Visual Development Environment for OpenVX

Alexey Syschikov, Boris Sedov, Konstantin Nedovodeev, Sergey Pakharev
Saint Petersburg State University of Aerospace Instrumentation
Saint Petersburg, Russia
{alexey.syschikov, boris.sedov, konstantin.nedovodeev, sergey.pakharev}@guap.ru

Abstract—OpenVX standard has appeared as an answer from the computer vision community to the challenge of accelerating vision applications on embedded heterogeneous platforms. It is designed as a low-level programming framework that enables software developers to leverage the computer vision hardware potential with functional and performance portability. In this paper, we present the visual environment for OpenVX programs development. To the best of our knowledge, this is the first time the graphical notation is used for OpenVX programming. Our environment addresses the need to design OpenVX graphs in a natural visual form with automatic generation of a full-fledged program, saving the programmer from writing a bunch of a boilerplate code. Using the VIPE visual IDE to develop OpenVX programs also makes it possible to work with our performance analysis tools. All the benefits gained from using the visual IDE are illustrated by the feature tracker example.

I. Introduction

Development of parallel programs, which should be efficiently executed on heterogeneous manycore platforms, is a hard challenge for embedded system developers. Such platforms are targeted to the domains like ADAS, cryptography, video surveillance, aerospace etc. Even today, there are many heterogeneous manycore platforms on the market from NVidia [1], Qualcomm [2], Imagination [3], AllWinner [4], Samsung [5], Mediatek [6] and other vendors. Tomorrow most of embedded systems will be heterogeneous manycores.

Computer vision experts, involved in many of the aforementioned application domains, most frequently require performance for their tasks, so effective use of platform resources is crucial for the success. Responding to the industry demand, Khronos Group developed the OpenVX standard. OpenVX [7] is a low-level programming framework for efficient access to computer vision hardware acceleration with both functional and performance portability. OpenVX was designed for diverse hardware platforms, providing a computer vision framework that efficiently addresses current and future hardware architectures with minimal impact on applications.

Our task was to make it possible for developers of computer vision applications to gain all the benefits provided by OpenVX. We successfully added support of OpenVX (spec. 1.0.1) in a VIPE IDE [8]. To the best of our knowledge, this is the first visual development interface for the OpenVX programming.

II. State of the art

OpenVX is intended to increase performance and reduce power consumption of machine vision applications. It is focused on embedded systems with real-time use cases such as face, body and gesture tracking, video surveillance, advanced driver assistance systems (ADAS), object and scene reconstruction, augmented reality, visual inspection etc.

The using of OpenVX standard functions is a way to ensure functional portability of the developed software to all hardware platforms that support OpenVX.

Since the OpenVX API is based on opaque data types, client-code need not be recompiled, when used with various OpenVX implementations. That is because such machine-specific details as memory alignment, byte packing etc. are hidden inside the vendor library, which is linked to the main program.

OpenVX uses a graph-based execution model [7] and incorporates data and task-level parallelism. This model allows OpenVX to solve a number of issues relevant to programs parallelization.

The graph program representation (Fig. 1) is crucial to OpenVX efficiency. Developers describe a graph of image processing operations using nodes (functions). Graph nodes could target any hardware computational unit. The graph model enables OpenVX implementations to optimize for power and performance. The host processor can set up a graph, which later can be executed almost autonomously [8]. Several nodes may be fused by the implementation to eliminate memory transfers [9]. Image processing can be tiled to fit data into a local scratchpad memory or cache. Host interaction during frame-rate graph execution can be minimized.

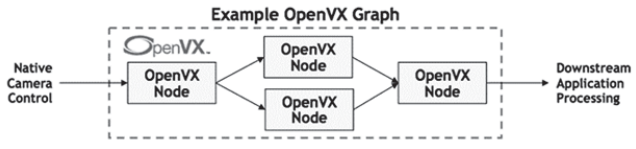

Fig. 1. An OpenVX graph [10]

It should be noted that the use of a visual graph notation for a software development gradually broadens. From our standpoint, there are mainly two tasks, which a visual graph notation is supposed to resolve.

ISSN 2305-7254

Firstly, nowadays developer teams create complex computing embedded systems [11-14]. Teams often include many experts from various domains. For an efficient problem solving such teams desperately need a common language for their project. According to the researches, a visual graph notation is a natural representation of an operations sequence [15]. Each member of a developer team explicitly or implicitly uses some kind of a graphical flow chart for his projects. It is better to have a single "big picture" of a whole project, to which all the members have a simultaneous access.

Secondly, developers face extremely complex and contradictory requirements; for example, they need to produce a high-quality embedded solution for some task within a tight time frame. Meanwhile, the volume of code vastly increases. When companies describe existing situation they compare it to the shift from writing programs solely in an assembly language to writing them on a high-level language. It is more comprehensible and productive and let teams cope with large projects. Modern projects are so huge and sophisticated that the high-level text-based language fall into a state of an assembler. It is not a coincidence that source-to-source compilers are used there.

