HIPA^{cc} Documentation

The Heterogeneous Image Processing Acceleration Framework
Version 0.6.1

Richard Membarth

May 28, 2013

Contents

1 Installation **3**
1.1 Dependencies 3
1.2 Heterogeneous Image Processing Acceleration (HIPA^{cc}) Installation 3

2 Domain Specific Language **6**
2.1 Built-in C++ Classes 6
2.2 Defining Image Operators 8
2.3 Memory Management 11
2.4 Supported Data Types and Operations 12

3 Framework Usage **15**
3.1 Optimization Selection 15
3.2 Sample Invocation 16
3.3 Supported Target Hardware 17

1 Installation

To install the HIPA^{cc} framework, which features a Clang-based source-to-source compiler, a version of Clang is required that was tested with HIPA^{cc}. Therefore, the file `dependencies.sh` lists the revision of Clang (and other dependencies) the current version of HIPA^{cc} works with. In addition to Clang, also LLVM and libcxx are required. Using Polly is optional.

1.1 Dependencies

- Clang: a C language family frontend for LLVM
- LLVM compiler infrastructure
- libc++: C++ standard library
- Polly: polyhedral optimizations for LLVM (optional)

Installation of dependencies:

```
cd <source_dir>
git clone http://llvm.org/git/llvm.git
cd llvm && git checkout <llvm_revision>
cd <source_dir>
git clone http://llvm.org/git/libcxx.git
cd libcxx && git checkout <libcxx_revision>
cd <source_dir>/llvm/tools
git clone http://llvm.org/git/clang.git
cd clang && git checkout <clang_revision>
// optional: installation of polly
cd <source_dir>/llvm/tools
git clone http://llvm.org/git/polly.git
cd polly && git checkout <polly_revision>
```

Configure the software packages using CMake, build and install using `make install`. Note: On GNU/Linux systems, libc++ has to be built using clang/clang++. The easiest way to do this is to use CMake and to specify the compilers at the command line: `CXX=clang++ CC=clang cmake ../ -DCMAKE_INSTALL_PREFIX=/opt/local`
Note: On Mac OS 10.6, `cxxabi.h` from http://home.roadrunner.com/ hinnant/libcp-pabi.zip is required to build libc++ successfully.

1.2 HIPA^{cc} Installation

Next, you have to download HIPA^{cc} from http://hipacc.sourceforege.net. HIPA^{cc} can be downloaded either as versioned tarball, or the latest version can be retrieved using git. Download the latest release (currently, the tarball hipacc-0.6.1.tar.gz) or get the latest sources using git:
`git clone git://git.code.sf.net/p/hipacc/code hipacc`

To build and install the HIPA^{cc} framework, CMake is used. In the main directory, the file INSTALL contains required instructions:

To configure the project, call cmake in the root directory. A working installation of Clang/LLVM (and Polly) is required. The llvm-config tool will be used to determine configuration for HIPA^{cc} and must be present in the environment.

The following variables can be set to tell cmake where to look for certain components:

- CMAKE_INSTALL_PREFIX: Installation prefix
- CMAKE_BUILD_TYPE: Build type (Debug or Release)
- OPENCL_INC_PATH: OpenCL include path
  (e.g., -DOPENCL_INC_PATH=/opt/cuda/include)
- OPENCL_LIB_PATH: OpenCL library path
  (e.g., -DOPENCL_LIB_PATH=/usr/lib64)
- CUDA_BIN_PATH: CUDA binary path
  (e.g., -DCUDA_BIN_PATH=/opt/cuda/bin)

The following options can be enabled or disabled:

- USE_POLLY: Use Polly for kernel analysis (e.g., -DUSE_POLLY=ON)
- USE_JIT_ESTIMATE: Use just-in-time compilation of generated kernels in order to get resource estimates - option only available for GNU/Linux systems

A possible configuration may look like in the following:

```
cd <hipacc_root>
mkdir build && cd build
mkdir install_dir
cmake ../ -DCMAKE_INSTALL_PREFIX=./install_dir
make && make install
```

