` file. All kernels described in this file will belong to the same OpenCL program.

KG_file : *programDefinition*

programDefinition : (*programStatement*)+

programStatement : (*typeDeclaration* | *nativeCodeSection* | *kernelDefinition*)+

All user defined types used in non-native sections must be declared prior to the use like in C as *typeDeclaration* statements. *typeDeclaration* supports the C type definition (`struct`, `union`, `typedef`, `enum`) with the restriction that `pointers`, `void` and function prototypes are forbidden.

Example:

```
typedef unsigned char uint8_t;
typedef struct { uint8_t Y; uint8_t U; uint8_t V;} YUV;
```

The native code will be directly copied to the output OpenCL-C program without being interpreted by KernelGenius, except by the pre-processor. It can be multi-lines. Note that functions contained into the native section should not manipulate pointers and should simply contain purely computational functions, pre-processor directives and variable declaration (like `const` OpenCL-C variables).

nativeCodeSection : '`${`' *nonIntepretedNativeCode* '`}$`'

2.3. Kernel definition

The kernel has a set of inputs that may be

- scalar (usually control data)

- mono- or multi-dimensional array (image or set of images in the image processing area, filter coefficients)

The prototype of the kernel simply declares inputs, while outputs are subsequently defined in the kernel body. This allows avoiding adding to kernel parameters a direction specifier (like in/out).

kernelDefinition : `kernel` *kernelDeclarator* '{' *kernelBody* '}'

kernelDeclarator : *kernelName* '(' *KernelInputParameterList* ')'

For the kernel parameter list, KernelGenius extends the C99 syntax and allows specifying a *value range* to a scalar parameter and a *value* or a *range* of the index space of an array.

1- *scalarDecl* : *typeSpecifier paramName* **('=' '[' *rangeValue* ']')?**
2- *arrayDimDecl* : '[' **(*rangeValue* | *value* | *scalarParamName*)?** ']'
valueRange* : *minValue* ':' *maxValue

Examples:

- Scalar parameter with a value range: `int imageHeight = [1:1080]`
- Array with potentially infinite index range : `float array[]`
- Array with fixed index range : `float array[1080]`
- Array with limited index range : `float array[480:1080]`
- Array with index range determined by a parameter : `float array[imageHeight]`

Kernel outputs and the function nodes that will generate these outputs from the inputs are specified in the kernel body. A kernel is purely *computational* and the programmer cannot express other side effects than the kernel outputs.

The kernel functionality is expressed as a graph of function nodes. This graph, if not reduced to a single node, is built from a sequence nodes declaration and a graph edge is implicitly created when the output of a node is taken as input of another node. Note that similarly to the C language, symbols cannot be backward referenced (used before being defined), which prevents by construction the programmer from creating circles in graphs, what is not allowed in KernelGenius. The kernel body is then a sequence of nodes that define the computation to perform, the dataflow relationship between nodes and return statements that define output of the kernel. All node outputs that are not defined a return value of the kernel can be optimized out by the KernelGenius compiler since internal to the kernel.

kernelBody : (*kernelSatement*)+

kernelStatement : (*nodeInstance* | *returnStatement*) ';'

returnStatement : `return` *nodeInstanceName*

2.4. Function Node

Fucntion nodes of kernel graphs are instiatiated from a function node templates. There are two kinds of node templates: *builtin* node templates (fixed or configurable) and *programmable* node templates. Both follow the same syntax. Each node instance has a name which

references the generated output data (a node cannot have more than one output). The node instantiation specifies an output base data type to a node template (with the C++ template syntax) and a set of properties. The list of available properties depends on each node, but some properties like can be common to different node types.

nodeInstance : *NodeTemplateName* '<' *baseOutputType* '>' *nodeInstanceBody*?

nodeInstanceBody : '{' *nodeInstanceProperty*+ '}'

nodeInstanceProperty : '.' *propertyName propertyParams*? ('=' *propertyValue*)? ';'

propertyParams : '(' *attachedParamList* ')'

The property names, parameters (optional) and values (optional) depend on each algorithmic node and will be described in the section describing the list of nodes existing KernelGenius.

