Harris Corner Detection on a NUMA Manycore

Claude Tadonki, Olfa Haggui, Lionel Lacassagne

Harris Corner Detection on a NUMA Manycore

Claude Tadonki[1], Olfa Haggui[2], and Lionel Lacassagne[3]

[1] Mines ParisTech - PSL Research University ,
Centre de Recherche en Informatique (CRI),
35, rue Saint-Honoré, 77305, Fontainebleau Cedex (France),
claude.tadonki@mines-paristech.fr
[2] Sousse National School of Engineering(Tunisia),
olfa.haggui@gmail.com
[3] University Pierre and Marie Curie (UPMC),
Laboratoire d'Informatique de Paris 6 (LIP6), SoC department / ALSoC team,
lionel.lacassagne@lip6.fr

September 24, 2017

Abstract

Corner detection is a key kernel for many image processing procedures including *pattern recognition* and *motion detection*. The latter, for instance, mainly relies on the corner points for which spatial analyses are performed, typically on (probably live) videos or temporal flows of images. Thus, highly efficient *corner detection* is essential to meet the real-time requirement of associated applications. In this paper, we consider the corner detection algorithm proposed by Harris, whose the main work-flow is a composition of basic operators represented by their approximations using 3×3 matrices. The corresponding data access patterns follow a stencil model, which is known to require careful memory organization and management. Cache misses and other additional hindering factors with NUMA architectures need to be skillfully addressed in order to reach an efficient scalable implementation. In addition, with an increasingly wide vector registers, an efficient SIMD version should be designed and explicitly implemented. In this paper, we study a direct and explicit implementation of common and novel optimization strategies, and provide a NUMA-aware parallelization. Experimental results on a dual-socket INTEL Bradwell-E/EP show a noticeably good scalability performance.

keywords *corner detection, Harris algorithm, multicore, multithread, NUMA, scalability*

1 Introduction

The common characteristic of image processing algorithms is the heavy use of convolution kernels. Indeed, the typical scheme is an iterative application of a stencil calculation at the pixel level. This yields non-local and unaligned memory accesses, thus making it hard to achieve a real-time performance implementation.

Harris corner (and edge) detection [27, 26] is an important kernel in image processing, especially for motion detection and object recognition/detection/tracking [28, 29]. Roughly speaking, the procedure is a serial combination of 3×3 filters (*derivatives* and *gaussians*), surrounded by basic arithmetic and selection operations. This leads to a *stencil computation* which exposes two correlated challenges concerning *memory accesses* and *redundant computation*. Since this kernel is likely to be called intensively on image processing applications, which includes the embedded context, fastest (at least real-time) implementations are crucial, hopefully on various hardware targets.

A thorough implementation study is provided by Lacassagne et al. in [25], where some of the basic ideas considered in this paper are mentioned, especially *arithmetic operations optimization* using the separability property of the filters, *computation reduction*, *memory accesses optimization* through continuous data reuses, *loop collapsing* [33] of the operators in order to reduce the lifetime

of intermediate data , and *array contraction* [30, 31, 32] which takes advantage of the shorter lifetime of intermediate variables to consider a compact storage through memory location reassignments. The SIMD part is left to the compiler (the native one for each considered architecture), and shared memory parallelism is implemented with OpenMP (through classical directives. Other studies of Harris corner detection and its applications can be found in [35, 36, 37, 38, 39].

In this paper, we follows the aforementioned basic ideas, but directly carried on in the SIMD context. Indeed, we consider $(i-1, j-1) \leftarrow (i, j)$ reindexation (as Kung-Lo-Lewis in [34] for the algebraic path problem) for each convolution and thereby obtain a perfectly aligned vector loads and stores at all levels of the loops. *Data reuse*, *factorization of the computations* and *optimal vector loads* are achieved through skillful registers shuffling and pipelining.

Regarding parallel implementation, we focus on the shared memory paradigm and basically apply a row-strip distribution of both input and output memory spaces among the working threads, each of them having its private space for intermediate values. We use Pthreads in order to have a full control of tasks allocation, data partitioning and thread/core association (we prevent thread migration). The main reason of this is that we are running on a NUMA architecture where any unaware memory scenario can incur a severe penalty [22, 23, 24]. We explicitly consider on-node memory allocations together with threads binding routines to implement an appropriate static partitioning, which minimizes remote accesses and internode bus (QPI) contention.

The rest of the paper is organized as follows. The next section (section 2) provides a fundamental description of the Harris corner detection procedure, followed by an overview of the related work in section 3. Section 4 introduces NUMA architectures and outlines the main related concerns. Our SIMD and SMP optimization strategies are described in section 5. Performance resultst are presented and commented in section 7. Section 8 concludes the paper.

2 The Harris Algorithm for Corner Detection

Harris and Stephen [2] interest point detection algorithm is an improved variant of the *Moravec corner detector* [1], used in computer vision for *feature extraction, motion detection, image matching, tracking, 3D reconstruction* and *object recognition*. An overview of the foundation of the algorithm, as described in [37] and similarly restated here, can be formulated as follows.

Let $I(x, y)$ denote the intensity of a pixel location (x, y) of the image, and λ a given threshold.

1. For each pixel (x, y) in the input image, compute the *Harris matrix* $G = \begin{pmatrix} g_{xx} & g_{xy} \\ g_{xy} & g_{yy} \end{pmatrix}$, with
$$g_{xx} = (\frac{\partial I}{\partial x})^2 \otimes w \qquad g_{xy} = (\frac{\partial I}{\partial x}\frac{\partial I}{\partial y}) \otimes w \qquad g_{yy} = (\frac{\partial I}{\partial y})^2 \otimes w, \tag{1}$$
where $\otimes$ denotes convolution operator and w is the Gaussian filter.

2. For each pixel (x, y), compute the Harris criterion given by
$$c(x, y) = det(G) - k(trace(G))^2, \tag{2}$$
where $det(G) = g_{xx}.g_{yy} - g_{xy}^2$, k an empirical constant, and $trace(G) = g_{xx} + g_{yy}$.

3. Set all $c(x, y)$ which are below λ to zero.

4. Extract points (x, y) having the maximum $c(x, y)$ within a window neighborhood. These points represent the corners.

Figure 1 illustrates the result of the algorithm for the corner detection case.

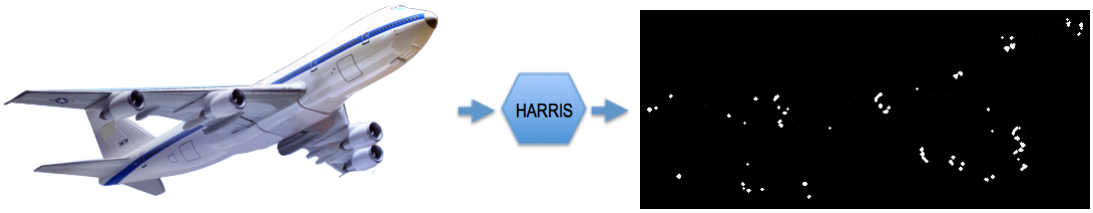

Figure 1: Illustration of the Harris-Stephens corner detection

The algorithm is mainly a successive application of convolution kernels that globally implement a discrete form of an autocorrelation S, given by

$$S(x,y) = \sum_{u,v} w(u,v)[I(x,y) - I(x-u, y-v)]^2, \tag{3}$$

where (x,y) is the location and $I(x,y)$ its intensity, and $u, v \in 1,2,3$ the components modeling the move on each dimension. At a given point (x,y) of the image, the value of $S(x,y)$ is compared to a suitable *threshold* in order to determine the nature of the corresponding pixel. Roughly speaking, the process is achieved by applying four discrete operators, namely *Sobel* (S), *Multiplication* (M), *Gauss* (G), and *Coarsity* (C). Figure 2 displays an overview of the global workflow.