The following tool suits already use visual graph notation for the computer vision domain:

- Adaptive Vision Studio [16] (graphical environment for development of data-flow based software for industrial computer vision);
- Intel Flow Graph Designer [17] (visualization tool that supports the analysis and design of parallel applications);
- Simulink [18] (graphical programming environment for modeling, simulating and analyzing multi-domain dynamic systems).

In our previous publications, we presented the visual integrated development environment VIPE to design portable software for embedded many-core systems. It allows creating DSL for a particular domain at hand [19]. Similarly to previously mentioned tools, it uses a visual graph notation for parallel programs representation.

OpenVX seems to be a promising framework to cope with machine vision embedded development. However, as for now, there are some drawbacks, most of which are directly or indirectly related to the need to describe OpenVX's graphs in a textual form while programming. These drawbacks will be considered further.

III. Visual OpenVX in VIPE IDE

The main goal of providing support for OpenVX in VIPE was in making OpenVX accessible to developers with various competence levels, e.g. for both programmers and domain experts. The goal was achieved by highlighting advantages and alleviating some drawbacks of the OpenVX framework.

The focus on a particular domain, portability of programs and heterogeneous platforms support are the key advantages of OpenVX. Nevertheless, it has some drawbacks, e.g. a requirement to write a lot of boilerplate code to make use of OpenVX graphs and functions and a limited set of standard functions (although it is possible to create user-defined functions it does not fully solve the problem). In this section, we will show how all these issues were addressed in VIPE.

From the programmer's perspective, the OpenVX specification contains three core types of a framework object: a computer vision function, an opaque data object and a graph.

A. OpenVX standard functions and data

A component model of the base VPL language includes terminal operators aimed at data processing, data objects, managing data, and control operators, such as loops, used for hierarchical composition and branching. The VPL language is so expressive that OpenVX components were integrated into a visual component library.

The main VIPE library was extended with new basic objects representing OpenVX functions and data objects. There is a VIPE incarnation of the *vx_kernel* object – *vxNode*, which is derived from the functional node. It can be called as an instance of a *vx_node* or as a standalone *vxu* function. There is a specific API devoted to writing templates for *vxNode* instances that differs from the API for the standard VPL functional node. Ports and custom parameters of such a node correspond to function parameters in a *vx_kernel*.

The library of OpenVX standard functions was integrated into VIPE as a domain-specific component library with a set of *vxNode*-based components (Fig. 2). Additionally, it contains such miscellaneous functions like "Read image/Write image" to access files, which are part of the auxiliary debug kernel library.

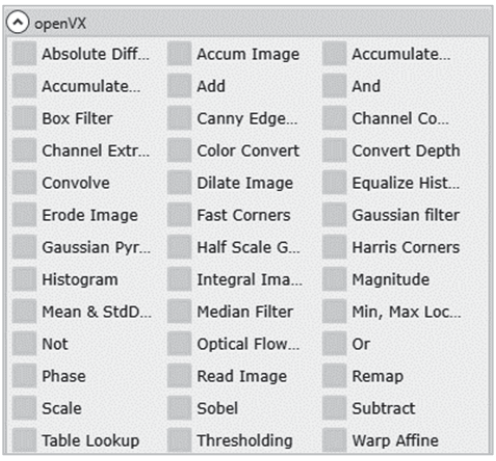

Fig. 2. The library of OpenVX functions in VIPE

Templates for such nodes have parameters for *vx_node* as well as for *vxu* form, because they can be different (Fig. 3).

OpenVX standard opaque data objects were integrated into VIPE as a domain-specific library (the domain-specific library interface is flexible enough) with a set of *vxData*-based components (Fig. 4). These components correspond to OpenVX data objects: *vx_image*, *vx_array*, *vx_pyramid* etc.

```
//<tdl>
//<module name="openvx">
//<submodules>
//<submodule template_type="Context">
//<params>
//  <param name="graph/context" type="Scope" num="1" dir="Input"/>
//  <param name="I" type="Port" num="2" dir="Input"/>
//  <param name="H" type="Port" num="3" dir="Input"/>
//  <param name="G" type="Numeric" num="4" dir="Input"/>
//  <param name="norm_type" type="Custom" num="5" dir="Input"/>
//  <param name="O" type="Port" num="6" dir="Output"/>
//</params>
//</submodule>
//<submodule template_type="Graph">
//<params>
//  <param name="graph/context" type="Scope" num="1" dir="Input"/>
//  <param name="I" type="Port" num="2" dir="Input"/>
//  <param name="H" type="Port" num="3" dir="Input"/>
//  <param name="G" type="Numeric" num="4" dir="Input"/>
//  <param name="norm_type" type="Custom" num="5" dir="Input"/>
//  <param name="O" type="Port" num="6" dir="Output"/>
//</params>
//</submodule>
//</submodules>
//</module>
//</tdl>
CannyEdgeDetector
```