The property *attached parameter list* specifies the list of node input parameters to which is attached the property. Example:

```
nodeType<float> myNode(in0, in1, in3) {
  .myprop(in0,in3) = 2;
}
```

Property *values* can be of several types that we list here:

- *Identifier* (name) which does not necessarily reference a symbol of the program
- *Constant Literal* which can be a scalar or a compound literal as defined in C99 with some extensions, since ranges are supported as index specifier in the literal type definition. Note that the range can cover the negative value space and is not necessarily symmetrical.
  Ex: `(float[-1:1]){1,2,1}  (float[-1:3]){1,2,3,4,5}`
- *String*. Ex: `"starting convolution execution trace"`
- *Native code statement*, surrounded by '`${`' and '`}$`' that, on the contrary to a string can be multi-lines.

3. Function node templates

3.1. Introduction

This section lists algorithmic nodes as they exist in the current version of KernelGenius. The KernelGenius syntax allows easily extending the set of node templates without any change in the language itself and this list will naturally grow in the future as the KernelGenius technology is developed further.

3.2. The `border` node property

The border management problematic is common to all algorithmic nodes that are susceptible to read data elements out of the data set. It concerns in particular the `Convolution` built-in node and the `Filter` programmable node.

There are different types of border management, depending on the use case and the programmer specifies which behaviour it expects with the `border` node property.

- *constantValue*: all values taken out of the image have specified value.
- `duplicate` : values read out of the input data set have the value of the closest data element of the border of the data set.
- `mirror` : the space out of the data set is the mirror of the input data set itself.
  For example, index *-1* becomes *0*, index *-2* becomes *1*, index *-3* becomes *2*, etc …
- `skip`: the output data elements that depend of at least one data element out of the input data are not computed and stay unchanged.
- `undef` : the value read out of the image have no defined value an can take any.
- *nativeCodeExpression* : the border property *is specific to the Filter programmable node*. It must be a native code expression enclosed as usually in '`${`' and '`}$`'. It provides the formula to compute the value of data elements read out of the input data set from an expression where indexes out of the iteration spaces are put back into the input. The native code section then follow rules that differ from other native code sections of KernelGenius:
  - Whatever number of dimensions in the iteration space of the input data set, the border expression specifies the behaviour on one single indexing space
  - The input is not referenced by any name since the border behaviour concerns potentially all inputs of a node. It will then be simply referenced with '$'

  Example: `.border = ${ $[0]-$[1] }$` means that of input data elements out of the input data set for a particular dimension will take the value computed as the subtraction of 2 valid input data element taking the index 0 and 1 in this dimension for the first border and 0 and -1 for the second border.

3.3. Builtin nodes

3.3.1. Simple operations

Simple operation nodes apply an operator on each individual element of inputs. Individual elements of inputs must be arithmetic: integral or floating point scalar. The number of parameters of simple operation nodes is variable and without limits. For these simple operations, there is no property customization.

- Arithmetic
  - `Add` : $\sum_{i=0}^{n} input_i$
  - `Sub` : $input_0$-$\sum_{i=1}^{n} input_i$
  - `Mult` : $\prod_{i=0}^{n} input_i$
  - `Absdiff` : $|input_0 - \sum_{i=1}^{n} input_i|$
  - `EuclideanNorm` : $\sqrt{\sum_{i=0}^{n} {input_i}^2}$
- Logical
  - `Or`
  - `And`
  - `Xor`
- Other
  - `Min`: min $(input_0, \dots, input_n)$
  - `Max`: min $(input_0, \dots, input_n)$

Note: this simple operation list has been voluntarily limited and feedbacks of users will allow extending this list with the most useful base function for the computer vision field.

3.3.2. Convolution

The `Convolution` node applies a specified and fixed coefficient filter to each elements of the input data set. The `Convolution` node must have one and only one input parameter.