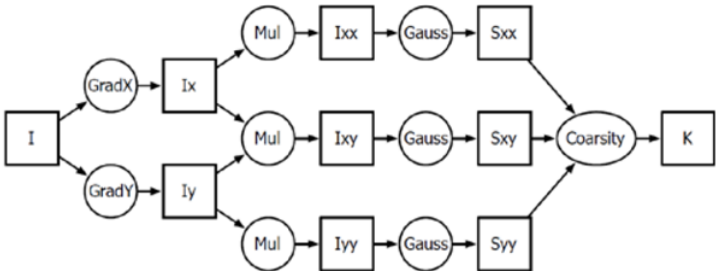

Figure 2: Harris detector workflow

Multiplication and *Coarsity* are point to point operators, while *Sobel* and *Gauss*, which approximate the first and second derivatives, are $9 \to 1$ or 3×3 operators defined by

$$S_x = \frac{1}{8}\begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} \quad S_y = \frac{1}{8}\begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix} \tag{4}$$

$$G = \frac{1}{16}\begin{pmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{pmatrix} \tag{5}$$

Applying a 3×3 operator to a given pixel (x,y) consists in a point-to-point multiplication of the corresponding 3×3 matrix by the following pixels matrix

$$\begin{pmatrix} I(x-1,y+1) & I(x,y+1) & I(x+1,y+1) \\ I(x-1,y) & I(x,y) & I(x+1,y) \\ I(x-1,y-1) & I(x,y-1) & I(x+1,y-1) \end{pmatrix} \tag{6}$$

Here comes the notion of *border*. In order to compute the output $H(x,y)$ for the pixel (x,y), we need its intensity $I(x,y)$ and those of its neighborhood. We say the operator is of *depth* 1. Operator depth is additive, means that if two operators f and g are of depth p and q respectively, then the depth of $f \circ g$ is $p+q$. Three problems are raised by the way the operators are applied:

- Loading the surrounding pixels yields a significant strides in the memory access pattern, which somehow breaks data locality and therefore becomes a potential source of cache misses.
- Adjacent pixels share a common border, which yields either a redundant memory access or a redundant computation, sometimes both, thus another performance issue.
- Applying each convolution kernel separately implies several read/write operations on/to the memory (persistent location or not), yet another source of performance penalty if not managed carefully.

There are several ways to deal with the above problems. One is to fuse or chain consecutive operators whenever possible. This overcomes the repetitive read/write of the entire image, at the price of data and computation redundancy (more border pixels), thus should be done under a certain compromise. The first two issues are well tackled by *tiling*, which could be considered with operators clustering. Although tiling is a more general technique, we really need a specific analysis in order to understand how the extra data that covers each incoming tile affect the global performance when dealing with operator-based algorithms [36].

3 Related Work on Harris Detector Implementation

Corner detection is a fundamental image processing kernel that is used in many applications like *object recognition* , *object tracking*, *motion detection*, to name the most common. The algorithm is likely to be memory-bound, and the real-time constraint yields the theed for efficiency. Several implementation have been proposed in the literature, each targeting a specific device or some particular aspects of the algorithm. Lacassagne et al. [25] analyze the core of the computation and an efficient management of the data, compiler-based vectorization and shared memory parallelism are also studied. Accelerated implementation is fundamentally investigated by Wu, Lam and Srikanthan in [6], and by Tadonki et al. in [36]. An implementation on the CELL processor is provided and discussed by Saidani et al. in [35]. The last two contributions also provide a qualitative and quantitative study of *tiling*. Pipelining Harris on FPGA is studied by Aydogdu, Demirci, and C. Kasnakoglu in [38]. Xiao, Zhou and Zhang [3] describe a CUDA implementation on GPU which shows a 60 times speedup over the serial CPU version. Luo and Zhang [4] also seek a GPU implementation using the Open Graphics Library (OpenGL) and the Graphics Library Shading Language (GLSL), with a reported speedup of 73. A low complexity detector in CUDA is proposed by Rajat et al. in [5]. Beside the real-time requirement which has put more focus on the high performance computing aspect, it is worth considering concerns related to robustness and accuracy. Zhang, Li and Yang describe a multi-scale variant of the Harris algorithm, which could overcome the drawback of the conventional version like detecting false corners due to noises [8]. A novel Harris algorithm is also proposed in [6] to reduce the amount of Gaussian smoothing links. In [9], Zhao et al. propose an auto-adaptive corner detector based on neighboring points elimination method to improve robustness. However, all the aforementioned improved Harris algorithms are executed on standard CPU, where the performance is still limited. The use of many-core processors can help to accelerate the execution, provided a skillful parallelism. In order to delivers better results in terms of speed up and accuracy, Johnyet al. [7, 11] propose a new resource-aware Harris corner detection algorithm using various pruning techniques on a manycore processors, with a noticeable improvement both in throughput and accuracy.

However, there is a lack of contribution on the NUMA context, and also on on explicit SIMD implementations of the major optimization strategies. These two concerns are the focus and the position of this paper.

4 NUMA Architectures and Related Concerns

Modern CPUs are typically made up with an increasing number of cores in order to deliver a higher peak performance. As the increase of the CPU clock frequency has been somehow frozen because of circuits integration capacity and energy concerns, the trend is to provide more and more cores within a single CPU, with a fully shared memory. Nowadays and future supercomputers are just an interconnected aggregation of such nodes [40, 41, 42]. The packaging of a high number of cores withing a single chip tends to look like a hardware connected block of conventional multicores, thus providing a global memory space with a non uniform access. This Non-Uniform Memory Access (NUMA) configuration is seamless to an ordinary programmer, as there is a unique virtual addressing. Within a NUMA node, the memory system is exactly the same as for an ordinary multicore. Between NUMA nodes, specific links, like the Quick Path Interconnect (QPI), connect *local* memories together following a specific topology. A memory access in a given NUMA node is said to be *local* (resp. *remote*) if the request comes from a core within that (resp. another) NUMA node. This looks like an on-chip distributed memory configuration. Figure 3 displays a typical single-socket NUMA configuration with 4 nodes, while figure 4 (from [22]) illustrates multi-socket cases.

