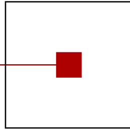

s c c h
software competence center
hagenberg

Advances in Integrated Software Sciences

Proceedings of the Master and PhD Seminar Summer term 2011, part 1

Softwarepark Hagenberg
SCCH, Room 0/2
15 April 2011

Software Competence Center Hagenberg
Softwarepark 21
A-4232 Hagenberg
Tel. +43 7236 3343 800
Fax +43 7236 3343 888
www.scch.at

Fuzzy Logic Laboratorium Linz
Softwarepark 21
A-4232 Hagenberg
Tel. +43 7236 3343 431
Fax +43 7236 3343 434
www.flll.jku.at

Program

Session 1. Chair: Bernhard Moser

9:00	Volkmar Wieser: Combining High Productivity and High Performance in Image Processing Using Single Assignment C
9:30	Dominic Looser: Semantics — Seeking the Big Picture. Part 1.

Combining High Productivity and High Performance in Image Processing Using Single Assignment C

Volkmar Wieser[a] and Bernhard Moser[a] and Sven-Bodo Scholz[b] and Stephan Herhut[b] and Jing Guo[b]

[a]Software Competence Center Hagenberg, Software Park 21, Hagenberg, Austria
[b]University of Hertfordshire, College Lane, Hatfield, AL10 9AB, UK

ABSTRACT

In this paper the problem of high performance software engineering is addressed in the context of image processing regarding productivity and optimized exploitation of hardware resources. Therefore, we introduce the functional array processing language Single Assignment C (SaC), which relies on a hardware virtualization concept for automated, parallel machine code generation. An illustrative benchmarking example proves both utility and adequacy of SaC for image processing.

Keywords: Functional Programming, Parallel Programming, Hardware Virtualization, Anisotropic Diffusion

1. INTRODUCTION

The landscape of parallel computing has substantially changed in the last years. It is not only obvious that “the future is parallel” but also current trends confirm that computing power through parallelism will be provided by many-core architectures.[1] General-purpose many-core architectures must conveniently support a wide range of programming styles and languages. If architectures prefer a particular model of parallel programming, they are not likely to become widely accepted, especially if such architectures require programming skills that probably overstrain the average programmer. Furthermore, the development of high performance applications on novel and ever-changing hardware environments like multi- and many-core systems, Graphics Processing Units (GPUs) or Field Programmable Gate Arrays (FPGAs) is cost- and time-intensive. What is needed is a convenient abstract language that supports automatic parallelization on different architectures without changing source code and robust performance benefits. Software engineers neither should need to design and develop in unintuitive ways nor to deal with a variety of hardware and language details just to avoid design mistakes or bottlenecks or just to achieve an attractive speedup. From the economical point of view, an approach that yields the desired performance with minimal effort will be preferred, particularly for real-time performance applications in industry as well as for less efficient hardware.

In the field of quality inspection of textured surfaces, e.g. for the production of foils or industrial woven fabrics, we have to cope with a scanning speed up to 300m/min, i.e. about 80MB/sec per camera systems, and a complex phenomenology of textures and defects. This requires the application of advanced cost-intensive algorithms of image processing as well as machine learning, the use of high-performance computational hardware like GPUs or multi-core systems and the exploitation of parallelization potentials. The analysis of the whole processing pipeline (image acquisition, preprocessing, feature extraction, registration, defect detection and classification) with standard languages regarding performance is a resource- and time-intensive challenge.

While image acquisition as well as major parts of preprocessing and feature extraction can be computed in a well-predictable way due to the pre-fixed size of filter operations and amount of data transfer known in advance, this is usually not the case for the high-level processing steps of pattern recognition and classification. With increasing semantics the processing steps involve operations of increasing complexity and computational costs. For example at the beginning of the processing pipe a typical scenario is to use local filter operations acting e.g. on 9x9 matrices while later on also global operations on the whole image data are applied like a registration with

Further author information: (Send correspondence to Volkmar Wieser)
Volkmar Wieser: E-mail: volkmar.wieser@scch.at, Telephone: +43 7236 3343 844

a reference model based on thousands of feature points. So far the processing steps are acting on a physical, appearance-based level which only depends on the image intensity values. Finally, defect candidates have to be identified, located and classified. This final step heavily depends on parameters that are not coded within the image, e.g. the customer's judgment whether some product quality aspects can be accepted or have to be rejected. The complexity of the classification step correlates with the complexity of the defect taxonomy. For industrial textile fabrics there are about 50 major categories of defects like wrong webbing, swelling, tension of yarn, surface wrinkling or staining problems, which require appropriate and sophisticated strategies for defect classification.