The `Convolution` node is customized by 2 properties:

- `coefficients` (mandatory): constant array literal.
  The number of dimensions of the coefficient must be lower or equal to the number of dimensions of the input of the node. The index range of arrays must be specified.
- `border` (optional): Since such filter usually require reading data out of the index range of the input, the user can define a `border` semantics. Since this property is common to other nodes, please refer to section 3.2 for more details about this property.

Example:

```
Convolution<float> myNode(in) {
	.border = 255;
	.coefficients = (float[-1:3]) {1, 0, 1, 2, 3};
}
```

3.4. `Filter` node

3.4.1. Base principle

The *Filter* programmable node is a powerful programmable node that can describe a large set of regular dense/semi-dense computation on matrixes or images by defining:

- The function of a *base filter*
- The way the *base filter* is replicated over the whole input and output dataset (images)

The function of the base filter is attached to a data element and defines a rectangular access window around the considered data element that can have various shown in Figure 1.

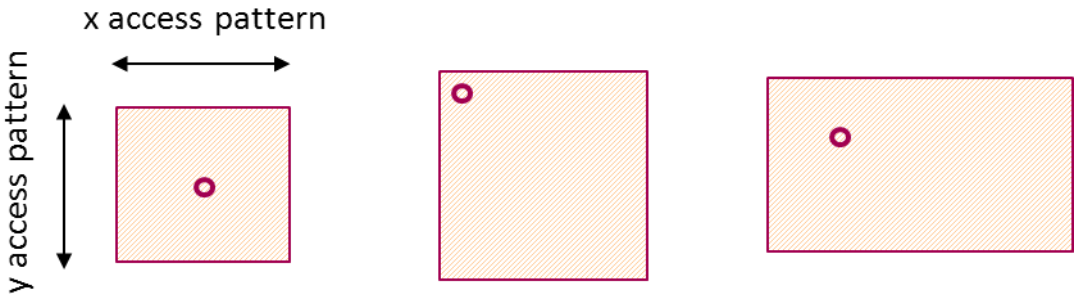


Figure 1: example of access window for the base filter

The base filter is *iterated* on the input dataset in each of its dimensions, starting from the first element (coordinate [0,0]), following a step determined by a *stride* as illustrated in Figure 2.

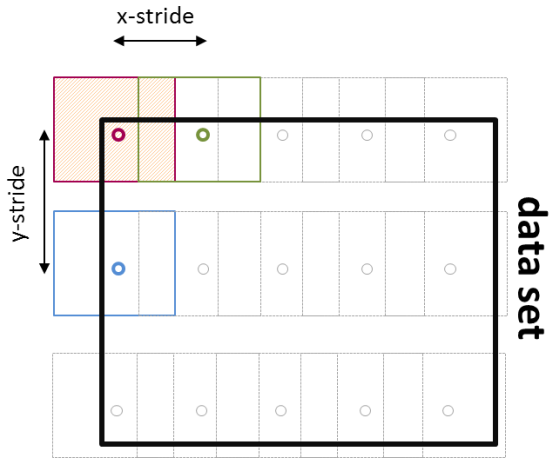


Figure 2: iteration of the base filter over the data set

These access and iteration pattern properties can be specified on inputs and output of the filter node and both can be different, as illustrated in Figure 3.

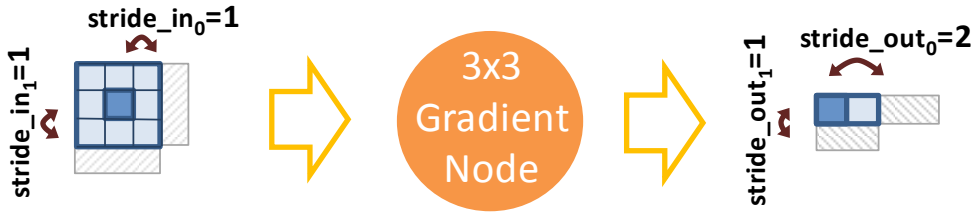


Figure 3 : Filter data pattern

3.4.2. Specification

The filter is programmed with the following properties:

- `Function` : it is a *native code block* that describes the basic filter function that is replicated over the input *data iteration space*. This `function` property defines then at the same time functionality of the filter and its *read* and *write* access pattern. The *read* elements for the input parameter on which the filter is iterated must be prefixed with the `$` marker. The *written* elements for the output must be prefixed with the `@` marker. An example is given in Figure 4.
  Note that the array indexing ordering follows the C language semantics where the last dimension is indexed first: `$myArray[z][y][x]` .

