DEFERRED IMAGE PROCESSING IN INTEL® IPP LIBRARY

Alexander Kibkalo (alexander.kibkalo@intel.com), Michael Lotkov (michael.lotkov@intel.com),
Ignat Rogozhkin (ignat.rogozhkin@intel.com), Alexander Turovets (alexander.turovets@intel.com)
Intel Corporation, 2 Parkovaya Street, Satis, Diveevo district, Nizhny Novgorod region, 607328, Russia

ABSTRACT
A new approach to fast processing of big images is proposed. The image size now often exceeds the size of CPU L2 cache. While performing operations on such images data access takes generally more time than operation itself. There are two main methods to accelerate calculations in this case. First – development of optimized image processing libraries containing fast implementations of main image processing algorithms (e.g. Intel IPP). Second – description of the task in a some abstract form and organization of delayed pipelined processing of the image by small fragments. We combined these two approaches in the new Deferred Mode Image Processing library (DMIP). It is the software layer over Intel IPP providing formula-level programming, pipelined processing of images and easy extension with user-defined operations. DMIP library allows for accelerating of image processing tasks up to 3 times.

KEY WORDS
Image manipulation, performance library, data pipeline

1. Introduction

Image processing, in general, is very time-consuming task. Besides, the size of processed images has the strong tendency to grow. Even mobile devices now capture images of several megapixels. New image processing algorithms require more calculations and more data. It is important to process images as fast as possible.
One approach to acceleration of image (and other data) processing is implemented in Intel® Integrated Performance Primitives library (IPP) [1]. It contains hundreds of image processing functions manually optimized for different Intel platforms. IPP functions provide the best performance for almost each separate function. But it does not always results in the best performance of the whole application performing several operations on images.
The main reason is that image processing applications work with big data arrays much larger than the CPU L2 cache. In such conditions the optimization of memory access is generally more important than the optimization of arithmetic calculations. So, the sequence of fast IPP function calls can be outperformed by processing the image data by small portion. This method is known as pipelined or deferred image processing. First, the sequence of operations on images is described by some way – calculations are deferred. Then this sequence is performed for small parts of images (tiles, blocks, slices etc) – calculation pipeline is organized.
The idea of pipelined data processing is not new. It is the base of the dataflow computing approach [2]. There are several applications of this idea to image or other multi-dimensional data.
Java Advanced Imaging API [3] allows for describing of the "imaging chain" or the image processing graph. It is a directed acyclic graph (DAG) with nodes for source images, operations and destination images. Operations with one or two inputs and one output are supported. JAI manual says nothing about nodes and graphs with more than one destination and about cloning the operation output. JAI discern rendered (for image with known size) and renderable (can be applied to arbitrary image) graphs. A graph is built via function calls (1-2 calls per node for a rendered graph, 3-5 calls – for renderable graph). JAI allows for extending the existing operation set with used-defined functions. There is no performance data in JAI manual but using of byte code for image pixel processing hardly could provide the high speed.
In the TaskGraph Meta-programming Library [4] a task is described as a DAG. The library supports run-time generation of domain-specific code and meta-programming (different types of code transformation including block unrolling and tiling). Such technique allows for generating the efficient code for different algorithms (e.g. image filtering, linear algebra, ray tracing etc.). TGM works with low-level code and optimizes it for separate functions.
The orthogonal approach to the deferred mode of image processing is Blitz++ system [5]. This system uses partial calculations and C++ templates to organize the efficient processing of N-dimensional arrays. Blitz++ supports compact and easy operator notation of calculations on arrays. Blitz++ generates loops for pipelined array processing including unrolling, cache optimization etc. Performance of generated code is close to performance of Fortran programs. Blitz++ is oriented to general purpose numeric calculations (PDE, linear algebra etc) and does not support many typical image processing operations.
The GEGL image processing library [6] uses the directed acyclic graph of an image operation to represent the whole image processing task. The task is described and

executed by calls of low-level C functions to create operation nodes, connect them, set property and perform operations. This library allows for creating of interactive image processing applications. The set of supported operations can be extended by writing plug-ins for new operations.
All these solutions either are not oriented for typical performance-oriented image processing tasks or do not provide efficient solution of them or require low-level programming..