Fig. 3. The OpenVX function template for the Canny edge detector kernel

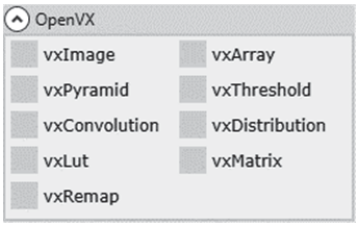

Fig. 4. The OpenVX data library in VIPE

Scalar data objects are generated automatically from the VPL links that connect *vxNode* operators with *vxData* ones or with other VPL objects (terminal nodes, constants, etc.). In case there is a description of a scalar parameter in a *vxNode* template, when the corresponding port receives data it will be interpreted as a scalar data object (Fig. 5).

```
vx_float32 link_9442_t;
vx_scalar link_9442_s;
vx_float32 link_9443_t;
vx_scalar link_9443_s;
vx_float32 link_9456_t;      memcpy(&link_9450_t, &link_9450, link_9450.Size);
vx_scalar link_9456_s;       link_9450_s = vxCreateScalar(context, VX_TYPE_BOOL, &link_9450_t);
vx_size link_9441_t;         link_9451_t = GetControlValue(&link_9451);
vx_scalar link_9441_s;       link_9451_s = vxCreateScalar(context, VX_TYPE_UINT32, &link_9451_t);
vx_bool link_9450_t;         link_9462_t = GetDoubleValue(&link_9462);
vx_scalar link_9450_s;       link_9462_s = vxCreateScalar(context, VX_TYPE_FLOAT32, &link_9462_t);
vx_uint32 link_9451_t;
vx_scalar link_9451_s;
vx_float32 link_9462_t;
vx_scalar link_9462_s;
```

Fig. 5. An example of a declaration and initialization of scalar data in generated code

A type of a scalar parameter will be either automatically set to a native one (*vx_int,* etc.) or will be wrapped up in a *vx_scalar,* the choice of a strategy depends on the type of the port parameter in the corresponding *vxNode*.

B. OpenVX graphs

The OpenVX model of computation is based on graphs. According to the specification, "graphs are composed of one or more nodes… and are linked together via data dependencies…" with some rules and limitations. Generally, it is very close to a classical Data-Flow model [20]. Asynchronous growing processes (AGP) model of computation [21] is the formal base of the VPL language. It is sufficiently rich to include the OpenVX model of computation as a special case.

To define OpenVX graphs in VPL language the *vxGraph* node was added, which is derived from the standard VPL *Complex* node. It is used to create an additional level of hierarchy, containing an OpenVX subgraph, and to put some constraints on a sub-scheme according to OpenVX graph requirements. The *vxGraph* body may contain only *vxNode* and *vxData* objects with some minor VPL objects (for example, constants) that can be transparently converted to appropriate OpenVX objects.

vxGraph can be internally interpreted both as an OpenVX graph with *vx_node* instances or as a set of distinct *vxu* nodes. This decision could be made either by a developer or automatically.

It should be noted that OpenVX objects could also be used outside of the *vxGraph*. In this case, they will operate as an ordinary terminal operator (in *vxu* mode) of a VPL program for data processing without the use of an OpenVX graph.

C. OpenVX in the IDE

An example of a visual OpenVX program (edge detection using the Canny Edge Detector function), which is open in VIPE, is presented on Fig. 6.

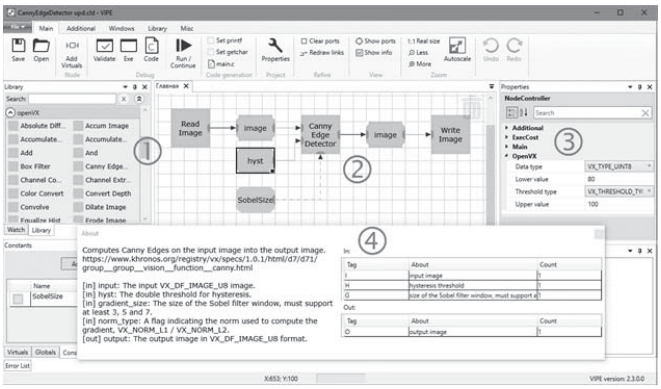

Fig. 6. The visual OpenVX program for edge detection in VIPE

The visual OpenVX library (Fig. 6, ①) is available in VIPE just like other DSL libraries. The graph shown here (Fig. 6, ②) contains only library objects. The program includes following components: the Canny Edge Detector operator from the base OpenVX standard, the "Read image" and "Write image" operators from the OpenVX debug extension, the source and final image containers, the threshold for the Canny hysteresis parameter, and the VPL constant for the Canny Sobel size parameter. The Canny normalization parameter is defined directly as an operator property. The threshold can also be configured directly in the object properties tab (Fig. 6, ③).

A brief memo is attached to every library operator, it contains a reference to the standard, a description of inputs and outputs (Fig. 6, ④) specifying their types, possible values, etc. There is no need for a developer to remember all the details of the OpenVX standard: all the necessary information is already presented in that description.