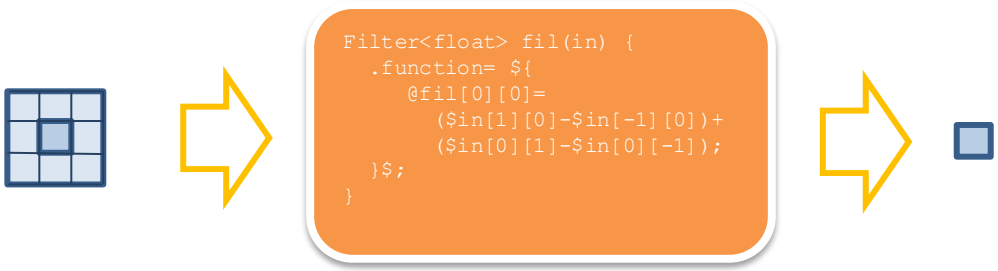


Figure 4: example of read and write access pattern defined by the function

- `stride_in / stride_out` (optional): this property is defined as an array initializer and defines the stride respectively for the input iterated dataset or output dataset in each dimension space of the dataset: `{x, y, z}`. By default, the stride is set to 1 in all dimensions.
  As an examples, the output stride of Figure 3 would be :
  ```
  .stride_in={1,1};
  .stride_out={2,1};
  ```
- `border` (optional): Since such filter usually requires reading data out of the index range of the input, the user can define a 'border' semantics. Please refer to section 3.2 for more details about this property
- `read_bound` (optional): In case the read accesses of the node function are dynamic, the KernelGenius compiler is not able to compute the read access pattern from the function. In this case, the user must specify the *read access bounds* with the `read_bound` property, which may be set independently on each node input. This property is defined as a mono- or multi-dimensional array range indexing following the C semantics : [z][y][x]

Examples:

```
.read_bound(in1,in3)=[-3:1][-2:2]; // Specific inputs
.read_bound = [-2:2][-1:1]; // All inputs
```

3.4.3. Assumption of dense output

KernelGenius requires algorithmic node to have a dense output in the sense that all elements of the output data should be written by the node, except the border when its content is explicitly specified as undefined (`undef` border option). The KernelGenius compiler will then generate an error if the *output stride pattern* is strictly greater than the *write pattern* that is deduced from the `.function` property. Inside the write pattern, all data elements that are not written by the filter are *undefined* and may take any value.

3.4.4. Limitations in term of stride

In its current version, KernelGenius only support input strides on the y-axis greater than 1 at the input of the graph and output strides on the y-axis greater than 1 at the output of the graph. It forced the graph to have changes of the data rate in the y-axis at the frontier of the graph. There is no such constraint in the x-axis.

Typically, a `Filter` node describing a Downsampling function node (that has an input stride of 2 in the y-axis) can only take as input an input of kernel and cannot take as input the output of another node. Reversely, an Upsampling `Filter` node that has an output stride of 2 in the y-axis) can only be inserted generate a graph output and cannot provide data to another node.

3.5. `Operator` node

The `Operator` node applies a statement specified by the programmer that combines together individual data elements located at the same coordinates in all the inputs. The `Operator` node can take any numbers of inputs.

In order to let the KernelGenius compiler understand how input and output data are accessed, the user must prefix read input references with the `$` marker and written output with the `@` marker.

It takes as properties:

- `function` (mandatory) : native code section the specifies the statement(s) that compute a output data element from input data element(s).