2. Proposed solution

We propose to combine memory access optimization by organizing of the image processing pipeline with the high calculation speed of IPP image processing functions and compact operator notation of image processing tasks. The main requirements to the new component (the draft name is IPP Deferred Mode of Image Processing) are:

- performance close to manually optimized pipelined code for big images;
- simple intuitive formula-level programming;
- low implementation cost: minimum of parsing, low-level code generation and other compiler-like manipulations;
- easy extension with user-defined operations.

DMIP library is the C++ layer modeling dataflow computing for source image. Simple overloading of C++ operators with calls of similar IPP functions (e.g. addition of images of all supported types) results in operations on whole images and could not increase performance on the application level.
Differently, IPP DMIP classes and functions describe images in memory, operations on images and sequences of such operations. An image processing task is described as a DAG. DAG nodes correspond to data in memory or operations on images, DAG edges – to data flow in it. Then the task graph is transformed to the sequence of IPP primitive calls for small image parts. In DMIP image is processed by slices. We call this stage "graph compilation" but really no low-level code is generated. In some sense IPP library is used as the code generator. DMIP chooses the proper size of image slice to adjust the task working set to the L2 cache of the computer. The proposed approach does not require source parsing or low-level code generation and uses big operation granularity (the typical size of the image part is about several thousands pixels). For big images the time of graph compilation is much less than the time of calculations. So, the possibility of graph editing or dynamic reconfiguration does not affect seriously the application performance.
DMIP allows for describing of image processing tasks as arithmetic expressions using C++ operators and operator-like functions. It makes programming simple and intuitive and hides details of voluminous IPP API. Classes for new user-defined operations are easily derived from basic DMIP classes. New operations can be used in arithmetic expressions with supported operators and functions. DMIP manual contains examples of user-derived classes.

3. Implementation basics

A sequence of operations on images is represented in IPP DMIP as a DAG. Nodes of such graph have input and output ports. Input nodes of a graph correspond to image data sources (images in memory or image generators) and have only one output port. Output nodes correspond to destination images or other data in memory and have only one input port. All other graph nodes describe operations on images and have an input port for each input image of the operation and the output port for each output image of the operation. For example, the node for image addition operation has two input ports and one output port. An output port of one node can be connected to an input port of another node. A connection (edge) corresponds to direction of an output of one operation to the input of another operation. The final graph of the image processing task is in some sense the data-flow program for the task.
After a graph is built it is compiled. The graph compiler does not perform parsing, syntactic analysis and code (C++, C, assembler or machine commands) generation. We understand under graph compilation the transformation of the graph to the sequence of IPP primitive calls for image slices. The compilation process starts with graph checking for absence of free ports and cycles. Next, the correct image type and size is defined for each operation in a graph. Formats of input and output data should be supported by IPP library. The graph compiler chooses in-place or not-in-place version of operations and allocated memory buffers in nodes for intermediate results. Then it chooses the size of the image slice to perform as many operations as possible on data in L2 cache to decrease the memory access overhead. At the last stage of graph compilation the table of parameters for IPP calls for image slices is generated. The graph compiler can optimize the sequence of calculations: use in-place operation if it is possible, eliminate common expressions, add hints for non-temporal write to a buffer etc.
We understand under graph execution successive calls of primitive functions for all slices with arguments generated while graph compiling. The graph executor processes each record of parameter table: tune input and output argument for slice positions in internal buffers and calls the corresponding function for a slice. A graph can be executed in parallel mode. If an operation allows for parallel processing of slice parts it is done by parallel threads.

4. DMIP API

The user interface if IPP DMIP library consist of two levels: the functional level that contains classes and

methods for graph building, compilation and execution and the symbolic level for a smart description of an image processing task as an arithmetic expression.

The functional level contain following classes:

– `Graph` – the directed acyclic graph – the main class for building, compiling and execution of the task;

– `Node` – the abstract parent class for a graph node;

– Derivative classes of `Node` for supported image processing operations, derivative image and store classes for different source and destination data;

– `Image` – the image in memory;

– `Store` – the container for non-image data in memory;

– `Kernel` – the convolution kernel;

– `Scalar` – the container for scalar arguments and pixel values;

– `Trace` – the tracing facility to get error and information messages.

