Adding Parallelism to the Hybrid Image Processing Library in Multi-Threading and Multi-Core Systems

Bogusław Cyganek

AGH University of Science and Technology
Al. Mickiewicza 30, 30-059 Kraków, Poland
cyganek@agh.edu.pl

DATICS-NESEA'11, 8-9 December 2011, Fremantle, Australia

An overview:

- Problem statement – what we try to achieve?
- Architecture of the hybrid image library (HIL).
- Adding parallelizm into the software layer with OpenMP.
- Code refactoring techniques.
- Experiments
- Conclusions.

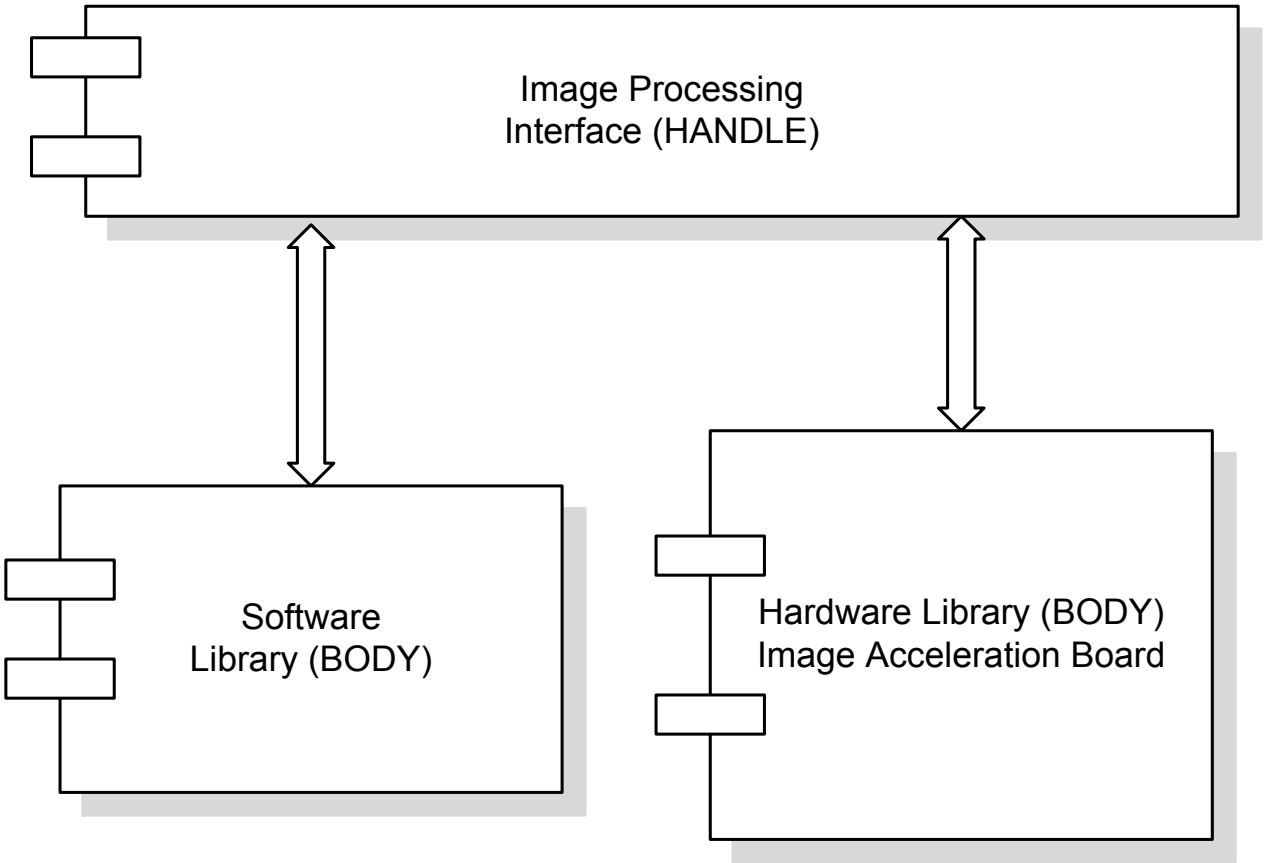
What we try to achieve?

Architecture of the System – An Overview

1. A hybrid image processing library to take full advantage of the *multi-threading* and *multi-core* computer systems.
2. Architecture of the library follows the handle-body design pattern.
3. The handle layer represents an abstract interface which *hides implementation details* and leaves the user's code untouched.
4. Software/hardware implementations are hidden in the body layers which allow for high level of parallelism and best exploitation of the available resources.
5. Software *refactoring* techniques to take advantage of the parallelizm offered with an advent of multi-core systems.

Architecture of the System – Basic Structure

Hybrid hardware / software structure.



Architecture of the System – Main Assumptions

1. Operations of the library are defined in terms of *processing objects*. There are two kinds of the processing objects:

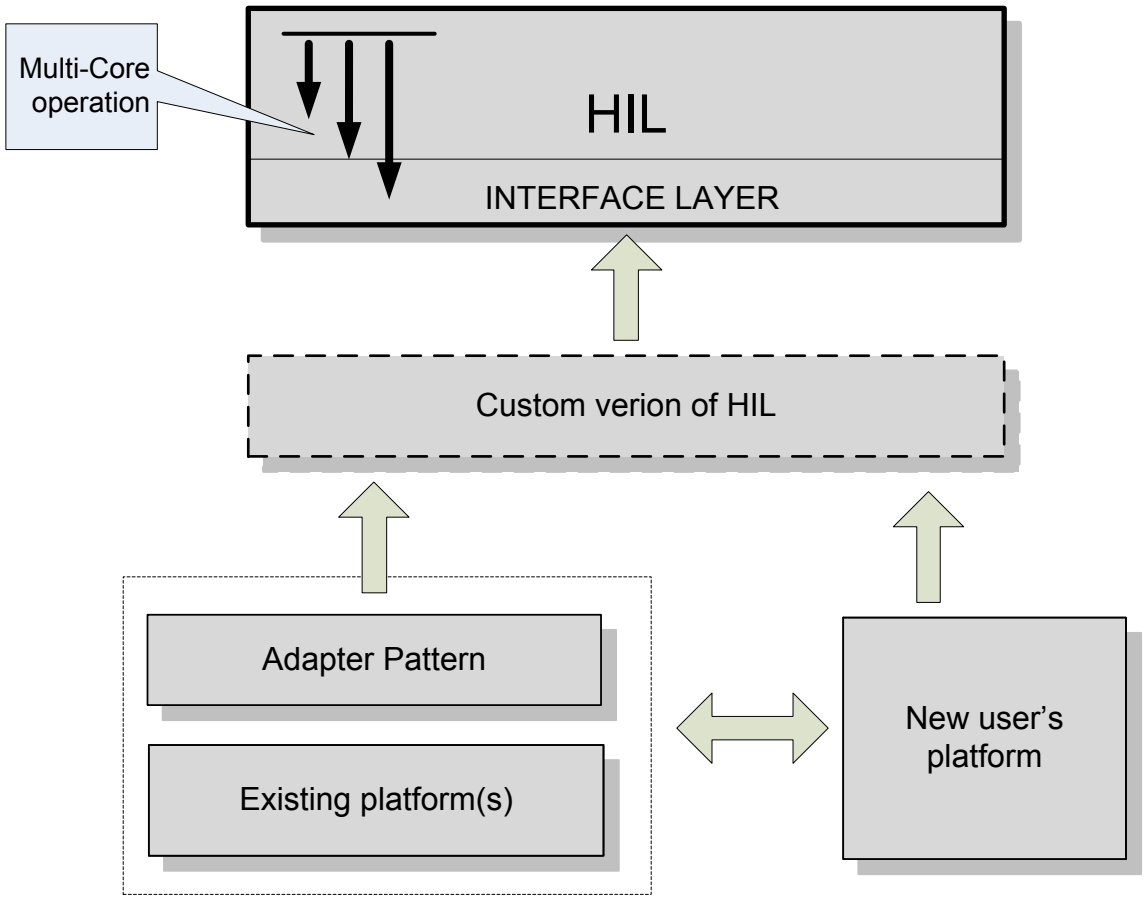
   A hardware processing object – a module implementing an IP operation.