Example:

```
Operator<float> myNode(in1, in2, in3) {
      .function = ${ @myNode = ($in1+2*$in2)/$in3; }$
}
```

4. KernelGenius compilation flow

4.1. Preprocessing

All `.kg` files are preprocessed by the standard C pre-processor, so that the programmer can for instance use `#define` macros, for instance to reuse nodes in several context, and include files, for instance to share types between `.kg` kernels and the C host application.

4.2. Generated files

The KernelGenius tool flow is described in Figure 5. Three files are generated for each KernelGenius `program`:

- Program host wrapper
  - <*program name*>.c
  - <*program name*>.h
- OpenCL-C program
  - <*program name*>.cl

The name of the program is by default the name of the `.kg` file but can be changed by a compiler option. For more information, the reader can refer to the *KernelGenius User Guide*.

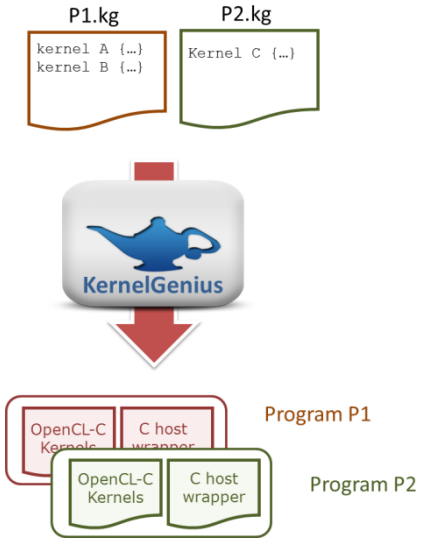


Figure 5 : KernelGenius flow

Generated OpenCL-C programs must be compiled by the STHORM OpenCL compiler and *C host wrapper* by the host CPU compiler. The host wrapper partially abstracts the fact that OpenCL is used as a backend, so that the host application needs to interact with functions of the *C host wrapper* and the OpenCL API in order execute the generated OpenCL kernels on STHORM.

4.3. Program host API

The functions provided to the application by the wrapper are the following:

For each `program`:

- `cl_program create<`*`program Name`*`>ProgramFromBinary(cl_context context, cl_device_id device);`
  Read the binary program file, create an OpenCL program object, build it and check if everything is fine. This function exists in case of error.
- `cl_program create<`*`program Name`*`>ProgramFromSource(cl_context context, cl_device_id device);`
  Read the source program file, create an OpenCL program object, build it and check if everything is fine. This function exists in case of error.

For each `kernel`:

- `void setKernelArgs_<`*`kernel name`*`>(cl_kernel kernel, <kernel output list>, <kernel parameter list>);`

Sets parameters to a kernel object. This function exists in case of error.
Note that the list of output has the same ordering as the order outputs are returned in the KernelGenius kernel. The input kernel list follows the ordering as defined in the KernelGenius kernel.

- `void run_<kernel name>(cl_command_queue commandQueue, cl_program program, int nbWG, int nbWIperWG, <kernel output list>, <kernel parameter list>)`
  Creates a new kernel object, sets arguments and run it as a 1D-Range of `nbWG` work-groups and `nbWIperWG` work-items per work-groups. This function exists in case of error.
  Notes:
  - In the current version, this 1D-Range should have a single work-group with from 1 to 16 work items. Future versions of KernelGenius will manage multiple work-groups.
  - If the application prefers to call a kernel directly with `clEnqueueNDRangeKernel`, it should be done only with a 1D-Range. In the current version, this 1D-Range should have a single work-group with from 1 to 16 work-items. Future versions will manage multiple work-groups.

5. KernelGenius by the example

5.1. Introduction

In this section, we illustrate KernelGenius through several examples based on classical image processing kernels in order to expose the principles discussed earlier in the document.