Classes for image processing operation have call-back functions to import non-image operation arguments, to execute an operation in in-place and not-in-place modes and to free if a working buffer or structure is required. The customer could derive its own class for an operation that is not supported by IPP DMIP and provide these functions.

The process of graph construction is intuitive and includes creation of nodes or subgraph, addition to the graph and linking of their ports. So, on the functional level the calculation of the expression

$$D = (A + B) \cdot \sqrt{\max(C, A + B)}$$

with the scaling factor 7 for multiplication and the scaling factor -4 for square root looks like (integer calculations with the scaling factor *n* means multiplication of the result by 2^{-n}):

```
Image *A, *B, *C, *D;
Graph *g = new Graph();
Node *in1=new SrcNode(A);
Node *in2=new SrcNode(B);
Node *in3=new SrcNode(C);
Node *out=new DstNode(D);
Node *add=new SimpleNode_2_1(idmAdd);
Node *mul=new SimpleNode_2_1(idmMul,7);
Node *max=new SimpleNode_2_1(idmMax);
Node *sqrt=new SimpleNode_1_1(idmSqrt,-4);
g->Add(in1);   // node 0
g->Add(in2);   // node 1
g->Add(add);   // node 2
g->Add(in3);   // node 3
g->Add(max);   // node 4
g->Add(sqr);   // node 5
g->Add(mul);   // node 6
g->Add(out);   // node 7
g->Link(0,0,2,0);
g->Link(1,0,2,1);
g->Link(2,0,4,0);
g->Link(3,0,4,1);
g->Link(4,0,5,0);
g->Link(2,0,6,0);
g->Link(5,0,6,1);
g->Link(6,0,7,0);
```

Receive data for source images `A, B, C`

```
g->Compile();
g->Execute();
```

Send data of destination image `D`

The DAG corresponding to this task is presented on Figure 1.

The acyclic graph with arbitrary number of inputs, operations and outputs can be built and executed by similar manner. `Add` function adds a node to a graph, `Link` function connects an output port of a node in the graph with an input port of another node in the same graph. `Compile` function compiles the graph, `Execute` function – executes it. These functions are primitives for implementation of higher symbolic level of IPP DMIP library.

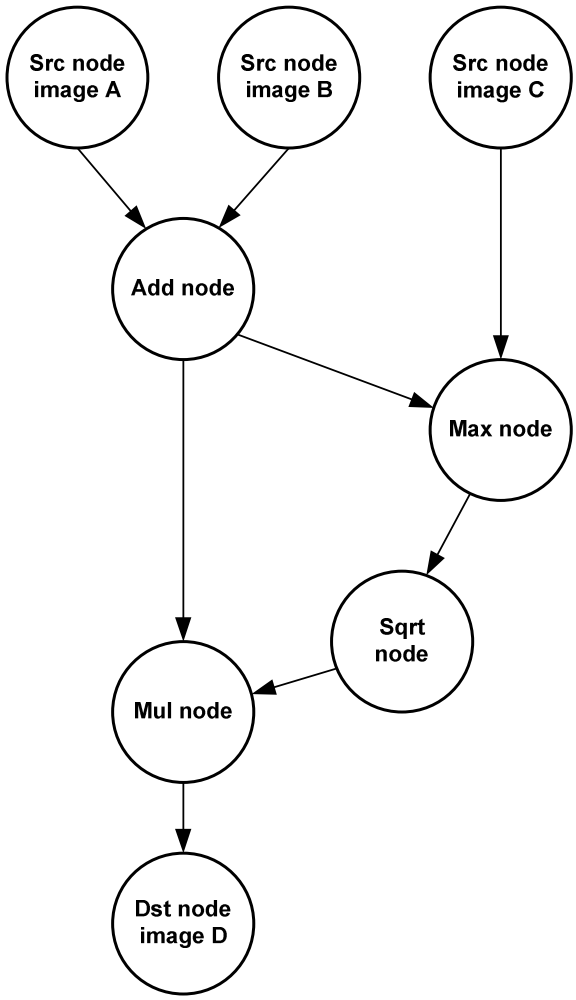

Figure 1. Directed acyclic graph of the example task.