Android Cross Compilation: Generating target code for Android requires cross compilation. Therefore, additional variables are recognized for cross compilation for Android (given are examples to target the Arndale Board with a Samsung Exynos 5 Dual (ARM Cortex-A15 & ARM Mali T604 GPU):

- ANDROID_SOURCE_DIR: Android source directory
  (e.g. -DANDROID_SOURCE_DIR=/opt/arndaleboard/android-jb-mr1)
- TARGET_NAME: Name of the target platform
  (e.g. -DTARGET_NAME=arndale)
- HOST_TYPE: Name of the local compile host type
  (e.g. -DHOST_TYPE=linux-x86)

- NDK_TOOLCHAIN_DIR: Android NDK directory
  (e.g. -DNDK_TOOLCHAIN_DIR=/opt/android/android-14-toolchain)
- RS_TARGET_API: Android API level
  (e.g. -DRS_TARGET_API=16)
- EMBEDDED_OPENCL_INC_PATH: OpenCL include path
  (e.g. -DEMBEDDED_OPENCL_INC_PATH=/opt/cuda/include)
- EMBEDDED_OPENCL_LIB_PATH: OpenCL library path within the target system
  (e.g. -DEMBEDDED_OPENCL_LIB_PATH=vendor/lib/egl)
- EMBEDDED_OPENCL_LIB: Name of the embedded OpenCL library
  (e.g. -DEMBEDDED_OPENCL_LIB=libGLES_mali.so)

2 Domain Specific Language

The Domain-Specific Language (DSL) can be separated into two parts: a) host code that describes the binding to standard C/C++ code and b) device code that describes the calculation on the graphics card.

2.1 Built-in C++ Classes

The library consists of built-in C++ classes that describe the following three basic components required to express image processing on an abstract level:

- *Image*: Describes data storage for the image pixels. Each pixel can be stored as an integer number, a floating point number, or in another format depending on instantiation of this templated class. The data layout is handled internally using multi-dimensional arrays. Syntax:

```
1 Image<type>(width, height);
```

- *Iteration Space*: Describes a rectangular region of interest in the output image, for example the complete image. Each pixel in this region is a point in the iteration space. Syntax:

```
1 IterationSpace<type>(Image, width, height, offset_x, offset_y);
```

  width, height, offset_x and offset_y are optional.

- *Kernel*: Describes an algorithm to be applied to each pixel in the *Iteration Space*. Syntax:

```
1 Kernel(IterationSpace, <parameters>);
```

  The parameters are defined in the kernel class itself by the user.

- *Mask*: Stores the coefficients that can be used by convolution kernels. Syntax:

```
1 Mask<type>(size_x, size_y);
```

- *BoundaryCondition*: Describes how the pixels of an *Accessor* are accessed when pixels are accessed out-of-bounds. The following boundary handling modes are supported:
  - `BOUNDARY_UNDEFINED`: No border handling specified → no border handling will be added by the compiler.
  - `BOUNDARY_CLAMP`: The x/y addresses will be set to the last valid value within the image.

- `BOUNDARY_REPEAT`: Accesses outside to the image are handled as if the image is repeated in each direction.
- `BOUNDARY_MIRROR`: Accesses outside to the image are handled as if the image is mirrored at the border.
- `BOUNDARY_CONSTANT`: Accesses outside to the image return a constant value.

Syntax:

```
BoundaryCondition<type>(Image, size_x, size_y,
    boundary_handling_mode, constant_value);
```

size_x and size_y define the domain where boundary handling is necessary (e. g., within a 5×5 convolution filter); boundary_handling_mode is one of the aforementioned constants. In case `BOUNDARY_CONSTANT` is used, the optional constant_value has to be specified.

- *Accessor*: Describes which pixels of an *Image* are *seen* within the *Kernel*. Similar to an *Iteration Space*, the Accessor defines an *Iteration Space* on an input image. Syntax:

```
Accessor<type>(Image, width, height, offset_x, offset_y);
```

width, height, offset_x and offset_y are optional.