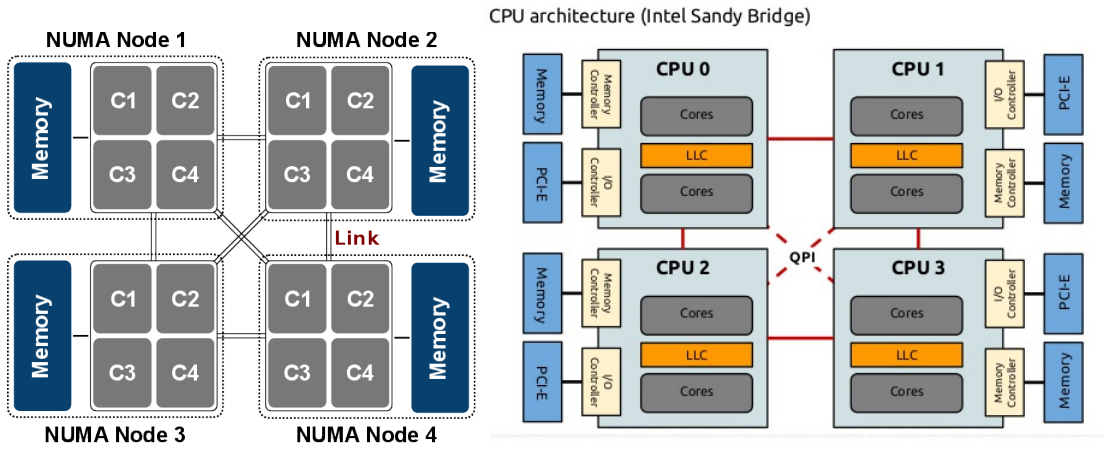

Figure 3: Sample NUMA configuration with 4 nodes

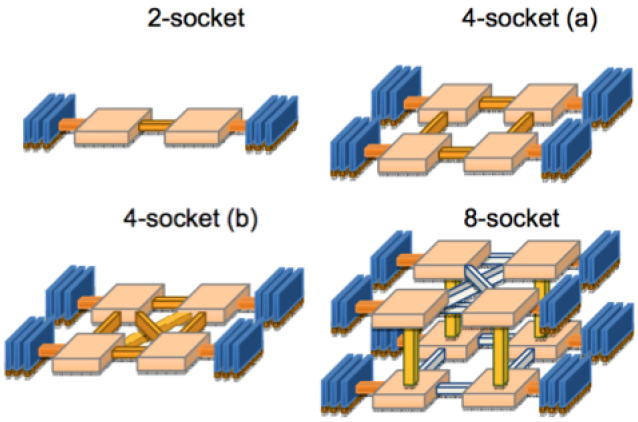

Figure 4: Sample multi-socket NUMA configurations

The NUMA configuration was designed to alleviate the bottleneck scenario where all CPU-cores use the same unique bus to access the main shared memory, thereby keeping a high probability of a good scalability over a large number of cores. Unfortunately, good scalability can be obtained only if memory accesses are mostly local. Indeed, remote accesses are more costly by nature and might incur more contention both on the QPI link and within the targeted NUMA node (because local accesses might be carried on at the same time). This potential memory controller saturation or QPI contention is the common culprit of speedup stagnation on NUMA manycores.

Efficient data placement and threads management for better scalability on NUMA systems is a hot topic. Stefan et al. propose a library for parallel programs on NUMA machines, based on array abstraction and memory allocation routines, which allows automatic tunning of data placement and accesses for better scalability [12]. Number of specific contributions [15, 16, 17, 18, 22] suggest a way to optimize thread and data placement in a NUMA system by combining data locality and thread binding, in order to reduce remote accesses. Lin et al. [13] propose an efficient stencil computations using many-core NUMA architectures, targeting higher performance and portability.

In this paper, we extend our parallel implementation of Harris detector to NUMA configurations and strive for good and stable overall scalability. Now let us describe the different implementation strategies considered in our work.

5 Optimization strategies

5.1 Separability of the Convolution Operators

All operators involved in the Harris corner detection are separable, means that each of the 3×3 convolution matrices is an outer product of two 1×3 vectors as expressed in (7), (8), and (9).

$$S_x = \frac{1}{8}\begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} = \frac{1}{8}\begin{pmatrix} 1 \\ 2 \\ 1 \end{pmatrix}\begin{pmatrix} -1 & 0 & 1 \end{pmatrix} \tag{7}$$

$$S_y = \frac{1}{8}\begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix} = \frac{1}{8}\begin{pmatrix} -1 \\ 0 \\ 1 \end{pmatrix}\begin{pmatrix} 1 & 2 & 1 \end{pmatrix} \tag{8}$$

$$G = \frac{1}{16}\begin{pmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{pmatrix} = \frac{1}{16}\begin{pmatrix} 1 \\ 2 \\ 1 \end{pmatrix}\begin{pmatrix} 1 & 2 & 1 \end{pmatrix} \tag{9}$$

The main advantage using the separated form of the operator is that it reduces the number of floating point operations (flops). Indeed, applying an operator written as $v^T u$ can be achieved by first applying v^T only for the entire row and afterwards apply u on that row. This factorization of the arithmetic operations reduces the flops, but an explicit management of intermediate storages.

5.2 Operators Clustering

Operators clustering aims at merging two operators into a single one, in order to reduce the lifetime of the intermediate results in between, thus expecting to lower or get ride of the corresponding accesses to main memory. For the Harris case, several combinations are possible [20]. The most balance one seems to be the so-called *half-pipe*, where the clusters are GRAD+MUL and GAUSS+COARSITY. Figure 5 illustrates the half-pipe workflow, whose a variant is obtained by pipelining the clusters GRAD+MUL and GAUSS+COARSITY. This version is the one considered in this paper, where we apply an array contraction technique in order to reduce the intermediate storage as in [21].

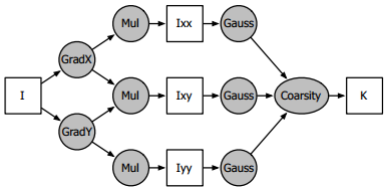

Figure 5: Harris clustering operator[20]

We are now going to explain our vectorization (SIMD) strategies, where we strive for a *minimum number* of *fully aligned* load and store statements. Although our approach is general, we consider 256-bit vector registers and AVX intrinsics for specific explanations and illustrations.

5.3 SIMD Implementation

We describe how our approach to get a nearly optimal vector implementation of the Harris-Stephensen algorithm. The main steps are the following:

5.3.1 Optimal storage strategy

One major concern when implementing a stencil algorithm is *alignment*. Just following the access pattern provided by the canonical formulation of the algorithm will lead to unaligned loads and stores, which severely impact on memory bandwidth and bus contention. To overcome this issue with the SOBEL and GAUSS operators, we (virtually) apply a upper-left shift for the output matrix, which means that (i, j) is stored at position $(i-1, j-1)$. Figure 6 illustrates our reindexation and the corresponding storage strategy.

