# Data- and task parallel image processing on a mixed SIMD-ILP platform using skeletons and asynchronous RPC*

## On-camera image processing

As processors are becoming faster, smaller, cheaper, and more efficient, new opportunities arise to integrate them into a wide range of devices. The SmartCam project investigates the specific field of intelligent cameras, where image processing hardware is integrated in a camera.

The processing done on an intelligent camera has very specific characteristics. On the one hand, low-level image processing operations such as interpolation, segmentation and edge enhancement are local, regular, and require vast amounts of bandwidth. On the other hand, high-level operations like classification, path planning, and control may be irregular while typically consuming less bandwidth. A typical smart camera architecture template therefore contains data-parallel (SIMD) as well as instruction-level parallel (ILP) processors.

## Heterogeneous hardware

One of the main goals of the project is keeping the system easy to program. As a consequence, the user should not be concerned with the heterogeneous nature of the platform, and his functions should map transparently to any of the processors in our architecture template. We use **algorithmic skeletons** to separate the data parallel implementation from the computation itself.

However, as a consequence of the use of heterogeneous hardware, not everything can be done data parallel. To keep all the processors busy, different operations need to run task-parallel, and we are using **asynchronous remote procedure call** to realize this while still presenting a familiar function-call interface to the user.

## Data parallelism: algorithmic skeletons

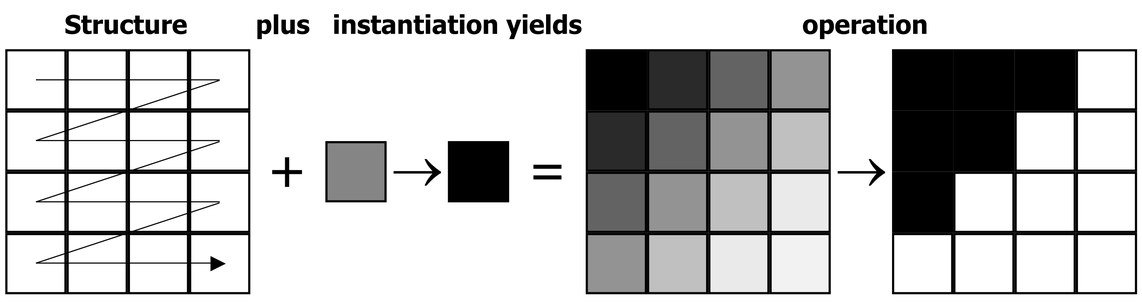
By separating the parallel implementation from the actual computation, algorithmic skeletons allow the user to avoid the bookkeeping normally associated with parallel processing. By writing different skeleton implementations for different processors, they also enable us to map a user function to any processor in our architecture template.

Algorithmic skeletons can be thought of as higher-order functions that repeatedly call a user-provided instantiation function. They provide the **structure** that, combined with the **instantiation**, results in the actual **operation**:

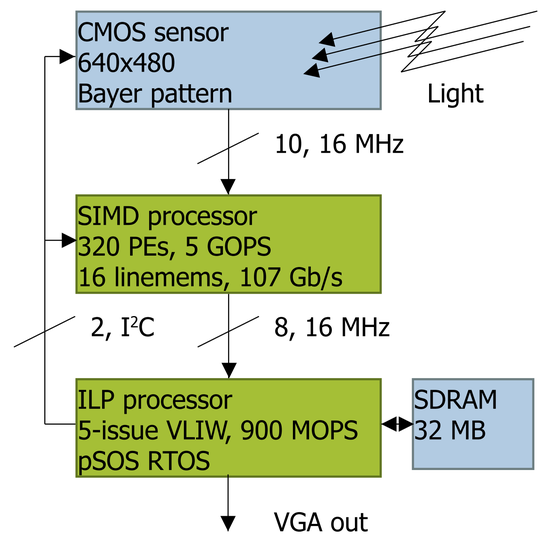
**Structure:**
```
void pixelop(PIXEL **in, PIXEL **out, ...) {
   for (y=0; y<HEIGHT: y++)
      for (x=0; x<WIDTH; x++)
         out[y][x] = op(in[y][x], ...);
}
```
**+**