In order to avoid out-of-bounds memory accesses, also a *BoundaryCondition* object can be specified instead of an *Image*. Syntax:

```
Accessor<type>(BoundaryCondition, width, height, offset_x,
    offset_y);
```

width, height, offset_x and offset_y are optional.

In case the *Iteration Space* (defined for the output image) does not match the region of interest defined by the *Accessor* (defined for an input image), interpolation is required. Therefore, HIPA^{cc} provides *Accessors* implementing different interpolation modes. Supported are nearest neighbor, linear filtering, bicubic filtering, and Lanczos filtering. Syntax:

```
AccessorNN<type>(Image, width, height, offset_x, offset_y);
AccessorLF<type>(Image, width, height, offset_x, offset_y);
AccessorCF<type>(Image, width, height, offset_x, offset_y);
AccessorL3<type>(Image, width, height, offset_x, offset_y);
```

Interpolation can be also combined with border handling. Syntax:

```
AccessorNN<type>(BoundaryCondition, width, height, offset_x,
    offset_y);
```

2.2 Defining Image Operators

In the following, the HIPA^{cc} framework is illustrated using a Gaussian filter, smoothing an image. By doing so, the Gaussian filter reduces image noise and detail. This filter is a local operator that is applied to a neighborhood (σ) of each pixel to produce a smoothed image (see Equation (1)). The filter mask of the Gaussian filter as described in Equation (2) depends only on the size of the considered neighborhood (σ) and is otherwise constant for the image. Therefore, the filter mask is typically precalculated and stored in a lookup table to avoid redundant calculations for each image pixel.

$$I_{Out}(x,y) = \sum_{ox=-\sigma}^{+\sigma} \sum_{oy=-\sigma}^{+\sigma} G((x,y),(x+ox,y+oy)) * I_{In}(x+ox,y+oy) \quad (1)$$

$$G((x,y),(x',y')) = \frac{1}{2\pi\sigma^2} e^{-\frac{\|(x,y)-(x',y')\|}{2\sigma^2}} \quad (2)$$

Device Code: To express this filter in our framework, the programmer derives a class from the built-in *Kernel* class and implements the virtual *kernel* function, as shown in Listing 1. To access the pixels of an input image, the parenthesis operator `()` is used, taking the column (`dx`) and row (`dy`) offsets as optional parameters. Similarly, coefficients of a filter *Mask* are accessed using the parenthesis operator `()`, specifying the desired column (`x`) and row (`y`) index. The output image as specified by the *Iteration Space* is accessed using the `output()` method provided by the built-in *Kernel* class. The user instantiates the class with input image accessors, one iteration space, and other parameters that are member variables of the class.

```
class GaussianFilter : public Kernel<float> {
  private:
    Accessor<float> &Input;
    const int size_x, size_y;

  public:
    GaussianFilter(IterationSpace<float> &IS, Accessor<float> &Input,
        const int size_x, const int size_y) :
      Kernel(IS),
      Input(Input),
      size_x(size_x),
      size_y(size_y)
    { addAccessor(&Input); }

    void kernel() {
      const int ax = size_x >> 1;
```

```
16        const int ay = size_y >> 1;
17        float sum = 0;
18
19        for (int yf = -ay; yf<=ay; yf++) {
20          for (int xf = -ax; xf<=ax; xf++) {
21            float gauss_constant = expf(-1.0f*((xf*xf)/(2.0f*size_x*
                  size_x) + (yf*yf)/(2.0f*size_y*size_y)));
22            sum += gauss_constant*Input(xf, yf);
23          }
24        }
25        output() = sum;
26      }
27  };
```

Listing 1: Gaussian filter, calculating the Gaussian mask for each pixel.