5.2. Driving the generated Code structuring

In this example, we want to generate a single OpenCL program called `Sobel` containing two OpenCL kernels called `Sobel3x3` and `Sobel5x5`

```
kernel Sobel3x3(int width, int height,
                  float in[height][width]) {
    // Kernel Body here
}

kernel Sobel5x5(int width, int height,
                  float in[height][width]) {
    // Kernel Body here
}
```

5.1.Defining the usage limits of the kernel

In this example, we want to get a `Sobel3x3` OpenCL kernel for which that input data is a 2D floating point image that will be called with an image format of HD 1080p maximum.

```
  kernel Sobel3x3(int width=[1:1920], int height=[1:1080],
                  float in[height][width]) {
    // Kernel Body here
  }
```

5.2. 3x3 Sobel

5.2.1. Introduction

In the previous section, we stated that we have two kinds of nodes: built-in and programmable. These nodes can be combined in a graph, but they have been created originally for 2 different usages of KernelGenius, that we can call high-level and low-level programming. In the high-level programming, the user assembles built-in nodes, while in the low-level programming, he may prefer to insert native OpenCL code in his KernelGenius source code and get more control on the way computation is finally performed. This section explains the different approaches by taking the same 3x3 Sobel filter example.

5.2.2. 3x3 Sobel: high level version – graph

The Sobel filter is the combination of two 2D 3x3 convolutions centred to the processed input data element. It can be implemented as a graph of 3 built-in nodes. In this case, we want pixels read out of the limits of the input images to have the value of pixels of the border, but it may be different in a real use case.

```
	Convolution<float> gx(in) {
		.border = duplicate;
		.coefficients= (float[-1:1][-1:1]) { {-1, 0, 1},
		                                     {-2, 0, 2},
		                                     {-1, 0, 1} };
	};

	Convolution<float> gy(in) {
		.border = duplicate;
		.coefficients= (float[-1:1][-1:1]) { {-1, -2, -1},
		                                     { 0,  0., 0},
		                                     { 1,  2 , 1} } ;
	};

	EuclideanNorm<float> merge(gx , gy);

	return merge;
```

Each node declares both the processing that must be performed and the output transient data that can be reused as input of another node. For instance, the results of the two convolutions (`gx` and `gy`) are used as input of the `merge` node. The `return` statement indicates which one should is a kernel output and then should not be optimized out be the compiler. Note that several outputs can be declared.

5.2.3. 3x3 Sobel: Low level version - Filter node

The Sobel filter can be described as a single `Filter` node. In this case, we may also want to use a specific fast square root function and not the OpenCL one. We then define a native section at the program level that will define user functions (`my_sqrt_f` in this case) accessible to all following kernels of the program.

```
${ // Native section
  float my_sqrt_f (float x) {
    // Implementation Code here
  }
}$ // End of Native section

kernel Sobel3x3(int width=[1:1920], int height=[1:1080], float in[height][width]) {
  Filter<float> sob(in) {
   .border = duplicate;
   .function = ${
     float gx=$in[-1][1]-$in[-1][-1]+2*($in[0][1]-$in[0][-1])+$in[1][1]-$in[1][-1];
     float gy=$in[1][-1]-$in[-1][-1]+2*($in[1][0]-$in[-1][0])+$in[1][1]-$in[-1][1];
     @sob[0][0]=my_sqrt_f(gx*gx+gy*gy);
   }$;
 };

  return sob;
}
```

5.1. Threshold kernel with the `Operator` node

The `Operator` node is a programmable node with a data access pattern limited to 1 elements for inputs and output, so it corresponds well to the need for writing a *threshold* filter. In this example, `in` is the *iterative input data space* of the filter, so that it is prefixed with the `$` marker, while `t` is not an iterative data set and is then left without marker.

```
kernel Threshold(int width=[1:1920], int height=[1:1080],
                 float in[height][width],
                 float t) {

  Operator<int> thresh(in,t) {
    .function = ${ @thresh = $in>=t?1:0; }$;
  };

  return thresh;
}
```