**Instantiation:**
```
inline PIXEL binarize(PIXEL p, int threshold) {
   return (p>threshold);
}
```
**=**

**Operation:**
```
pixelop(binarize)(PIXEL **in, PIXEL **out, int threshold) {
   for (y=0; y<HEIGHT: y++)
      for (x=0; x<WIDTH; x++)
         out[y][x] = (in[y][x]>threshold);
}
```





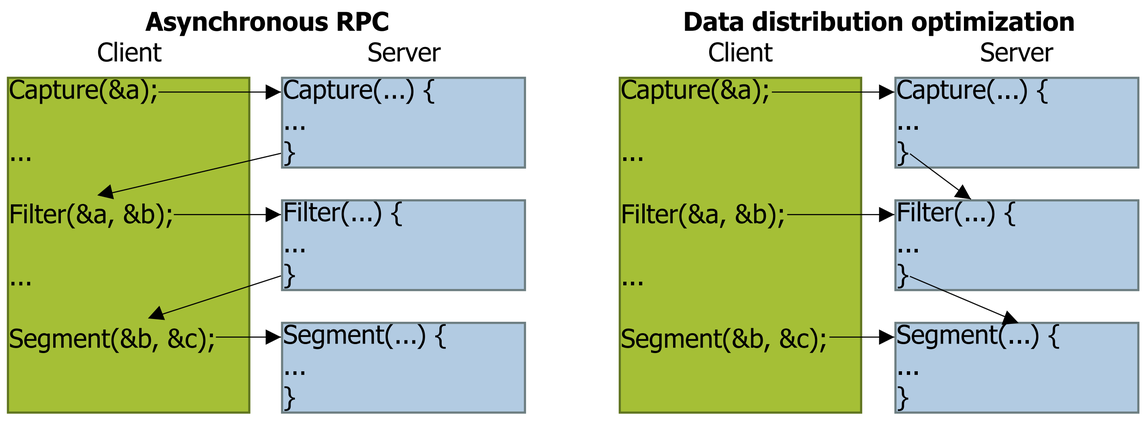


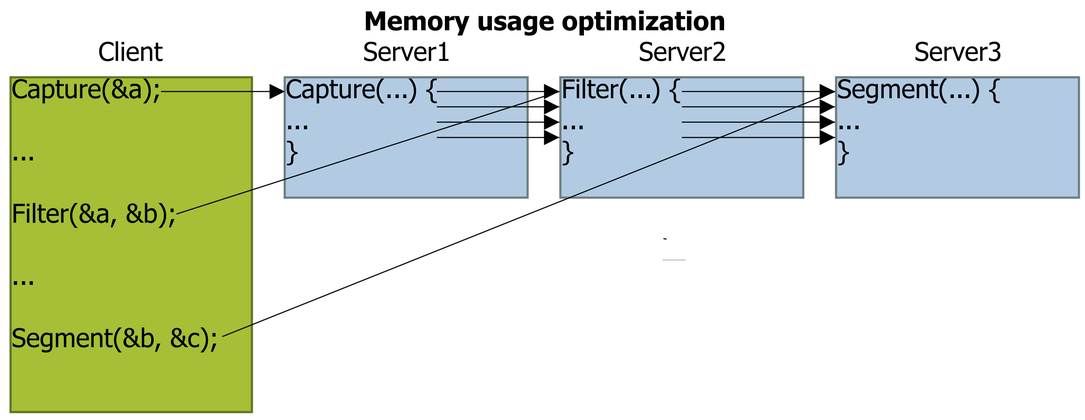

*The Philips CFT Inca+ smart camera contains a XeTaL 320-PE SIMD processor as well as a TriMedia 5-issue VLIW processor. Pixel-level operations are run on the SIMD, while more irregular computations are carried out on the VLIW.*

## Task parallelism: asynchronous RPC

In RPC, a client program calls stubs which signal a server to perform the actual computation. In our case, the application is the client program running on a control processor, while the instantiated skeleton operations are run on the coprocessors. This alone does not imply parallelism, because the stub waits for the results of the server before returning. In asynchronous RPC, therefore, the stub returns immediately, and the client has to block on a certain operation before accessing the result. This allows the client program to run concurrent to the server program, as well as multiple server programs to run in parallel.

