OpenCL parallel Processing using General Purpose Graphical Processing units

–TiViPE software development

Technical Report

Tino Lourens
TiViPE
Kanaaldijk ZW 11
5706 LD Helmond
The Netherlands
tino@tivipe.com

November 6, 2012

Contents

Abstract

The aim of this report to elaborate TiViPE modules that make use of Open Computing Language (OpenCL) programming. OpenCL is available in TiViPE from version 2.1.0.

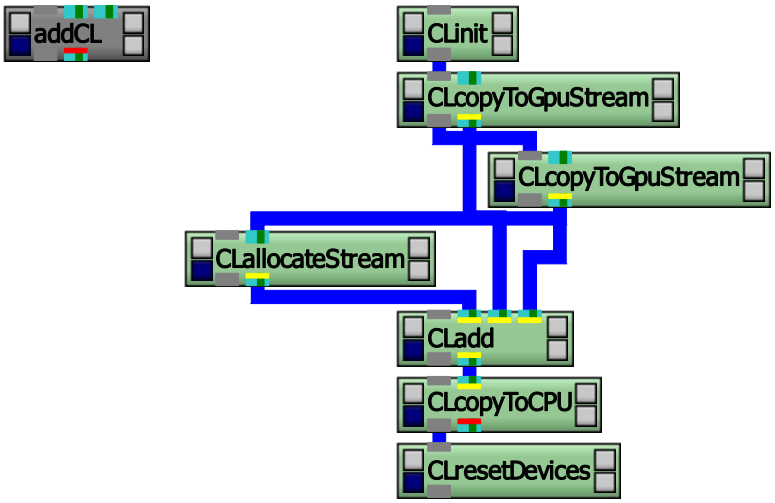

Figure 1: *OpenCL example: addCL is a merged module with two inputs and one output. On the right the atomic modules given in green.*

1 Introduction

The aim of TiViPE is to integrate different technologies in a seamless way using graphical icons [1]. Due to these icons the user does not need to have in depth knowledge of the underlying hardware architecture [2].

In this report the focus is on how to integrate the Open Computing Language (OpenCL) into TiViPE. OpenCL is an open standard for parallel programming of heterogeneous systems, it has been intitated by Apple and is coordinated by the Khronos group. The aim of OpenCL is to support parallel programming on many different computational devices, including Central Processing Units (CPUs) and Graphical Processing Units (GPUs). The terminology for CPU and GPU becomes somewhat outdated because GPUs can be used as generic processing units also known as GP-GPUs. It is the reason that AMD has redefined its processing units to Accelerated Processing Units (APU), since both CPU and GPU can be used in the same way when using the OpenCL SDK (Softwre Development Kit).

Objectively spoken OpenCL mostly benefits if the hardware is massively parallel and for the moment this implies that the GPU processing units are most suitable. The NVIDIA CUDA (Compute Unified Device Architecture) SDK is a competitor of OpenCL. The only drawback of CUDA compared to OpenCL is that only NVIDIA (GPU) hardware is supported.

OpenCL has a bunch of restrictions compared to CUDA and TiViPE has put an effort to alleviate the gap between CUDA and OpenCL. Our aim is to make OpenCL and CUDA programming very similar and make modular integration in TiViPE the same. In this document we make a step by step comparison between the 2 technologies using a simple example.

2 GPUadd using CUDA

The addCL TiViPE module illustrated in Figure 1 performs an addition for every element. In formula:

$$d_i = a_i + b_i \forall i \tag{1}$$

A very simple module to program, but several steps need to be taken to make such a module be integrated in an environment like TiViPE. Figure 1 already illustrates that the memory allocation and transfer needs to be taken in consideration. The reason to do so is that

1. allocation of memory on a GPU takes time
2. freeing memory also
3. transfer from CPU to GPU memory is time consuming
4. and vice versa the transfer from GPU to CPU memory

In the design for an efficient algorithm it is therefore important to allocate memory in the initialization phase and use the allocated memory during an iterative process. When the process has finished memory is being freed.