Therefore, from the economical point of view the main question is how to find the optimal tradeoff between performance and economical resources. Single Assignment C (SaC)[2] provides a design scope which definitely reduces the period for design and development and hence saves resources by increasing productivity. In this paper we demonstrate this benefit on the basis of an application-oriented example (Perona-Malik Anisotropic Diffusion[1]) which has poor performance characteristics by default. First, we introduce the functional array language SaC in Section 2. Afterwards, we compare the SaC optimization strategies against those of the opencv2.0[3] in Section 3. Finally, we illustrate benchmarking results of both implementations on a many-core system in Section 4.

2. SINGLE ASSIGNMENT C

SaC is a purely functional and data-parallel programming language. All its basic language constructs are identical to those of C, not only with respect to their syntax but also with respect to their semantics. Despite this rather imperative look and feel, a side-effect free semantics is enforced by the exclusion of a few features of C, most notably the absence of pointers. As a replacement, extensive support for compiler-managed n-dimensional arrays as well as a very powerful language construct for expressing data-parallel operations called with-loop have been added. Based on this language construct, various MATLAB-like operations have been defined. Similar to the STL of C++, these operations are provided within a library and come as a part of the SaC disrtribution*.

Various application studies demonstrate that this setting enables (i) sequential runtimes competitive with those of hand-written C and Fortran codes, and (ii) almost linear speedups from auto-parallelization for shared memory systems.[4]

A first introduction of SaC is presented in "Single Assignment C - efficient support for high-level array operations in a functional setting".[5] This paper discusses the general approach for integrating arrays with access time $\mathcal{O}(1)$ into functional languages as well as the shape-invariant language construct called With-loop. Finally, a runtime comparison between SaC and High Performance Fortran (HPF)[6] implementation is done. Further experiments with respect to runtime performance are done in "SaC - from high-level programming with arrays to efficient parallel execution".[2] Beside a general introduction this paper demonstrate the major steps in compiling rank- and shape-invariant high-level SaC specifications into efficiently, executable multithreaded code. The measurement of runtime performance is taken in SaC , HPF[6] and ZPL[7] (A Machine Independent Programming Language for Parallel Computers). More detailed introductions of SaC can be found in further readings (e.g. design[8] and compilation[2] aspects, shape-invariant programming facilities,[9] accelerating APL† programs,[10] generic programs on arrays,[11] and with-loop-fusion[12]).

At present, in the scope of the EU-funded project advance,[13] further benchmarks with industrial applications will transfer to SaC and the stream processing language S-Net.[14] These applications are benchmarked regarding throughput, scalability, and runtime performance on Unibench,[15] where first results are available. To demonstrate the combination of SaC's array support with standard C code, we present a small example (see Algorithm 1).

Algorithm 1 shows a program fragment that is typical for many scientific codes. It contains the definition of a function `relax` which takes two arguments: a scalar double `eps` and a two-dimensional array (matrix) of double elements `a`. In the body of the function the array `a` is repeatedly modified within a `do`-loop. Each modification is expressed by means of the with-loop in lines 5-10. It specifies that all inner elements of the new version of `a` are

*available from www.sac-home.org
†APL (A Programming Language) is an interactive array-oriented language

Algorithm 1 Program fragment for combination of SaC and C code

```
double[.,.] relax( double eps, double[.,.] a){
 1: do{
 2:    old_a = a;
 3:    a = with {
 4:       ([1, 1] <= iv < shape(a) − 1){
 5:       res = 0.25 ∗ (a[iv − [0, 1]] + a[iv + [0, 1]]
 6:                     a[iv − [1, 0]] + a[iv + [1, 0]]);
 7:       } : res;
 8:    }:modarray(a);
 9: }while (maxval(abs(a − old_a)) > eps);
10: return(a);
 }
```

computed as a mean of the four direct neighbor elements of the old version of `a`. Note that indexing into arrays in SaC is done by means of index vectors rather than sequences of scalar indices. The function `shape` used in line 6 returns a vector containing the number of elements of the given array; the subtraction of `1` is applied to all elements of that vector, indexing starts at `0`. As a consequence, all indices between `[1,1]` (inclusive) and `shape(a)-1` (exclusive) denote exactly all inner elements of `a`. Furthermore, it is important to note that the semantics of the WITH-loop ensures that all array modifications, conceptually, happen at the same time, i.e., they can be executed in parallel.

