

# Generating GPU Code from a High-level Representation for Image Processing Kernels

Richard Membarth<sup>1\*</sup>, Anton Lokhmotov<sup>2</sup>, and Jürgen Teich<sup>1</sup>

<sup>1</sup> Hardware/Software Co-Design, Department of Computer Science,  
University of Erlangen-Nuremberg, Germany.  
{richard.membarth, teich}@cs.fau.de

<sup>2</sup> Media Processing Division, ARM,  
Cambridge, United Kingdom.  
anton.lokhmotov@arm.com

**Abstract.** We present a framework for representing image processing kernels based on decoupled access/execute metadata, which allow the programmer to specify both execution constraints and memory access pattern of a kernel. The framework performs source-to-source translation of kernels expressed in high-level framework-specific C++ classes into low-level CUDA or OpenCL code with effective device-dependent optimizations such as global memory padding for memory coalescing and optimal memory bandwidth utilization. We evaluate the framework on several image filters, comparing generated code against highly-optimized CPU and GPU versions in the popular OpenCV library.

## 1 Introduction

Computer systems are increasingly heterogeneous, as many important computational tasks, such as multimedia processing, can be *accelerated* by special-purpose processors that outperform general-purpose processors by 1–2 orders of magnitude, importantly, in terms of energy efficiency as well as in terms of execution speed.

Until recently, every accelerator vendor provided their own application programming interface (API), typically based on the C language. For example, NVIDIA’s API called CUDA C [6] targets systems accelerated with Graphics Processing Units (GPUs). In CUDA, the programmer dispatches compute-intensive data-parallel functions (*kernels*) to the GPU, and manages the interaction between the CPU and the GPU via API calls. Ryoo et al. [7] highlight the complexity of CUDA programming, in particular, the need for exploring thoroughly the space of possible implementations and configuration options. OpenCL [8], a new industry-backed standard API that inherits many traits from CUDA, aims to provide software portability across heterogeneous systems: correct OpenCL programs will run on any standard-compliant implementation. OpenCL per se, however, does not address the problem of *performance portability*; that is, OpenCL code optimized for one accelerator device may perform dismally on another, since performance may significantly depend on low-level details, such as data layout and iteration space mapping [4].

---

\* This work was partly done during the author’s internship at ARM, which was sponsored by the European Network of Excellence on High Performance and Embedded Architectures and Compilation (HiPEAC).

Low-level programming increases the cost of software development and maintenance: whilst low-level languages can be robustly compiled into efficient machine code, they effectively lack support for creating portable and composable software.

High-level languages with domain-specific features are more attractive to domain experts, who do not necessarily wish to become target system experts. To compete with low-level languages for programming accelerated systems, however, domain-specific languages should have an acceptable performance penalty.

We present a framework for image processing that allows programmers to concentrate on developing algorithms and applications, rather than on mapping them to the target hardware. While previous work shows that running the same kernels (e. g., written in OpenCL) on different hardware (from AMD and NVIDIA) can have significant impact on the performance [3], this framework serves to protect investments in software in the face of the ever changing landscape of computer systems.

The framework is implemented as a library of C++ classes (§2.1) and a Clang-based compiler producing host and device code in CUDA C and OpenCL (§2.2). Our framework is most similar in spirit to Cornwall *et al.*'s work on indexed metadata for visual effects [2] but introduces additional device-specific optimizations such as global memory padding for memory coalescing and optimal bandwidth utilization. We evaluate the framework by comparing generated code against highly-optimized CPU and GPU versions in the popular OpenCV library (§3).

## 2 Image Processing Framework

Our framework provides a library of C++ classes for representing image processing kernels (§2.1) and a source-to-source compiler for translating library constructs into host and device code in CUDA or OpenCL (§2.2). The library is based on the concept of decoupled access/execute metadata, which capture both execution constraints and memory access patterns of a kernel [4]. The compiler is built using Clang [1], an open source frontend for C-family languages.

### 2.1 Library

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 such as RGB, depending on instantiation of this templated class. The data layout is handled internally using multi-dimensional arrays.
- *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.
- *Kernel*: Describes an algorithm to be applied to each pixel in the region of interest.

These components are an instance of decoupled access/execute metadata [4]: the *Iteration Space* specification provides ordering and partitioning constraints (execute metadata); the *Kernel* specification provides a pattern of accesses to uniform memory (access metadata). Currently, the access/execute metadata is mostly implicit: we assume that the iteration space is parallel in all dimensions and has a 1:1 mapping to work-items (threads), and that the memory access pattern is obvious from the kernel code.