We provide two optimizations to this basic paradigm: data distribution and memory usage optimization. The data distribution optimization avoids a scatter-gather bottleneck by using **futures** to pass data between server processes instead of gathering it to the client, while the memory usage optimization uses **pipelining** to enable processors that cannot store an entire image to participate in the framework.



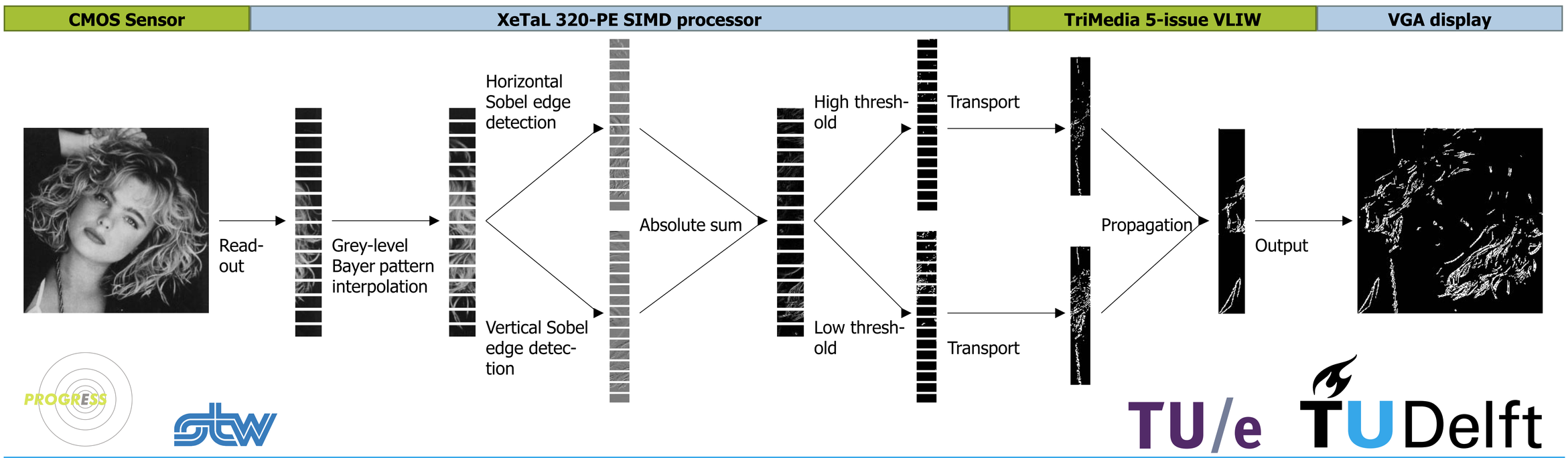




## Example: double thresholding edge detection on the Philips CFT Inca+ smart camera architecture

As an example application, we have implemented a simple double thresholding edge detection algorithm using our framework. This consists of an interpolation, grey-level edge detection, binarization at two levels, and a propagation of the high threshold over the low threshold. The propagation cannot run on the XeTaL processor, because it requires a frame memory, but the other steps can be pipelined.

The Bayer pattern interpolator, being a 2x2 filter, requires a 2-line input buffer. The horizontal and vertical Sobel operators share the same 3-line input buffer, while they output to a 1-line buffer for the absolute sum. The two thresholds again share a 1-line buffer, and their outputs are routed to the TriMedia. Here, the propagation step has a frame memory as internal state, and it outputs to a VGA display.

In summary, the XeTaL requires only 10 line memories for the 6 operations that are scheduled to it, while the user doesn't have to write the program in a pipelined way: that is handled by the skeletons and RPC framework. Also, the data is only forwarded to the TriMedia if it is necessary for an operation there.




*) Wouter Caarls, Quantitative Imaging, Delft University of Technology
Pieter Jonker, Quantitative Imaging, Delft University of Technology
Henk Corporaal, Information and Communication Systems, Eindhoven University of Technology
Delft University of Technology

PROGRESS project number EES.5411. http://www.qi.tnw.tudelft.nl/~wcaarls/smartcam