   A software processing objects which is constructed in a calling software to perform a given action in the library.

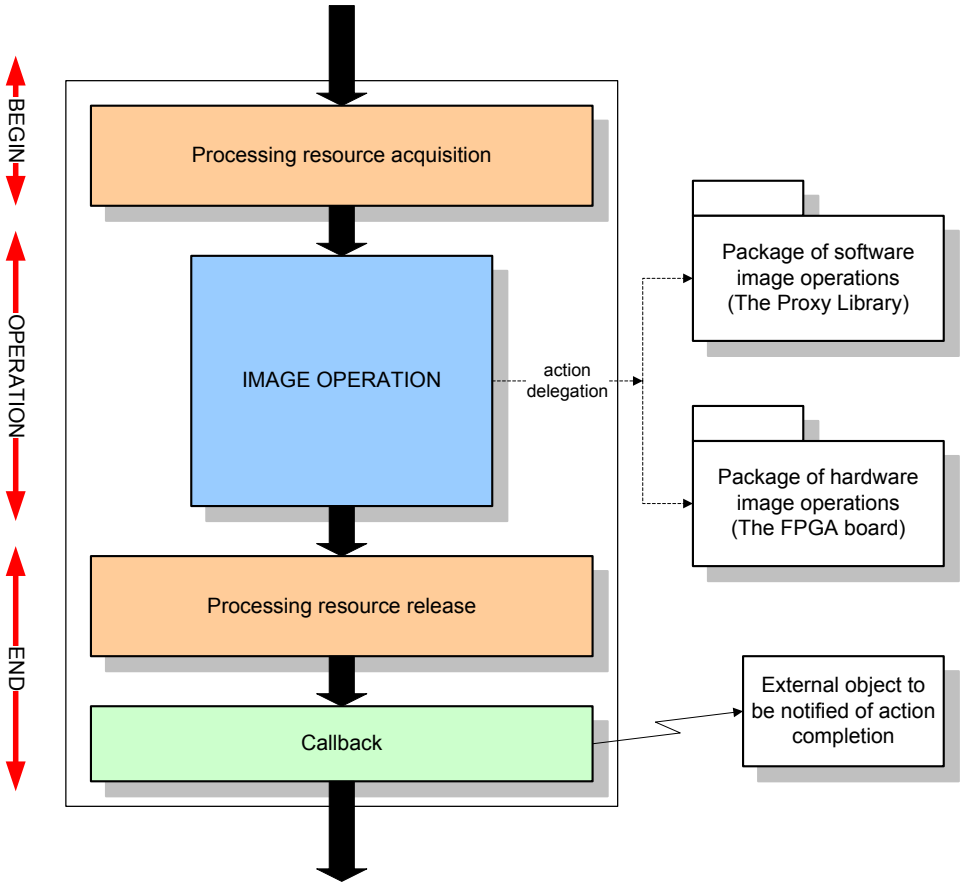
2. The library allows many image operations to be performed in a hardware pipeline. The initial pipeline can be fed by a library user by constructing a composition of processing objects and run as a batch-job in hardware.
3. The input images allow multiple types of pixels.
4. Hardware processing modules can operate in two of the arithmetic modes:
5. Integer format – in this representation all operations are performed only with an integer arithmetic (e.g. searching for a maximum value in an image, adding two images, etc.)
6. Fixed point format – in this representation the 16 bits per pixel is assumed for input and output images, although pixels are in the fixed point format. However, the single 16 bits/pixel channels can be further joined to form an *extended precision* fixed point representation (e.g. 2×16, 3×16, …, $m\times16$). Similarly, the input parameters can be supplied in a fixed point format.