**Example** We illustrate our image processing framework using a grayscale vertical mean image filter, for which the output pixel with coordinates  $(x,y)$  is the average of  $D$  input column pixels:

$$\mathbf{O}_{x,y} = \frac{1}{D} \sum_{k=0}^{D-1} \mathbf{I}_{x,y+k}, \text{ where } 0 \leq x < W, 0 \leq y < H - D. \quad (1)$$

- $\mathbf{I}$  is an input image of  $W \times H$  pixels;
- $\mathbf{O}$  is an output image of  $W \times H$  pixels;
- $D$  is the *diameter* of the filter, that is, the number of input pixels over which the mean is computed (typically,  $D \ll H$ ).

To express this filter, the framework user derives a class from the built-in *Kernel* class and implements the virtual *kernel* function, as shown in Listing 1. The *kernel* function (line 10) takes an *ElementIterator* argument that represents the output pixel for which the algorithm is run. To access the pixels of an image, the parenthesis operator  $()$  is used, taking the *ElementIterator* argument as a mandatory parameter, and the column ( $dx$ ) and row ( $dy$ ) offsets as optional parameters. The user instantiates the class with input and output images, an iteration space, and other parameters that are member variables of the class.

```

1 class VerticalMeanFilter : public Kernel {
2     private:
3         Image<float> &Input, &Output;
4         int d;
5
6     public:
7         VerticalMeanFilter(IterationSpace &IS, Image<float> &Input,
8             Image<float> &Output, int d) :
9             Kernel(IS), Input(Input), Output(Output), d(d) {}
10
11         void kernel(IterationSpace::ElementIterator EI) {
12             float sum = 0.0f;
13
14             for (int k=0; k<d; ++k) {
15                 sum += Input(EI, 0, k);
16             }
17
18             Output(EI) = sum/(float)d;
19         }
20 };

```

Listing 1: The vertical mean filter expressed in our framework.

In Listing 2, 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 8–9). The *Image* object IN is initialized with the `host_in` pointer to a plain C array with raw image data, which invokes the `=` operator of the *Image* class (line 12). The region of interest VIS contains all image columns but excludes the last  $d$  rows to simplify border handling in this example (line 15). The kernel is initialized with the

iteration space object, image objects and kernel diameter  $d$  (line 18), and executed by a call to the `execute()` method (line 21). To retrieve the output image, the `host_out` pointer to a plain C data array is assigned the `Image` object `OUT`, which invokes the `getData()` operator (line 24).

```
1  const int width = 5120, height = 3200, d = 40;
2
3  // pointers to raw image data
4  float *host_in = ...;
5  float *host_out = ...;
6
7  // input and output images
8  Image<float> IN(width, height);
9  Image<float> OUT(width, height);
10
11 // initialize input image
12 IN = host_in; // operator=
13
14 // define region of interest
15 IterationSpace VIS(width, height-d);
16
17 // define kernel
18 VerticalMeanFilter VMF(VIS, IN, OUT, d);
19
20 // execute kernel
21 VMF.execute();
22
23 // retrieve output image
24 host_out = OUT.getData();
```

Listing 2: Example code that initializes and executes vertical mean filtering.

## 2.2 Compiler

This section describes the design of our source-to-source compiler and the single steps taken to create CUDA C and OpenCL code from a high-level description of image objects, iteration space objects and kernel objects.

Our source-to-source compiler is based on the latest Clang/LLVM compiler framework. The Clang frontend for C/C++ is used to parse the input files and to generate an AST representation of the source code. Our backend uses this AST representation to generate host and device code in CUDA or OpenCL.

**Kernel Code** The compiler creates the kernel code AST in multiple steps.

First, the kernel declaration is created. The kernel parameters are identified from the `Kernel` class constructor. Each variable, reference, or pointer has to be initialized in the constructor of the `Kernel` class and a corresponding kernel parameter is added to the declaration. In doing so, references to image objects are replaced by global memory pointers to the pixel type. The existing kernel method argument—the `ElementIterator`—is removed. Some additional parameters such as the image width and height are added for index calculations and for future uses like border handling.