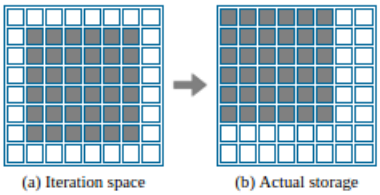

Figure 6: Upper-left shift storage

Such a storage ensures aligned write-accesses and provides a similar shape of the input for the next operator (GAUSS in this case). Thus, the shape for the computation of GAUSS (since we always exclude the border) is illustrated by figure 7.

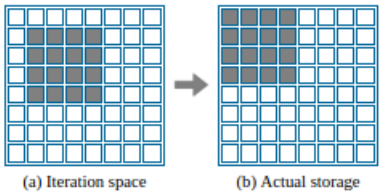

Figure 7: Storage Layout for GAUSS

In addition, we avoid useless control statements by fully computing the last block-vector of each row and store it with its two "dirty" values, which we do not consider anyway because they are located out of the actual output region. Figure 8 illustrates the view.

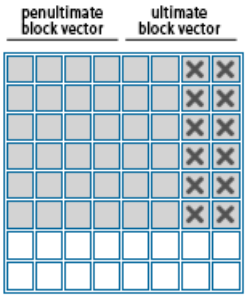

Figure 8: Storage of the last vector in a row

In case of a 8x8 image and 4-components vector registers as considered in our illustrations, we will always compute and store 2 vector registers per row of the output. Further steps will just ignore the last two columns (and of course the last two rows). Now let see how we do the core computation.

5.3.2 In-registers computation

Once a vector register is filled with input data from memory, we should strive to fully use it to compute dependent variables. For this purpose, we proceed as follows. All our filters are *separable*, means they have a generic form which can be described by 3×3 mask of the generic form (10)

$$A = \begin{pmatrix} a & \alpha a & \beta a \\ b & \alpha b & \beta b \\ c & \alpha c & \beta c \end{pmatrix} = \begin{pmatrix} a \\ b \\ c \end{pmatrix} \begin{pmatrix} 1 & \alpha & \beta \end{pmatrix}. \tag{10}$$

Assume we need to apply operator ([**?**]) at a point $I_{i,j}$ (denotes by $@_{(i,j)}$) of the input image. We have (11)

$$\begin{array}{rlll} @_{(i,j)} = & aI_{i-1,j-1} & +\alpha aI_{i-1,j} & +\beta aI_{i-1,j+1} \\ + & bI_{i,j-1} & +\alpha bI_{i,j} & +\beta bI_{i,j+1} \\ + & cI_{i+1,j-1} & +\alpha cI_{i+1,j} & +\beta cI_{i+1,j+1} \end{array} \tag{11}$$

which can be written as (12)

$$\begin{aligned} @_{(i,j)} = \ & (aI_{i-1,j-1} + bI_{i,j-1} + cI_{i+1,j-1}) + \\ & \alpha(aI_{i-1,j} + bI_{i,j} + cI_{i+1,j}) + \\ & \beta(aI_{i-1,j+1} + bI_{i,j+1} + cI_{i+1,j+1}) \ . \end{aligned} \tag{12}$$

Thus, in order to computer a block vector in a row i, starting from position j, we need the vector registers staring at $(i-1, j-1)$, $(i, j-1)$, and $(i+1, j-1)$, and also their next adjacent ones. Figure 9 illustrates the configuration, where the block vector to be computed is represented by gray cells, and requires four registers r0, r1, r2, r3 (we assume 4-components vector registers).

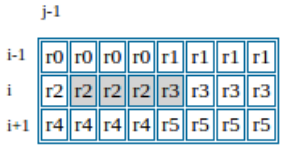

Figure 9: SIMD memory mapping

Given two vector registers r and s, we denote by $\lhd_k(r, s)$ the vector register obtained from r by performing k left shifts, and taking for the last k components the first k components of s. For example, if $u = (u_0, u_1, u_2, u_3)$ and $v = (v_0, v_1, v_2, v_3)$, then $\lhd_1(u, v) = (u_1, u_2, u_3, v_0)$ and $\lhd_2(u, v) = (u_2, u_3, v_0, v_1)$. Given this notation, the computation of our block vector as described in (12) can be specified by expression (13)

$$\begin{aligned} & [a \times r_0 + b \times \lhd_1(r_0, r_1) + c \times \lhd_2(r_0, r_1)] + \\ & \alpha \times [a \times r_2 + b \times \lhd_1(r_2, r_3) + c \times \lhd_2(r_2, r_3)] + \\ & \beta \times [a \times r_4 + b \times \lhd_1(r_4, r_5) + c \times \lhd_2(r_4, r_5)] \end{aligned} \tag{13}$$

For the computation of the next block, we just need to do $r_0 := r_1$, $r_2 := r_3$, $r_4 := r_5$, and only reload r_1, r_3, and r_5, thereby saving three loads. Thus, we always use a combination of non overlapping registers, and each of them is used twice to compute two consecutive block vectors of the current row.

5.3.3 Factorization of the computation

The previously described in-registers computation can be applied using the partial application of the mask restricted to its first column. Indeed, (13) can be written as (14)

$$\begin{aligned} & a \times [r_0 + \alpha \times r_2 + \beta r_4] \\ & b \times [\lhd_1(r_0, r_1) + \alpha \lhd_1 (r_2, r_3) + \beta \lhd_1 (r_4, r_5)] \\ & c \times [\lhd_2(r_0, r_1) + \alpha \lhd_2 (r_2, r_3) + \beta \lhd_2 (r_4, r_5)] \end{aligned} \tag{14}$$

which, considering the distributivity of $\lhd_k$ over addition, yields (15)

$$\begin{aligned} & a \times [r_0 + \alpha \times r_2 + \beta r_4] \\ & b \times [\lhd_1(r_0 + \alpha \times r_2 + \beta r_4, r_1 + \alpha \times r_3 + \beta r_5)] \\ & c \times [\lhd_2(r_0 + \alpha \times r_2 + \beta r_4, r_1 + \alpha \times r_3 + \beta r_5)] \end{aligned} \tag{15}$$

Finally, if $u = r_0 + \alpha \times r_2 + \beta \times r_4$ and $v = r_1 + \alpha \times r_3 + \beta \times r_5$, then (13) can be written as (16)

$$a \times u + b \times \lhd_1(u, v) + c \times \lhd_2(u, v). \tag{16}$$

Thus, for the computation of an entire row, the typical steps at each iteration are displayed in Table 1, where r is first computed within the prologue phase.

1: $u = load(I_{i-1,j})$; $v = load(I_{i,j})$; $w = load(I_{i+1,j})$;
2: $s := u + \alpha \times v + \beta \times w$;
3: $t := a \times r + b \times \lhd_1(r, s) + c \times \lhd_2(r, s)$;
4: $store(t)$;
5: $j := next(j)$; {next block vector}
6: $r := s$;

Table 1: Typical computation of a block vector

For the special cases of a null row or a null column, for instance $\alpha = 0$ for SOBELX and $b = 0$ for SOBELY, and/or if one entry of the mask equals ± 1, the code should be simplified accordingly. More precisely, line 2 becomes $s := u + \beta \times w$ if $\alpha = 0$ (we avoid one register calculation) and line 3 becomes $t := a \times r + c \times \lhd_2(r, s)$ (we avoid one shift).