While in Listing 1, the Gaussian filter mask was calculated for each pixel (according to Equation (2)), the Gaussian filter mask can be precalculated and stored to a `Mask`. This is shown in Listing 2 where the mask coefficient is retrieved from a `Mask` object.

```
1  class GaussianFilter : public Kernel<float> {
2    private:
3      Accessor<float> &Input;
4      Mask<float> &cMask;
5      const int size_x, size_y;
6
7    public:
8      GaussianFilter(IterationSpace<float> &IS, Accessor<float> &Input,
           Mask<float> &cMask, const int size_x, const int size_y) :
9        Kernel(IS),
10       Input(Input),
11       cMask(cMask),
12       size_x(size_x),
13       size_y(size_y)
14     { addAccessor(&Input); }
15
16     void kernel() {
17       const int ax = size_x >> 1;
18       const int ay = size_y >> 1;
19       float sum = 0;
20
21       for (int yf = -ay; yf<=ay; yf++) {
22         for (int xf = -ax; xf<=ax; xf++) {
23           sum += cMask(xf, yf)*Input(xf, yf);
24         }
25       }
26       output() = sum;
27     }
28 };
```

Listing 2: Gaussian filter, using a precalculated the Gaussian mask.

As an alternative, the convolution can be expressed using the `convolve` method, taking three parameters: a) the mask itself, b) the reduction operator for each element, and c)

the calculation instruction for one element of the mask with pixels of the image. This is shown in Listing 3.

```
1 class GaussianFilter : public Kernel<float> {
2   private:
3     Accessor<float> &Input;
4     Mask<float> &cMask;
5
6   public:
7     GaussianFilter(IterationSpace<float> &IS, Accessor<float> &Input,
          Mask<float> &cMask) :
8       Kernel(IS),
9       Input(Input),
10      cMask(cMask)
11    { addAccessor(&Input); }
12
13    void kernel() {
14      output() = convolve(cMask, HipaccSUM, [&] () -> float {
15        return cMask()*Input(cMask);
16        });
17    }
18 };
```

Listing 3: Gaussian filter, using the convolve function.

Host Code: In Listing 4, the input and output *Image* objects `IN` and `OUT` are defined as two-dimensional $W \times H$ grayscale images, having pixels represented as floating-point numbers (lines 10–11). The *Image* object `IN` is initialized with the `host_in` pointer to a plain C array (line 14). The Gaussian filter *Mask* object `GMask` is defined (line 17) and is initialized (line 18) for the filter size. Because of accessing neighboring pixels in the Gaussian filter, border handling is required. In line 21, a *Boundary Condition* object specifying mirroring as boundary mode for the filter size is defined. The region of interest `IsOut` contains the whole image (line 24) and the *Accessor* `AccIn` is defined on the input image taking the boundary condition into account (line 27). The kernel is initialized with the iteration space, accessor, and filter mask objects as well as filter size parameters $size\_x$ and $size\_y$ (line 30), and executed by a call to the `execute()` method (line 33). To retrieve the output image, the `host_out` pointer is assigned the *Image* object `OUT`, invoking the `getData()` operator (line 36).

```
1 const int width=1024, height=1024, size_x=3, size_y=3;
2
3 // pointers to raw image data
4 float *host_in = ...;
5 float *host_out = ...;
6 // pointer to Gaussian filter mask
7 float *filter_mask = ...;
8
9 // input and output images
10 Image<float> IN(width, height);
11 Image<float> OUT(width, height);
```

```
12
13 // initialize input image
14 IN = host_in; // operator=
15
16 // filter Mask for Gaussian filter
17 Mask<float> GMask(size_x, size_y);
18 GMask = filter_mask;
19
20 // Boundary handling mode for out of bounds accesses
21 BoundaryCondition<float> BcInMirror(IN, GMask, BOUNDARY_MIRROR);
22
23 // define region of interest
24 IterationSpace<float> IsOut(OUT);
25
26 // Accessor used to access image pixels with the defined boundary
      handling mode
27 Accessor<float> AccIn(BcInMirror);
28
29 // define kernel
30 GaussianFilter GF(IsOut, AccIn, GMask, size_x, size_y);
31
32 // execute kernel
33 GF.execute();
34
35 // retrieve output image
36 host_out = OUT.getData();
```

Listing 4: Host code, instantiating and executing the Gaussian filter.