In line 12 we see the termination condition of the `do`-loop. As for the description of the indexing range in line 6, we make extensive use of the array operations from the SaC standard library. The expression (a - old_a) denotes a two-dimensional array containing all the differences between the old version of `a` and the current one. The function `abs` computes their absolute values and `maxval` determines their maximum value. Only if this value exceeds a given epsilon `eps`, computation proceeds.

Note here that the nature of the termination condition always requires two generations of the array `a` (`a` and old_a) to be available. This is achieved by the assignment in line 4. It conceptually creates a copy of the current array `a`. However, the compiler/runtime system of SaC ensures that these copies in fact do not materialize at runtime. More details on the technology that facilitates this kind of optimization can be found in.

3. A BENCHMARKING EXPERIMENT

In this paper we claim that SaC allows high-productivity and high-performance. In order to underpin this claim we want to work out an example by showing its style of implementations and performance behavior in comparison with an OpenCV implementation. As example we choose the anisotropic diffusion filter operation of Perona-Malik[1] which is widely-used within the image processing community. The execution time of the filter only depends on the dimension of the input image and not on the content. This characteristic allows an objective analysis of the scalability on multi/many-core architectures as well as a good demonstration of the automatic data-parallelism feature of SaC.

3.1 The Algorithmic Background of Perona-Malik Anisotropic Diffusion

First, the idea of Perona-Malik anisotropic diffusion is to locally steer the diffusion in a specific direction by introducing $D(.,.)$

$$\frac{\partial}{\partial t}\varphi = div(D\Delta\varphi), \tag{1}$$

with boundary condition

$$\varphi(.,.,0) = \varphi, \tag{2}$$

where D depends on the local derivatives. Perona-Malik propose two different derivates

$$D = 1/(1 + c\,\|\Delta\varphi\|^2), \tag{3}$$

and

$$D = exp(-\|\Delta\varphi\|/c), \tag{4}$$

where Equation (3) acts like a smoothing filter that suppresses fine (noisy) structures, while Equation (4) strengthens high contrast edges. For an illustration, see Figure 1 to Figure 3.

Algorithm 2 outlines a pseudo-code for an implementation of Perona-Malik anisotropic diffusion, which contains the degree of smoothing by the number of iteration steps in the for-loop (line 3 till 14). This means that the result of the first iteration is the input of the next iteration and so on, which limits the scope of parallelization.

Algorithm 2 Pseudo code for Perona-Malik anisotropic diffusion

1: $diffl \leftarrow$ create $px1$ border around image
2: $diff \leftarrow$ image
3: **for** $i = 1$ to NITER **do**
4: shift(diffl, $px1$ to North, South, East and West)
5: $delta \leftarrow$ substract(diff from shifted image)
6: **if** $type == 1$ **then**
7: $c = exp(-(delta/kappa))$
8: **else**
9: $c = 1/(1 + kappa * delta^2)$
10: **end if**
11: $c \leftarrow c * delta$
12: $c \leftarrow add(allshiftedimages) * LAMBDA$
13: $diff \leftarrow diff + c$
14: **end for**
15: $result \leftarrow diff$

Figure 2 illustrates the use of the deviation in Equation (3), where we can see that only the connected wide regions are left, whereas noise structure is largely removed. The use of the deviation in Equation (4) in Figure 3 shows us that beside the big deep scratch in the middle also fine, noisy, high contrast edges are left.