5.3.4 Shifting virtually concatenated registers

With SSE or AVX, getting $vv = \lhd_1(v1, v2)$ from two given vector registers u and v can be achieved using basic register-register instructions only, thus no need for any kind of memory access. We provided the steps corresponding to AVX case considered in this work. Figure 10 describes how to compute $\lhd_1(v1, v2)$ with single precision components.

```
vv  = _mm256_permute_ps(v1, 0b00111001);
u1  = _mm256_permute2f128_ps(vv, vv, 0x81);
u1  = _mm256_blend_ps(vv, u1, 0b10001000);
vv  = _mm256_permute_ps(v2, 0b00111001);
u2  = _mm256_permute2f128_ps(vv, vv, 0);
vv  = _mm256_blend_ps(u1, u2, 0b10000000);
```

Figure 10: Left-shift of two registers in AVX

If necessary, a step-by-step simulation of the instructions in figure 10 is advised for ordinary readers in order to understand the code and get convinced. Figure 11 provides the code for $vv = \lhd_2(v1, v2)$.

```
vv = _mm256_permute_ps(v1, 0b01001110);
u1 = _mm256_permute2f128_ps(vv, vv, 0x81);
u1 = _mm256_blend_ps(vv, u1, 0b11001100);
vv = _mm256_permute_ps(v2, 0b01001110);
u2 = _mm256_permute2f128_ps(vv, vv, 0);
vv = _mm256_blend_ps(u1, u2, 0b11000000);
```

Figure 11: Two-left-shift of two registers in AVX

5.4 Loop Transformations and Array Contraction

When it comes to stencil computation, cache memory needs a special consideration. Indeed, each point of the iteration space contributes to the calculation associated to its neighborhood. If we consider our SIMD scheduling as previously described, we can see that each line i is loaded three time for lines $i-1$, i, and $i+1$. We can reduce this load redundancy by computing two lines at each iteration (as described in table 1). In fact, computing line i (resp. $i+1$) requires lines $\{i-1, i, i+1\}$ (resp. $\{i, i+1, i+2\}$), thus each time we compute a chunk of line i, we can compute the corresponding chunk for line $i+1$ by loading the only missing block from line $i+2$. Figure 12 illustrates the computation flow.

	j-1							
i-1	r0	r0	r0	r0	r1	r1	r1	r1
i	r2	r2	r2	r2	r3	r3	r3	r3
i+1	r4	r4	r4	r4	r5	r5	r5	r5
i+2	r6	r6	r6	r6	r7	r7	r7	r7

Figure 12: Data layout illustration for double-row computation

This approach clearly tries to reduce the number of loads. However, there are two factors that already help to reduce the effective overheads of the loads. First, we manipulate 256-wide vector registers (32 bytes) and the size of the cache line is 64 bytes, so each load brings into the cache enough data for two registers, and our access pattern matches this granted prefetch. In addition, after a given line i is visited (for $i-1$), it is reused twice in the next two iterations (for lines i and $i+1$). Since that row is still at least in the L2 cache, the aforementioned two reloads will not have a noticeable impact. Moreover, for the two SOBEL (SOLBELX and SOBELY), we use the same input, thus another advantageous fact for the loads. Considering the MULTIPLY that follows the SOBEL, we produce three results, thus three (distant) stores. Thereby, in addition to the fact that

write-cache mechanism is more complex, the cost of write accesses is likely to be more important than that of read accesses. The situation is reversed with GAUSS+COARSITY where we produce a single output from three inputs.

The second strategy to benefit from the cache is the so-called *array contraction*, which aims at reducing the memory footprint when collapsing two loops. Thereby, intermediate writes and reads hopefully occur at a certain level of the cache (typically L2). In our case, the first loop is for SOBEL+MUL and the second is for GAUSS+COARSITY. As previously explained, we need three consecutive rows from SOBEL+MUL in order to compute one row with GAUSS+COARSITY. Thus, after rows 0 and 1 are produced by SOBEL+MUL, we can start iterating on the collapsed loop body [SOBEL+MUL]→[GAUSS+COARSITY] to produce consecutive rows of the final output (see Table 2).

1: $\{l_0^{(I)}, l_1^{(I)}, l_2^{(I)}\} \rightarrow$ [SOBEL+MUL] $\rightarrow \{l_0^{(S)}, l_1^{(S)}\}$;
2: $for(i = 2; i < n - 2; i++)\{$
3: $\quad \{l_{i-1}^{(I)}, l_i^{(I)}, l_{i+1}^{(I)}\} \rightarrow$ [SOBEL+MUL] $\rightarrow l_i^{(S)}$;
4: $\quad \{l_{i-2}^{(S)}, l_{i-1}^{(S)}, l_i^{(S)}\} \rightarrow$ [GAUSS+COARSITY] $\rightarrow l_{i-2}^{(O)}$;
5: $\}$

Table 2: Collapsed Sobel and Gauss

At step i (resp. $i+1$), we need to be able to house $l_{i-2}^{(S)}$, $l_{i-1}^{(S)}$, and $l_i^{(S)}$ (resp. $l_{i-1}^{(S)}$, $l_i^{(S)}$, and $l_{i+1}^{(S)}$), thus we can just replace $l_{i-2}^{(S)}$ by $l_{i+1}^{(S)}$. Thereby, a space for 3 rows can be used for intermediate data between [SOBEL+MUL] and [GAUSS+COARSITY] following a cyclic (mod 3) assignment. This contraction yields a recurrent reuse of a smaller memory space, which will certainly remain into the cache during all the iterations, thus a noticeable performance improvement.

5.5 Another Optimization from Thresholding

As heavy write accesses certainly degrade absolute performance as well as scalability, a noticeable benefit can be expected from any strategy that aims at reducing write accesses. In the case of Harris algorithm, the *thresholding* can be used to avoid writing the coarsity values of unselected pixels. This can be done by the AVX code of figure 13.

```
cmp  = _mm256_cmp_ps(reg_coasity, reg_threshold, _CMP_LE_OS); // compare for <= 0
mask = _mm256_movemask_ps(cmp);                               // mask of the comparison result
if (mask != 0)                                                // then some pixels are selected
     _mm256_store_ps(&C[i,j], reg_coarsity);                  // store this coarsity register
```

Figure 13: Write accesses reduction from the thresholding

As the number of selected pixels is a very low fraction of the overall number of pixels (even 0.01% seems too much), there is a clear opportunity for a significant saving in write accesses. The coarsity matrix can be initialized to zero (which corresponds to what the thresholding would have done for unselected pixels), then updated afterwards by the computed values for the selected pixels. Another way is to completely forget unselected pixels and write the coarsity of the selected ones in a compact storage (non-sparse way). For our experimental results, we consider the former strategy.