2.3 Memory Management

For each *Image* defined in the DSL, memory is allocated on the compute device (e. g., GPU). Synchronization between the memory allocated on the compute device and the data assigned to the *Image* instance is explicitly done by the programmer. Assigning a memory pointer to an *Image* triggers the memory transfer from the host to the compute device. Copying the data back to the host is initiated by the `getData()` operator.

Once the data is on the compute device, data can be directly copied between *Images* and *Accessors*. Listing 5 shows the possibilities of memory assignments between *Images* and *Accessors* as well as the data transfer to and from the compute device.

```
1 // input Image
2 int width, height;
3 uchar *image = read_image(&width, &height, "input.pgm");
4 Image<uchar> IN(width, height);
5
6 // copy data to the device: host -> device
7 IN = image;
8
9 // define second Image
10 Image<uchar> TMP(width, height);
```

```
11
12 // copy from IN to TMP: device -> device
13 TMP = IN;
14
15
16 // define ROI on IN (Accessor)
17 Accessor<uchar> AccIn(IN, roi_width, roi_height, offset_x, offset_y);
18
19 // define ROI on TMP (Accessor)
20 Accessor<uchar> AccTmp(TMP, roi_width, roi_height, 0, 0);
21
22 // copy from ROI on IN to ROI on TMP: device -> device
23 AccTmp = AccIn;
24
25
26 // output image
27 Image<uchar> OUT(roi_width, roi_height);
28
29 // copy from ROI on TMP to OUT: device -> device
30 OUT = AccTmp;
31
32 // copy from Accessor to Image: device -> device
33 AccTmp = OUT;
34
35 // copy data from device to host: device -> host
36 OUT.getData();
```

Listing 5: Data transfer possibilities in HIPA^{cc}.

2.4 Supported Data Types and Operations

Data Types: HIPA^{cc} supports all built-in (primitive) data types supported in C/C++ and provides vector types (currently only with 4 vector elements) for these data types. Table 1 lists the supported built-in data types in C/C++ and the corresponding scalar and vector data types in HIPA^{cc}.

Convert Functions: While casting and implicit conversion between built-in scalar data types is provided by the C/C++ languages, no such support is provided for vector data types. In order to convert between different vector data types, *convert* functions are provided by the HIPA^{cc} framework (see Table 2).

Math Functions: Standard math functions (math.h / cmath) are supported on scalar data types. For vector data types, corresponding math functions are provided by HIPA^{cc} in the *hipacc::math* namespace. Listing 6 shows the usage of vector types and math functions on vector types.

```
1 using namespace hipacc;
2 using namespace hipacc::math;
3
```

C/C++ built-in type	scalar type	vector type
char	char	char4
unsigned char	uchar	uchar4
short	short	short4
unsigned short	ushort	ushort4
int	int	int4
unsigned int	uint	uint4
long	long	long4
unsigned long	ulong	ulong4
float	float	float4
double	double	double4

Table 1: Supported built-in types and vector types by the HIPA^{cc} framework.

convert function	return type	argument type
convert_char4	char4	any vector data type
convert_uchar4	uchar4	any vector data type
convert_short4	short4	any vector data type
convert_ushort4	ushort4	any vector data type
convert_int4	int4	any vector data type
convert_uint4	uint4	any vector data type
convert_long4	long4	any vector data type
convert_ulong4	ulong4	any vector data type
convert_float4	float4	any vector data type
convert_double4	double4	any vector data type

Table 2: Convert functions for vector types provided by the HIPA^{cc} framework.

```
4 ushort4 pixel_s = { 0, 0, 0, 0};
5
6 uchar4 pixel;
7 pixel.x = 204;
8 pixel.y = 0;
9 pixel.z = 0;
10 pixel.w = 0;
11
12 float4 tmp;
13 // using sin from hipacc::math
14 tmp = sin(convert_float4(pixel));
15 // calling sin from hipacc::math directly
16 tmp = hipacc::math::sin(convert_float4(pixel));
17
18 pixel_s = convert_uchar(tmp);
```

Listing 6: Example usage of vector types and math functions.