All the intricacies are handled automatically during translation to an appropriate OpenVX code according to standard rules: with all type conversions involved, “SetAttribute”-s, virtual vs. real images, etc., while the visual OpenVX graph stays easy to grasp.

Visual OpenVX graphs can be easily modified by adding more processing steps (Fig. 7), parameters and other attributes, or by reconfiguration of links.

D. Alleviation of OpenVX drawbacks

As it was mentioned earlier, the OpenVX standard has some drawbacks that significantly raise the cost of its usage in programs.

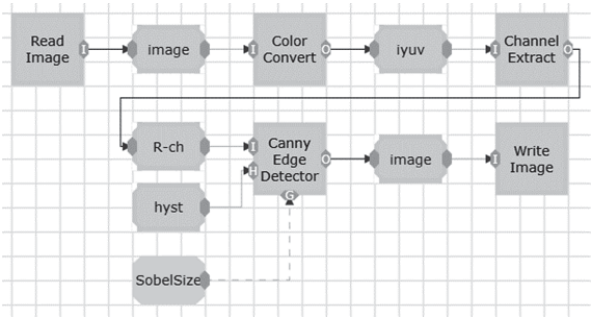

Fig. 7. The visual OpenVX program for edge detection (a modified version)

One of the main drawbacks is a requirement to write a lot of boilerplate code. This is the first thing that various OpenVX implementations try to conceal. For example, AMDOVX uses a scripting language [22], PythonOpenVX wraps it inside Python functions [23], etc. VIPE shields a developer from such a “code bloat” using code generation. It becomes particularly valuable when working with OpenVX. An illustration of what a developer is spared from even for a simple one-function example presented earlier (Fig. 6) is shown below.

The first block of code is a declaration of auxiliary variables and creation of base objects (Fig. 8): *status*, *context*, *graph*, loading of additional kernel sets (*vxLoadKernels*), *graph* execution and release of everything.

```
vx_graph graph_637;

int vxGraph_Create_637(){ ... }

int main()
{
    status = VX_FAILURE;
    context = vxCreateContext();
    vxLoadKernels(context, "openvx-debug");
    vxGraph_Create_637();
    DataLink link_668 = { NULL, 0, 0 };
    status = vxProcessGraph(graph_637);
    FreeLink(&link_668);
    vxReleaseGraph(&graph_637);
    vxReleaseContext(&context);
    return 0;
}
```

Fig. 8. The first code snippet: working with graph

Graph creation is the second block of code. It includes:

- declaration and initialization of variables, creation of OpenVX data objects, setting their parameters (Fig. 9);

```
DataLink link_668 = { NULL, 0, 0 };
vx_image vxImage_639 = vxCreateVirtualImage(graph_637, 640, 480, VX_DF_IMAGE_U8);
vx_int32 vxThreshold_656_lower = 80;
vx_int32 vxThreshold_656_upper = 100;
vx_threshold vxThreshold_656 = vxCreateThreshold(context, VX_THRESHOLD_TYPE_RANGE,
                                                 VX_TYPE_UINT8);
vx_image vxImage_660 = vxCreateVirtualImage(graph_637, 640, 480, VX_DF_IMAGE_U8);
vx_int32 link_668_t;
/*Assign constant 658 to 3 */
link_668.Size = 4;
link_668.Data = (char*)malloc(4);
SetControlValue(3, &link_668);
link_668_t = GetControlValue(&link_668);
status |= vxSetThresholdAttribute(vxThreshold_656, VX_THRESHOLD_ATTRIBUTE_THRESHOLD_LOWER,
    &vxThreshold_656_lower, sizeof(vxThreshold_656_lower));
status |= vxSetThresholdAttribute(vxThreshold_656, VX_THRESHOLD_ATTRIBUTE_THRESHOLD_UPPER,
    &vxThreshold_656_upper, sizeof(vxThreshold_656_upper));
```

Fig. 9. The second code snippet: graph creation (variables and data objects)

- nodes creation and linking (Fig. 10);

```
vx_node nodes[] =
{
    vxFReadImageNode(graph_637, "image.pgm", vxImage_639),
    vxCannyEdgeDetectorNode(graph_637, vxImage_639,
            vxThreshold_656, link_668_t, VX_NORM_L1, vxImage_660),
    vxFWriteImageNode(graph_637, vxImage_660, "imageCanny.pgm"),
};
```

Fig. 10. The second code snipped (continued): graph creation (nodes creation and linking)

- nodes and graph verification with result checking and error reporting (Fig. 11);

```
for (int i = 0; i < dimof(nodes); i++)
{
    if (nodes[i] == NULL)
    {
        printf("nodes[%u] is NULL\n", i);
        nodeCheck = 0;
    }
}
if (nodeCheck == 0)
    print_exit(1);
status = vxVerifyGraph(graph_637);
if (status != VX_SUCCESS)
{
    printf("Graph failed (%d)\n", status);
    print_exit(1);
}
```

Fig. 11. The second code snipped (continued): graph creation (verification)

It is worth mentioning that all the generated definitions of data objects are correct, they are real or virtual according to their location in a graph, so a developer does not need to ensure its correctness.