6 Shared Memory Parallel Implementation

6.1 Basic Parallel Allocation

We basically consider a row-strip distribution of each working 2D array, and assign each block of rows to a specific thread. This straightforward partitioning is applied for every operator of the Harris algorithm. In addition, we apply thread binding in order to avoid thread migration and prepare for NUMA considerations. For intermediate storage, each thread has its one memory space (indexed by its *id*). Since each thread computes the full Harris for a block of rows of the final output, there is no need for any kind of synchronization. On a NUMA configuration, all allocations

are performed on node 0, which means that all reads (resp. writes) are done from (resp. on) node 0. For sure, such a NUMA-unaware is likely to show a poor scalability over the entire machine. Note that this is configuration that a programmer might obtain with basic OpenMP directives. The *first touch policy* will just change the node where everything will be definitely physically allocated. We now describe how we handle NUMA considerations.

6.2 NUMA-aware Adaptation

Considering a NUMA configuration, we first choose to load (resp. store) and keep the entire input (resp. output) image on node 0, and use an internal storage for all intermediate memory allocations and accesses (thus making them local). Figure 14 illustrates our organization of the memory allocation and the associated diagram.

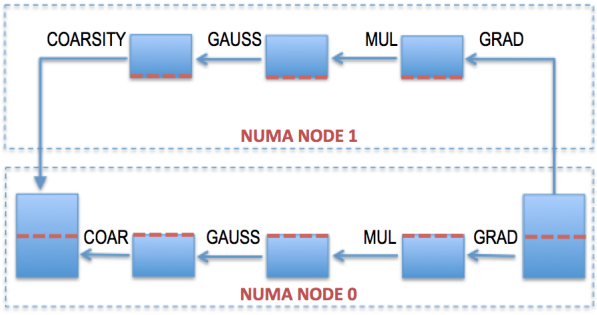

Figure 14: Our basic parallel scheduling

This organization, which can be easily implemented with specialized memory allocator routines (*libnuma* library for instance) still contains remote memory accesses which might become really hindering with larger images. To solve this, we chose to implement a block-cyclic allocation [43] using a memory binding routine, which can perform a remapping of a given memory chunk onto a specific node. The core of the corresponding C is provided in figure 15.

```
for(i=0;i<nnodes;i++){
  newnode = 1 << (nodemask[i]);
  err = mbind((void *)t, (nb_pages*PAGE_SIZE), MPOL_BIND , &newnode, nb_nodes, MPOL_MF_MOVE);
  t = t + (nb_pages*PAGE_SIZE);
}
```

Figure 15: Block memory binding

Let us now provide and comment our experimental performance results.

7 Performance results and comments

7.1 Technical Context

In order to evaluate our implementation strategies on a NUMA many-core machine, we consider an Intel Xeon E5-2669 v4 (Broadwell-EP) CPU having a total of 44 cores. This is a dual-socket configuration with 4 11-cores NUMA nodes. We chose to consider 8 cores per node in order to have a balanced task and data distribution with 2^k threads, as we plan to consider square images with power of two sizes. Figure 16 displays the main technical specifications of our machine.

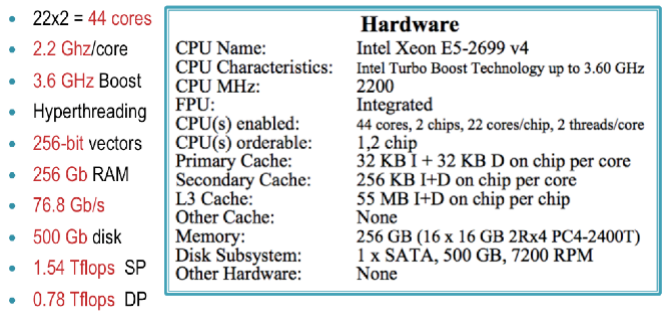

Figure 16: Our Broadwell configuration

7.2 Basic Complexity Analysis

Given a $n \times n$ image, the computation of a separable 3×3 filter $u^T v$ can be achieved with a minimum number of floating operations (flops) given by

$$(o(u) + o(v))n^2, \tag{17}$$

where $o(u)$ (resp. v) is the flops corresponding to a single application of u (resp. v). For this case of the Harris algorithm, we have:
SOBELX: $u = (1, 2, 1)$, $v = (1, 0, 1)$, $o(u) + o(v) = 3 + 1 = 4$
SOBELY: $u = (1, 0, 1)$, $v = (1, 2, 1)$, $o(u) + o(v) = 1 + 3 = 4$
GAUSS: $u = (1, 2, 1)$, $v = (1, 2, 1)$, $o(u) + o(v) = 3 + 3 = 6$
In addition, we need to consider 1 MUL (3 flops) and 1 COARSITY (7 flops). Instead of applying the factor $1/8$ to the gradients and $1/16$ to the gaussians, we multiply the final result by $1/(8 * 16)$. Thus, an optimal computation of Harris consumes a number of flops per pixel equals to

$$4 + 4 + 3 + 6 \times 3 + 7 + 1 = 37. \tag{18}$$

For each execution on a n^2 image (single precision), the GFLOPS performance is given by

$$(37 \times n^2)/(10^9 \times t(s)), \tag{19}$$

where $t(s)$ is the execution time in seconds.

7.3 Measured Performance Results

We consider square images with power of two sizes ranging from 1024 (2^{10}) to 32768 (2^{15}). The intensity of the pixels are generated randomly using the *rand*() function in C.

7.3.1 Sequential Case

N	T(s)	GFlops	%Peak
1 024	0.00312	12.49	64
2 048	0.01168	13.28	69
4 096	0.04829	12.85	66
8 192	0.22431	10.07	57
16 384	0.97930	10.14	52
32 768	3.84535	10.33	53

Table 3: Performance results of our sequential implementation

The first remark is that we are always above 50% of the peak performance, with a weak deviation. Since we are dealing with a memory-bound application (exacerbated by our flops optimizations) and our hardware provides an important Flops capability, we can claim from the measured timings (Table 3) that our implementation is globally efficient. We now examine the behavior of its parallelization.

7.3.2 NUMA-unaware Parallel Case

	1024×1024		2048×2048		4096×4096		8192×8192		16384×16384	
#cores	T(s)	Acc	T(s)	Acc	T(s)	Acc	T(s)	Acc	T(s)	Acc
1	0.00303	1.00	0.01166	1.00	0.04756	1.00	0.22143	1.00	0.97425	1.00
2	0.00159	1.90	0.00587	1.99	0.02408	1.98	0.12409	1.78	0.48822	2.00
4	0.00085	3.56	0.00299	3.90	0.01311	3.63	0.06421	3.45	0.25957	3.75
8	0.00050	6.11	0.00191	6.09	0.00835	5.70	0.03493	6.34	0.14260	6.83
16	0.00074	4.10	0.00239	4.88	0.01101	4.32	0.04478	4.94	0.18017	5.41
32	0.00111	2.73	0.00263	4.43	0.01055	4.51	0.05252	4.22	0.21648	4.50

Table 4: Performance results of our NUMA-unaware parallelization

Table 4 shows a good speedup withing one NUMA node (each NUMA node executes a maximum of 8 threads). However, when more that one nodes are involved, a severe slowdown can be observed. This is the typical effect of remote accesses and associated memory controller saturation in a NUMA configuration. We now see the impact of our NUMA-aware adaptation.