Architecture of the System – Using HIL from External Systems

Collaboration diagram of the HIL library with new and existing platforms



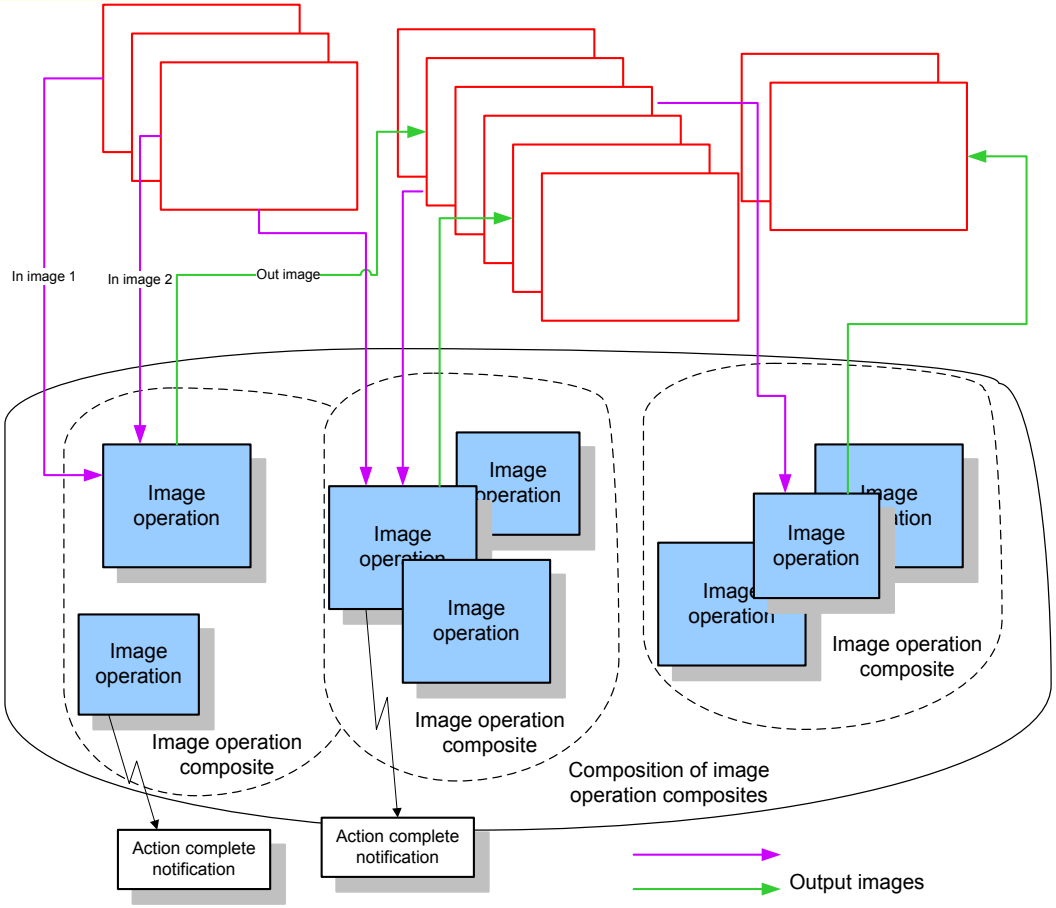
Architecture of the System – Operation Processing Steps



Flow chart of each image operation. There are three stages of execution:

1. Operation preamble which consists of the acquisition of processing resources,
2. The main image operation, and
3. The operation finishing sequence which consists of resource release and callback (notification) mechanism

Architecture of the System – Multi-Operation Processing Steps

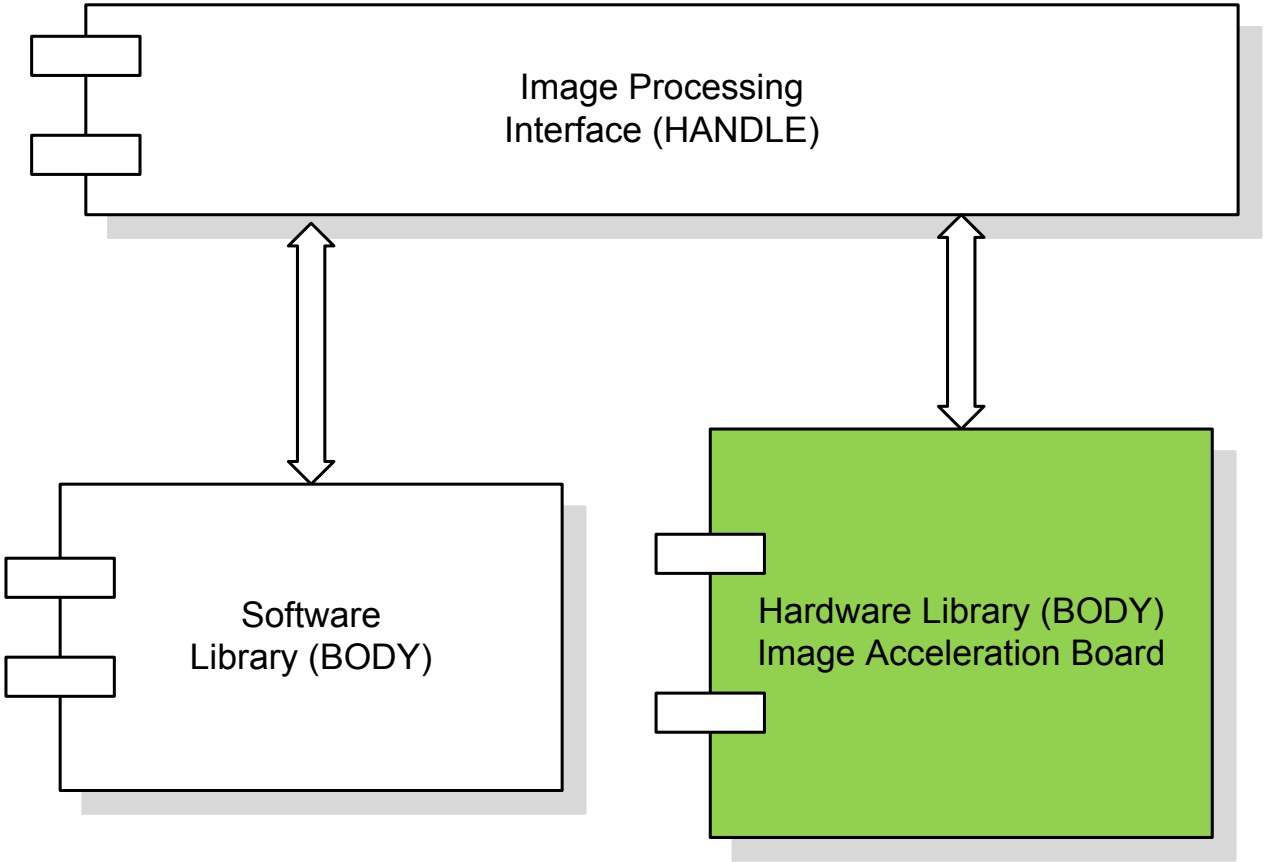


A scenario diagram of image operations. Operations accept many images and are grouped in certain compositions. Each operation has associated number of input images and a single output image. An image can constitute an input for some operations and output for the other. Order of execution is determined by position of that operation in enclosing composition object. Some operations can launch callback notification upon completion. Each operation can be supplied with resource access object

How can we improve perfromance?