When coding, a developer has to make an effort each time he wants to intermix OpenVX with other technologies or libraries. VIPE provides seamless integration of OpenVX graphs into programs built using other libraries or user-defined functions (Fig. 12). The only thing required in that case is adding some conversion operators: from some “foreign” to the OpenVX data object format and vice versa. This feature allows compensating a lack of standard functions providing easy access to other libraries.

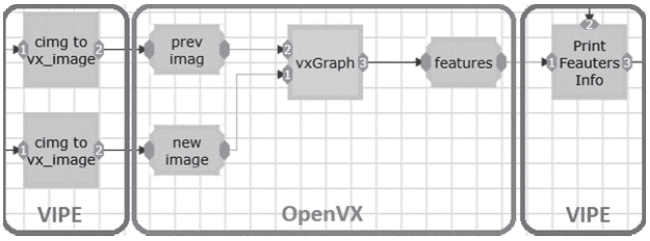

Fig. 12. Integration of OpenVX graphs into a VPL program

According to the OpenVX standard guideline [7], a graph should be created, verified after creation and each time any structural changes happen and then executed multiple times (Fig. 13).

This is reasonable, since graph construction and subsequent verification is a time-consuming process and repetitive verification on every processing iteration leads to a significant penalty. However, it puts a burden on a developer to conform to this guideline.

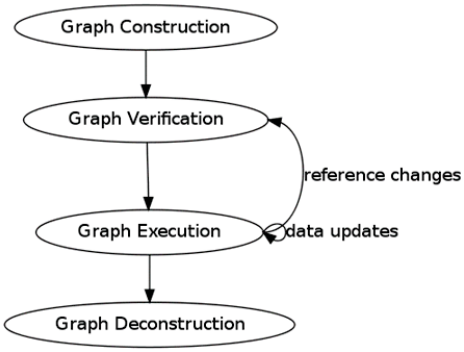

Fig. 13. OpenVX graph lifecycle guideline [7]

Writing of real code, which often contains complex control flow, makes it inevitable either to create a graph before any processing as a global variable, or pass it through all the functions as a parameter. As an additional burden, a developer has to manage visibility of the following components: graph input and output data objects (to feed data into and gather results after graph execution), parameters of graph nodes for configuration purposes, etc.

VIPE eliminates that "manual labor" generating target code. Generated OpenVX graphs conform to the guideline and thus are executed efficiently.

Initially, OpenVX graphs in VIPE had a plenty of ways to use scalar data computed at a higher-level, variables, etc. However, a graph needs to be verified in case of any change (even parameter values of any graph node). Therefore, we put constraints on VPL constructs when interfacing with OpenVX graphs in VIPE due to the rule of the OpenVX standard.

IV. Enabling analysis tools for OpenVX

VIPE IDE includes a growing set of performance analysis and debugging tools [24].

While adding OpenVX support into VIPE, the OpenVX model of computation was smoothly absorbed by the AGP model and OpenVX functions and data objects were seamlessly integrated into the VPL language. Formally speaking, OpenVX support was added without making any modifications to the core components of VIPE.

This makes it possible to apply existing VIPE tools to OpenVX programs slightly "patching" the code generator so that the analysis results stay relevant.

The automated code profiling can now be used for OpenVX graphs. The current version unrolls graph to a set of *vxu* functions and profiles them separately, after that the visual graph is marked with profile data. In the upcoming releases, the profiler will use the OpenVX performance measurement interface to increase profiling precision.

The visual profiler and the static performance analyzer did not require any modifications and thus could be used out-of-the-box. Application of these tools to OpenVX programs is demonstrated in the next section.

Visual debugging is a new feature of the VIPE IDE, which is currently being developed. It will allow debugging and analyzing the behavior of both native VPL and OpenVX programs in a visual manner. It will display flow of execution, data consumption, intermediate graph data values and images together with their "processing history" (Fig. 14).

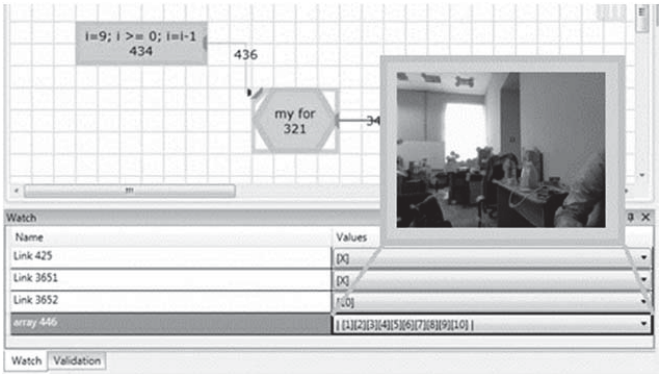

Fig. 14. An interface of the visual debugger

V. Use case illustration

As an illustrative example for showing all the benefits of VIPE usage for OpenVX programming, the feature tracker use-case, developed with AMDOVX open-source implementation [22], was selected. This program takes each pair of consecutive frames from camera, analyzes it and mark distinctive features on an image (Fig. 15).Since there is only a few of available OpenVX implementations, the feature tracker in VIPE uses the Khronos sample OpenVX implementation [25]. All the experiments with this program were conducted on an x86 platform.