Second, the kernel body is created from the *kernel* method of the class. To get an AST for the kernel body, the original AST is copied with certain AST nodes replaced. References to *Image* objects are replaced with references to corresponding arrays. Instead of using the *ElementIterator* to calculate the image index, the compiler adds statements at the beginning of the kernel that calculate the pixel location from the thread index and block index in CUDA C or the global indices in OpenCL. Similarly, each class member expression – access to a member variable of the kernel class – is translated to a reference to the corresponding kernel function parameter. After the translation, we get an AST that can be used for further transformations.

After transformations, the AST is pretty printed and stored to a file. During pretty printing, CUDA C and OpenCL C specific function and variable qualifiers are emitted. For example, the `__global__` qualifier in CUDA C and the `__kernel` qualifier in OpenCL are emitted for entry functions.

**Host Code** Unlike for device code, we create no AST for host code. Rather, we use Clang’s *Rewriter* functionality to change the textual representation of AST nodes, whilst leaving the nodes intact.

To invoke previously generated device kernels, the framework code gets translated into corresponding CUDA or OpenCL API calls as follows:

- *Image declarations* (line 8 and 9): Get translated into device memory allocation using `cudaMalloc` or `clCreateBuffer`.
- *Memory assignments* (line 12): Get translated into memory transfers using `cudaMemcpy` or `clEnqueueWriteBuffer`.
- *IterationSpace declaration* (line 15): Defines the kernel execution configuration.
- *Kernel declaration* (line 18): Gets translated into loading the kernel source. For CUDA C, this step is not required. For OpenCL, the kernel source is loaded from a file, an OpenCL program for the loaded source is created, and the kernel is compiled.
- *Kernel execution* (line 21): Gets translated into launching the kernel, using the execution configuration obtained from from the corresponding *IterationSpace* declaration.
- *Memory assignments* (line 24): Get translated into memory transfers using `cudaMemcpy` or `clEnqueueReadBuffer`.

In addition to the above changes, further changes are required to get proper CUDA C or OpenCL code. First of all, include directives for the CUDA C or OpenCL headers are added. In addition, the CUDA C kernel sources are included at the beginning of the file. To initialize the runtime, we add the corresponding functionality at the beginning of the main function. In particular, for OpenCL this initialization is an important part since it sets up the platform and device to be used for execution. After these changes, the generated host and device files can be compiled.

**Padding Support** To avoid conflicts in accessing image pixels in global memory, our compiler adds padding to allocated host and device memory. For host code, special memory allocation functions are required to allocate memory so that each image row is padded to get the desired data alignment. For device code, array index calculations are changed to take padding into account. The compiler handles padding automatically, given the desired alignment amount for the target device.

## 3 Results

### 3.1 Vertical Mean Filtering

A naïve parallel algorithm can run  $N = W \times (H - D)$  threads, each producing a single output element, which requires  $\Theta(ND)$  reads and arithmetic operations. A good parallel algorithm, however, must be efficient and scalable [5]. Therefore, we use an algorithm that *strips* the computation, where up to  $T$  outputs in the same strip are computed serially in two *phases* [4]: The first phase computes  $\mathbf{O}_{x,y_0}$  according to (1), while the second phase computes  $\mathbf{O}_{x,y}$  for  $y \geq y_0 + 1$  as  $\mathbf{O}_{x,y-1} + (\mathbf{I}_{x,y+D-1} - \mathbf{I}_{x,y-1})/D$ .

This algorithm performs  $\Theta(N + ND/T)$  reads and arithmetic operations, considerably reducing memory bandwidth and compute requirements for  $T \gg D$ , whilst allowing up to  $\lceil N/T \rceil$  threads to run in parallel. Thus, this algorithm trades off work efficiency against parallelism.

Listing 3 shows the implementation of this algorithm in our framework. Since our framework supports currently only a 1:1 mapping of output pixels to threads, we use the offset specification to calculate the pixel location for a 1:N mapping. We will provide special syntax for a 1:N mapping in the future.

```
1 class VerticalMeanFilterRollingSum : public Kernel {
2     ...
3     void kernel(IterationSpace::ElementIterator EI) {
4         float sum = 0.0f;
5         int t0 = EI.getY();
6
7         // first phase: convolution
8         for (int k=0; k<d; ++k) {
9             sum += Input(EI, 0, k + (t0*NT-t0));
10        }
11        Output(EI, 0, (t0*NT-t0)) = sum/(float)d;
12
13        // second phase: rolling sum
14        for (int dt=1; dt<min(NT, height-d-(t0*NT)); ++dt) {
15            int t = (t0*NT-t0) + dt;
16            sum -= Input(EI, 0, t-1);
17            sum += Input(EI, 0, t-1+d);
18            Output(EI, 0, t) = sum/(float)d;
19        }
20    }
21};
```

Listing 3: Kernel description of the vertical mean filter using a rolling sum.