Figure 1. stainless steel plate with noise surface and scratch

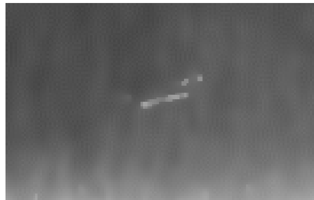

Figure 2. Application of Equation (3) on image of Figure 1; NITER=5; $delta = 1/3$; kappa=10;

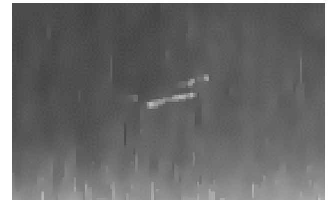

Figure 3. Application of Equation (4) on image of Figure 1; NITER=5; $delta = 1/3$; kappa=10;

In Chapter 4 we evaluate the execution time and the scalability of the anisotropic diffusion, which is implemented as shown in Appendix A.1 for OpenCV and Appendix A.2 for SaC.

4. BENCHMARKING RESULTS

For our test scenario we use three different hardware environments, i.e., a DELL Precision™ 690 for CPU performance tests, a NVIDIA GeForce GTX 480 graphic card for GPU computing, and a benchmarking environment named UNIBENCH[15] to demonstrate scaleability. The Dell Precision™ 690 has two seperate Intel®Xeon®5060 with 3.2GHz, i.e., 8 cores in total, and 2GB full buffered DDR2 memory. The NVIDIA GeForce GTX 480 has a core frequency of 1.4 GHz and in total 480 cores, i.e., the graphic card has a bandwith of about 177.4 GB per second. UNIBENCH has a Symmetric Multiprocessing (SMP) architecture, where four AMD Opteron™ 6174 processors are connected to a single shared main memory. Each processor has 12 cores and a clock speed of 2.2 GHz, i.e., that are in total 48 cores. The advantage of SMP machines is the large amount of physical memory of about 256 GB, visible to all 48 cores, which allows multi-threaded applications. The input data of the anisotropic filter range from 256×256 pixels to 4096×4096 pixels with pseudo-randomly generated 8-bit values between 0 and 255 since in this example only the dimension of the data affects the execution time.

Therefore, in Table 1 we present five different input sizes and propagate for each of them Equation (3) ten times. In comparison to OpenCV, with the input size of 4096×4096 pixels, using the multithreaded SAC-MT compilation we can reduce the execution time of the filter and, consequently, the execution time of the application on 8 cores by 27.72 seconds. This reduction is realized with a similar effort of development. This improvement can be a time buffer for using more complex algorithms or having a significant competitive advantage against other applications.

	$px256 \times 256$	$px512 \times 512$	$px1024 \times 1024$	$px2048 \times 2048$	$px4096 \times 4096$
OpenCV	0.09 sec	0.51 sec	2.31 sec	9.23 sec	37.43 sec
SAC-SEQ	0.16 sec	0.66 sec	2.67 sec	10.65 sec	42.75 sec
SAC-MT	0.15 sec	0.21 sec	0.69 sec	2.41 sec	9.71 sec
OpenCV vs. SAC-MT	0.6×	2.4×	3.3×	3.8×	3.8×

Table 1. Comparison of OpenCV and SAC implementation of anisotropic filter using DELL Precision™ 690

The reason of the performance gain of SAC is the optimal processor load on all 8 cores and the functional programming paradigm, which results in a low memory usage of about 350 MB on average. The standard OpenCV implementation does not support the distribution on the available cores. Therefore, the OpenCV anisotropic filter is executed only on one core and needs many times more memory of about 1.9 GB on average due to data materialization. For further analysis of SAC we restricted the execution of the anisotropic filter to a single-core. As we can see in Table 1, the sequential execution time of SAC-SEQ is close to OpenCV but with slightly higher communication overhead.

If we need more performance for the application scenario, we have the possibility either to re-implement the whole algorithm with NVIDIAs CUDA framework or automatically generate executable GPU code with the SAC-CUDA backend. A re-implementation definitely requires higher development costs and programming know-how from experts, where SAC-CUDA allows a flexible time and cost efficient development. For the moment being, it is not possible to outperform a hand coded implementation by means of SAC-CUDA. While for the automated SAC-CUDA code we achieve a speedup about 40× (with respect to OpenCV implementation with input size of $px4096 \times 4096$), a manually optimized CUDA implementation allows a speedup of 197× for the same example. However, note that SAC-CUDA is still under development, which in the near future will bring further improvements and speedups by means of SAC-CUDA.