5.2. 5x5 Separable convolution with a graph of `Filter` nodes

The predefined `Convolution` node assumes hard coded coefficients and a non-separable convolution. One can write a separable convolution with generic coefficients by combining two `Filter` nodes in a graph, computing successively a 1D convolution in the x-axis and in the y-axis. In this example, `in` and `conv_x` are the input *iterative data spaces* of a node, so that they are prefixed with the `$` marker in the function native code section, while `filter` is not an iterative data set and is left without marker.

```
kernel SeparableConvolution5x5(int width=[1:1920], int height=[1:1080],
                    float in[height][width], float filter[5]) {

      Filter<float> conv_x(in,filter) {
             .border = BORDER;
             .function= ${
               @conv_x[0][0]=filter[0]*$in[0][-2] + filter[1]*$in[0][-1] +
                             filter[2]*$in[0][0]  +
                              filter[3]*$in[0][1]  + filter[4]*$in[0][2];
             }$;
      };

      Filter<float> conv_y(conv_x,filter) {
             .border = BORDER;
             .function= ${
               @conv_y[0][0]=filter[0]*$conv_x[-2][0] + filter[1]*$conv_x[-1][0] +
                             filter[2]*$conv_x[0][0]  +
                              filter[3]*$conv_x[1][0]  + filter[4]*$conv_x[2][0];
             }$;
      };

      return conv_y;
}
```

5.1. Upsampling with a graph of `Filter` nodes

In this case, we want to create an upsampling kernel with a simple interpolation formula that creates intermediate pixels as the average of two successive pixels in the input image in the x- and y- axis.

```
kernel Upsampling(int width=[1:1920], int height=[1:1080],
                  float in[height][width]) {
  Filter<float> gx(in) {
    .border = duplicate;
    .stride_out = {2};
    .function = ${
      @gx[0][0]=$in[0][0];
      @gx[0][1]=0.5f*($in[0][0]+$in[0][1]);
     }$;
  };
```

```
    Filter<float> gy(gx) {
      .border = duplicate;
      .stride_out = {1,2};
      .function = ${
        @gy[0][0]=$gx[0][0];
        @gy[1][0]=0.5f*($gx[0][0]+$gx[1][0]);
       }$;
    };

    return gy;
  }
}
```

5.2. *Gradient Filter*

The gradient filter computes 2 values for each pixel of the input image that are the length and the angle of the gradient vector. This gradient is computed from a 3x3 windows, taking the average distance per pixel in the x- and y- axis: 0.5*(pixel[1] - pixel[-1]). At the border of the image, the formula changes and this average per pixel becomes on the bottom and right edge of the image : pixel[0] - pixel[-1], and on the top and the left of the image pixel[1] - pixel[0]. Since the `function` of the `Filter` node stays the same for all pixels of the input image, we specify an expression determining the value of the pixels out of the image that is necessary to get the correct formula at the edge. For instance, at the right edge of the input image, gx will take as value 0.5*((2*pixel[0]-pixel[-1]) - pixel[-1]) which is equal to
pixel[0] - pixel[-1] as expected at this border. The stride and access patterns of the gradient are depicted in Figure 5.

```
kernel Gradient3x3(int width=[1:1920], int height=[1:1080],
                   float in[height][width]) {

      Filter<float> grad(in) {
            // $[0] is the pixel of the border
            // $[1] is the of the pixel inside the image next to the border
            .border = "2*$[0]-$[1]";
            .stride_out = {2};
            .function = ${
               float gx=0.5f*($in[0][1]-$in[0][-1]);
               float gy=0.5f*($in[1][0]-$in[-1][0]);
               @grad[0][0]= sqrt(gx*gx+gy*gy);
               @grad[0][1]= atan2(gy, gx);
             }$;
      };

      return grad;
}
```

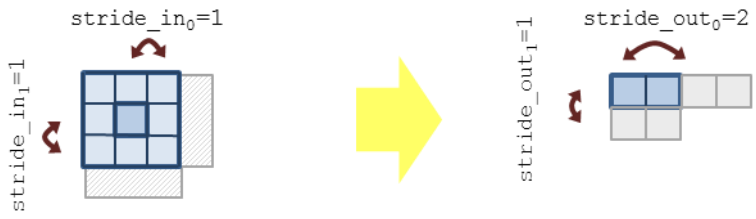


Figure 6: Gradient data patterns