7.3.3 NUMA-aware Parallel Case

	1024×1024		2048×2048		4096×4096		8192×8192		16384×16384	
#cores	T(s)	Acc	T(s)	Acc	T(s)	Acc	T(s)	Acc	T(s)	Acc
1	0.00306	1.00	0.01176	1.00	0.04777	1.00	0.22193	1.00	0.97619	1.00
2	0.00160	1.92	0.00587	2.00	0.02398	1.99	0.11857	1.87	0.50325	1.94
4	0.00100	3.06	0.00300	3.92	0.01277	3.74	0.06344	3.50	0.25939	3.76
8	0.00071	4.34	0.00203	5.79	0.00847	5.64	0.03549	6.25	0.14251	6.85
16	0.00063	4.90	0.00135	8.68	0.00460	10.38	0.01850	12.00	0.07259	13.45
32	0.00132	2.32	0.00069	17.11	0.00257	18.60	0.01072	20.71	0.04270	22.86

Table 5: Performance results of our NUMA-aware parallelization

We can see from Table 5 that there is a significant improvement of the speedup with several NUMA nodes. The pathologic 1024 case is certainly due to the typical thread starvation effect, there is no need to use more that one NUMA node for small instances. Let now check how write avoiding through thresholding affect our execution timings.

7.3.4 Thresholding Optimization Case

	1024×1024		2048×2048		4096×4096		8192×8192		16384×16384	
#cores	T(s)	Acc	T(s)	Acc	T(s)	Acc	T(s)	Acc	T(s)	Acc
1	0.00293	1.00	0.01127	1.00	0.04685	1.00	0.21433	1.00	0.93773	1.00
2	0.00153	1.92	0.00569	1.98	0.02359	1.99	0.10751	1.99	0.46875	2.00
4	0.00080	3.69	0.00293	3.85	0.01172	4.00	0.05875	3.65	0.24492	3.83
8	0.00047	6.29	0.00153	7.38	0.00605	7.75	0.03152	6.80	0.12462	7.52
16	0.00055	5.32	0.00092	12.29	0.00334	14.04	0.01642	13.06	0.06365	14.73
32	0.00115	2.54	0.00055	20.68	0.00181	25.83	0.00879	24.40	0.03276	28.62

Table 6: Evaluation of the thresholding optimization

We can see another improvement in the speedup, but a rather marginal one in the absolute performance on a single core. The former is certainly due to the fact that there is a less stress on the bus to the main memory. Table 7 illustrates the effect of the thresholding optimization with 0.01% pixels selected. There is an observed improvement, making it more significant requires more investigations.

N	T(s)	GFlops	%Peak	%Peak (no threshold)
1 024	0.00293	13.30	72	64
2 048	0.01127	14.31	71	69
4 096	0.04685	13.24	68	66
8 192	0.21433	10.53	60	57
16 384	0.93773	10.58	54	52
32 768	3.60662	11.01	57	53

Table 7: Impact of the thresholding optimization

8 Conclusion

In this paper, we revisit the Harris corner detection algorithm. Targeting a NUMA manycore, we first start by striving for an efficient single-core implementation, emphasizing on an optimal memory organization. Indeed, as memory bus and controller are shared by the cores, it is crucial to reduce the number of compulsory memory requests and to remain on cache as long as possible. In addition, we study the effect of NUMA configuration and provide a solution based on a standard memory distribution onto the NUMA nodes through memory binding routines. Using the thresholding selection to reduce the number of write access to the main memory is also explored. This last aspect needs more investigation. An extension of the SIMD implementation with `short int` data type (thus a wider vectorization) is also worth considering.

References

[1] Hans Moravec, *Obstacle avoidance and navigation in the real world by a seeing robot rover, In tech.*, report CMU-RI-TR-80-03, Robotics Institute, Carnegie Mellon University & doctoral dissertation, Stanford University. September 1980.

[2] C. Harris and M. Stephens, *A combined corner and edge detector*, 4th ALVEY Vision Conference, 1988.

[3] Xiao Han, Zhou Qinglei, Zhang Zuxun,*Parallel Algorithm of Harris Corner Detection Based on Multi-GPU*. Geomatics and Information Science of Wuhan University. 37 (7): 876-881, 2012

[4] Shuhua Luo, Jun Zhang,*Accelerating Harris Algorithm with GPU for Corner Detection*, Seventh International Conference on Image and Graphics,978-0-7695-5050-3/13 26.00 © 2013 IEEE,DOI 10.1109/ICIG.2013.36.

[5] Rajat Phull, Pradip Mainali, Qiong Yang,Patrice Rondao Alface,*Low Complexity Corner Detector Using CUDA for Multimedia Applications*, MMEDIA 2011 : The Third International Conferences on Advances in Multimedia,IARIA, 2011. ISBN: 978-1-61208-129-8.

[6] M. Wu, N. Ramakrishnan, S.-K. Lam, and T. Srikanthan, *Low-complexity pruning for accelerating corner detection.* In Circuits and Systems (ISCAS), 2012 IEEE International Symposium on, pages 1684–1687. IEEE, 2012.

[7] Johny Paul1, Walter Stechele1 ,Manfred Krohnert ¨ 2, Tamim Asfour2 ,Benjamin Oechslein3, Christoph Erhardt3, Jens Schedel3, Daniel Lohmann3, and Wolfgang Schroder-Preikschat,*Resource-Aware Harris Corner Detection based on Adaptive Pruning* ,German Research Foundation (DFG) as part of the Transregional Collaborative Research Centre "Invasive Computing" (SFB/TR 89).

[8] Zhang Xiaohong, Li Bo, Yang Dan. *A Novel Harris Mullti-scale Corner Detection Algorithm* . Journal of Electronics Information Technology. 29 (7): 1735-1738, 2007.

[9] Zhao Wanjin, Gong Shengrong, Liu Chunping, Shen Xiangjun. *ADaptive Harris Corner Detection Algorithm* . Computer Engineering. 34 (10): 212-215, 2008.

[10] Guo Chenguang, Li Xianglong, Zhong Linfeng, Luo Xiang. *A Fast and Accurate Corner Detector Based on Harris Algorithm* . Third International Symposium on Intelligent Information Technology Application. Nanjing, China, 49-51, 2009.

[11] Johny Paul, Walter Stechele,Ericles Sousa, Vahid Lari,Frank Hannig, Jurgen Teich,Manfred Krohnert,Tamim Asfour, *Self-Adaptive Harris Corner Detector on Heterogeneous Many-Core Processor* , German Research Foundation (DFG) as part of the Transregional Collaborative Research Centre "Invasive Computing"(SFB/TR 89).

[12] Stefan Kaestle, Reto Achermann, Timothy Roscoe, Tim Harris,*Shoal: smart allocation and replication of memory for parallel programs* ,USENIX Annual Technical Conference,July 8–10, 2015 • Santa Clara, CA, USA ISBN 978-1-931971-225,2015.

[13] P. Lin, Q. Yi, D. Quinlan, C. Liao, Y. Yan, *Automatically Optimizing Stencil Computationson Many-core NUMA Architectures*,International Workshop on Languages and Compilers for Parallel Computing Rochester, NY, United States September 28, 2016 through September 30, 2016.