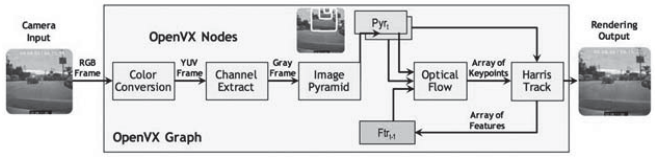

Fig. 15. Feature tracker program structure [26]

OpenVX is not a fully featured library as, for example, OpenCV, and the range of tasks it can be applied to is thus limited. In the feature tracker program, OpenVX was used to identify features on the captured image and calculate its motion between the two consecutive frames (Fig. 16).

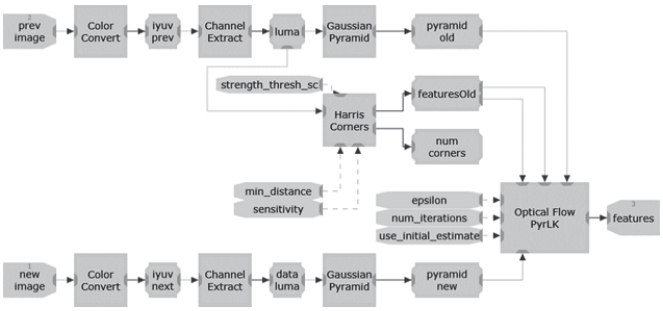

Fig. 16. An OpenVX subgraph for the feature tracker program

The CImg library [27] was used to capture images from a camera and to display features on the screen, the program also uses some auxiliary library functions (Fig. 17).

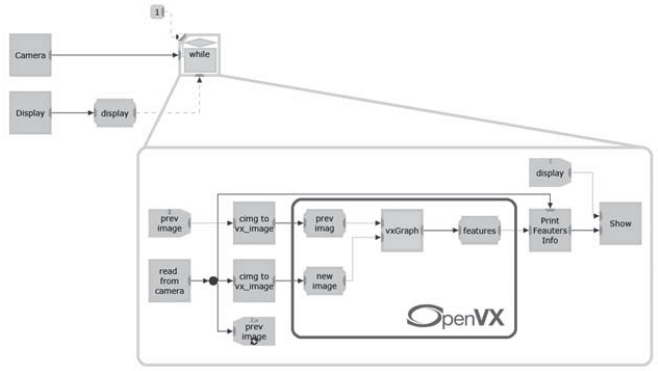

Fig. 17. CImg and its auxiliary functions used in the feature tracker program

With the standard conforming generation of the *vxGraph* code, it required to change simple "cimg->vx_image" operator to the specially designed *vxWriter* nodes. The reason was that there is no simple way to feed a new image to the verified graph. The program needs to "patch" the existing input image data without affecting the image reference.

The application shows frames captured from the camera and draws feature markers on them as circle marks. Resulting data (i.e. coordinates of each feature) can be used as an input to later stages of the image processing pipeline (Fig. 18).

It is the ability of VIPE to interpret OpenVX subprograms both as OpenVX graphs and as programs with regular OpenVX functions that enables automated profiling and performance analysis for programs containing OpenVX fragments. Performance analysis shows a relatively good speedup for two cores, a moderate speedup for three cores and no speedup for more cores (Fig. 19).

The easiest way to extract more parallelism was to use a multi-frame buffer and parallelize a camera frame-processing loop synchronizing writes the output data. It does add some latency, but this solution is quite suitable for a non-real-time system. Such a kind of refactoring is easy to perform on the ready-made VPL program without making any significant modifications.