Improving performance

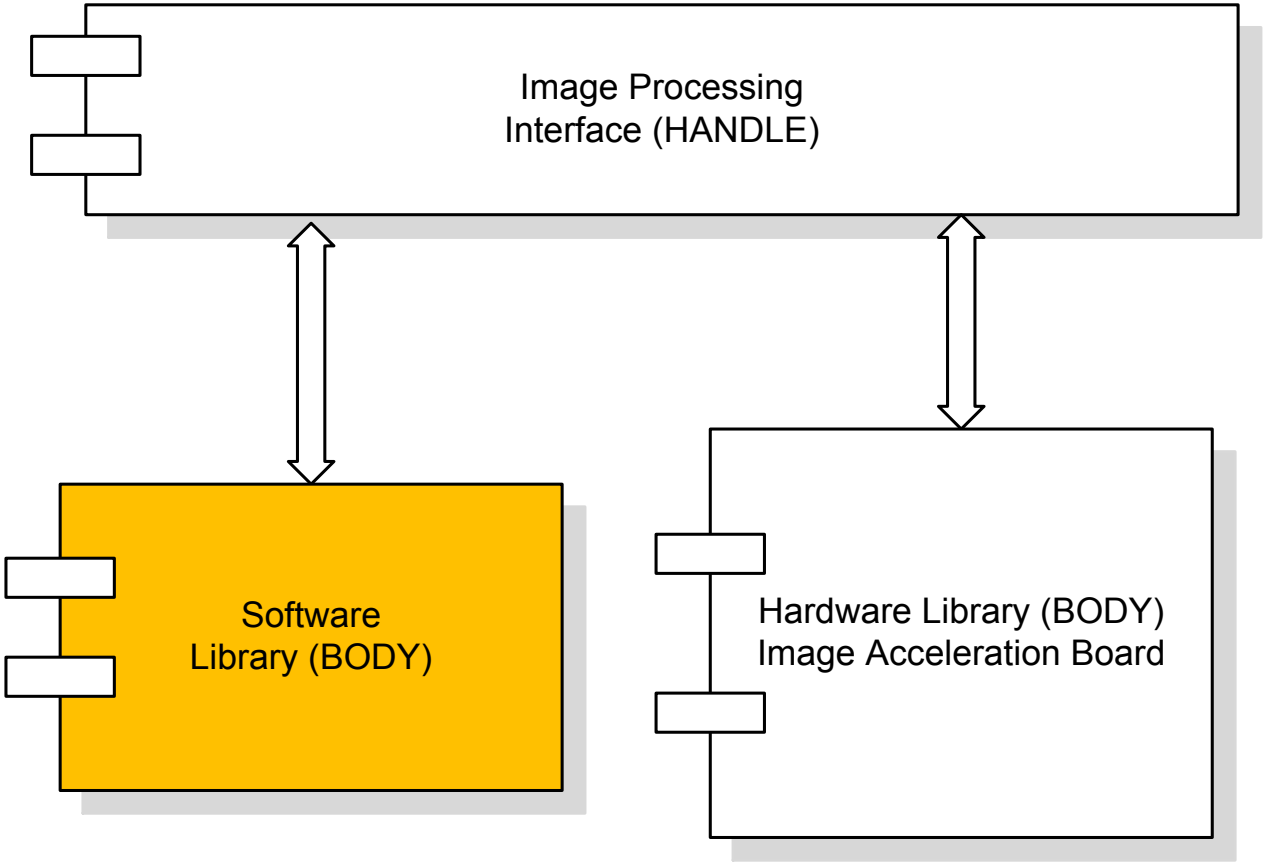
Adding parallelizm with hardware acceleration board



Very attractive, but …

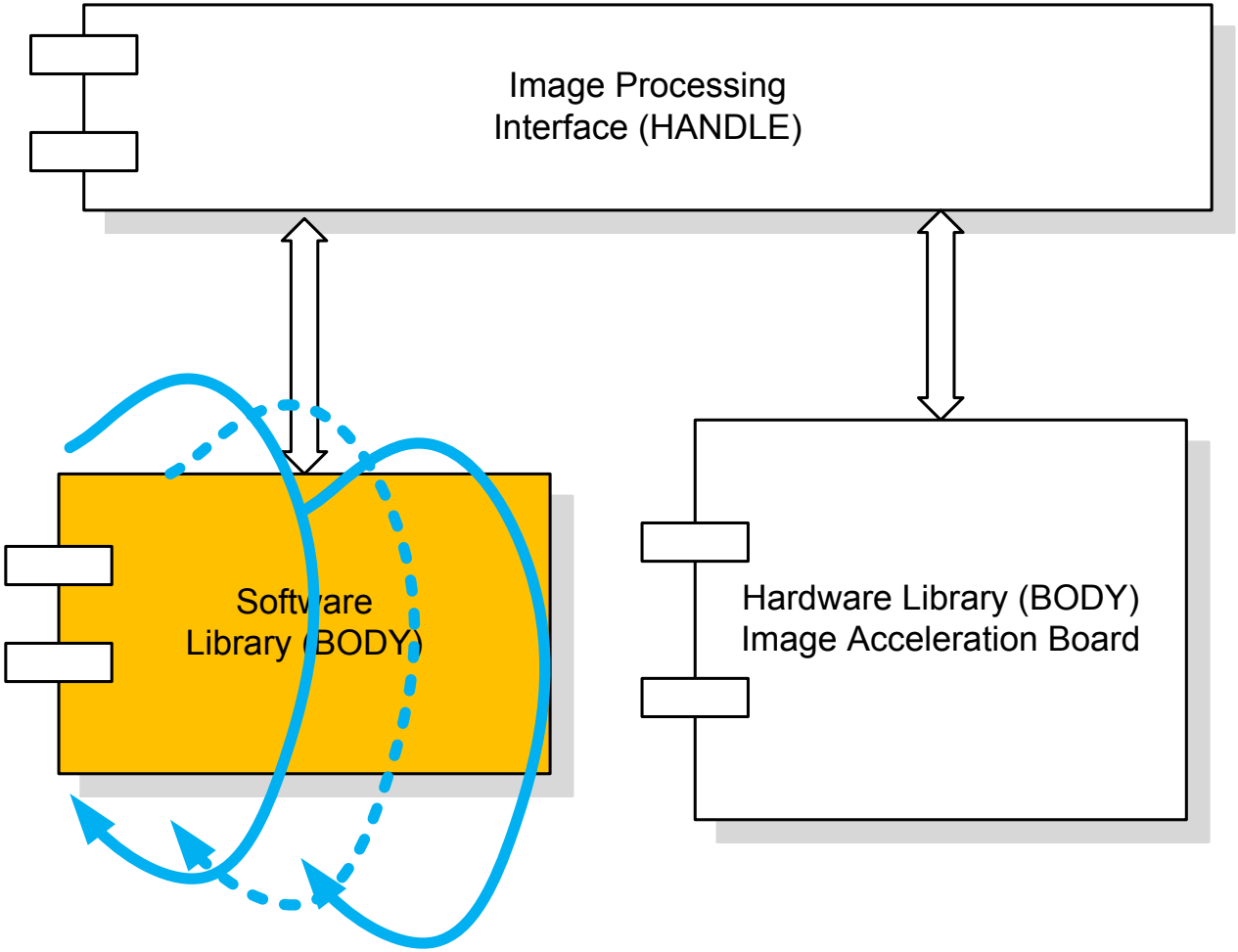
Improving performance

Improving software …



Improving performance

Adding **parallelizm to the software layer**



Improving performance with **OpenMP**

- OpenMP is a parallel programming library for shared memory multithreading execution.
- It works most effective in the multi-processor or multi-core computer systems.
- OpenMP allows easy refactoring of the loop constructs which process significant amount of iterations (most efficient if the operations inside a loop are independent – no synchronizations)