Modules in TiViPE require to be constructed and merged to a single module where IO to CPU memory is included, as illustrated in Figure 1. The construction of

$$e_i = a_i + b_i + c_i \forall i \tag{2}$$

can be reprogrammed but in this case be constructed using the basic building block twice:

$$d_i = a_i + b_i \forall i \tag{3}$$
$$e_i = d_i + c_i \forall i \tag{4}$$

The `CLinit` module, initializes all available modules, for all available detected OpenCL capable devices. For instance, an ASUS EEE notebook PC this can contain 2 different OpenCL devices: an "AMD E-350 Processor" a dual core E-350 CPU and a "Loveland" which is a Radeon 6310 GPU. While on a desktop this can be a single NVIDIA GPU device like "GeForce GTX 560". This `CLinit` module uses a platform number and a device number to give the user control over these devices. The module, when executed, compiles all modules to binaries when they do not exist for a given device. In some occasions this is not necessary, because the binaries used for different devices are the same. The module for this reason has 2 additional parameters to map devices by providing string commands. For instance, when the devices are "GeForce GTX 580, GeForce GTX 570" they can be mapped to the 560 gtx binaries, by providing "GeForce GTX 560,GeForce GTX 560" as mapped devices. The `CLinit` module is required as the first module for every TiViPE network that uses OpenCL. Likewise, the `CLresetDevices` is required as last module in every TiViPE network.

The `CLadd` module performs the addition as described by (1) using the generic `Data5D` datatype. The inputs and outputs are made yellow to indicate that memory resides on a OpenCL device rather than CPU memory. This holds for instance for a GPU device with own GPU memory. The module has three inputs, the first input is used to pre-allocate output memory, while the content of the second and third input are used to perform addition itself. This construction avoids the overhead of memory allocation and freeing in the `CLadd` module itself. For the three inputs memory is allocated the first time only. The second and third input copy data from CPU memory to the OpenCL device. The output in turn is copied from the device memory to CPU memory.

2.1 Cuda implementation GPUadd implementation

Figure 1 illustrates a TiViPE network using OpenCL commands. The green `CLadd` module in this figure is in essence the same as the `GPUadd` module. The code information file GPUadd.ci in TiViPE reveals that the following interface parameters have been used

```
Input("Input pre-allocated output data",
      "-preo", DATA5D, DATA5D_ANYTYPE, IGNORED) [1];
Input("Input 1",
```

```
      "-input1", DATA5D, DATA5D_ANYTYPE, IGNORED) [2];
Input("Input 2",
      "-input2", DATA5D, DATA5D_ANYTYPE, IGNORED) [3];
Output("Output",
       "-output", DATA5D, DATA5D_ANYTYPE, IGNORED) [F];
```

The term IGNORED denotes that memory does not reside on the CPU (but on the GPU).

The class and routine are given by code are `TVPgpuArithmetic` and `TVPgpuAdd`. This routine is part of library `TVPgpu` that is found in the `$(TVPHOME)/lib` directory. The source code given by files `TVPgpuarithmetic.h` and `TVPgpuarithmetic.cpp` are found in directory `$(TVPHOME)/Libraries/TVPgpu/src`. In the header file this is given by

```
  Data5D TVPgpuAdd (Data5D gpu_ret,
                    Data5D gpu_d1,
                    Data5D gpu_d2);
```

and we find the routine call itself back in file `TVPgpuarithmetic.cpp`. The routine itself checks if the parameters are set properly, if so it performs the following:

```
 TVPcudaAdd3 (gpu_ret, gpu_d1, gpu_d2);
 return gpu_ret;
```

The `TVPcudaAdd3` routine is found in the `TVPcuda` library in the file `TVPcudaarithmetic.h` it is found as

```
extern "C" TVPCUDA_EXPORT
void TVPcudaAdd3 (Data5D ret,
                  Data5D d1,
                  Data5D d2);
```