The symbolic level includes operators and operator-like functions on IPP DMIP objects for `Graph`, `Image`, `Store` and `Scalar` operands are supported. The result of operator (function) is the graph containing operands

and the node for corresponding operation. The assignment to the `Graph` object results in saving a graph built in the right part, the assignment to an `Image` or a `Store` object causes addition of the destination node to the right part `Graph`, compilation and execution of it. While programming on the symbolic level we should remember C++ rules for precedence of overloaded operators. The code for the previous example looks like:

```
Image A(…), B(…), C(…), D(…);
D=((A+B)*(Sqrt(Max(C,A+B)),-4),7);
```

We can avoid double calculation of `A+B` by assignment this expression to another graph object. Two entries of the subgraph `G` will be detected and identified while graph compilation.

```
Image A(…), B(…), C(…), D(…);
Graph G=A+B;
D=(G*(Sqrt(Max(C,G)),-4),7);
```

We can calculate several results (e.g. the expression and the histogram of `A+B` using one pass of input images) using graph concatenation:

```
Image A(…), B(…), C(…), D(…);
HistogramStore H(256);
Graph G=A+B;
Graph I=(Histogram(G,255,0,256)>>H);
D=((G*(Sqrt(Max(C,G)),-4),7),I);
```

The functional level is more powerful than the symbolic level. It allows for building of arbitrary graphs with several destination images. Graphs that can be built by the operator level are, generally speaking, trees or forests with identified common subtrees. It is possible to combine symbolic and functional levels, for example if we perform same operations on many sets of source images:

```
Image A(…), B(…), C(…), D(…);
Graph G=A+B;
Graph I=D<<(G*(Sqrt(Max(C,G)),-4),7);
idmStatus st = G.Compile();
if (st != idmOK) …

While (…) {
    Receive data for source images A, B, C
    st = G.Execute();
    if (st >= idmError) …
    Send data of destination image D
}
```

Different events and errors can occur while graph building, compilation and execution (e.g. a graph contains a cycle, an operation is not supported for the used data type etc). DMIP provides two methods of special case handling: the return status and the tracing facility. Functions of the DMIP functional level return special values if an error occurs. Left-side object of the assignment operator (graph, image and store) acquires the return status after executing the operator.

The DMIP tracing facility can collect records on all events while graph building, compiling and executing. It facilitates debugging of code using DMIP. The tracing facility can be switched on by `Trace::StartTrace(filename, level)` function that starts to collect messages on DMIP events. The level argument defines type of messages to be collected. Level 0 causes tracing of error messages preventing further execution (wrong graph topology, incompatible data types or image sizes, wrong parameters of operations, memory allocation fails, wrong return codes of IPP functions etc.). Warning messages that allow for continuing further graph processing are traced at the level 1. Level 2 causes tracing of all DMIP messages including object creation and deletion, all manipulations with graphs and nodes.

5. Performance

The using of DMIP allows for accelerating of image processing tasks up to 3 times. We present here performance data for two typical tasks.

The image harmonization filter (one stage of medical image processing) of the source image A is calculated by the formula:

$$D = \min\left(T_{\max}, \max\left(T_{\min}, (A - (A - F_{box}(A)) \cdot c)\right)\right),$$

where F_{box} is the 7×7 box filter, $T_{\max}$ and $T_{\min}$ – thresholds and c is a constant. Input and output images have one channel 8-bit unsigned integer data, internal calculations are performed in 32-bit floating point data. The DMIP code of the harmonization filtering looks like:

```
Image A(inData,Ipp8u,IppC1,…);
Image D(outData,Ipp8u,IppC1,…);
Kernel K(idmFilterBox,…);
Ipp32f c;
Ipp8u Tmax,Tmin;

Graph O=To32f(A);
D=Max(Min(To8u(O-(O-O*K)*c),Tmax),Tmin);
```