We compare the performance of code generated by our framework against that of hand-written code reported in [4].<sup>3</sup> We run the vertical mean filter with different values for  $T$ , that is, changing the number of pixels calculated by one thread. Figure 1 shows the execution times of the vertical mean filter applied to an image of  $5120 \times 3200$  pixels. Processing more than one pixel increases the throughput from 0.53 Gpixel/s for  $T = 1$ , up to the peak throughput of 6.6 Gpixel/s at several points (e. g., for  $T = 528$ ).

<sup>3</sup> We use the same configuration: thread block dimensions  $128 \times 1$ , kernel diameter  $D = 40$ . However, we use Quadro FX 5800, rather than GTX 280.

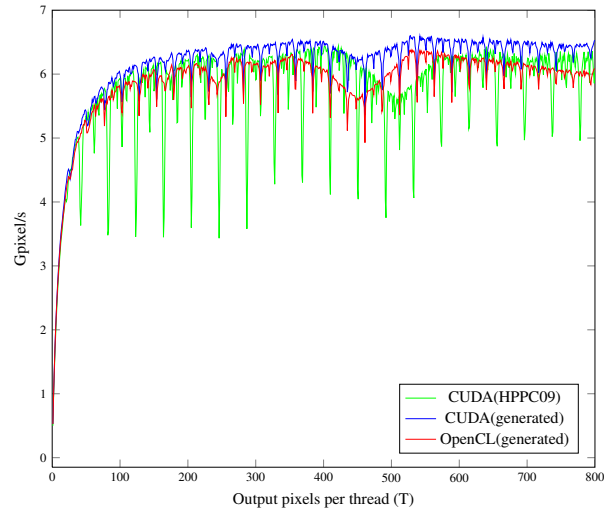


Fig. 1: Throughput of the generated CUDA C/OpenCL sources in Gpixel/s for the vertical mean filter on an image of  $5120 \times 3200$  pixels in comparison to the hand-written CUDA code from [4].

The results show that the generated CUDA code achieves the same performance as the optimized hand-written CUDA code.<sup>4</sup> However, our high-level implementation is concise and has only a fraction of the complexity of the low-level implementation of [4]. For instance, in terms of lines of code, the low-level implementation consists of about 500 lines of host and device code, whilst the high-level implementation consists of fewer than 50 lines of code.

In the previous example, the image width of 5120 is a multiple of the SIMD width of the underlying hardware (which is 32). This results in optimal memory transfers utilizing memory bandwidth best. However, if the image width is not a multiple of the SIMD width and not properly aligned, bandwidth throughput drops. For instance, increasing image width by one pixel using an image of  $5121 \times 3200$  pixels, gives us a peak throughput of 3.9 Gpixel/s which is roughly half of the throughput we got before. Using our framework allows to pad images and changes the kernel source to take padding into account. The amount of padding required for best performance depends on the underlying hardware. For the used graphics hardware, best memory throughput can be achieved when the image is padded to a multiple of the memory transaction size that can be handled by the GPU in one transaction. This size can be 32-, 64-, and 128-byte segments of aligned memory. Doing so improves the peak throughput as shown in Fig. 2 for an image of  $5121 \times 3200$  pixels with image lines padded to the different memory transaction sizes. The peak throughput of 6.4 Gpixel/s is achieved for aligning to 256-bytes, which is double the maximum transaction size.

### 3.2 OpenCV Library

One widely used library for image processing is the *Open Source Computer Vision* (OpenCV) library [9]. Image processing algorithms are optimized in OpenCV to make

<sup>4</sup> The generated OpenCL code is slightly slower than the generated CUDA code, which we attribute to the relative immaturity of the OpenCL implementation.