The cuda code itself is not revealed by TiViPE but it is implemented as follows

```
void TVPcudaAdd3 (Data5D ret,
                  Data5D d1,
                  Data5D d2)
{
#if defined (TVPCUDA_DEBUG) || defined (ALL_DEBUG)
  printf ("TVPcudaAdd3\n");
#endif
  /*****************************************************
   * Setup block and grid sizes
   *****************************************************/
  int w, h, len;
  TVPcudaImageSize (d1, w, h, len);
  int wh = w * h;
  int Bx, By, Bz, Gx, Gy, Gz;
  TVPcudaBlockGridSize (Bx, By, Bz, Gx, Gy, Gz,
                        w, h, len, CUDA_NO_CHECK);
  dim3 block (Bx, By, Bz);
  dim3 grid (Gx, Gy, Gz);
  /*****************************************************
```

```
   * Add
   * indata and outdata are coupled to d2 and d1,
   * respectively.
   ****************************************************/
  switch (d1.datatype)
    {
    case DATA5D_FLOAT:
      {
        float *indata1   = (float *) d1.data;
        float *indata2   = (float *) d2.data;
        float *lindata1  = indata1;
        float *lindata2  = indata2;
        float *outdata  = (float *) ret.data;
        float *loutdata = outdata;
        /***********************************************
         * iterate over the different images
         ***********************************************/
        for (int i = 0; i < len; i++)
          {
            TVPcudaAddKernel3<float><<<grid, block>>>
              (loutdata, lindata1, lindata2, w);
            lindata1 += wh;
            lindata2 += wh;
            loutdata += wh;
          }
      }
      break;
    case DATA5D_CHAR:
      /*
       * Copy all data from the case above
       * Replace all float by char
       */
    case DATA5D_UCHAR:
    case DATA5D_SHORT:
    case DATA5D_USHORT:
    case DATA5D_INT:
    case DATA5D_UINT:
    case DATA5D_LONG:
    case DATA5D_ULONG:
    case DATA5D_DOUBLE:
    case DATA5D_COMPLEXFLOAT:
    case DATA5D_COMPLEXDOUBLE:
      printf ("Float datatype is implemented only.\n");
      break;
    case DATA5D_NOTYPE:
      break;
    }
}
```

The `TVPcudaAddKernel3` routine is the so-called CUDA kernel module which is implemented as follows:

```
template <typename DType>
__global__ void TVPcudaAddKernel (DType *ret,
                                  DType *d1,
                                  DType *d2,
                                  int   width)
{
  /*****************************************************
   Add
   *****************************************************/
  int x = blockIdx.x * blockDim.x + threadIdx.x;
  int y = blockIdx.y * blockDim.y + threadIdx.y;
  int i = y * width + x;
  ret[i] = (DType) (d1[i] + d2[i]);
}
```

The code above is compiled and made available in the TVPcuda library.

3 CLadd using OpenCL

Lets try to achieve the same in OpenCL. In header `TVPclArithmetic.h` the TVPgpuAdd the routine name has changed to TVPclAdd:

```
  Data5D TVPclAdd (Data5D cl_ret,
                   Data5D cl_d1,
                   Data5D cl_d2);
```

but the parameters are the same. The implementation of this routine given in file `TVPclarithmetic.cpp` also this is very similar to the CUDA version:

```
 TVPclAddPriv (cl_ret, cl_d1, cl_d2);
 return cl_ret;
```

The `TVPclAddPriv` routine is found in the same file and not in a separated library. The routine is implemented as follows:

```
cl_event TVPclArithmetic::TVPclAddPriv (Data5D ret,
                                        Data5D d1,
                                        Data5D d2,
                                        bool   useEvent)
{
#if defined (TVPCL_DEBUG) || defined (ALL_DEBUG)
  qDebug () << "TVPclArithmetic::TVPclAddPriv" << flush;
#endif
  /*****************************************************
   * Setup block and grid sizes
   *****************************************************/
  int w, h, len;
```

```
TVPclImageSize (d1, w, h, len);
int wh = w * h;
int Bx, By, Bz, Gx, Gy, Gz;
TVPclBlockGridSize (Bx, By, Bz, Gx, Gy, Gz,
                    w, h, len, CL_NO_CHECK);
const int dims = 3;
size_t block[dims] = {Bx, By, Bz};
size_t grid[dims]  = {Gx, Gy, Gz};
/*****************************************************
 * get first kernel index of TVPclAddKernel
 *****************************************************/
int index = TVPclKernelIndex ("TVPclAddKernel");
switch (d1.datatype)
  {
  case DATA5D_CHAR:
    index += 0;
    break;
  case DATA5D_UCHAR:
    index += 1;
    break;
  case DATA5D_SHORT:
    index += 2;
    break;
  case DATA5D_USHORT:
    index += 3;
    break;
  case DATA5D_INT:
    index += 4;
    break;
  case DATA5D_UINT:
    index += 5;
    break;
  case DATA5D_LONG:
    index += 6;
    break;
  case DATA5D_ULONG:
    index += 7;
    break;
  case DATA5D_FLOAT:
    index += 8;
    break;
  case DATA5D_DOUBLE:
    index += 9;
    break;
  case DATA5D_COMPLEXFLOAT:
    index += 10;
    break;
  case DATA5D_COMPLEXDOUBLE:
```

```
      index += 11;
      break;
    case DATA5D_NOTYPE:
    default:
      error ("TVPclArithmetic", "TVPclAddPriv",
             QObject::tr ("No type has been set."));
      break;
    }
  /*******************************************************
   * Fetch appropriate kernel
   *******************************************************/
  cl_kernel kernel = TVPclKernel (index);
  /*******************************************************
   * Set arguments
   *******************************************************/
  cl_int ok;
  ok  = clSetKernelArg (kernel, 0, sizeof (cl_mem),
                        (void*) &(ret.data));
  ok |= clSetKernelArg (kernel, 1, sizeof (cl_mem),
                        (void*) &(d1.data));
  ok |= clSetKernelArg (kernel, 2, sizeof (cl_mem),
                        (void*) &(d2.data));
  ok |= clSetKernelArg (kernel, 3, sizeof (int),
                        (void*) &w);
  ok |= clSetKernelArg (kernel, 4, sizeof (int),
                        (void*) &len);
  ok |= clSetKernelArg (kernel, 5, sizeof (int),
                        (void*) &wh);
  if (ok != CL_SUCCESS)
    error ("TVPclArithmetic", "TVPclAddPriv",
           QObject::tr ("Unable to set kernel arguments."));
  /*******************************************************
   * Launch kernel
   *******************************************************/
  cl_event event = TVPclLaunchKernel (kernel, dims, block,
                                      grid, useEvent);
  return event;
}
```