	$px256 \times 256$	$px512 \times 512$	$px1024 \times 1024$	$px2048 \times 2048$	$px4096 \times 4096$
SAC-CUDA	0.01 sec	0.02 sec	0.06 sec	0.24 sec	0.94 sec
CUDA-hand coded	0.005 sec	0.007 sec	0.016 sec	0.05 sec	0.19 sec
OpenCV vs. SAC-CUDA	9×	25.5×	38.5×	38.5×	40×
OpenCV vs. CUDA-hand	18×	72×	144×	184×	197×

Table 2. Comparison of hand-coded CUDA code and SAC-CUDA produced code on NVIDIA GeForce GTX 480

To analyse the scalability of SAC compiled code, we can see in Table 3 that the scalability of the anisotropic diffusion using the SAC implementation is nearly linear by increasing the number of cores.

input size	$px2048 \times 2048$	$px2048 \times 2048$	$px2048 \times 2048$	$px2048 \times 2048$
number of cores	2	8	24	48
speedup	2×	8×	23×	42×

Table 3. Scalability of SaC code using unibench

It is interesting to observe that the OpenCV and SaC implementations have a similar programming style. Particularly, the handling of matrix operations is comparably comfortable in both languages.

Concerning the optimization strategies, we can say that OpenCV offers an optimized Streaming SIMD Extensions 2 (SSE2) code. SSE2 is a processor supplementary instruction set for modern 32-bit x86 and 64-bit x64 Single Instruction, Multiply Data (SIMD) architectures, where many of the basic arithmetic functions can run significantly faster. OpenCV also contains Intel®Threading Building Blocks (TBB)[16] support for several functions. TBB is a C++ template library which offers a complete threading mechanism on modern multi-core processors. The advantages of this library are easy and efficient handling (software engineers do not need to be threading experts), scalable performance and a higher-level, task-based parallelism. In our example TBB is irrelevant as we do not use OpenCV functions that support this library. OpenCV applies the TBB only to OpenCV applications, e.g., haartraining, traincascade, and not to basic arithmetic/filter operations.

The optimization strategy of SaC is different. One of the major design principles of SaC is the With-loop construct, which supports the specification of shape-invariant array operations. All primitive array operations of SaC can be defined as with-loops within a standard library rather than being implemented as part of the compiler. The basic idea is to use with-loops as a universal representation for array operations and to develop a general transformation scheme that allows the transforming of two consecutive with-loops into a single one. Using this mechanism called with-loop-folding,[17] any nesting of primitive array operations can be stepwise transformed into a single loop construct that contains an element-wise specification of the resulting array. During the compilation process of the SaC-compiler various conventional optimization techniques,[18] such as function inlining, constant folding, constant propagation, loop unrolling, and dead code removal, are applied to produce efficiently executable, multithreaded C code with Posix™ threads.[19]

5. CONCLUSION

In this paper we showed the advantage of the functional array language Single Assignment C (SaC) in the field of image processing, especially for the anisotropic diffusion filter. Therefore, we conducted a benchmarking experiment in which we compared the effort of the development using the common image library opencv2.0. Compared to the language syntax, SaC is similar to matlab because of the definition of various matlab-like operations. Furthermore, the period of development can be reduced by the well-known C/C++ semantics, however, by applying a side-effect free semantics most notably due to the absence of pointers and hardware virtualization. The hardware virtualization allows flexible and fast development on architectures corresponding to CPUs, GPUs or FPGAs using the same language and the same implementation. Moreover, on multi/many-core architectures we achieve a nearly linear speedup due to the auto-parallelization of SaC. This is indispensable for inline quality inspection applications due to a scanning speed up to 300m/min and about 80MB/sec per camera systems, and the complex phenomenology of textures and defects. Additionally, from the economical point of view, SaC provides us with an extremely good balance between time of development and performance.

Although SaC is well suited for image processing because of the data-parallelism and the n-dimensional array support, there does not exist an image library like opencv. In addition, during the development of SaC applications the support of debugging tools is limited. This will be changed in the future by an intensive enhancement of SaC and community building.