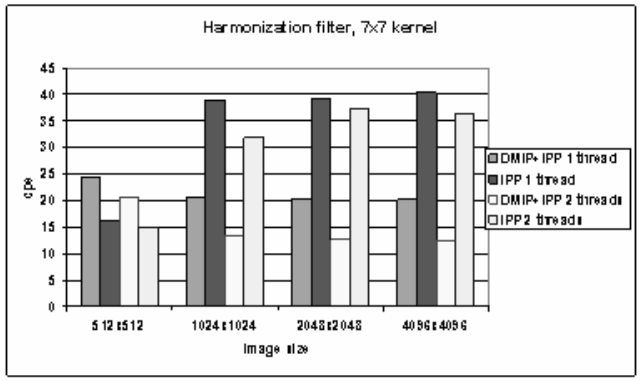

Figure 2. Performance of harmonization filter (clocks per pixel). 2-way 2.67 GHz Xeon, L2 cache 4 MByte.

Figure 2 shows performance numbers of the harmonization filter for "traditional" implementation by calls of IPP functions for the whole image and for using of DMIP and IPP libraries. Performance is measured in CPU clocks per image element (the lower the better). DMIP library accelerates filtering of big image two times for sequential code or three times for parallel code. We see no acceleration for the smaller image because the size of the task working set is close to the size of the L2 cache. Another case – the Sobel edge detector [7] for input 3-channel image:

$$D = \left|S_{hor}(Gray(A))\right| + \left|S_{vert}(Gray(A))\right|,$$

Where S_{hor} and S_{vert} are horizontal and vertical Sobel filters, $Gray(A)$ is conversion to 1-channel gray image. Source and destination images are 8-bit unsigned integer, the filtering result is the 16-bit signed integer image. The DMIP code of the Sobel edge detector looks like:

```
Image A(inData,Ipp8u,IppC3,…);
Image D(outData,Ipp8u,IppC1,…);
Kernel KH(SobelHoriz,…,ipp8u,ipp16s);
Kernel KV(SobelVert,…,ipp8u,ipp16s);

Graph O= ColorToGray(*image_src);
D=To8u(Abs(O*KH)+Abs(O*KV));
```

Figure 3 shows performance numbers of the Sobel edge detector. Again, the using of DMIP substantially accelerates calculations 1.6-1.75 times for big images.

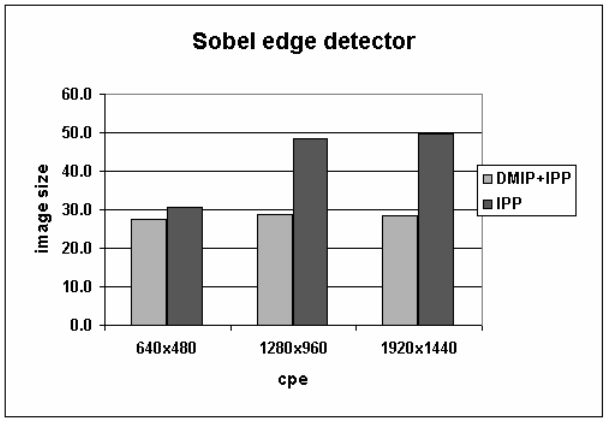

Figure 3. Performance of Sobel edge detector (clocks per pixel). 1.8 GHz Centrino, L2 cache 1 MByte.

We considered here image processing operations that can be performed in pipelined manner. In general, all these operations are pixel-wise. Some of them (filters) cause the delay in calculations on several image rows depending on kernel size and anchor. But there exist image processing operations that do not allow for row-wise pipelined calculations (FFT, image geometric transforms, image segmentation etc.). In DMIP terms they require the delay on the whole image. Such operations can be integrated into DMIP framework but the performance gain for them is not obvious.

6. Current state

Now DMIP library supports general purpose image processing operations corresponding to IPP functions. More than 100 operators and functions are implemented on the symbolic level:

– arithmetic operators (`+`, `-`, `*`, `/`) and functions (`Abs`, `Sqrt`, `Ln`, `Min`, `Max` etc.);
– logical operations (`&`, `|`, `^`, ~ operators);
– thresholding operations;
– alpha composition;
– statistical operations producing non-image data (norms, histograms, `Mean`, `Sum`, `Min`, `Max` etc.),
– image filters (`*` with a kernel): general linear, special (median, box, maximum, minimum), small kernel fixed filters;
– data type conversions;
– color space conversions;
– image channel manipulations.