Using vector types, the Gaussian filter can be also applied to images using 4-channel pixels as shown in Listing 7.

```
1 class GaussianFilter : public Kernel<uchar4> {
2   private:
3     Accessor<uchar4> &Input;
4     Mask<float> &cMask;
5
6   public:
7     GaussianFilter(IterationSpace<uchar4> &IS, Accessor<uchar4> &
        Input, Mask<float> &cMask) :
8       Kernel(IS),
9       Input(Input),
10      cMask(cMask)
11    { addAccessor(&Input); }
12
13    void kernel() {
14      output() = convert_uchar4(convolve(cMask, HipaccSUM, [&] () ->
          float4 {
15            return cMask() * convert_float4(Input(cMask));
16            }));
17    }
18 };
```

Listing 7: Gaussian filter on 4 channel pixels, using the convolve function.

3 Framework Usage

In order to generate target code for a Graphics Processing Unit (GPU) accelerator, the user invokes the `hipacc` compiler providing an input file and specifying the output file using the `-o <file>` option. In addition, the `-target <n>` option specifies the target hardware. Supported devices are listed in Table 3.

3.1 Optimization Selection

The code variant (i. e., combination of optimizations) for a particular target device is automatically chosen by the HIPA^{cc} framework according to an expert system and based on heuristics. For manual testing, the user can enable or disable optimizations using corresponding command line options. For example, the user can specify that local memory or texture memory should be turned on or off. Similar, the amount of padding or the unroll factor can be set by the user. The `--time-kernels` compiler flag generates code that executes each kernel 10 times for calculating the execution time. This timing information (in ms) can be retrieved for the kernel executed last using the `hipaccGetLastKernelTiming()` function.

Below, all options of the source-to-source compiler are listed.

```
membarth@codesign75:~/projects/hipacc/build/release > ./bin/hipacc --help

Copyright (c) 2012, University of Erlangen-Nuremberg
Copyright (c) 2012, Siemens AG
Copyright (c) 2010, ARM Limited
All rights reserved.

OVERVIEW: HIPAcc - Heterogeneous Image Processing Acceleration framework

USAGE:  hipacc [options] <input>

OPTIONS:

  -emit-cuda               Emit CUDA code; default is OpenCL code
  -emit-opencl-cpu         Emit OpenCL code for CPU devices, no padding supported
  -emit-renderscript       Emit Renderscript code for Android
  -emit-renderscript-gpu   Emit Renderscript code for Android (force GPU execution)
  -emit-filterscript       Emit Filterscript code for Android
  -emit-padding <n>        Emit CUDA/OpenCL/Renderscript image padding, using alignment of <n> bytes for GPU devices
  -target <n>              Generate code for GPUs with code name <n>.
                           Code names for CUDA/OpenCL on NVIDIA devices are:
                             'Tesla-10', 'Tesla-11', 'Tesla-12', and 'Tesla-13' for Tesla architecture.
```

```
23                              'Fermi-20' and 'Fermi-21' for Fermi
                                   architecture.
24                              'Kepler-30' and 'Kepler-35' for Kepler
                                   architecture.
25                            Code names for for OpenCL on AMD devices
                                   are:
26                              'Evergreen'      for Evergreen
                                   architecture (Radeon HD5xxx).
27                              'NorthernIsland' for Northern Island
                                   architecture (Radeon HD6xxx).
28                            Code names for for OpenCL/Renderscript on
                                   ARM devices are:
29                              'Midgard' for Mali-T6xx' for Mali.
30    -explore-config         Emit code that explores all possible kernel
          configuration and print its performance
31    -use-config <nxm>       Emit code that uses a configuration of nxm
         threads, e.g. 128x1
32    -time-kernels           Emit code that executes each kernel
         multiple times to get accurate timings
33    -use-textures <o>       Enable/disable usage of textures (cached)
         in CUDA/OpenCL to read/write image pixels - for GPU devices only
34                            Valid values for CUDA on NVIDIA devices: '
                                   off', 'Linear1D', 'Linear2D', and '
                                   Array2D'
35                            Valid values for OpenCL: 'off' and 'Array2D
                                   '
36    -use-local <o>          Enable/disable usage of shared/local memory
          in CUDA/OpenCL to stage image pixels to scratchpad
37                            Valid values: 'on' and 'off'
38    -vectorize <o>          Enable/disable vectorization of generated
         CUDA/OpenCL code
39                            Valid values: 'on' and 'off'
40    -pixels-per-thread <n>  Specify how many pixels should be
         calculated per thread
41    -o <file>               Write output to <file>
42    --help                  Display available options
43    --version               Display version information
```