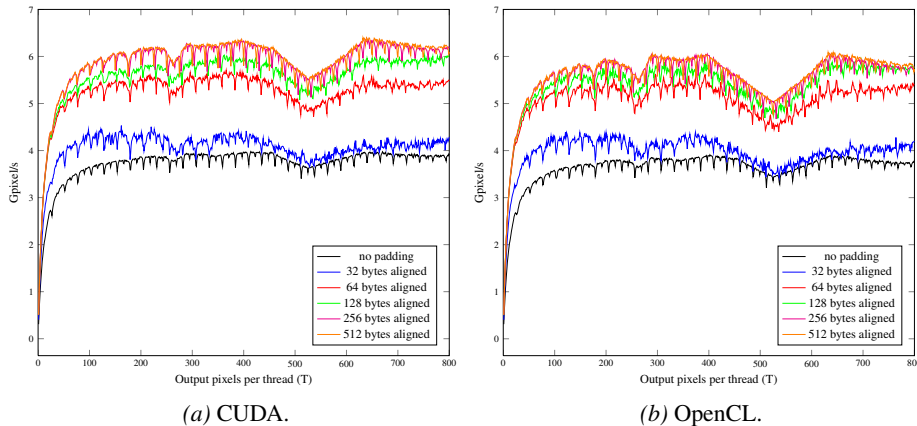
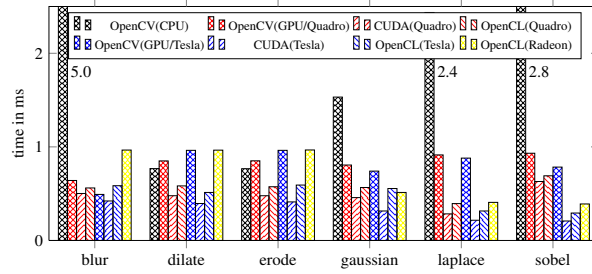


Fig. 2: Throughput of the generated CUDA C sources in Gpixel/s for the vertical mean filter on an image of  $5121 \times 3200$  pixels with padding. The generated CUDA C and OpenCL source pads the image width to a multiple of 32-, 64-, 128-, 256-, or 512-bytes.

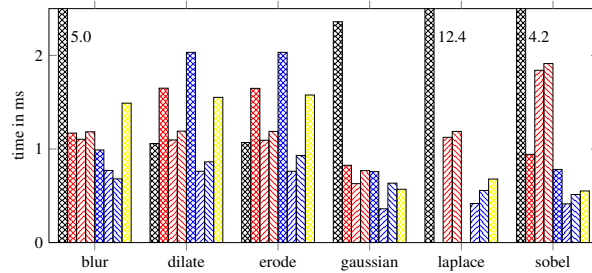
use of the SIMD units and multiple cores of modern processors. Beginning with version 2.2, selected algorithms (mostly convolution kernels) can also be executed on the GPU. Instead of implementing these kernels from scratch, OpenCV relies on the NVIDIA Performance Primitives (NPP) library. To compare the performance of code generated by our framework to such state-of-the-art approaches, we used the framework to implement all six convolution kernels from OpenCV that utilize NVIDIA GPUs. These kernels mostly support the 8-bit *unsigned char* type and the  $3 \times 3$  and  $5 \times 5$  window dimensions, which we use for evaluation. (Note, there is no  $5 \times 5$  GPU implementation of the laplace convolution filter.) However, we can also generate code for other configurations with only minor modifications to the high-level description as for the  $5 \times 5$  laplace convolution filter.

Figure 3 shows the execution times of the OpenCV implementations on a CPU (Core 2 Quad @3.00 GHz) and three GPUs: NVIDIA’s Quadro FX 5800 and Tesla C2050 and AMD’s Radeon HD 5870. For the NVIDIA cards, the OpenCV implementation and CUDA/OpenCL code generated by our framework are compared, while on the AMD card only generated OpenCL code is available. Generated code is as fast as OpenCV code (actually, faster in most cases). With larger filter window size also execution time increases. Again, the generated CUDA code is slightly faster than the OpenCL code. The GPU implementation of OpenCV relies on NPP, resulting in longer execution times. While our DSL approach generates GPU code from a high-level representation of the desired convolution kernel, the OpenCV library and NPP<sup>5</sup> provide more general implementations that are not optimized for the selected convolution kernel properties like the kernel size. The performance of the vectorized OpenCV code varies considerably. For some convolution kernels, their CPU implementation is almost as fast as our generated GPU code (e. g., for dilate and erode); for most kernels, however, their CPU implementation is an order of magnitude slower than generated GPU code (e. g., for blur, laplace, and gaussian). One big advantage of our framework is that we can

<sup>5</sup> The NPP source code is not available for detailed analysis.



(a)  $3 \times 3$  window size.



(b)  $5 \times 5$  window size.