Improving performance with **OpenMP**

- OpenMP is a parallel programming library for shared memory multithreading execution.
- It works most effective in the multi-processor or multi-core computer systems.
- OpenMP allows easy refactoring of the loop constructs which process significant amount of iterations (most efficient if the operations inside a loop are independent – no synchronizations)

[1] *www.openmp.org*

[2] Chapman B., Jost G., Van Der Pas A.R.: Using OpenMP. Portable Shared Memory Parallel Programming. MIT Press, 2008.

Improving performance with **OpenMP** – An example

1. Find a function which processes massive data.

```
template< class T >
double SumAllImagePixels( const TImageFor< T > & in )
{
        register double sum = 0.0;
        register T * src_data_ptr = in.GetImageData();
        register int data_num = in.GetElems();

        while( data_num -- != 0 )
                sum += * src_data_ptr ++;

        return sum;
}
```

Improving performance with **OpenMP** – An example

1. Find a function which processes massive data.

```
template< class T >
double SumAllImagePixels( const TImageFor< T > & in )
{
        register double sum = 0.0;
        register T * src_data_ptr = in.GetImageData();
        register int data_num = in.GetElems();

        while( data_num -- != 0 )
                sum += * src_data_ptr ++;

        return sum;
}
```

2. Find a loop of interest (convert to corresponding **for()** loop).

Improving performance with **OpenMP** – An example

1. Find a function which processes massive data.

```
template< class T >
double SumAllImagePixels( const TImageFor< T > & in )
{
        register double sum = 0.0;
        register T * src_data_ptr = in.GetImageData();
        register int data_num = in.GetElems();

        while( data_num -- != 0 )
                sum += * src_data_ptr ++;

        return sum;
}
```

2. Find a loop of interest (convert to corresponding **for()** loop).

3. Identify all shared (synchronization !!) and private variables.

Improving performance with **OpenMP** – An example

```
template< class T >
double SumAllImagePixels( const TImageFor< T > & in )
{
        register double sum = 0.0;
        register T * src_data_ptr = in.GetImageData();
        register int data_num = in.GetElems();

#if USE_OPEN_MP

        register int i = 0;

        #pragma omp parallel for default( none ) \
                shared( src_data_ptr, data_num ) \
                private( i ) \
                reduction( + : sum ) \
                schedule( static )
        for( i = 0; i < data_num; ++ i )
        {
                sum += (double) src_data_ptr[ i ];
        }       // here we have a common barrier

#else // USE_OPEN_MP

        while( data_num -- != 0 )
                sum += * src_data_ptr ++;

#endif // USE_OPEN_MP

        return sum;
}
```

Use C++ preprocessor to keep two versions of code

4. Write new version of the loop which is OpenMP aware.

Improving performance with **OpenMP** – An example

```
register double sum = 0.0;
register T * src_data_ptr = in.GetImageData();
register int data_num = in.GetElems();

register int i = 0;

#pragma omp parallel for default( none ) \
        shared( src_data_ptr, data_num ) \
        private( i ) \
        reduction( + : sum ) \
        schedule( static )

for( i = 0; i < data_num; ++ i )
{
        sum += (double) src_data_ptr[ i ];
}       // here we have a common barrier
```

Improving performance with **OpenMP** – An example

```
register double sum = 0.0;
register T * src_data_ptr = in.GetImageData();
register int data_num = in.GetElems();

register int i = 0;

#pragma omp parallel for default( none ) \
        shared( src_data_ptr, data_num ) \
        private( i ) \
        reduction( + : sum ) \
        schedule( static )

for( i = 0; i < data_num; ++ i )
{
        sum += (double) src_data_ptr[ i ];
}       // here we have a common barrier
```

Improving performance with **OpenMP** – An example

```
register double sum = 0.0;
register T * src_data_ptr = in.GetImageData();
register int data_num = in.GetElems();

register int i = 0;

#pragma omp parallel for default( none ) \
        shared( src_data_ptr, data_num ) \
        private( i ) \
        reduction( + : sum ) \
        schedule( static )

for( i = 0; i < data_num; ++ i )
{
        sum += (double) src_data_ptr[ i ];
}       // here we have a common barrier
```

Improving performance with **OpenMP** – An example

```
register double sum = 0.0;
register T * src_data_ptr = in.GetImageData();
register int data_num = in.GetElems();

register int i = 0;

#pragma omp parallel for default( none ) \
        shared( src_data_ptr, data_num ) \
        private( i ) \
        reduction( + : sum ) \
        schedule( static )

for( i = 0; i < data_num; ++ i )
{
        sum += (double) src_data_ptr[ i ];
}       // here we have a common barrier
```

Improving performance with **OpenMP** – An example

```
register double sum = 0.0;
register T * src_data_ptr = in.GetImageData();
register int data_num = in.GetElems();

register int i = 0;

#pragma omp parallel for default( none ) \
        shared( src_data_ptr, data_num ) \
        private( i ) \
        reduction( + : sum ) \
        schedule( static )

for( i = 0; i < data_num; ++ i )
{
        sum += (double) src_data_ptr[ i ];
}       // here we have a common barrier
```

Improving performance with **OpenMP** – An example

```
register double sum = 0.0;
register T * src_data_ptr = in.GetImageData();
register int data_num = in.GetElems();

register int i = 0;

#pragma omp parallel for default( none ) \
        shared( src_data_ptr, data_num ) \
        private( i ) \
        reduction( + : sum ) \
        schedule( static )

for( i = 0; i < data_num; ++ i )
{
        sum += (double) src_data_ptr[ i ];
}       // here we have a common barrier
```

Improving performance with **OpenMP** – An example

```
register double sum = 0.0;
register T * src_data_ptr = in.GetImageData();
register int data_num = in.GetElems();

register int i = 0;

#pragma omp parallel for default( none ) \
        shared( src_data_ptr, data_num ) \
        private( i ) \
        reduction( + : sum ) \
        schedule( static )

for( i = 0; i < data_num; ++ i )
{
        sum += (double) src_data_ptr[ i ];
}       // here we have a common barrier
```