3.2 Sample Invocation

The installation of the HIPA^{cc} framework provides a set of example programs and a Makefile for getting started easily. The installation directory contains the `tests` directory with sample programs. Setting the `TEST_CASE` environment variable to one of these directories and the `HIPACC_TARGET` for the graphics card in the system is all to get started. Afterwards, the `make cuda` and `make opencl` targets can be used to generate code using the CUDA and OpenCL back ends, respectively.

- TEST_CASE: directory of the example that should be compiled using the HIPA^{cc} compiler.

target architecture	supported devices
Tesla-10	NVIDIA GeForce 8800 GTX, 9800 GT, Tesla C870
Tesla-11	NVIDIA GeForce 8800 GTS, 9800 GTX
Tesla-12	NVIDIA GeForce GT 240
Tesla-13	NVIDIA GeForce GTX 285, Tesla C1060
Fermi-20	NVIDIA GeForce GTX 590, Tesla C2050
Fermi-21	NVIDIA GeForce GTX 560 Ti
Kepler-30	NVIDIA GeForce GTX 680
Kepler-35	NVIDIA GeForce GTX TITAN
Evergreen	AMD Radeon HD 58xx
NorthernIsland	AMD Radeon HD 69xx
Midgard	ARM Mali T6xx

Table 3: Target architecture and sample GPU devices supported by the HIPA^{cc} framework.

- HIPACC_TARGET: specified the target architecture for which the compiler should optimize for

Here are sample definition of these variables:

```
# compile the bilateral filter example
export TEST_CASE=./tests/bilateral_filter

# generate target code for a Quadro FX 5800 graphics card from NVIDIA
export HIPACC_TARGET=Tesla-13
```

3.3 Supported Target Hardware

The target hardware as supported by HIPA^{cc} is categorized according to a target architecture. The target architecture corresponds to the code name of NVIDIA GPUs with compute capability appended and corresponds to the series specification for GPUs from AMD and ARM. Table 3 lists the devices currently supported by the HIPA^{cc} framework.

References

[MHT+12a] Richard Membarth, Frank Hannig, Jürgen Teich, Mario Körner, and Wieland Eckert. Automatic Optimization of In-Flight Memory Transactions for GPU Accelerators based on a Domain-Specific Language for Medical Imaging. In *Proceedings of the 11th International Symposium on Parallel and Distributed Computing (ISPDC)*, pages 211–218, Munich, Germany, June 2012.

[MHT+12b] Richard Membarth, Frank Hannig, Jürgen Teich, Mario Körner, and Wieland Eckert. Generating Device-specific GPU Code for Local Operators in Medical Imaging. In *Proceedings of the 26th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, pages 569–581, Shanghai, China, May 2012.

[MHT+12c] Richard Membarth, Frank Hannig, Jürgen Teich, Mario Körner, and Wieland Eckert. Mastering Software Variant Explosion for GPU Accelerators. In *Proceedings of the 10th International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Platforms (HeteroPar)*, pages 123–132, Rhodes Island, Greece, August 2012.

[MLT11] Richard Membarth, Anton Lokhmotov, and Jürgen Teich. Generating GPU Code from a High-level Representation for Image Processing Kernels. In *Proceedings of the 5th Workshop on Highly Parallel Processing on a Chip (HPPC)*, pages 270–280, Bordeaux, France, August 2011.