REFERENCES

1. P. Perona and J. Malik, "Scale-space and edge detection using anisotropic diffusion," *IEEE Transactions on Pattern Analysis and Machine Intelligence* **12**, pp. 629–639, 1990.
2. C. Grelck and S. Scholz, "SaC – from High-Level Programming with Arrays to Efficient Parallel Execution," *Parallel Processing Letters* **13**(3), pp. 401–412, 2003.
3. G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools* , 2000.
4. D. Rolls, C. Joslin, A. Kudryavtsev, S.-B. Scholz, and A. V. Shafarenko, "Numerical simulations of unsteady shock wave interactions using sac and fortran-90," in *PaCT*, pp. 445–456, 2009.
5. S.-B. Scholz, "Single Assignment C — efficient support for high-level array operations in a functional setting," *Journal of Functional Programming* **13**(6), pp. 1005–1059, 2003.
6. J. M. Department and J. Merlin, "High performance fortran," 1997.
7. B. L. Chamberlain, S.-E. Choi, E. C. Lewis, C. Lin, L. Snyder, and W. D. Weathersby, "Zpl: A machine independent programming language for parallel computers," *IEEE Transactions on Software Engineering* **26**, pp. 197–211, 2000.
8. C. Grelck and S.-B. Scholz, "A Functional Array Language for Efficient Multithreaded Execution," *International Journal of Parallel Programming* , 2006. to appear.
9. S.-B. Scholz, "On Defining Application-Specific High-Level Operations by Means of Shape-Invariant Programming Facilities," in *Proceedings of the Array Processing Language Conference 98*, S. Picchi and M. Micocci, eds., pp. 40–45, ACM-SIGAPL, 1998.
10. C. Grelck and S. Scholz, "Accelerating APL Programs with SAC," in *Proceedings of the Array Processing Language Conference 99*, O. Lefevre, ed., pp. 50–57, ACM-SIGAPL, 1999.
11. C. Grelck, S.-B. Scholz, and A. Shafarenko, "A Binding-Scope Analysis for Generic Programs on Arrays," in *Implementation and Application of Functional Languages, 17th INternational Workshop, IFL'05*, A. Butterfield, ed., *LNCS*, Springer, 2006. to appear.
12. C. Grelck, K. Hinkfu, and S.-B. Scholz, "With-Loop Fusion for Data Locality and Parallelism," in *Implementation and Application of Functional Languages, 17th INternational Workshop, IFL'05, Selected Papers*, A. Butterfield, ed., *LNCS*, Springer, 2006. to appear.
13. SAC team, "EU Project ADVANCE," *http://www.project-advance.eu* , last visited 15th November 2010.
14. A. Shafarenko, S.-B. Scholz, and C. Grelck, "Streaming networks for coordinating data-parallel programs," in *Proceedings of the 6th International Andrei Ershov Memorial Conference on Perspectives of System Informatics (PSI'06), Novosibirsk, Russia*, I. Virbitskaite and A. Voronkov, eds., pp. 274–276, A.P. Ershov Institute of Informatics Systems 2006, Novosisibirsk, Russia, 2006.
15. UNIBENCH team, "Benchmarking System UNIBENCH," *http://unibench.sac-home.org* , last visited 15th November 2010.
16. J. Reinders, *Intel threading building blocks*, O'Reilly & Associates, Inc., Sebastopol, CA, USA, first ed., 2007.
17. S. bodo Scholz, "With-loop-folding in sac condensing consecutive array operations," in *Proceedings of the 9th International Workshop on Implementation of Functional Languages (IFL97*, pp. 72–92, Springer-Verlag, 1998.
18. K. Kennedy and J. R. Allen, *Optimizing compilers for modern architectures: a dependence-based approach*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.
19. "IEEE Standard for Information Technology - Portable Operating System Interface (POSIX). Base Definitions," *IEEE Std 1003.1, 2004 Edition. The Open Group Technical Standard Base Specifications, Issue 6. Includes IEEE Std 1003.1-2001, IEEE Std 1003.1-2001/Cor 1-2002 and IEEE Std 1003.1-2001/Cor 2-2004. Base* , p. 0 1, 2004.