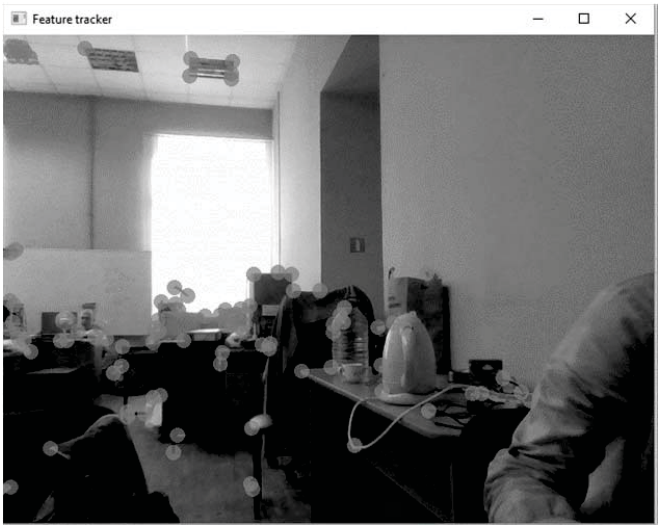

Fig. 18. Running feature tracker

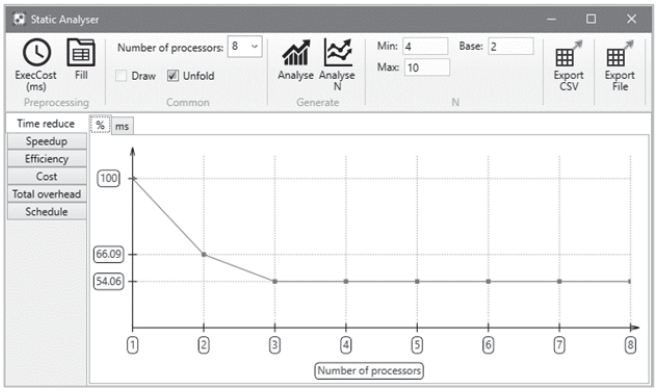

Fig. 19. Results of performance analysis of the feature tracker program

Analysis of the results shows near linear time reduction (Fig. 20).

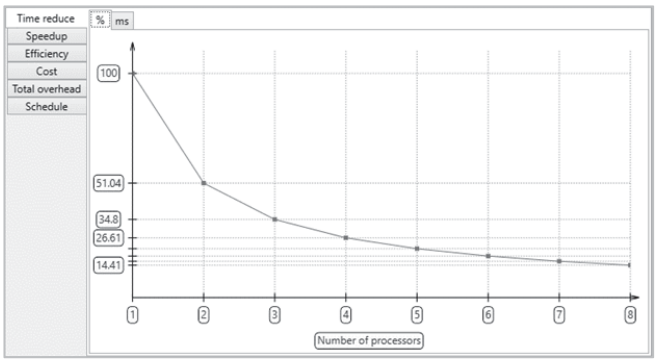

Fig. 20. Results of performance analysis of the feature tracker (with parallel processing of frames)

A more convenient way of raising the level of parallelism is to apply tiling using the OpenVX tiling extension [28]. This requires adding support for the OpenVX tiling extension to VIPE and is a subject of future work.

As visual profiler shows a lot of time is spent inside image conversion functions (convert to the *vx_image* format and back, see Fig. 21).

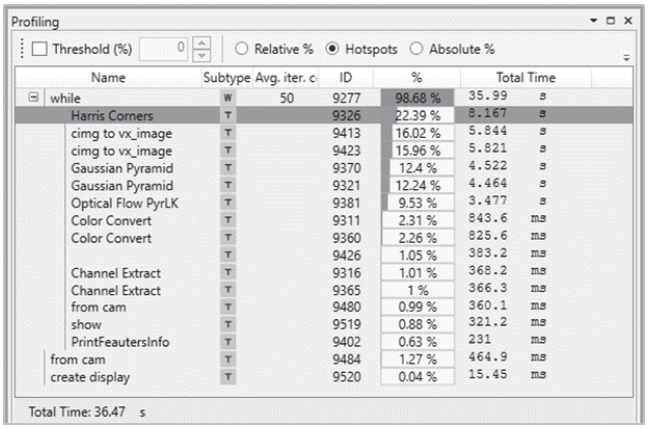

Fig. 21. Visual profiling results for the feature tracker program

This should be a subject of further optimization. Another optimization step is to use a *vxDelay* object for a native way of passing data between two graph invocations.

VI. Conclusion

In this paper, we introduced the visual environment for parallel programs development with support of the OpenVX standard – VIPE, which is based on the VPL language.

VIPE IDE allows programmers to compose OpenVX graphs in a natural graphical form. The process is less error prone, because all the boilerplate code, including the names of all the input-output parameters, is generated automatically. Thus, changing structure of a previously created graph becomes an easy task. Moreover, domain expert could grasp the intent of a program at a glance, thanks to a higher "signal-to-noise ratio" of a visual representation.

Other (non-OpenVX) components could be used in the same VPL program, say, OpenCV functions, and could be connected with OpenVX graphs to produce valuable results. This may be convenient when the full image processing pipeline should be implemented.

We also demonstrated that any OpenVX subgraph is a first-class citizen for our profiling and performance analysis tools, thus making the IDE friendlier to a developer.

Acknowledgment

The research leading to these results has received funding from the Ministry of Education and Science of the Russian Federation under the contract RFMEFI57816X0214.

References

[1] NVIDIA Whitepaper, “NVIDIA Tegra K1: A New Era in Mobile Computing”, Web: http://www.nvidia.com/content/PDF/tegra_white_papers/tegra-K1-whitepaper.pdf.