The code above fetches a kernel, sets parameters as kernel arguments, and launches the kernel, and appears as easy to use as the CUDA example. In fact it was aimed to be very similar to the cuda code with the exception that it is not possible to iterate over the kernel, where the kernel is given a certain index. This whole loop needs to be performed in the OpenCL code and is found in the file `TVPclAddKernel.cl`:

```
__kernel void TVPclAddKernel<DType> (__global DType *ret,
                                     __global DType *d1,
                                     __global DType *d2,
                                     const int      width,
```

```
                                        const int       len,
                                        const int       offset)
{
  int x = get_global_id (0);
  int y = get_global_id (1);
  int i = y * width + x;
  /*****************************************************
   * add
   *****************************************************/
  for (int j = 0; j < len; j++)
    {
      ret[i] = d1[i] + d2[i];
      i += offset;
    }
}
```

This loop is the only difference compared to the `TVPcudaAddKernel3` CUDA routine. This is a structural difference with the CUDA kernels, but can be alleviated easily as given in the code above.

OpenCL does not support templates by default, hence the code above is not according to the specifications given in OpenCL version 1.1 or 1.2. Templating is thus a TiViPE extension to OpenCL. TiViPE only allows very strict templating rules. That is a templated routine name always look like

```
...clkernelname<DType> ...
...clkernelname<DType,DType1> ...
...clkernelname<DType,DType1,DType2> ...
```

and there are no spaces between routinename and comma's. Up to three different templates are supported. In case of one template only "DType" is used in case of two templates "DType" and "DType1" are used. Note that the routine name includes thes

Figure 2 illustrates the normal situation for using a routine call from a library. The standard case (as depicted in blue in the figure) a header file and a library are required. The header file contains a description of the routine that is being called which is available in the library. For NVIDIA CUDA there is a compiler available which compiles to a library, and the way a routine is called from a library is thus the same, as illustrated in green.

The OpenCL setup given in yellow is different because OpenCL code does not compile to a library, but instead it is assumed that there are OpenCL kernels represented as text strings, these kernels are internally compiled before they are launched. The concept therefore is that OpenCL assumes that the source code is available as a text string in a C or C++ program. We would like to change this to the concept of having a file with OpenCL source code, compile this to a binary object file[1], and store these into a library.

There is no way to create a library without any tools, rules, or administration. In order to get the concept working properly we need to make some rules:

1. The code above couples a kernel name "TVPclAddKernel" to a file "/src/CL/TVPclAddKernel.cl".

2. The kernel module is administrated in the file "/src/CL/TVPclKernels.txt". This is done as: "TVPclAddKernel DT12;"

[1] This binary for NVIDIA still is text, the so-called ptx code. Latter is compiled by a just in time (JIT) compiler to a real binary.