APPENDIX A. IMPLEMENTATION DETAILS OF ANISOTROPIC DIFFUSION

A.1 "OpenCV 2.0" implementation

```
void executeAnisotropiefilter(const cv::Mat& image, cv::Mat& result, int NITER, int type)
{
    /*allocate memory*/
    cv::Mat diffl(img.rows+2, img.cols+2, CV_64FC1);
    ...

    /*convert image to double*/
    image.convertTo(diff, diff.type(), 1, 0);

    /*select region of interest*/
    cv::Rect rectl(1, 1, diff.cols, diff.rows);
    cv::Rect rectN(1, 0, diff.cols, diff.rows);
    cv::Rect rectS(1, 2, diff.cols, diff.rows);
    cv::Rect rectE(2, 1, diff.cols, diff.rows);
    cv::Rect rectW(0, 1, diff.cols, diff.rows);

    for(int i=0; i<NITER; i++)
    {
      /*select roi*/
      cv::Mat diffl_roi = diffl(rectl); diff.copyTo(diffl_roi);

      cv::Mat diffl_roiN = diffl(rectN);
      cv::Mat diffl_roiS = diffl(rectS);
      cv::Mat diffl_roiE = diffl(rectE);
      cv::Mat diffl_roiW = diffl(rectW);

      deltaN = diffl_roiN - diff;
      deltaS = diffl_roiS - diff;
      deltaE = diffl_roiE - diff;
      deltaW = diffl_roiW - diff;

      conduction(cN, deltaN, type);
      conduction(cS, deltaS, type);
      conduction(cE, deltaE, type);
      conduction(cW, deltaW, type);

      cN = cN.mul(deltaN);
      cS = cS.mul(deltaS);
      cE = cE.mul(deltaE);
      cW = cW.mul(deltaW);

      diff = diff + (cN + cS + cE + cW) * LAMBDA;
    }

    /*convert image back to int*/
    diff.convertTo(result, image.type(), 1, 0);
}

inline
void conduction(cv::Mat& c, cv::Mat& delta, int type)
{
  if(type == 1){
    c = delta / KAPPA;
    c = c.mul(c, -1);
    cv::exp(c, c);
  }else{
    c = pow(delta, 2.0);
    c = c.mul(KAPPA);
    cv::add(c, 1, c);
    c = 1 / c;
  }
}
```

Figure 4. OPENCV 2.0 implementation of Perona-Malik Anisotropic filter

A.2 "Single Assignment C" implementation

```
int[.,.] executeAnisotropicFilter(int[.,.] image, int niter, int type)
{
        /*convert image to double*/
        result = tod(image);

        for (i=0; i<niter; i++)
        {
                imageN = shift( 0,-1, 0.0, result);
                imageS = shift( 0, 1, 0.0, result);
                imageE = shift( 1,-1, 0.0, result);
                imageW = shift( 1, 1, 0.0, result);

                deltaN = result - imageN;
                deltaS = result - imageS;
                deltaE = result - imageE;
                deltaW = result - imageW;

                cN = conduction(deltaN, kappa, type);
                cS = conduction(deltaS, kappa, type);
                cE = conduction(deltaE, kappa, type);
                cW = conduction(deltaW, kappa, type);

                mN = cN * deltaN;
                mS = cS * deltaS;
                mE = cE * deltaE;
                mW = cW * deltaW;

                result = result + (lambda*(mN + mS + mE + mW));
        }

        /*convert image back to int and return*/
        return( toi(result));
}

inline
double[.,.] conduction(  double[.,.] delta, double kappa, int type)
{
        /*privileges high-contrast edges over low-contrast ones.*/
        if (type == 1)
                c = exp(-(delta / kappa));

        /*privileges wide regions over smaller ones.*/
        else
                c = 1.0 / (1.0 + pow((delta / kappa),2.0));

        return( c );
}
```

Figure 5. SaC implementation of Perona-Malik Anisotropic filter

Semantics: Seeking the Big Picture. Part 1

Dominic Looser

Abstract

There is a recent surge of use of the term *semantic* in computer science. This is due to developments that can be subsumed under the notion of the semantic web. But the field of semantics also has a rich history and is highly connected to various scientific disciplines. Its study quickly leads to general questions concerning the relations of humans, mind, signs, and reality. This series of talks seeks to approximate the big picture of semantics, leading from semiotics to the semantic web.