Improving performance with **OpenMP** – An example

```
register double sum = 0.0;
register T * src_data_ptr = in.GetImageData();
register int data_num = in.GetElems();

register int i = 0;

#pragma omp parallel for default( none ) \
        shared( src_data_ptr, data_num ) \
        private( i ) \
        reduction( + : sum ) \
        schedule( static )

for( i = 0; i < data_num; ++ i )
{
        sum += (double) src_data_ptr[ i ];
}       // here we have a common barrier
```

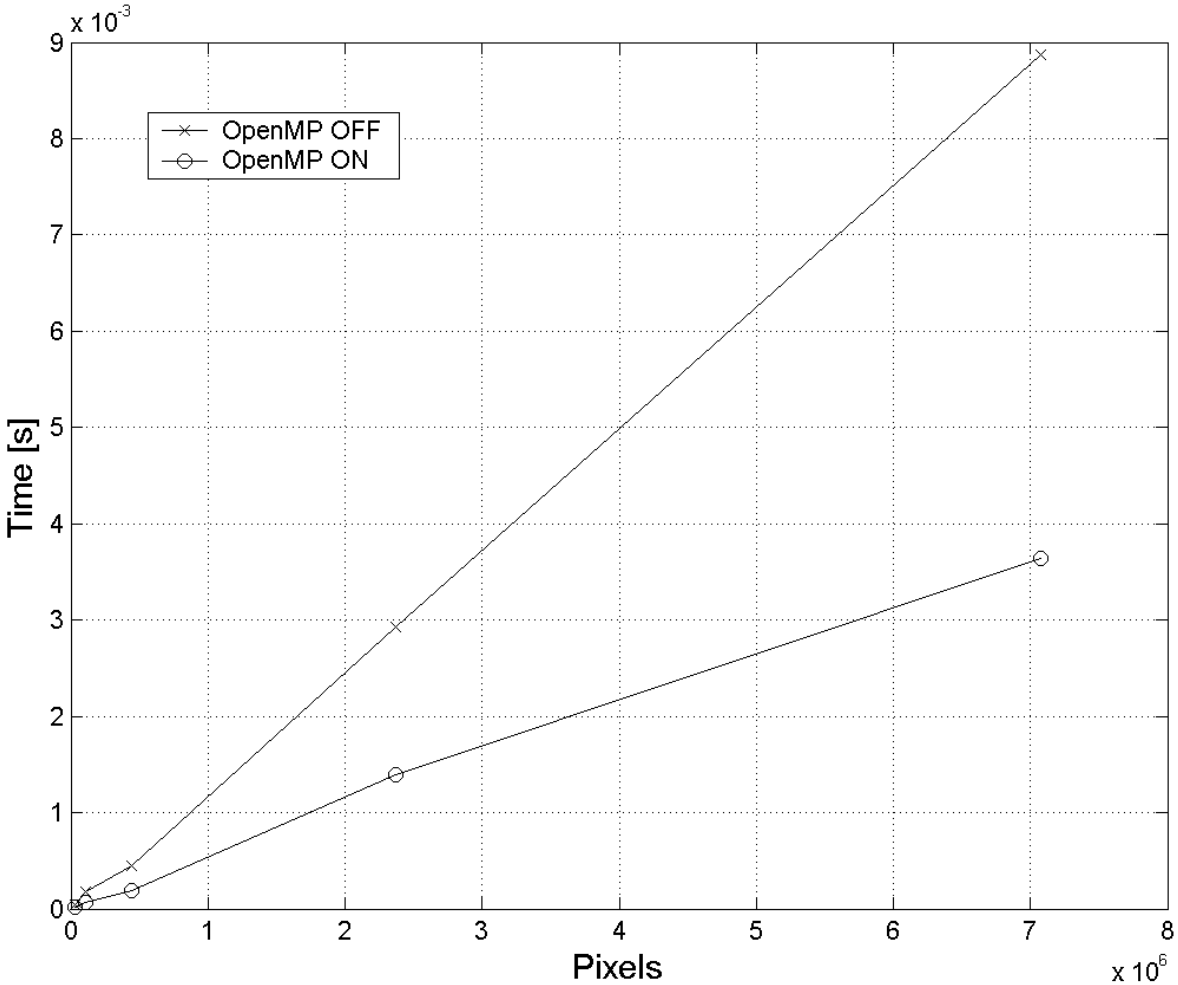
Experiments

Experimental setup

- Dell® Precision M6500;
- Windows 7 Professional;
- Intel® Core i7 CPU Q820 @ 1.73GHz quad-core processor and 8 GB of RAM. (8 threads due to the Intel's Hyper-Threading Technology);
- Dual core AMD® E-350 (Zacate) @ 1.6GHz;
- Software developed with the Microsoft Visual Studio® .NET 2010 C++ compiler;

Experimental results

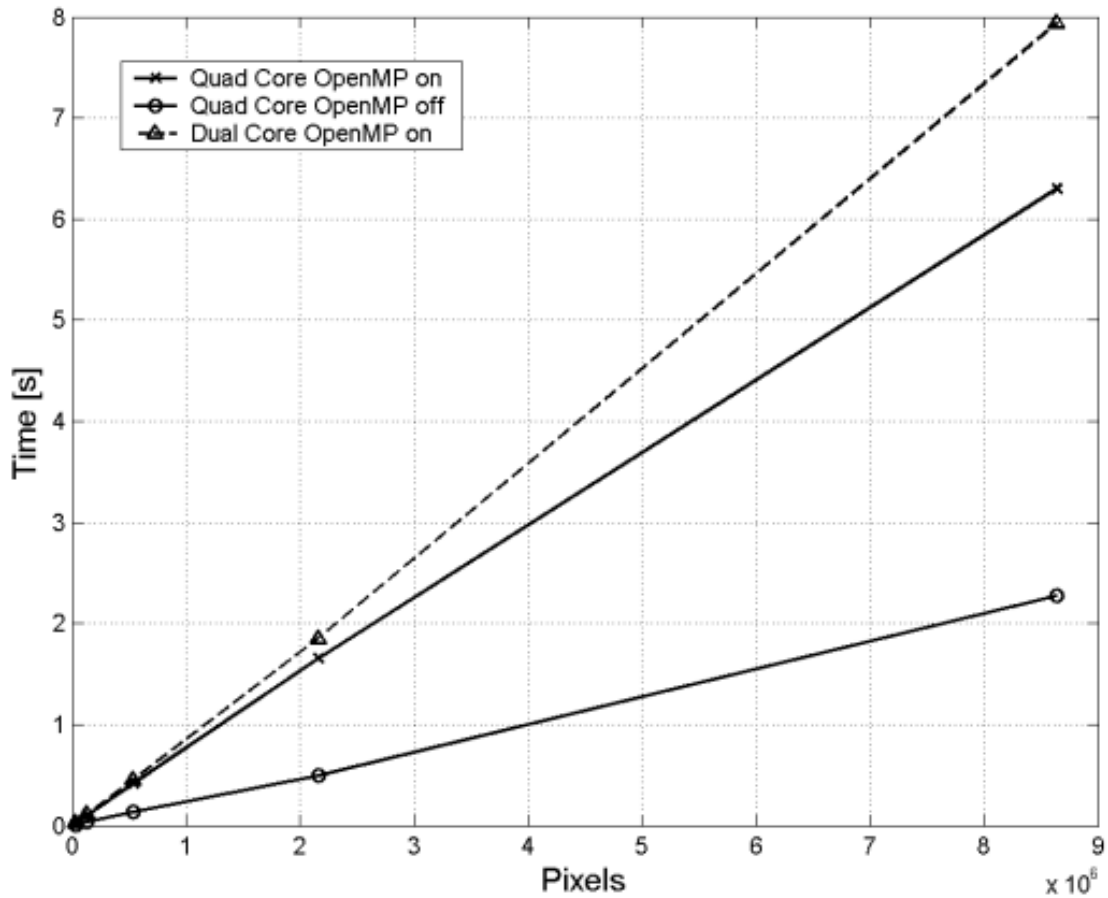
System performance with and without OpenMP, measured for the presented function **SumAllImagePixels** on Intel® Core i7 CPU Q820 quad core