[14] Adrian Schüpbach, Simon Peter, Andrew Baumann, Timothy Roscoe,Paul Barham, Tim Harris, Rebecca Isaacs,*Embracing diversity in the Barrelfish manycore operating system* ,MMCS'08, Boston, Massachusetts, USA, June 24, 2008.

[15] Mohammad Dashti,Alexandra Fedorova,Justin Funston,Fabien Gaud,Renaud Lachaize ,Baptiste Lepers,Vivien Quema ,Mark Roth,Traffic Management:*A Holistic Approach to Memory Placement on NUMA Systems*,ASPLOS'13, March 16–20, 2013, Houston, Texas, USA 2013 ACM 978-1-4503-1870-9/13/03.

[16] Baptiste Lepers, Vivien Quéma, and Alexandra Fedorova. *Thread and memory placement on NUMA systems: asymmetry matters* ,In Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '15). USENIX Association, Berkeley, CA, USA, 277-289

[17] Renaud Lachaize, Baptiste Lepers, and Vivien Quéma.*MemProf:A memory Profiler for NUMA Multicore Systems*, In USENIX ATC,2012

[18] Collins, Alexander and Harris, Tim and Cole, Murray and Fensch, Christian. *LIRA: Adaptive Contention-Aware Thread Placement for Parallel Runtime Systems*, In ROSS, 2015

[19] *wikipedia.org/wiki/Non uniform memory access* wikipedia

[20] Antoine Pédron Florence Laguzet Tarik Saidani Pierre Courbin Lionel Lacassagne Michèle Gouiffès, *Parallélisation d'opérateurs de TI:multi-coeurs, Cell ou GPU ?*,Traitement du Signal. Volume 27 – n ° 2/2010, pages 161 à 183.

[21] Lionel Lacassagne, Daniel Etiemble, Hassan Zahraee, Alain Dominguez, Pascal Vezoll,*High Level Transforms for SIMD and Low-Level Computer Vision Algorithms* Symposium on Principles and Practice of Parallel Programming / WPMVP, Feb 2014, Orlando, Florida, United States. pp.8 .

[22] Y. Li, I. Pandis, R. Mueller, V. Raman, and G. Lohman, *NUMA-aware algorithms: the case of data shuffling*
http://www.pandis.net/resources/cidr13numashuffling.pdf
2013.

[23] C. Tadonki, *Scalable NUMA-Aware Wilson-Dirac on Supercomputers*, 2017 International Conference on High Performance Computing & Simulation (HPCS 2017), Genoa - Italy, Jul. 2017

[24] R. Al-Omairy, G. Miranda, H. Ltaief, R. M. Badia, X. Martorell, J. Labarta, and D. Keyes, *Dense Matrix Computations on NUMA Architectures with Distance-Aware Work Stealing*, Upercomputing Frontiers and Innovations, vol. 2(1), 2015.

[25] L. Lacassagne, D. Etiemble, A. Hassan Zahraee, A. Dominguez, P. Vezolle, *High Level Transforms for SIMD and Low-level Computer Vision Algorithms*, Workshop on Programming Models for SIMD/Vector Processing (WPMVP 14), pp.49-56 ,Orlando Florida - USA, 2014.

[26] H. Moravec, *Obstacle avoidance and navigation in the real world by a seeing robot rover*, Tech Report CMU-RI-TR-3, Carnegie-Mellon University, Robotics Institute, September 1980.

[27] C. Harris and M. Stephens, *A combined corner and edge detector*, in Proceedings of the 4th ALVEY Vision Conference, pages 147–151, 1988.

[28] A. Yilmaz, O. Javed, and M. Shah, *Object tracking: A survey*, ACM Computing Surveys 38(4), 2006.

[29] P.M. Roth and M. Winter, *Survey of appearance-based methods for object recognition*, Technical Report ICG-TR-01/08, Inst. for Computer Graphics and Vision, Graz University of Technology, 2008.

[30] C. Alias, F. Baray and A. Darte, *Bee+Cl@k: An Implementation of Lattice-Based Array Contraction in the Source-to-Source Translator ROSE*, LCTES'07, San Diego - California - USA, June 13-16, 2007.

[31] S. Bhaskaracharya, U. Bondhugula, A. Cohen, *SMO: An Integrated Approach to Intra-array and Inter-array Storage Optimization*, ACM Symposium on Principles of Programming Languages (POPL 2016), Florida-USA, Jan 20-23 , 2016.

[32] Y. Song, R. Xu, C. Wang and Z. Li, *Improving Data Locality by Array Contraction*, IEEE Transactions on Computers, Vol. 53 (9), September, 2004.

[33] P. Clauss, E. Altintas, M. Kuhn, *Automatic Collapsing of Non-Rectangular Loops*, Parallel and Distributed Processing Symposium (IPDPS), Orlando - Florida - USA, 29 May-2 June, 2017.

[34] S. Y. Kung, S. C. Lo and P. S. Lewis, *An Optimal Systolic Design for the Transitive Closure and the Shortest Path Problems*, IEEE Transactions on Computers, vol. C-36, no 5, p. 603-614, 1987.

[35] T. Saidani, L. Lacassagne, J. Falcou, C. Tadonki, Samir Bouaziz, *Parallelization Schemes for Memory Optimization on the Cell Processor : A Case Study on the Harris Corner Detector*, Transactions on High-Performance Embedded Architectures and Compilers, volume 3(3) 2011.

[36] C. Tadonki, L. Lacassagne, E. Dadi, M. Daoudi, *Accelerator-based implementation of the Harris algorithm*, 5th International Conference on Image Processing (ICISP 2012), Agadir, Morocco, June 28-30, 2012.

[37] F. Hosseini, A. Fijany and J.-G. Fontaine, *Highly Parallel Implementation of Harris Corner Detector on CSX SIMD Architecture*, th Workshop on Highly Parallel Processing on a Chip (HPPC), Ischia- Naples - Italy, August 31, 2010.

[38] M. F. Aydogdu, M. F. Demirci, and C. Kasnakoglu, *Pipelining Harris Corner Detection with a Tiny FPGA for a Mobile Robot*, IEEE International Conference on Robotics and Biomimetics (ROBIO)Shenzhen, China, December 2013

[39] F. Mokhtarian and F. Mohanna, *A performance evaluation of corner detectors using consistency and accuracy measures*,Computer Vision and Image Understanding, vol.102, 2006, pp. 81-94.

[40] C. Tadonki, *High Performance Computing as a Combination of Machines and Methods and Programming*, Habilitation Thesis, University Paris-Sud, May 2013.

[41] P. Kogge et al., *ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems*, DARPA report, 2008.

[42] E. W. Nagel, D. B. Krner,and M. M. Resch (Eds.), *High Performance Computing in Science and Engineering* , Springer Book Archives, 2013.

[43] C. P. Ribeiro — J.-F. Méhaut, *MAi: Memory Affinity interface*, INRIA Research Report n° 0359, 2008. ,