<http://docs.mitk.org/nightly/MitkOpenCL_BinaryThresholdFilter.html>

Implementing Own OpenCL-based Image Filter

This example demonstrates how to implement an image-to-image filter with some computations done on an OpenCL device (most likely a GPU). This particular example presents a simple threshold filter, which uses an upper and lower threshold to map the pixel to the specified inside and outside value.

Since the filter we want to implement is an image-to-image filter, we will inherit from the [mitk::OclImageToImageFilter](http://docs.mitk.org/nightly/classmitk_1_1OclImageToImageFilter.html) class.

In the constructor, we set the source file of the filter. In this case, it is the file located in the default directory placed in the module itself. We also set the unique filter id string, which serves as an identifier for storing and getting the associated program through the [OclResourceService](http://docs.mitk.org/nightly/classOclResourceService.html).

std::string path = "BinaryThresholdFilter.cl";

this->SetSourceFile( path.c\_str() );

this->m\_FilterID = "BinaryThreshold";

if ( OclFilter::Initialize() )

{

this->m\_ckBinaryThreshold = clCreateKernel( this->m\_clProgram, "ckBinaryThreshold", &clErr);

buildErr |= [CHECK\_OCL\_ERR](http://docs.mitk.org/nightly/mitkOclUtils_8h.html#a9b9284ab430935b639a286fc7f334ecc)( clErr );

}

For preparing the execution we call the InitExec method provided by the superclass. We also catch exceptions signalizing some initialization errors

try

{

this->InitExec( this->m\_ckBinaryThreshold );

}

catch( const [mitk::Exception](http://docs.mitk.org/nightly/classmitk_1_1Exception.html)& e)

{

[MITK\_ERROR](http://docs.mitk.org/nightly/mitkLogMacros_8h.html#a5b2267738332420c278264d869c1aa55) << "Catched exception while initializing filter: " << e.what();

return;

}

In this example, the header of the OpenCL-Kernel to be executed is the following

\_\_kernel void ckBinaryThreshold(

\_\_read\_only image3d\_t dSource, // input image

\_\_global uchar\* dDest, // output buffer

int lowerT, int upperT, int outsideVal, int insideVal // parameters

)

The InitExec method takes care of initializing the input and output images and their OpenCL counterparts. In addition, both parameters are passed to the filter and the input data is copied (if necessary). If the first initialization part was successful, we continue by passing in the values for the filter specific parameters:

clErr = clSetKernelArg( this->m\_ckBinaryThreshold, 2, sizeof(cl\_int), &(this->m\_lowerThr) );

clErr |= clSetKernelArg( this->m\_ckBinaryThreshold, 3, sizeof(cl\_int), &(this->m\_upperThr) );

clErr |= clSetKernelArg( this->m\_ckBinaryThreshold, 4, sizeof(cl\_int), &(this->m\_outsideVal) );

clErr |= clSetKernelArg( this->m\_ckBinaryThreshold, 5, sizeof(cl\_int), &(this->m\_insideVal) );

[CHECK\_OCL\_ERR](http://docs.mitk.org/nightly/mitkOclUtils_8h.html#a9b9284ab430935b639a286fc7f334ecc)( clErr );

If this passes, we tell the superclass to execute our compiled kernel. The second argument specifies the dimensionality of the OpenCL NDRange (the parallel execution grid). In this example we use the 3D-Grid:

this->ExecuteKernel( m\_ckBinaryThreshold, 3);

// signalize the GPU-side data changed

m\_output->Modified( [GPU\_DATA](http://docs.mitk.org/nightly/mitkOclDataSet_8h.html#a7b07a39445cd345f36e5b2c1f02be775) );

Through the last line we signalize that the OpenCL-side data of the output image is modified. This is essential for getting the result data when calling GetOutput() afterwards.