Experimental results

System performance with and without OpenMP, measured for the **median** operation with the 7x7 structure element. Measurements performed for two microprocessors:

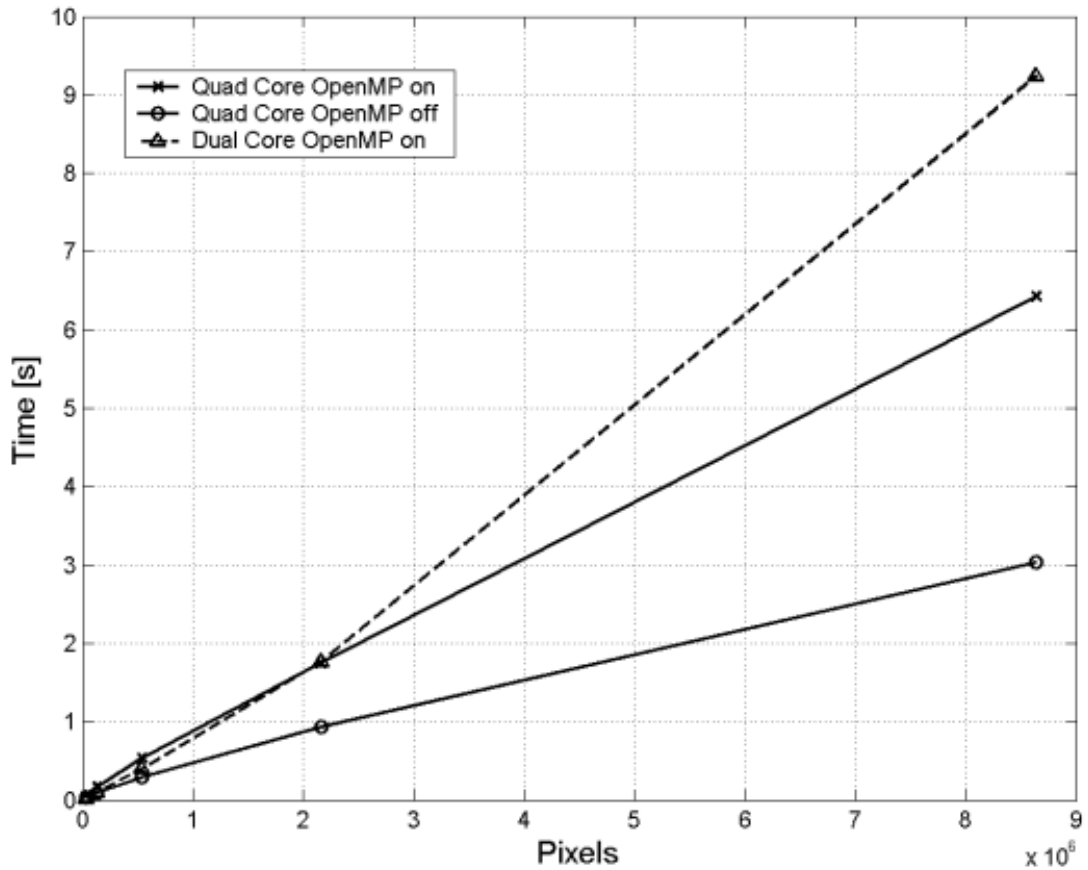
1. Intel® Core i7 CPU Q820 quad core (solid),
2. Dual core AMD® E-350 (Zacate) @ 1.6GHz (dashed)