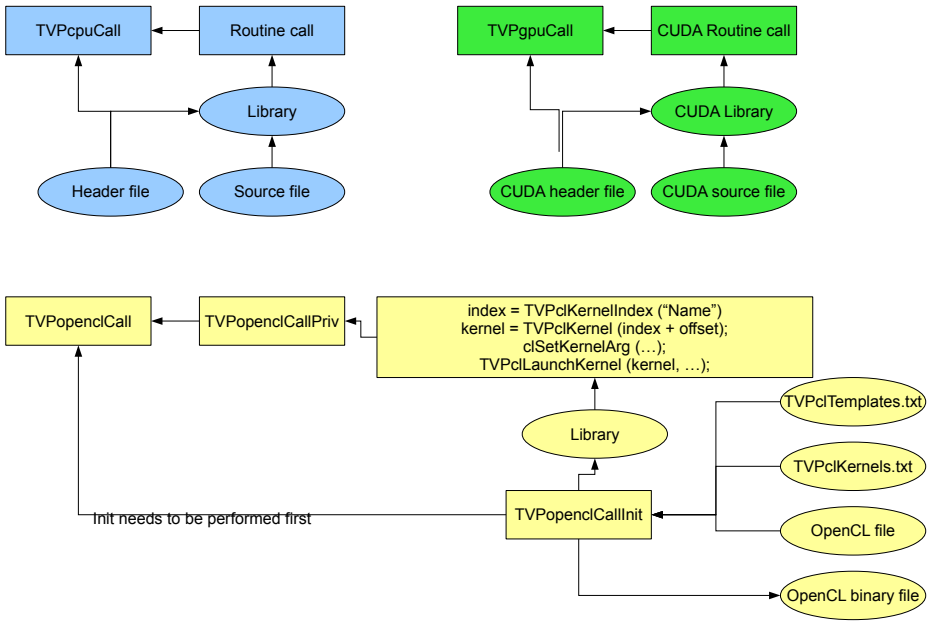

Figure 2: *Programmers infrastructure for calling a routine from a library. In blue the normal situation. In green the situation for NVIDIA CUDA. In yellow the way it is done by TiViPE for OpenCL.*

3. Template "DT12", which stands for 12 basic datatypes is coupled to "TVPclAddKernel" and this template is administrated in "/src/CL/TVPclTemplates.txt". In this file the template is given as "DT12 char 0 uchar 0 short 0 ushort 0 int 0 uint 0 long 0 ulong 0 float 0 double 1 float2 0 double2 1;". In OpenCL hardware not all devices support double floating point arithmetic, and thus for every datatype in the template a flag is created to indicate whether or not the data type requires double operand floating point arithmetic.

4. For open source development it might be preferred to use the OpenCL source code only, and (on the fly) compile the required modules. For companies, like ours it is not always possible to provide source code only. We'd prefer to be able to use solely binaries too. Binaries are not generic, so for every OpenCL capable platform (that we intend to support) we need to generate binaries. TiViPE compiles all kernels given in `TVPclKernels.txt` to "/obj/ [device] /TVPclAddKernel [DType] [DType1] [DType2].o" object files. Here `device` is the hardware device supported, and `DType, DType1, DType2` are the (optional) template datatype substitutions. For instance the TVPclAdd routine with integer datatype for a Geforce GTX 560 compile to and use the binary code obtained from "/obj/Geforce GTX 560/TVPclAddKernelint.o".

4 Programming a TiViPE module using OpenCL

Programming a module in TiViPE takes 3 steps:

1. Registration of kernel name in file "/src/CL/TVPclKernels.txt". This is done by adding a line of text, for instance "TVPclMyKernelDouble 1;", "TVPclMyKernel Datatype1 Datatype2;"), or the add kernel "TVPclAddKernel DT12;". In case the datatypes have not been created they need to be added to "/src/CL/TVPclTemplates.txt".