Main IPP image formats are supported: integer (8, 16 or 32 bit signed and unsigned) and floating-point (32 bit) data, images with one, three or four channels. IPP functions working in in-place (if one of source images is simultaneously the destination image) and not-in-place (if the result is written to another image) modes are used. DMIP automatically calls corresponding primitive functions for image slices. IPP primitives are used if they exist and own C functions otherwise.

7. User-defined functions in DMIP

Although IPP contains hundreds of imaging functions it does not cover all possible methods of image processing. DMIP library proposes the simple method of extension with new functions. DMIP contains abstract classes for nodes with different number of inputs and outputs derived from `Node` class and leaf node classes for operations implemented in DMIP. The new node class can be derived from one of intermediate or leaf node classes and should contain following functions:

– Constructor and destructor, initialization of variables controlling operation execution in in-place and parallel modes;
– Node initialization function with possible additional parameters of the new operation;
– Node copy function;
– The node name for the tracing facility (optional);
– Functions to check input and output data types and channel numbers (optional);
– Functions to allocate and free additional memory or structures for the operation (optional);
– Functions to execute the operation in in-place and not-in-place modes;
– Operators or functions of the symbolic level.

DMIP contains examples of external derivative classes that can be used as a pattern for new user-defined operations. Changes of the pattern for the new operation are generally formal.

8. DMIP extension

In some sense DMIP models dataflow computing where the minimal unit of data is an image row. Operations on images differently affect the data flow in such model. Filtering operations cause finite delay on several lines depending on kernel size and anchor. Geometric or linear transforms (FFT or affine transform) of the image could require stall on the whole image. Some operations (resize or pyramids) require the change of the image size.
Now DMIP supports operations with finite delay, same input and output image, no more than two inputs and no more than one output. We plan to extend the scope of DMIP operations in following directions:
– other formats of image (e.g. images with plane data layout);
– more operations of supported types;
– operations with full image delay (as FFT);
– operations changing the image size (as image resize);
– optimization of the graph compilation stage;
– new examples of DMIP usage including image codecs.
Besides, some IPP functions do not fit well to the paradigm of pipelined image processing. Efficient implementations of many algorithms could be split to a prologue, data accumulation and a epilogue. For example, the fastest method to calculate the histogram of a 8-bit integer image implemented in IPP consists of accumulator zeroing (prologue), accumulation of the full histogram and recalculation of the histogram for required levels (epilogue). When DMIP uses such IPP functions the prologue and the epilogue is executed for each slice. So, the efficient pipelined processing could require new optimized primitive functions.

9. Conclusion

DMIP library combines benefits of pipelined image processing and using of optimized IPP library and gives several times acceleration. It provides the intuitive interface for formula-level programming and does not require knowing low-level interfaces or optimization details. DMIP library provides also the simple method of adding new operation and their execution in the pipelined manner.

References

[1] *Intel Integrated Performance Primitives for Intel Architecture*. Reference Manual. Vol. 2: Image and Video Processing, June 2007.
[2] J. Herath, Y. Yamaguchi, N. Saito, T. Yuba. Dataflow Computing Models, Languages, and Machines for Intelligence Computations. *IEEE Transactions on Software Engineering*, Vol. 14, No. 12, December 1988.
[3] *Programming in Java™ Advanced Imaging*, Release 1.0.1, http://java.sun.com/javase, November 1999.
[4] O. Beckmann, A. Houghton, P. H. J. Kelly and M. Mellor, Run-time code generation in C++ as a foundation for domain-specific optimization. In *Domain-Specific Program Generation International Seminar*, Dagstuhl Castle, Germany, March 23-28, 2003,
[5] T. Veldhuizen. *Blitz++ User's Guide*. http://www.oonumerics.org/blitz, March 2006.
[6] GEGL (Generic Graphics Library) http://www.gegl.org
[7] M. Nixon, A. Aguado, *Feature Extraction & Image Processing*, Newnes, 2002