Experimental results

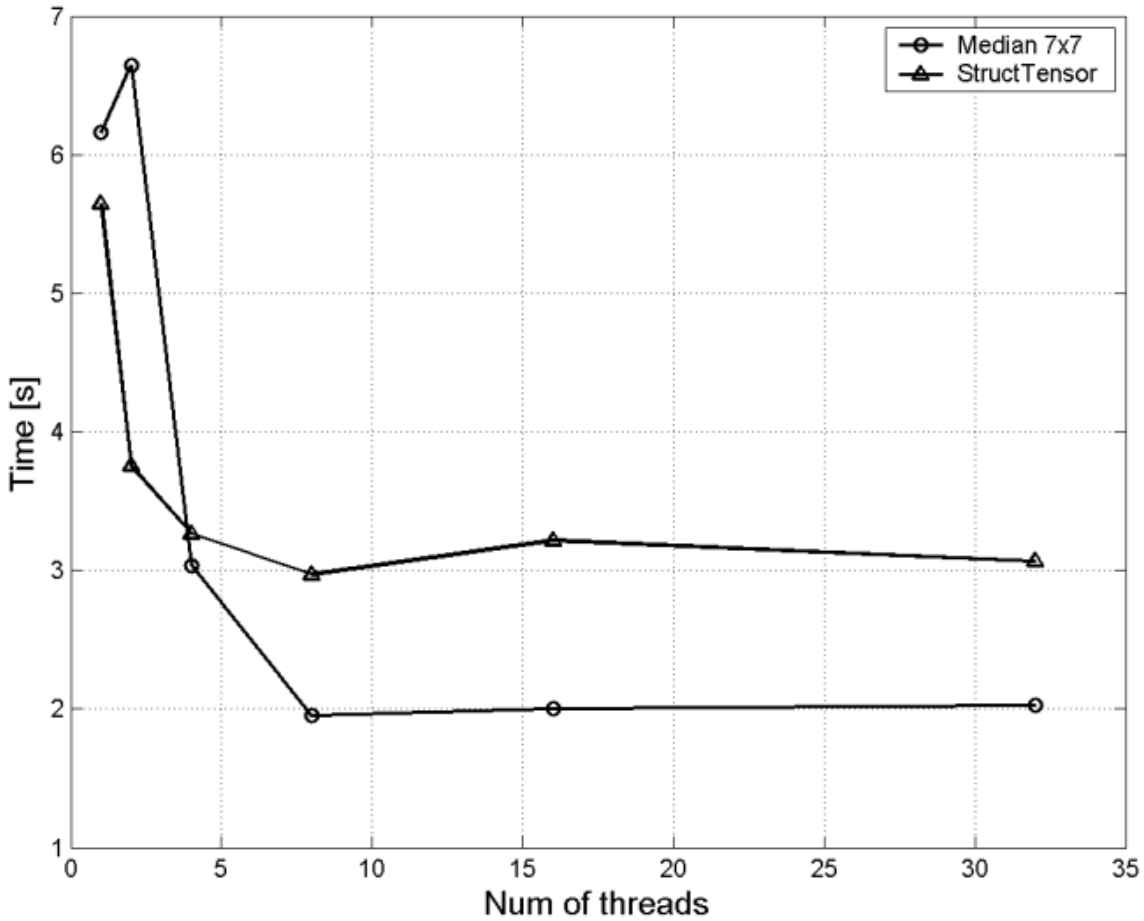
System performance with and without OpenMP, measured for the **median** operation with the 7x7 structure element. Measurements performed for two microprocessors:

1. Intel® Core i7 CPU Q820 quad core (solid),
2. Dual core AMD® E-350 (Zacate) @ 1.6GHz (dashed)



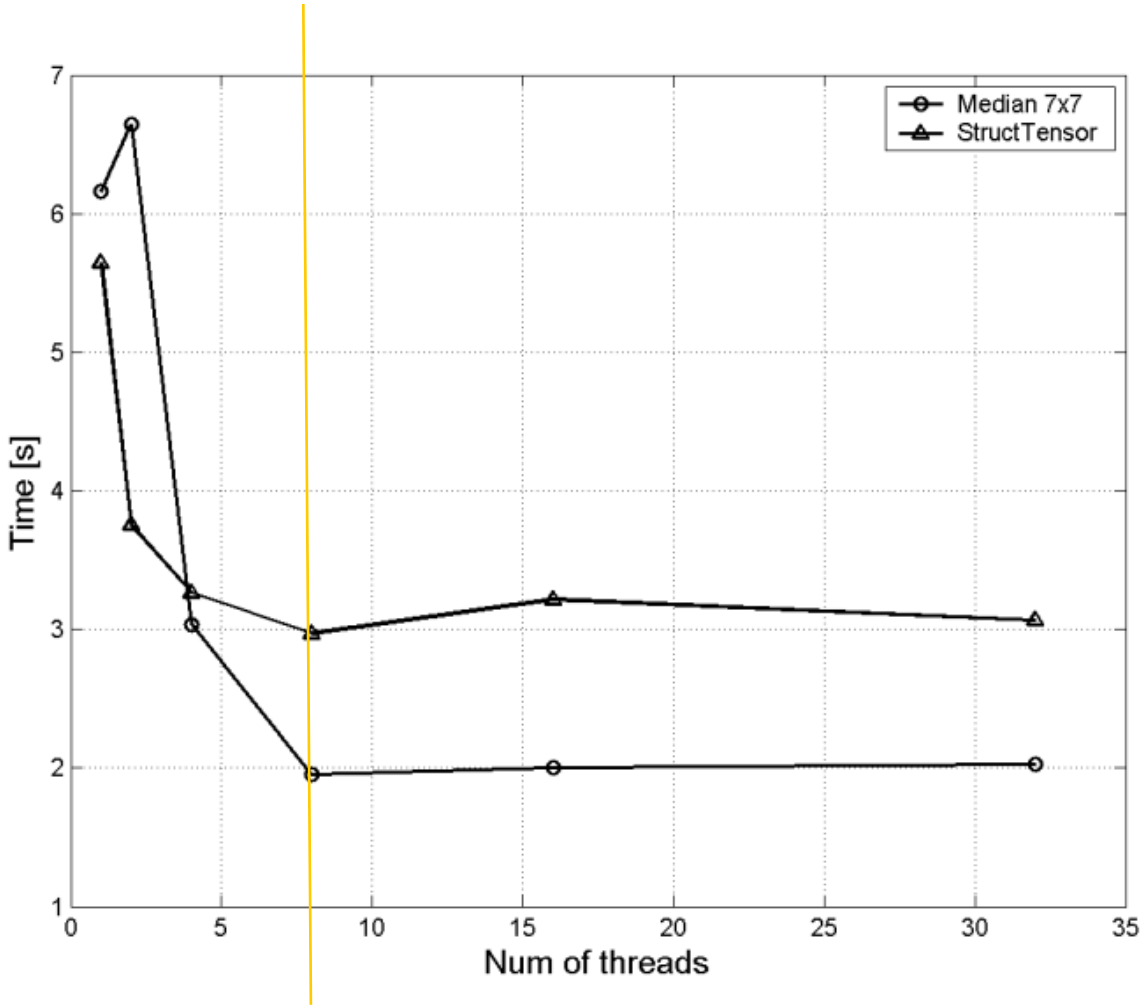
Experimental results

System performance in respect to **the number of threads** measured for the **median** with the 7x7 structure element (a), and computation of the **structural tensor** (b). Measured for Intel® Core i7 CPU Q820 quad core microprocessor



Experimental results

System performance in respect to **the number of threads** measured for the **median** with the 7x7 structure element (a), and computation of the **structural tensor** (b). Measured for Intel® Core i7 CPU Q820 quad core microprocessor



Conclusions

Conclusions:

- Proper design of hybrid hardware/software systems meets massive data processing needs;
- Handle/body architecture allows easy changes in implementations (hardware and/or software + OpenMP);
- Built-in parallelism into the software layer - with the OpenMP library
- Low cost refactoring method of existing code to gain full multi-core benefits.
- Experimental results show a speed-up ratio of 2 up to **3.5 times** on image processing operations (compared to the serial implementations)
- The presented multi-core version of the library is available on the Internet (http://home.agh.edu.pl/~cyganek/HIL_OpenMP.zip)

Thank you!