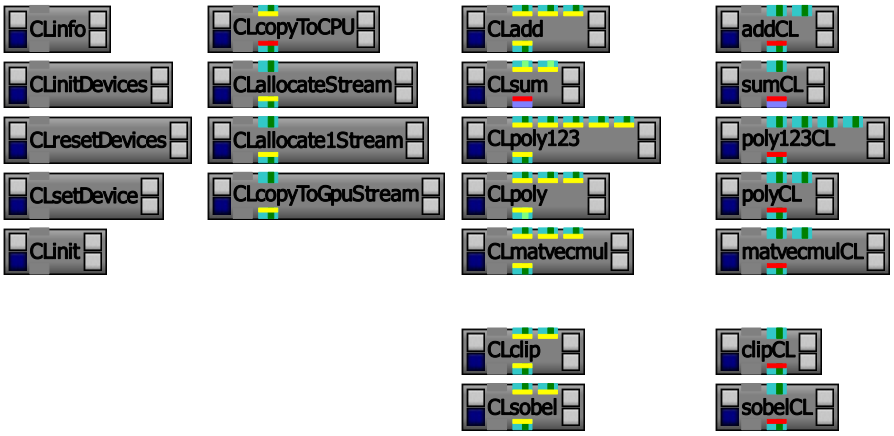

Figure 3: *All available OpenCL modules in TiViPE version 2.1.0.*

2. Write a kernel module "/src/CL/TVPclMyKernel.cl".

3. Create the interface routines.

The interface routine contains a routine to select the appropriate kernel:

```
index = TVPclKernelIndex ("TVPclAddKernel");
TVPclKernel (index + offset);
```

here the offset determines the datatype used, for "DT12" it is indexed from 0 to 11. For every kernel the arguments used, that is the input, output, and internal parameters, need to be set to the kernel. As last step the kernel is launched:

```
TVPclLaunchKernel (kernel, dims, block, grid, useEvent);
```

The rest of the OpenCL specific coding and complexity is hidden within the core TiViPE OpenCL routines.

5 OpenCL components

In TiViPE version 2.1.0 OpenCL has been incorporated and 23 new modules developed. It will grow in the following versions. The number of available modules for CUDA is currently 68, and it likely that most of them will be made available for OpenCL as well.

The current release involves 5 generic basic modules, 4 IO transfer modules, and 7 computational blocks. The 7 merged modules allow the user to use these computational blocks.

So far most effort has gone in an appropriate framework for OpenCL with the aim to easily embed new OpenCL functionality in the future.

5.1 Device information and selection

Obtaining information from all OpenCL devices is obtained by using `CLinfo`, for a graphical representation see the left side of Figure 3. This module gives, depending on the hardware used, the following output:

```
==============================
Available kernels on Loveland
TVPclAddKernel : char uchar short ushort int uint long ulong
                 float float2
...
==============================
Available kernels on AMD E-350 Processor
TVPclAddKernel : char uchar short ushort int uint long ulong
                 float double float2 double2
...
==============================
Number of Platforms: 1
==============================
Platform Number: 0
Platform Name: AMD Accelerated Parallel Processing
Platform Profile: FULL_PROFILE
Platform Version: OpenCL 1.1 AMD-APP (851.4)
Platform Vendor: Advanced Micro Devices, Inc.
Platform Extensions: cl_khr_icd cl_amd_event_callback ...
Number of Devices: 2
------------------------------
Device Number: 0
Name: Loveland
Type: CL_DEVICE_TYPE_GPU
Profile: FULL_PROFILE
Version: OpenCL 1.1 AMD-APP (851.4)
Vendor: Advanced Micro Devices, Inc.
Vendor Id: 4098
Extensions: cl_khr_global_int32_base_atomics
            cl_khr_global_int32_extended_atomics ...
Driver Version: CAL 1.4.1664 (VM)
Address Bits: 32
Available: Yes
Compiler Available:Yes
Double Precision Floating Point Capability
* Denorms: No
* Quiet NaNs: No
* Round to nearest even: No
* Round to zero: No
* Round to +ve and infinity: No
* IEEE754-2008 fused multiply-add: No
...
------------------------------
Device Number: 1
Name: AMD E-350 Processor
Type: CL_DEVICE_TYPE_CPU
Profile: FULL_PROFILE
Version: OpenCL 1.1 AMD-APP (851.4)
```

. . .

5.2 Data transportation

The basic components for data handling on OpenCL devices are data allocation and freeing, data transfer on the device and from device to CPU and vice versa. The available modules are illustrated in the second column of Figure 3.

5.3 Computational modules

The available computational modules illustrated in the rightmost 2 columns of Figure 3 is divided in arithmetic (add, sum, matrix vector multiplication, and 2 polynomial functions) and filters (clipping and Sobel).

References

[1] T. Lourens. Tivipe –tino's visual programming environment. In *The 28th Annual International Computer Software & Applications Conference, IEEE COMPSAC 2004*, pages 10–15, 2004.

[2] T. Lourens. The art of parallel processing using general purpose graphical processing units – hardware, cuda introduction, and software architecture. Technical report, TiViPE, 2009.