[2] F. Cheng, “Meet the Snapdragon 835: a next-gen processor made for power users”, *Qualcomm Snapdragon Blog*, Jan. 2017. Web: https://www.qualcomm.com/news/snapdragon/2017/01/03/meet-snapdragon-835-next-gen-processor-made-power-users.

[3] A. Voica, “ELVEES goes full purple: MIPS, PowerVR and Ensigma united in one vision chip”, *Imagination Blog*, Jun. 2015, Web: https://www.imgtec.com/blog/elvees-goes-full-purple-mips-powervr-and-ensigma-united-in-one-vision-chip.

[4] Allwinner Technologies official website, Series “A” Processors, Web: http://www.allwinnertech.com/index.php?c=product&a=index&pid=2.

[5] Samsung official website, Samsung Exynos Processor, Web: http://www.samsung.com/semiconductor/minisite/Exynos/w/.

[6] MediaTek official website, MediaTek helio X20/X25, Web: http://mediatek-helio.com/x20/.

[7] Khronos Vision Working Group, “The OpenVX™ Specification v1.1”, Web: https://www.khronos.org/registry/OpenVX/specs/1.1/OpenVX_Specification_1_1.pdf

[8] A. Syschikov, Y. Sheynin, B. Sedov, V. Ivanova. “Domain-Specific Programming Environment for Heterogeneous Multicore Embedded Systems”. International Journal of Embedded and Real-Time Communication Systems (IJERTCS), vol. 5, №. 4, 2014. pp. 1-23

[9] Rainey, Erik, et al. "Addressing system-level optimization with OpenVX graphs." Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops. 2014.

[10] Khronos Group Inc. official website, OpenVX, Web: https://www.khronos.org/openvx/.

[11] D. Ghosh, "DSL for the Uninitiated", *Communications of the ACM*, vol. 54, № 7, 2011, pp. 44-50.

[12] Co-Modeling of Embedded Networks Using SystemC and SDL / V.Olenev, A.Rabin, A.Stepanov, I.Lavrovskaya, S. Balandin, M. Gillet // International Journal of Embedded and Real-Time Communication Systems (JERTCS) – Тампере. 2011. – #2(1) – С. 24-49.

[13] E. Evans, *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004

[14] S. Balandin, M. Gillet, “Embedded Network in Mobile Devices”, *International Journal of Embedded and Real-Time Communication Systems* (IJERTCS), vol. 1, № 1, 2010, pp 22-36.

[15] S.J. Mellor, M. Balcer, and I. Jacoboson, *Executable UML: A foundation for model-driven architectures*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[16] Adaptive Vision Studio official website, Web: http://www.adaptive-vision.com/en/software.

[17] V. Michael, “Flow Graph Analyzer”, *Intel Blog*, Mar. 2014, Web: https://software.intel.com/en-us/articles/flow-graph-designer.

[18] MathWorks official website, Simulation and Model-Based Design, Web: https://www.mathworks.com/products/simulink.html?s_tid=hp_products_simulink.

[19] Ivanova, Vera, et al. "Domain-specific languages for embedded systems portable software development." Open Innovations Association (FRUCT16), 2014 16th Conference of. IEEE, 2014

[20] Jack B Dennis, John Fosseen, John Linderman. Data Flow Schemas. In International Symposium on Theoretical Programming, 1972.

[21] Ivanov, V., Y. Sheynin, and A. Syschikov. "Programming model for coarse-grained distributed heterogeneous architecture." XI International Symposium on Problems of Redundancy in Information and Control Systems: Proceedings, SUAI. 2007.

[22] GPUOpen official website, AMD OpenVX (AMDOVX), Web: http://gpuopen.com/compute-product/amd-openvx/.

[23] O. Heimlich, E. Ezra Tsur, “OpenVX-based Python Framework for real-time cross platform acceleration of embedded computer vision applications”, *Frontiers in ICT*, vol. 3, 2016. p. 28.

[24] A. Syschikov, B. Sedov, Y. Sheynin, “Domain-Specific Programming Technology for Heterogeneous Manycore Platforms” *in Proc. of the 12th Central and Eastern European Software Engineering Conf. in Russia*, 2016, p. 15.

[25] Khronos Group Inc. official website, OpenVX 1.0.1 Sample Code, Web: https://www.khronos.org/registry/OpenVX/sample/openvx_sample_1.0.1.tar.bz2/.

[26] Khronos Vision Working Group, “OpenVX Webinar”, Web: https://www.khronos.org/assets/uploads/developers/library/2016-openvx-webinar/Khronos-OpenVXwebinar-June2016.pdf.

[27] CImg Library official website, Web: http://cimg.eu.

[28] Khronos Vision Working Group, “OpenVX User Kernel Tiling Extension”, Web: https://www.khronos.org/registry/OpenVX/extensions/vx_khr_tiling/1.0/OpenVX_Tiling_Extension_1_0.pdf.