Fig. 3: Comparison of the execution time of convolution kernels from OpenCV and our framework for an image of  $1024 \times 1024$  pixels on a Quadro FX 5800, Tesla C2050, and Radeon HD 5870. The results for a window size of  $3 \times 3$  is shown in (a) and for a window size of  $5 \times 5$  in (b).

generate code for any pixel data type, while the OpenCV implementations are mostly restricted to *unsigned char*.

#### 4 Future Work

The framework presented in this paper allows abstract description of algorithms for image processing which is translated and transformed into device-dependent, optimized source codes. While this works for simple kernels and convolution kernels, we are planning to extend our current framework to provide better support for a broader range of image processing specific features and applications.

The current version of our framework does not support border handling for image processing. The user has to specify border handling in the high-level algorithm description. Instead, our compiler can generate border handling support for images, like clamping to the last valid value, repeating the values beyond the border, mirroring the values at the border, or using a constant value.

Currently, the access/execute metadata is mostly implicit: we assume that the iteration space is parallel in all dimensions and has a 1:1 mapping to work-items (threads), and that the memory access pattern is obvious from the kernel code. In the vertical mean filter example, we use the offset specification to realize a 1:N mapping. More elegant, native support for such mappings allow not only more concise code, but also optimizations on the generated code.

The configuration for a kernel can be specified by a user or determined by the framework. Currently, our framework falls back to a default configuration of  $128 \times 1$ , which is generally suboptimal. However, the compiler can detect a suitable configuration for a kernel and use this setting.

Often, to reduce overheads for dispatching kernels and communicating intermediate data between them, the programmers *manually* bundle several kernels (related by data flow, not necessarily by the application logic!) into a single kernel. Since we have AST information for kernel functions, we can perform kernel fusion and other optimizations automatically if they are allowed by iteration space specifications and data dependences.

## 5 Conclusion

In this paper, we introduced a performance-portable framework for image processing. Our framework provides C++ classes that allow to describe image processing kernels based on decoupled access/execute metadata, which allows programmers to concentrate on developing algorithms and applications, rather than on mapping them to the target hardware.

The framework performs source-to-source translation of kernels expressed in high-level framework-specific C++ classes into low-level CUDA C and OpenCL code with effective device-dependent optimizations such as global memory padding for memory coalescing and optimal memory bandwidth utilization. Our source-to-source compiler is based on Clang and creates AST for kernel functions, which leaves room for future intra- and inter-kernel optimizations.

Our experiments show that code generated from our abstract description is as fast as hand-optimized and hand-tuned CUDA code for vertical mean filtering. While the performance for images that lead to misaligned memory layouts decreases almost by 50 %, our compiler pads the memory layout so that almost no performance penalty can be observed. In terms of lines of code, our concise high-level description requires only one tenth of the hand-written CUDA implementation. Supporting different backends, our source-to-source compiler produces CUDA C and OpenCL code that is faster than the OpenCV/NPP implementation for the available OpenCV convolution kernels that run on the GPU.

## References

1. Clang: Clang: A C Language Family Frontend for LLVM. <http://clang.llvm.org> (2007–2011)
2. Cornwall, J., Howes, L., Kelly, P., Parsonage, P., Nicoletti, B.: High-Performance SIMT Code Generation in an Active Visual Effects Library. In: Proceedings of the 6th ACM Conference on Computing Frontiers. pp. 175–184. ACM (2009)
3. Du, P., Weber, R., Luszczek, P., Tomov, S., Peterson, G., Dongarra, J.: From CUDA to OpenCL: Towards a Performance-portable Solution for Multi-platform GPU Programming. Tech. rep. (2010)
4. Howes, L., Lohmotov, A., Donaldson, A., Kelly, P.: Towards Metaprogramming for Parallel Systems on a Chip. In: Euro-Par 2009–Parallel Processing Workshops. pp. 36–45. Springer (2010)
5. Lin, C., Snyder, L.: Principles of Parallel Programming. Addison-Wesley Publishing Company, USA (2008)
6. NVIDIA: CUDA. <http://www.nvidia.com/cuda> (2006–2011)
7. Ryoo, S., Rodrigues, C., Stone, S., Stratton, J., Ueng, S., Bagsorkhi, S., Hwu, W.: Program Optimization Carving for GPU Computing. Journal of Parallel and Distributed Computing 68(10), 1389–1401 (2008)
8. The Khronos Group: OpenCL. <http://www.khronos.org/opencl> (2008–2011)
9. Willow Garage: Open Source Computer Vision (OpenCV). <http://opencv.willowgarage.com/wiki> (1999–2011)